Technische Universität München

Fakultät für Maschinenwesen
LRT – Lehrstuhl für Raumfahrttechnik

Kompatibilitätsmodellierung im Systems-Engineering Umfeld

Dipl. Inf. Univ. Markus Franz Brandstätter

Vollständiger Abdruck der von der Fakultät für Maschinwesen der Technischen Universität München zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigten Dissertation.

Vorsitzender: Univ. Prof. Dr.-Ing. Frank Schiller
1. Univ. Prof. Dr. rer. nat. Ulrich Walter
2. Univ. Prof. Dr. rer. nat. Dr. rer. nat. habil. Dr. h.c. Manfred Broy

Die Dissertation wurde am 19. Januar 2009 bei der Technischen Universität München eingereicht und durch die Fakultät für Maschinenwesen am 09. Juni 2009 angenommen.

Danksagung

Die vorliegende Arbeit entstand im Rahmen meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Lehrstuhl für Raumfahrttechnik – Systems Engineering Group – an der Technischen Universität München in den Jahren 2004 bis 2009.

Sehr herzlich bedanke ich mich bei meinem Doktorvater Herrn Prof. Dr. rer. nat. Ulrich Walter für die warmherzige Aufnahme an seinem Lehrstuhl, das angenehme Arbeitsklima, die vielen fruchtbaren fachlichen Diskussionen, das ständig offene Ohr für seine Mitarbeiter und vor allem für die wissenschaftliche Freiheit die ich an seinem Lehrstuhl hatte, um meine Ideen zu verwirklichen. Besonders möchte ich mich auch bei Herrn Prof. Dr.-Ing. E. Igenbergs bedanken, der mich so wunderbar in die hohe Kunst des Systems Engineerings eingeführt hat. Dabei denke ich vor allem an die zahlreichen anregenden fachlichen Diskussionen und Kommentare zum Thema „Sichtweise eines Maschinenbauingenieurs“ und vor allem seine unermüdliche Hilfsbereitschaft.

Bedanken möchte ich mich auch bei Herrn Prof. Dr. rer. nat. Dr. h.c. Manfred Broy für die Übernahme des zweiten Gutachtens sowie Herrn Prof. Dr.-Ing. Frank Schiller für den Prüfungsvorsitz.

Insbesondere möchte ich Frau Dipl. Inf. Dagmar Koß für die ausgezeichnete Zusammenarbeit während des vom BMBF geförderten Forschungsvorhabens MOKOMA und darüber hinaus danken. Ohne die gute Zusammenarbeit wären weder die gemeinsame Entwicklung der theoretischen Grundlagen für die Kompatibilitätsmodellierung noch die graphische Modellierungssprache (U)CML möglich gewesen. In diesem Zusammenhang möchte ich mich auch bei allen Studenten bedanken, die an der Entwicklung des graphischen Editors *(U)CML-ed* mitgearbeitet haben.

Meinen besonderen Dank möchte ich Frau Dipl. Inf. Carolin Eckl aussprechen für das gewissenhafte Korrekturlesen und die zahlreichen guten Kommentare und Anregungen, durch die sowohl die sprachliche als auch die inhaltliche Qualität dieser Arbeit erheblich verbessert wurde.

Außerdem möchte ich mich bei allen Mitarbeiten des Lehrstuhls für Raumfahrttechnik, und insbesondere der Systems Engineering Group, danken für die herzliche Aufnahme, die unzähligen Gespräche und fachlichen Diskussionen und vor allem die inspirierende Zusammenarbeit. Dabei denke ich vor allem an die zahlreichen fruchtbaren fachlichen Diskussionen mit Frau Dipl. Inf. Carolin Eckl, Herrn Dr.-Ing. Andreas Peukert und Herrn Dr.-Ing. Michael Schiffner.

Aus tiefstem Herzen danke ich meinem Vater für die anhaltende, unermüdliche Unterstützung und seine guten inspirierenden Anregungen und vor allem seinen Beistand während der „heißen Phase“ der Dissertationsfertigstellung.

Markus Brandstätter München, Januar 2009

Zusammenfassung

Die vorliegende Arbeit beschäftigt sich sowohl mit der formalen kompatibilitätskonformen Modellierung von komplexen technischen Systemen als auch mit der Integration des objektorientierten Kompatibilitätsmodellbildungsprozesses in das Systems Engineering Umfeld.

Um dies zu erreichen, werden in dieser Arbeit zunächst die grundlegenden Begriffe und Definitionen aus dem Systems Engineering Umfeld, die für die Kompatibilitätsmodellierung notwendig sind, vorgestellt und erläutert, bevor mit der Beschreibung der Grundlagen der objektorientierten modellbasierten Kompatibilitätsmodellierung und -bestimmung begonnen wird. Nachdem die Eckpfeiler der objektorientierten Kompatibilitätsmodellierung gesetzt worden sind, werden die wesentlichen Anforderungen an eine interdisziplinär anwendbare Kompatibilitätsmodellierungssprache für technische Systeme festgelegt. Aufbauend auf diesen Anforderungen werden zunächst vier unterschiedliche Modellierungssprachen und Methoden für technische Systeme eingeführt, sowie deren Anwendbarkeit für die Kompatibilitätsmodellierung und -bewertung erörtert. Anschließend wird die neu entwickelte interdisziplinär anwendbare, domänenübergreifende Kompatibilitätsmodellierungssprache (U)CML, sowie das dazugehörige graphische Werkzeug *(U)CML-ed* vorgestellt. Abgeschlossen wird diese Arbeit mit der praktischen Anwendung der (U)CML anhand der Fallstudie Gleitschutzsystem.

Abstract

This work is concerned with both the formal compatibility-oriented modeling of complex technical systems and the integration of an object-oriented process for modeling compatibility into systems engineering practice.

To accomplish both of these goals, this work will begin with the introduction of fundamental terms and concepts from the systems engineering surrounding, which are essential for modeling compatibility. This will be followed by a description of the principles of object-oriented modeling and examining systems for compatibility. Significant requirements for any multidisciplinary modeling language for determining the compatibility of technical systems are provided after the foundations of object-oriented modeling of compatibility have been described. Based on these requirements, four different modeling languages and methods for technical systems are described and their applicability for modeling and determining compatibility discussed. Following this comparison, the newly developed modeling language (U)CML for modeling compatibility as well as its associated graphical tool *(U)CML-ed* will be introduced. This work will conclude with the practical application of (U)CML in a case study on an anti-slip regulation system for trains.

Inhaltsverzeichnis

Einleitung und Motivation

Seit Beginn der industriellen Revolution Ende des 19./Anfang des 20. Jahrhunderts bis heute hat sich der technologische Fortschritt auf fast allen Gebieten der Technik rasant entwickelt. Noch vor wenigen Jahrzehnten war es aus technologischer Sicht undenkbar, dass mobile Telefone so klein, hochintegriert und vor allem billig sein werden, dass statistisch gesehen fast 78% der Bundesbürger im Jahre 2007 ein Mobiltelefon besitzen[1]. So hat der technologische Fortschritt heutzutage in fast allen Lebensbereichen Einzug gehalten, ohne dass wir dies heute noch bewusst wahrnehmen, da wir uns an diese unterschiedlichen Systeme gewöhnt haben. Seien es elektronische Terminkalender (Organizer), MP3-Player oder ein Kühlschrank mit Internetzugang. Der stetig schneller voranschreitende Fortschritt auf dem Gebiet der Mikroelektronik und insbesondere bei der Integration von Hard- und Software, zu so genannten *eingebetteten Systemen*, bestehend aus mechanischen-, elektrischen/elektronischen- und Softwarekomponenten, sowie die damit einhergehende Kostenreduktion der Produkte, hat diese Entwicklung der letzten Jahre erst möglich gemacht.

Situationsanalyse
Der galoppierende technologische Fortschritt, vor allem auf dem Gebiet der eingebetteten Systeme, hat jedoch auch seine Schattenseiten. Zum einen wird die Produktlebenszeit von einer Produktgeneration zur nächsten ständig kürzer da der Markt immer schneller nach neuen Produkten mit erweiterten Eigenschaften und Funktionalität verlangt. Zum anderen sinkt aufgrund der kürzer werdenden Entwicklungszeiten die Qualität der Produkte zum Teil dramatisch, wodurch häufig Nachbesserungen notwendig werden. Die kürzer werdenden Produktlebenszeiten haben neben der sinkenden Produktqualität noch eine weit aus schlimmere Folge – die langfristige Versorgung mit passenden Ersatzteilen. Bereits nach relativ kurzer Zeit gibt es für die meisten am Markt befindlichen Produkte keine passenden Ersatzteile mehr, aufgrund der Tatsache, dass das Produkt am Markt nicht mehr gefragt wird, weil es nicht über die erweiterten Eigenschaften sowie die Funktionalität der neuen Produktgeneration verfügt. Mag diese Entwicklung für Unterhaltungs- und Konsumgüter wie beispielsweise mobile Telefone oder Computer akzeptabel sein, so ist sie für langlebige Industriegüter, wie z.B. für Schienenfahrzeuge, Automobile oder Flugzeuge undenkbar. Hier herrschen zum Teil Produktlebenszeiten von zum Teil deutlich mehr als 30 Jahren, in denen Ersatzteile vom Hersteller aufgrund von Verträgen und Vereinbarungen bereitgestellt werden müssen.

Um während der extrem langen Produktlebenszeiten das Produkt mit passenden Ersatzteilen versorgen zu können, bieten sich zwei grundsätzlich unterschiedliche Herangehensweisen an. Zum einen kann ein Hersteller für jedes seiner Produkte mehrere Ersatzteil einlagern, bis diese schließlich benötigt werden. Zum anderen kann bereits bei der Entwicklung neuer Produkte bzw. von Baugruppen der Produktlinie darauf geachtet werden, dass diese *kompatibel* zur Vorgängerversion des Produkts sind. Auch Mischformen aus den beiden geschilderten Extrema sind möglich, also die möglichst kompatible Weiterentwicklung von bereits vorhandenen Baugruppen eines Systems sowie die gleichzeitige Einlagerung von Baugruppen, die nicht weiter entwickelt werden sollen.

In der Praxis hat es sich in den letzten Jahren gezeigt, dass die Lagerhaltung von alten Baugruppen eines Systems nahezu unmöglich ist, aufgrund der Tatsache, dass niemand über einen Zeitraum von mehr als 30 Jahren vorhersagen kann, welche und vor allem wie viele Baugruppen eines bestimmten Typs eingelagert werden müssen, um die garantierte Versorgung mit Ersatzteilen zu gewährleisten. Darüber hinaus wird durch die Lagerhaltung sehr viel Kapital im Unternehmen gebunden, das dann nicht für die Verbesserung und Weiterentwicklung der Produktpalette eingesetzt werden kann. Aus diesen Gründen tendieren heutzutage fast alle Hersteller von Industriegütern mit sehr langen Produktlebenszeiten dazu, ihren Lagerbestand zu minimieren und statt dessen ihre Produkte möglichst kompatibel weiter zu entwickeln. Jedoch gerade die kompatible Weiterentwicklung von Baugruppen hat sich in den letzten Jahren als sehr große Herausforderung für die Entwicklerteams erwiesen. Dies liegt zum einen an der stärker werdenden Komplexität der Produkte und zum anderen an der ständig anwachsenden Interdisziplinarität in der Produktentwicklung. Moderne Produkte bestehen heutzutage fast immer aus mechanischen, elektrischen/elektronischen, sowie zunehmend auch Softwarekomponenten. Dadurch wird die kompatibilitätsgerechte Neu- bzw. Weiterentwicklung einer Baugruppe zum Teil erheblich erschwert, da Entwickler aus unterschiedlichen Domänen gemeinsam an einem Produkt arbeiten, jedoch selten die „Sprache der anderen“ sprechen. Dieser Umstand führt häufig zu erheblichen Dissonanzen zwischen den, an der Produktentwicklung beteiligen, Entwicklerteams und resultiert schließlich meistens in „hausgemachten“ Kompatibilitätsproblemen.

Eine weitere Schwierigkeit, die sowohl bei der Neu- als auch bei der Weiterentwicklung von Systemen oder Systembestandteilen häufig zu Tage tritt, ist sehr eng mit dem oben beschriebenen „Verständigungsproblem“ zwischen den unterschiedlichen, an der Systementwicklung beteiligten Ingenieur-Teams, verbunden. So fehlt beispielsweise bei den meisten Neu- oder Weiterentwicklungen einzelner Baugruppen oder des gesamten Systems ein gemeinsames, zentrales Modell des Systems, in dem sämtliche (kompatibilitätsrelevanten) Eigenschaften so hinterlegt sind, dass Ingenieure aus allen an der Entwicklung beteiligten Domänen dieses Modell lesen, verstehen und gegebenenfalls anpassen können. Die Einführung eines zentralen domänenübergreifenden Modells des zu entwickelnden Systems reicht jedoch bei weitem nicht aus da es zur Zeit keine Modellierungssprache am Markt gibt, die von Ingenieuren aus allen Domänen gleichermaßen zur Modellierung eingesetzt werden kann und vor allem die die speziellen Anforderungen an die Kompatibilitätsmodellierung und -bewertung beinhaltet.

[1] Siehe hierzu insbesondere [Sta08].

Lösungsansatz
Um den unterschiedlichen Problemen der geschilderten Szenarien dennoch wirkungsvoll entgegenzutreten, wurde im Rahmen der *Softwareinitiative 2006* der Bundesregierung und des BMBF[2] das geförderte Forschungsvorhaben MOKOMA[3] ins Leben gerufen. Ziel dieses Vorhabens war es, ein domänenunabhängiges Modellierungskonzept für die Beschreibung und Bewertung der Kompatibilität von eingebetteten softwarelastigen Systemen zu erarbeiten, um sämtliche an der Entwicklung des Systems beteiligten interdisziplinären Ingenieur-Teams bei der kompatibilitätskonformen Weiterentwicklung der unterschiedlichen Baugruppen des Systems unter die Arme zu greifen. Des Weiteren sollte die erarbeitete Kompatibilitätsbeschreibungsmethode nicht nur während des Entwicklungsprozesses, sondern gleichermaßen für den Austausch einer Baugruppe des Systems eingesetzt werden können, um das oben erwähnte Lagerhaltungsproblem in den Griff zu bekommen. Um eine möglichst ausgeglichene Projektstruktur innerhalb des Forschungsvorhabens MOKOMA zu erreichen, setzte sich das Forschungsvorhaben aus drei Industrieunternehmen (Knorr-Bremse AG, Validas AG und 3D Systems Engineering GmbH), dem Lehrstuhl für Software und Systems Engineering sowie dem Lehrstuhl für Raumfahrttechnik – Systems Engineering Gruppe – der Technischen Universität München (TUM) zusammen. Die Firma Knorr-Bremse AG fungierte als Anwender der auch das Pilotsystem vorgegeben hat. Die Firma Validas AG war für die Werkzeugunterstützung (AutoFOCUS[4]) zuständig, während die Firma 3D Systems Engineering GmbH die Projektpartner koordinierte. Die beiden Lehrstühle der TUM waren für die Erarbeitung der theoretischen Grundlagen zuständig.

Zielsetzung dieser Arbeit
Der Schwerpunkt dieser Arbeit ist die Einführung einer ganzheitlichen, interdisziplinären und generischen Methode zur objektorientierten Modellierung von komplexen technischen Systemen, der Überprüfung und Bewertung der Modelle auf Kompatibilität sowie der Integration der Kompatibilitätsmodellierung und -bestimmung in das Systems Engineering Umfeld.

Inhalt der Arbeit in Stichpunkten:

- Theoretische Grundlagen der objektorientierten Modellbildung und insbesondere der Kompatibilitätsmodellierung und -bewertung der Modelle
- Integration der objektorientierten Kompatibilitätsmodellierung und -bestimmung in das Systems Engineering Umfeld
- Vorstellung einer generischen Methode zur Kompatibilitätsmodellierung und -bestimmung
- Einführung einer allgemein gültigen Methode für die Identifikation von (kompatibilitätsrelevanten) Eigenschaften und Funktionen eines technischen Systems
- Vergleich und Bewertung existierender Modellierungssprachen und Konzepte in Bezug auf die Nutzung für die Modellierung und Bewertung von Kompatibilität
- Entwicklung einer domänenübergreifenden generischen Modellierungssprache für die Modellierung und Bewertung von Kompatibilität in technischen Systemen – (U)CML[5]
- Prototypische Anwendung der Kompatibilitätsmodellierungssprache (U)CML

Um die unterschiedlichen Ziele dieser Arbeit „unter einen Hut“ zu bekommen, wurde folgende Struktur für die Arbeit gewählt.

Aufbau und Struktur der Arbeit
Diese Arbeit unterteilt sich thematisch in drei große aufeinander aufbauende Bereiche und umfasst darüber hinaus die Einleitung, den Anhang, ein Glossar der wichtigsten Begriffe und Definitionen sowie einem Stichwortverzeichnis. Im ersten Abschnitt der Arbeit werden zunächst sämtliche Grundlagen für die modellbasierte Kompatibilitätsmodellierung und -bestimmung gelegt, sowie deren Integration in das Systems Engineering Umfeld diskutiert. Darauf folgt die praktische Umsetzung der theoretischen Grundlagen anhand des konkreten Beispielsystems Gleitschutz. Abgeschlossen wird die Arbeit mit der Zusammenfassung der Ergebnisse sowie einem Ausblick auf weitere verwandte Themenbereiche, die in dieser Arbeit nur gestreift wurden.

Die folgende Aufzählung bzw. die Abbildung 0.1 auf Seite xv beschreibt den Aufbau, die Struktur sowie den inhaltlichen Schwerpunkt der einzelnen Kapitel der Arbeit.

Abschnitt 1 – Theoretische Grundlagen und Methoden der ganzheitlichen Systemmodellierung

Im ersten Kapitel dieses Abschnitts werden zunächst die grundlegenden Begriffe und Definitionen rund um den Themenschwerpunkt Kompatibilität eingeführt und erläutert, die für das Verständnis der weiteren Arbeit notwendig sind. Daran anschließend folgt im zweiten Kapitel zunächst die Einführung der Grundlagen der objektorientierten Kompatibilitätsmodellierung, bevor im dritten Kapitel unterschiedliche Modellierungssprachen und Techniken für technische Systeme eingeführt und anhand des durchgängigen Beispiels „KOMPTEST“ erläutert werden. Abgeschlossen wird das dritte Kapitel mit der Bewertung der zuvor vorgestellten Modellierungssprachen im Hinblick auf die Modellierung und Bewertung von Kompatibilität.

- Themenschwerpunkte des Kapitels „*Grundlegende Begriffe und Definitionen*“
  - Definition des Kompatibilitätsbegriffs
  - Vorstellung verschiedener Kompatibilitätsarten

[2] Das Akronym *BMBF* steht für Bundesministerium für Bildung und Forschung.
[3] Das Akronym MOKOMA steht für Modellbasiertes Engineering und Kompatibilitätsmanagement von komplexen eingebetteten Softwarelösungen und Elektroniknetzwerken in mobilen Anlagen (Eisenbahn, Automobil, Truck, Flugzeugen, Landmaschinen und Schiffen).
[4] *AutoFOCUS* ist ein graphisches Werkzeug zur Spezifikation von verteilten Systemen. Nähere Informationen zu AutoFOCUS findet sich unter [HEB08].
[5] Das Akronym *(U)CML* steht für (U)nified Compatibility Modelling Language. Die (U)CML wird im Kapitel „3.6 *Einführung in die Kompatibilitätsmodellierungssprache (U)CML*“ ab Seite 153 eingeführt.

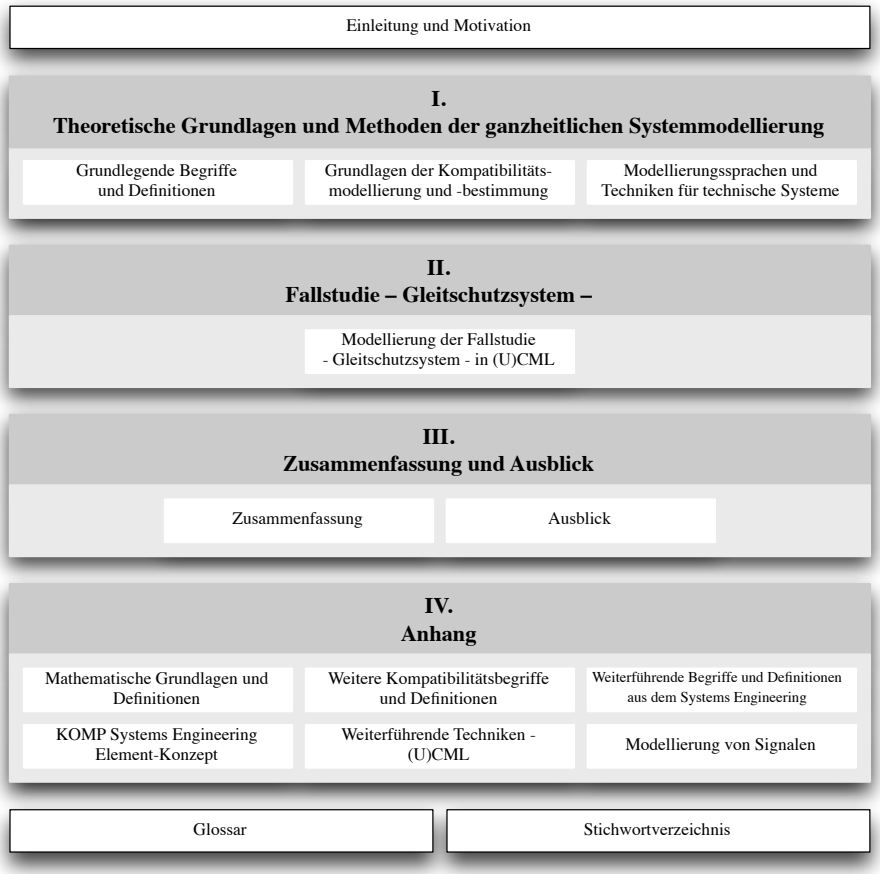


Abbildung 0.1.: Aufbau und Struktur der Arbeit.

  - Vorstellung unterschiedlicher Kompatibilitätsszenarien
  - Einführung eines Kompatibilitätsregelwerks
  - Integration des Kompatibilitätsmanagements in das Systems Engineering Umfeld
- Themenschwerpunkte des Kapitels *„Grundlagen der Kompatibilitätsmodellierung und -bestimmung“*
  - Einführung in die Grundlagen der klassischen sowie der objektorientierten Modellbildung
  - Erweiterung der objektorientierten Modellbildung für die Kompatibilitätsmodellierung und -bestimmung
  - Identifikation von kompatibilitätsrelevanten Eigenschaften eines Systems
  - Identifikation von kompatibilitätsrelevanten verhaltensbeschreibenden Methoden eines Systems
  - Vorstellung des objektorientierten Kompatibilitätsmodellierungsprozesses
  - Einführung eines objektorientierten Kompatibilitätsregelwerks
  - Objektorientierte Kompatibilitätsprüfung
  - Integriertes domänenübergreifendes Systemmodell
- Themenschwerpunkte des Kapitels *„Modellierungssprachen und -techniken für technische Systeme“*
  - Anforderungen an eine domänenübergreifende Kompatibilitätsmodellierungssprache
  - Technische Spezifikation des Beispielsystems „KOMPTEST“
  - Vorstellung existierender Modellierungssprachen und Paradigmen für technische Systeme
  - Einführung in die Kompatibilitätsmodellierungssprache (U)CML
  - Vergleich und Bewertung der unterschiedlichen Modellierungssprachen auf Kompatibilität
  - Vorstellung des graphischen Editors *(U)CML-ed*

Abschnitt 2 – Fallstudie – Gleitschutzsystem

Nach dem im ersten Abschnitt die theoretischen Grundlagen für die kompatibilitätskonforme Systemmodellierung und Bewertung gelegt worden sind, folgt nun deren prototypische Umsetzung anhand des Beispielsystems Gleitschutz. Zunächst wird das System Gleitschutz mit Hilfe des vorgestellten Kompatibilitätsmodellierungsprozesses in der Kompatibilitätsmodellierungssprache (U)CML modelliert und anschließend auf Kompatibilität untersucht und das Ergebnis bewertet. Im zweiten Teil der Fallstudie wird eine Baugruppe aus dem Gleitschutzsystem entfernt und gegen eine neuere Variante ersetzt. Daran anschließen wird das neu entstandene Gleitschutzsystem auf Kompatibilität zum ursprünglichen System untersucht und bewertet.

- Themenschwerpunkte des Kapitels „*Modellierung der Fallstudie – Gleitschutzsystem – in (U)CML*“
  - Modellierung des Gleitschutzsystems in (U)CML
  - Austausch einer Baugruppe des Gleitschutzsystems
  - Bewertung der Ergebnisse der Fallstudie

Abschnitt 3 – Zusammenfassung und Ausblick

In diesem Abschnitt werden die Ergebnisse der gesamten Arbeit zusammengefasst, sowie ein Ausblick auf mögliche Erweiterungen und Ergänzungen dieser Arbeit vorgestellt.

- 5 *Zusammenfassung der Arbeit* ab Seite 311
- 6 *Ausblick* ab Seite 313

Abschnitt 4 – Anhang

Im Abschnitt Anhang werden weitere grundlegende Definitionen, Erweiterungen und weiterführende Techniken vorgestellt, die für den Zusammenhang der Arbeit notwendig sind, jedoch im Hauptteil nicht enthalten sind, bzw. dort nur gestreift aber nicht ausgeführt worden sind.

- A *Mathematische Grundlagen und Definitionen* ab Seite 317
- B *Weitere Kompatibilitätsbegriffe und Definitionen* ab Seite 325
- C *Weiterführende Begriffe und Definitionen aus dem Systems Engineering* ab Seite 327
- D KOMP *Systems Engineering Element-Konzept* ab Seite 331
- E *Weiterführende Techniken – (U)CML* ab Seite 337
- F *Modellierung von Signalen* ab Seite 341

Abgerundet wird die Arbeit mit einem Glossar der wichtigsten Begriffe und Definitionen aus dem Umfeld der objektorientierten Kompatibilitätsmodellbildung, sowie dem Stichwortverzeichnis.

Teil I.

Theoretische Grundlagen und Methoden der ganzheitlichen Systemmodellierung

In diesem Abschnitt werden zunächst grundlegende Methoden der ganzheitlichen Systemmodellierung vorgestellt und anhand von einfachen Beispielen erläutert. Daran anschließend folgt die Einführung in die Grundlagen der Kompatibilitätsmodellierung. Abschlossen wird der erste Abschnitt mit der Einführung verschiedener Modellierungstechniken für technische Systeme.

Kapitel:

Kapitel 1.

Grundlegende Begriffe und Definitionen

> Man kann ein Problem nicht mit der Denkweise lösen, die es geschaffen hat.
>
> *(Albert Einstein)*

Ziel dieses einführenden Kapitels ist es, die grundlegenden Begriffe und Definitionen aus den beiden Bereichen *Systems Engineering* sowie der *Modellierung* und *Bewertung* von eingebetteten softwarelastigen Systemen[6], zu definieren und diese anhand von einfachen Beispielen zu erläutern. Vor allem wird in diesem Kapitel ein grundlegender Überblick über die Kompatibilitätsproblematik, wie sie bei der Modellierung, der Beschreibung sowie der Bewertung von technischen Systemen, bestehend aus Hard- und Software, auftreten kann ausführlich erläutert und anhand von einfachen begleitenden Beispielen beschrieben. Sämtliche in diesem Kapitel eingeführten Begriffe und Definitionen dienen als Grundlage für alle nachfolgenden Kapitel dieser Arbeit.

1.1. Aufbau und Struktur des Kapitels

Das Kapitel „*Grundlegende Begriffe und Definitionen*“ ist logisch in drei aufeinander aufbauende Teilbereiche unterteilt. Im ersten Teil wird die provokante Frage „Was hat Systems Engineering mit modellbasierter Kompatibilitätsbewertung zu tun?“ geklärt. Im darauf folgenden Abschnitt werden einige wesentliche Begriffe aus dem Systems Engineering Umfeld erläutert, die für die ganzheitliche Systembetrachtung notwendig sind. Daran anschließend wird die zentrale Bedeutung der Einbettung der modellbasierten Kompatibilitätsbestimmung in das Systems Engineering Umfeld, und somit die obige Frage geklärt.

Nachdem die grundlegenden Begriffe und Definitionen aus dem Systems Engineering Umfeld eingeführt worden sind, folgt nun die Klärung des für diese Arbeit zentralen Begriffs *Kompatibilität*. In diesem Abschnitt werden unterschiedliche Arten von Kompatibilität, mit dem Fokus der Anwendbarkeit auf technische Systeme, eingeführt und anhand von begleitenden Beispielen erläutert. Abgeschlossen wird das Kapitel mit der Einführung verschiedener, auf den oben eingeführten Kompatibilitätsbegriffen aufbauenden Kompatibilitätsszenarien.

1.2. Was hat Systems Engineering mit modellbasierter Kompatibilitätsbewertung zu tun?

Zu Beginn der industriellen Revolution am Ende des 19./Anfang des 20. Jahrhunderts, konnte ein Erfinder/Ingenieur ein technisches System alleine ohne fremde Hilfe entwickeln und in kleinen Serien bauen. Bereits zu Beginn des 20. Jahrhunderts wurden aufgrund des rasanten wissenschaftlich und technischen Fortschritts die Systeme zunehmend komplexer, so dass es für einen einzelnen Ingenieur fast unmöglich geworden war, ein komplexes technische System alleine zu entwickeln. Mitte des 20. Jahrhunderts und insbesondere zu Beginn der zweiten technischen Revolution (dem Elektronik- und Computerzeitalter) wurden technische Systeme so komplex, dass nicht nur Ingenieurteams einer Fachrichtung, sondern Ingenieure unterschiedlicher Fachrichtungen (Domänen) nur noch gemeinsam in der Lage waren technische Systeme zu entwickeln. Um den neuen bzw. geänderten Anforderungen bei der Systementwicklung gerecht zu werden und vor allem der ständig wachsenden Komplexität und Vernetzung der Systeme Rechnung zu tragen, wurde Mitte des 20. Jahrhunderts eine neue Wissenschaft, das *Systems Engineering*[7] geboren. Ziel des Systems Engineerings war und ist es, Ingenieure bei der Entwicklung von komplexen Systemen zu unterstützen.

Was ist Systems Engineering?
Das Systems Engineering befasst sich sowohl mit der Modellierung als auch der Beschreibung von komplexen Systemen jeglicher Art, als auch der Integration und Zusammenarbeit unterschiedlicher Fachdisziplinen bei der Systementwicklung. Im Besonderen beschäftigt sich das Systems Engineering mit der modellbasierten Beschreibung von technischen Systemen. Es bildet somit die Grundlage vieler unterschiedlicher Fachdisziplinen, wie beispielsweise der Elektrotechnik, der Mechanik, den Wirtschaftswissenschaften oder der Softwaretechnik, und ermöglicht so eine effiziente domänenübergreifende Betrachtung des gesamten Systems. M. Griffin, Administrator der NASA, definiert das Systems Engineering folgendermaßen: *„System engineering is the art and science of developing an operable system capable of meeting requirements within imposed constraints.“* (Michael Griffin, NASA Administrator [Gri07]).

Nach der Definition von M. Griffin ist das Systems Engineering eine umfassende Wissenschaft, die sich mit der ganzheitlichen Betrachtung von Systemen beschäftigt. Dieser Ansatz ist vor allem für die in dieser Arbeit vorgestellten Methode der interdisziplinären Kompatibilitätsmodellierung, Beschreibung und Bewertung von technischen Systemen von entscheidender Bedeutung, da hier im Allgemeinen sehr viele Ingenieure unterschiedlicher Fachrichtungen zusammen an einem System arbeiten. Eine weitere Definition des Begriffs Systems Engineering bzw. der Fachdisziplin Systems Engineering wurde von der internationalen Dachorganisation

[6] *Eingebettete Systeme* bestehen im Allgemeinen aus elektrischen/elektronischen, mechanischen sowie Software Komponenten. Bei *eingebetteten softwarelastigen Systemen* nimmt der Softwareanteil innerhalb des Systems eine dominante Rolle gegenüber den „restlichen“ Komponenten des Systems ein.

[7] Im weiteren Text wird der Begriff *Systems Engineering* oft mit dem Akronym *SE* abgekürzt.

INCOSE[8] [SE07] geprägt, in der sich viele unterschiedliche Fachdisziplinen vereinigt haben, um gemeinsame Standards für die Systementwicklung festzulegen und zu beschreiben.

Definition 1.1 ***Systems Engineering nach* INCOSE[9] *[OOS07]***
Systems Engineering ist ein interdisziplinärer Ansatz mit dem Ziel erfolgreich Systeme zu realisieren. Systems Engineering konzentriert sich auf die Definition und Dokumentation der Systemanforderungen in der frühen Entwicklungsphase, die Erarbeitung eines Systemdesigns und die Überprüfung des Systems auf Einhaltung der gestellten Anforderungen unter Berücksichtigung des Gesamtproblems:

- *Betrieb*
- *Zeit*
- *Test*
- *Erstellung*
- *Kosten und Planung*
- *Training und Support*
- *Entsorgung*

Das Systems Engineering integriert alle Disziplinen und bildet einen strukturierten Entwicklungsprozess vom Konzept über die Produktion- bis zur Betriebsphase. Es werden sowohl die technischen als auch die wirtschaftlichen Aspekte betrachtet, um ein System zu entwickeln, dass den Benutzerbedürfnissen entspricht.

Auch die Systems Engineering Definition 1.1[10] nach INCOSE verdeutlicht die unterschiedlichen Ziele, die bei der Entwicklung von komplexen Systemen verfolgt und berücksichtigt werden müssen. Abbildung 1.1 hingegen zeigt den grundsätzlichen Ablauf/Prozess bei der System- bzw. Produktentwicklung. Sie beginnt bei der Identifikation des Problems/Idee, geht über das eigentliche Systems Engineering bis hin zum fertigen System/Produkt. Dabei durchläuft jedes zu entwickelnde System bzw. Produkt den gesamten Entwicklungszyklus.

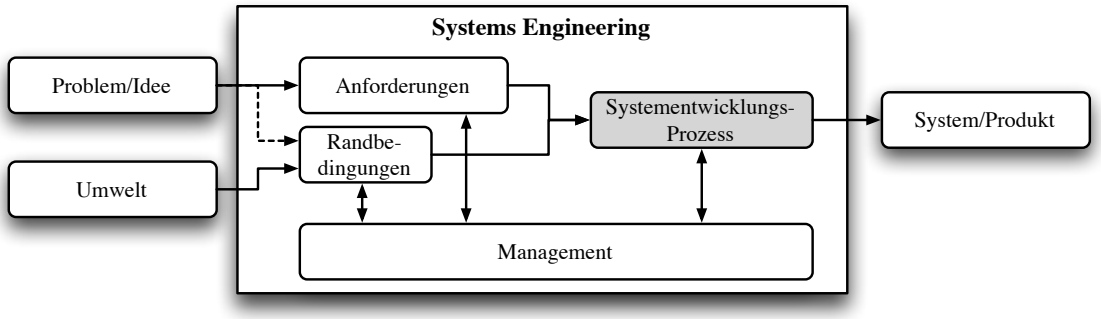


Abbildung 1.1.: Grundsätzlicher Ablauf des Systems Engineerings nach [Wal08].

Die Systementwicklung beginnt stets mit einer Idee oder einem konkreten Problem. Um die Idee in ein konkretes System/Produkt zu transferieren, bzw. ein gegebenes Problem zu lösen, wird der Systems Engineering Prozess angestoßen (vgl. Abbildung: 1.1). Dabei werden zunächst, zusammen mit dem Kunden, die Anforderungen und Randbedingungen an das zu entwickelnde System erfasst und deren Notwendigkeit bzw. die Umsetzungsmöglichkeit sowie die daraus resultierende Entwicklungszeit und Kosten erörtert. Aus den Anforderungen und Randbedingungen wird im nächsten Schritt ein (abstraktes) Modell des zu entwickelnden Systems erstellt. Aus dem Modell wird dann das konkrete System/Produkt entwickelt und gebaut. Dabei muss während des gesamten Entwicklungszyklus stets darauf geachtet werden, dass sowohl die Anforderungen, als auch die Randbedingungen an das zu entwickelnde System/Produkt eingehalten werden. Sämtliche Steuerungs- und Kontrollaufgaben werden dabei, während des gesamten Systems Engineering Prozesses, vom Management wahrgenommen und durchgeführt. Dazu kommuniziert das Management mit allen an der Entwicklung beteiligten Bereichen. Die Abbildung 1.2 auf der nächsten Seite illustriert den generischen Systems Engineering Prozess, so wie er bei jeder Systementwicklung angewendet wird. Dabei beschreibt die Abbildung die logische Abfolge bei der Systementwicklung, ohne jedoch auf die Fertigung des Produkts bzw. das Management einzugehen.

Die Abbildung 1.2 auf der nächsten Seite stellt ebenfalls den zeitlichen Ablauf des generischen Systems Engineering Prozesses graphisch dar. Begonnen wird auch hier mit dem zugrunde liegenden Problem bzw. der Idee (1). Durch den semantischen Prozess werden Informationen und Daten vom Kundenlevel in den Ingenieurslevel übertragen und dabei analysiert $(1 \rightarrow 2)$[11]. Dabei gehen im Allgemeinen Informationen und Daten über das zu entwickelnde System/Produkt durch die Übersetzung vom Sprachschatz des Kunden in den des Ingenieurs verloren, die durch den anschließenden Systems Engineering Prozess $(2 \rightarrow 3)$ wieder hinzugefügt werden müssen, um das vom Kunden gewünschte System/Produkt erfolgreich zu entwickeln. Um den Informationsverlust

[8] Das Akronym INCOSE steht für **In**ternational **C**ouncil **o**n **S**ystems **E**ngineering.
[9] Die ursprüngliche englischsprachige INCOSE Definition wurde von OOSE [OOS07] ins Deutsche übersetzt.
[10] Anmerkung:
Im Anhang ab Seite 327 sind einige weitere Definitionen des Begriffs „Systems Engineering“ aufgelistet, die verdeutlichen, dass das Systems Engineering eine sehr weit gefächerte Fachdisziplin ist, die in allen anderen Disziplinen ebenfalls stark verankert sein muss, um möglichst effizient und kostengünstig ein voll funktionsfähiges System entwickeln zu können.
[11] Dieser Prozess wird oft auch als *Anforderungsanalyse* bezeichnet.

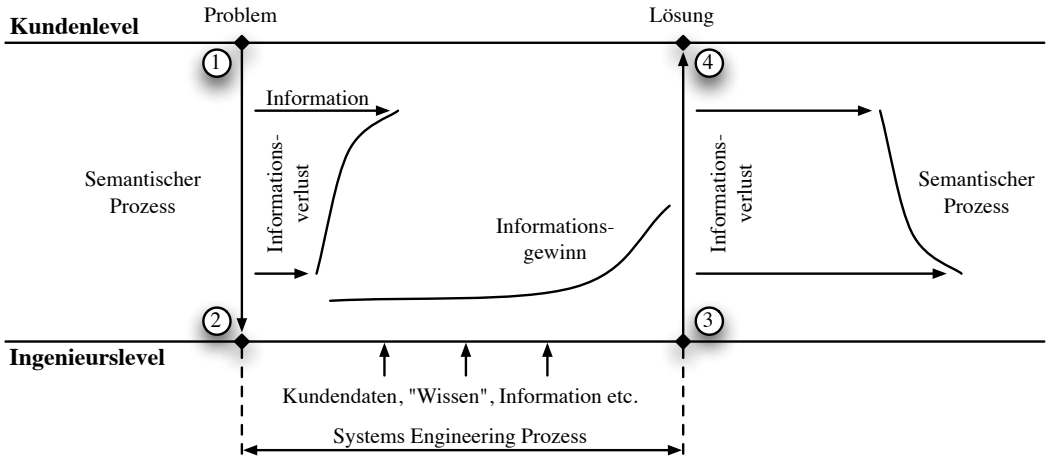


Abbildung 1.2.: Ganzheitliche Systementwicklungsprozess nach Prof. Dr.-Ing. E. Igenbergs.

zu kompensieren, werden während des gesamten Systems Engineering Prozess unterschiedliche Entwicklungsstrategien und Methoden, wie beispielsweise Vorgehensmodelle oder Entwicklungsparadigmen wie top-down Entwurf oder Work-Breakdown-Structure (WBS) verwendet, um dem Informationsverlust entgegenzuwirken. Insbesondere werden während des gesamten Systems Engineering Prozesses verschiedene (abstrakte) Modelle des zu entwickelnden Systems/Produkts eingesetzt. Nachdem das Systemmodell während des Systems Engineering Prozesses entwickelt und dabei stets iterativ verfeinert worden ist, wird es zurück vom Ingenieurslevel auf den Kundenlevel übertragen ($3 \rightarrow 4$). Nachdem der Systems Engineering Prozess abgeschlossen worden ist, sollte das ursprüngliche Problem/Idee (1) gelöst worden sein, so dass nun das konkrete System/Produkt gebaut werden kann (4).

Aus den unterschiedlichen Zielen (vgl. Definition: 1.1 auf Seite 4), sowie dem grundsätzlichen Systems Engineering Prozess (vgl. Abbildung: 1.1 sowie 1.2), lassen sind einige grundlegende Paradigmen ableiten, die bei jeder Systementwicklung angewendet werden sollten. In der nachfolgenden Aufzählung sind einige der grundlegenden Paradigmen aufgelistet, die für eine erfolgreiche Systementwicklung notwendig sind (Auszug bzw. Erweiterung aus [MS07, 31]).

- **Integrierte Systementwicklung**
  Um ein beliebiges System erfolgreich entwickeln zu können, ist es notwendig, dass neben der reinen System- bzw. Produktentwicklung Ziele wie Termintreue, Kostenkontrolle, Personalplanung etc. eingehalten werden. Um dieses Systementwicklungsparadigma zu erreichen, muss während des gesamten Systementwicklungszyklus dafür Sorge getragen werden, dass sämtliche Entwicklungsziele des Systems „gleichzeitig“ verfolgt und eingehalten werden. Voraussetzung für eine erfolgreiche integrierte Systementwicklung ist die Anwendung der nachfolgenden Entwicklungsparadigmen.

- **Prozess- und Systemdenken**
  Mit Hilfe des durch das Systems Engineering propagierten Systemdenkens soll das zu entwickelnde System als Ganzes – als System – wahrgenommen werden. So ist es zum Beispiel bei der Entwicklung eines Satelliten notwendig, nicht nur den Satelliten als isoliertes System, sondern auch die zur Verfügung stehenden Trägersysteme bei der Systementwicklung mit einzubeziehen. So darf beispielsweise ein Kommunikationssatellit auf keinen Fall schwerer sein als dass er mit einem zur Verfügung stehenden Trägersystem gestartet werden kann. Ansonsten ist zusätzlich die Entwicklung eines geeigneten Trägersystems notwendig. Zusätzlich zum Systemdenken ist für eine erfolgreiche termin- und kostengerechte Systementwicklung ein penibel geplantes Vorgehen notwendig. Dies ist umso wichtiger, je komplexer das zu entwickelnde System wird, bzw. je mehr unterschiedliche Personen und Fachabteilungen an der Entwicklung beteiligt sind.

  Für die „Steuerung und Lenkung“ der Systementwicklung werden unterschiedliche Vorgehensmodelle, auch Makro-Prozesse genannt, verwendet, die zum Teil speziell an die Bedürfnisse einer Domäne angepasst worden sind. In der Raumfahrt bzw. bei militärischen Projekten wird beispielsweise das ECSS[12] Phasenmodell verwendet. Beim Phasenmodell wird die komplette Systementwicklung in einzelne Phasen unterteilt. In jeder Phase wird eine spezielle Tätigkeit durchgeführt und die Ergebnisse an die darauf folgende Phase weitergegeben. Nach jeder Phase wird das Ergebnis der Phase begutachtet und entschieden ob das Projekt eingestellt oder weiterverfolgt wird.

  Die Abbildung 1.3 auf der nächsten Seite zeigt das Phasenmodell nach [sta96], so wie es von der europäischen Raumfahrtbehörde (ESA) für die Entwicklung, zum Beispiel eines Satelliten, vorgeschrieben wird. Dabei wird das gesamte Projekt in sieben aufeinander folgende Phasen unterteilt (Phase 0 bis F[13]). Nach jeder Phase wird das Ergebnis durch ein unabhängiges

[12] Das Akronym *ECSS* steht für **E**uropean **C**ooperation for **S**pace **S**tandardization
[13] Phasenbezeichnungen nach dem ECSS Standard [RD08][ECS08]:

- Phase 0 – Missionsanalyse/Identifizierung der Anforderungen
- Phase A – Durchführbarkeit
- Phase B – Vordefinition (Projekt/Produkt)
- Phase C – Detaildefinition (Produkt)
- Phase D – Produktion/Bodenqualifikationstest
- Phase E – Betrieb
- Phase F – Entsorgung

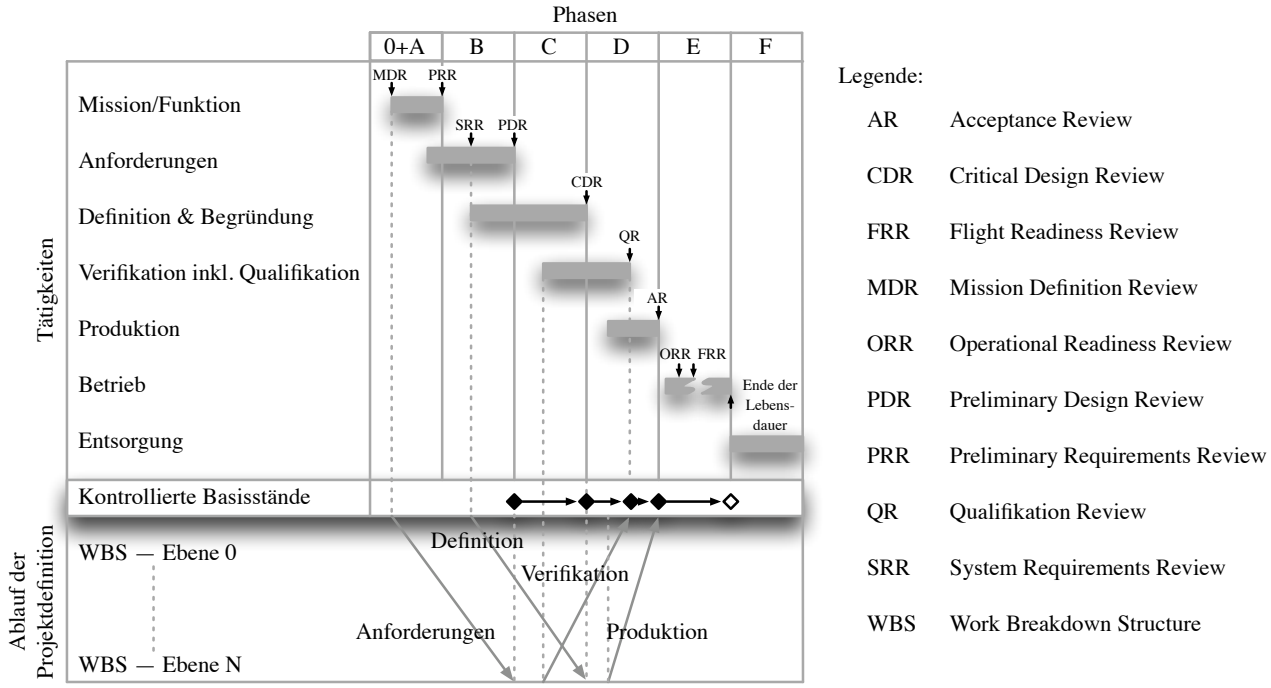


Abbildung 1.3.: Aufbau und Struktur des Phasenmodells nach [sta96, 11].

Team von Experten begutachtet (Review) und entschieden, ob weitergearbeitet, nachgebessert oder das Projekt beendet wird, da die Projektziele (die Anforderungen des Kunden) nicht mehr erreicht werden können.

- **Abstraktion und Modellbildung**
  Die meisten (technischen) Systeme sind zu komplex, um sie vollständig im Detail modellieren und beschreiben zu können. Aus diesem Grund ist eine Abstraktion des zu entwickelnden Systems notwendig, in dem lediglich die Daten und Informationen enthalten sind, die für die Systementwicklung benötigt werden. Dies sind neben den Systemdaten (z.B. CAD Zeichnungen, Schaltplänen, Quellcodes) auch die im Punkt integrierte Systementwicklung angesprochenen Zusatzdaten (z.B. Termin- und Kostenpläne), ohne die das System/Produkt nur erschwert entwickelt werden kann. Die abstrahierten Daten werden im darauf folgenden Schritt, der Modellbildung (Phase 0 bis C), in ein Modell des Systems transferiert. Dieses Modell enthält somit sämtliche System- und Zusatzdaten, die für die Systementwicklung benötigt werden.

  Die Abstraktion und Modellbildung repräsentiert den Kern dieser Arbeit. Aus diesem Grund werden im Kapitel „*Modellbildung*“ die Grundlagen der Modellbildung vorgestellt und anhand einfacher Beispiele erläutert. Nachdem das allgemeine Vorgehen bei der Modellbildung erläutert worden ist, wird im Kapitel „*Klassen und Objekte eines Systems*“ auf die objektorientierte Modellbildung eingegangen, so wie sie im weiteren Verlauf dieser Arbeit verwendet wird. Dabei wird insbesondere auf die Modellierung von kompatibilitätsrelevanten Systemeigenschaften eingebetteter technischer Systeme eingegangen. Schließlich werden im Kapitel „*Modellierungssprachen und -techniken für technische Systeme*“ unterschiedliche Modellierungssprachen und -techniken vorgestellt und erläutert, mit deren Hilfe die Kompatibilitätsmodellierung und -bewertung durchgeführt werden kann.

- **Interdisziplinarität**
  Moderne Systeme bestehen heutzutage aus vielen unterschiedlichen technischen Komponenten. Diese werden von spezialisierten Fachabteilungen erstellt und anschließend zu einem gemeinsamen System/Produkt integriert. Eine wichtige Aufgabe des Systems Engineerings ist es, die Zusammenarbeit zwischen heterogenen Entwicklungsteams zu unterstützen, um dabei auftretende Reibungsverluste zum Beispiel die domänenspezifische Sprachbarriere („Fachchinesisch“), zwischen den unterschiedlichen Domänen zu minimieren.

Einige der hier stichpunktartig aufgelisteten Systementwicklungsparadigmen werden im weiteren Verlauf dieser Arbeit noch einmal aufgegriffen, aufgrund der Tatsache, dass sie für die Kompatibilitätsmodellierung und Bestimmung von zentraler Bedeutung sind. Die übrigen Ziele und Paradigmen aus dem Systems Engineering Umfeld, die nicht direkt für die Kompatibilitätsmodellierung und -bestimmung benötigt werden, werden im weiteren Verlauf dieser Arbeit nicht weiter verfolgt.

Integrierte Systementwicklung – Der Systems Engineering Ansatz
Wie die beiden obigen Systems Engineering Definitionen nahe legen, ist das Systems Engineering eine domänenübergreifende, integrierende Wissenschaft, deren Ziel es ist, eine schnelle und effiziente Systementwicklung über Fachgrenzen hinweg zu gewährleisten. Ausgehend von der Grundidee, sämtliche an der Systementwicklung beteiligte Fachrichtungen zu integrieren, liegt es nahe, auch die Kompatibilitätsmodellierung und -bestimmung mit in die allgemeinen Systems Engineering Aufgaben

Anmerkung:
Die einzelnen Phasen können je nach Anwendungsgebiet auch andere Namen tragen. Zum Beispiel im Anlagenbau werden die Phasen auch mit (A) Konzeptphase, (B) Definitionsphase, (C/D) Entwurfs- und Entwicklungsphase, (E/F) Fertigungs-, Betriebs- und Wartungsphase und (G) Stilllegungsphase bezeichnet [Wik08m].

aufzunehmen. Die Abbildung 1.4 zeigt eine Auswahl an unterschiedlichen Aufgabenbereichen, die im Systems Engineering Umfeld bei der Systementwicklung erledigt werden müssen.

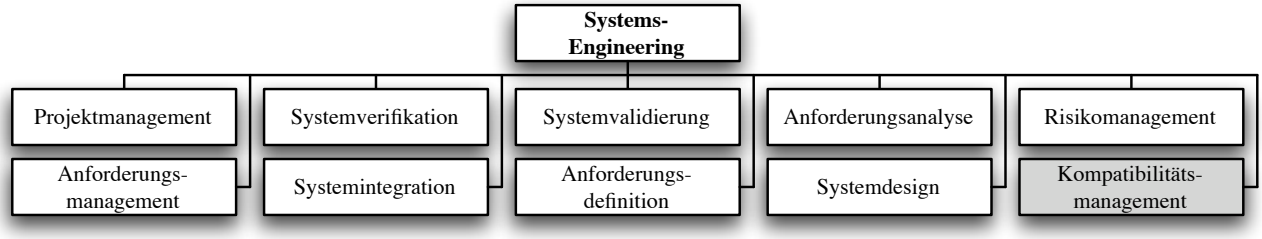


Abbildung 1.4.: Aufgabenbereiche des Systems Engineerings (in Anlehnung an [Wei06, 12]).

Zusätzlich zu den allgemeinen Aufgaben des Systems Engineerings ist in der Abbildung 1.4 (rechts) das Kompatibilitätsmanagement als weitere Teilaufgabe aufgenommen worden. Das Hauptziel des Kompatibilitätsmanagements im Systems Engineering Umfeld ist es, Methoden und Prozesse zu definieren und bereitzustellen, mit deren Hilfe die Kompatibilität von technischen Systemen bei der Entwicklung und Wartung sichergestellt werden kann. F. Bornemann beschreibt in seiner Veröffentlichung „Managing compatibility throughout the product life cycle of embedded systems – Definition and application of an effective process to control compatibility“ ([BDW06]) die Notwendigkeit des Kompatibilitätsmanagements bei der Entwicklung und Wartung von technischen Systemen und insbesondere von langlebigen eingebetteten softwarelastigen Systemen bestehend aus Elektrotechnik-, Mechanik- und Softwarekomponenten.

Integration von Kompatibilitätsmanagement in die Systems Engineering Welt
Um ein komplexes technisches System effizient entwickeln und warten zu können, hat F. Bornemann den Mikro-Kompatibilitätssicherungsprozess DIBMUK – **D**efinition, **I**dentifikation, **B**ewertung, **M**aßnahmen, **U**msetzung und **K**ontrolle[14] – in seiner Veröffentlichung vorgeschlagen, der während des gesamten Systementwicklungsprozesses, begonnen bei der (Vor-) Entwicklung über den Betrieb bis zur Wartung des Systems, eingesetzt werden kann. Der DIBMUK-Prozess unterteilt sich dabei in sechs aufeinander aufbauende Teilprozesse, die sicher stellen, dass angefangen bei der Systementwicklung über den Betrieb bis hin zur Wartung und Pflege eines technischen Systems, die Kompatibilitätssicherung wahrgenommen werden kann. In der Abbildung 1.5 ist der Mikro-Kompatibilitätssicherungsprozess DIBMUK mit seinen sechs nacheinander ablaufenden Phasen anschaulich dargestellt.

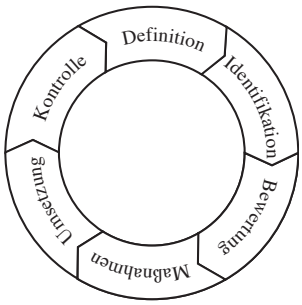


Abbildung 1.5.: Der DIBMUK-Micro-Kompatibilitätssicherungsprozess.

Die Steuerung, Leitung und Koordination des DIBMUK-Prozesses wird dabei vom Kompatibilitätsmanagement bzw. vom Kompatibilitätsverantwortlichen wahrgenommen. Er überwacht und koordiniert sämtliche Prozessschritte die während des DIBMUK-Prozesses auftreten. Die nachfolgende Aufzählung beschreibt die einzelnen Phasen des DIBMUK-Prozesses zur Wahrung der Kompatibilität während des Entwicklungs- bzw. Wartungsprozesses eines technischen Systems.

- **Definition** der Kompatibilitätsanforderungen
  Die erste Phase des DIBMUK-Prozess beschäftigt sich mit der Definition der Kompatibilitätsanforderungen sowie mit der Identifikation der für die Kompatibilitätsbestimmung relevanten Eigenschaften des Systems. Sämtliche hier identifizierten kompatibilitätsrelevanten Eigenschaften bilden die Basis für die spätere Identifikation und Bewertung von Inkompatibilitäten.
  Aufgaben und Ziele:
  - Vollständige Identifikation und Sammlung aller kompatibilitätsrelevanten Eigenschaften des Systems.
  - Beschreibung der gültigen Wertebereiche aller kompatibilitätsrelevanten Eigenschaften des Systems.
  - Alle an der Entwicklung und Wartung beteiligten Ingenieuren müssen obige kompatibilitätsrelevanten Eigenschaften des Systems bekannt sein um möglichen Inkompatibilitäten während der Systementwicklung entgegenzuwirken.
  - Nachträgliche Änderungen der Anforderungen sowie der kompatibilitätsrelevanten Systemeigenschaften sind allen beteiligten Fachabteilungen mitzuteilen und gemeinsam darüber abzustimmen.

[14] Übersetzung aus dem englischen Akronym *DIEMIC*. Siehe auch [BKB+07].

Die Abteilung Kompatibilitätsmanagement bzw. der Kompatibilitätsverantwortliche sorgt in dieser Phase dafür, dass alle identifizierten kompatibilitätsrelevanten Systemeigenschaften in einem zentralen System für alle Abteilungen zugänglich hinterlegt werden. Mit Hilfe dieser Datenbasis wird die Kompatibilität aller Systembestandteile während des gesamten Entwicklungs- und Wartungsprozess des Systems überprüft.

- **Identifikation** der Störung/Inkompatibilität
  Sämtliche Störungen und mögliche Inkompatibilitäten während der Systementwicklungsphase bzw. während des Betriebs des Systems werden in dieser DIBMUK-Phase durch den Kompatibilitätsverantwortlichen gesammelt und bei dem Verdacht einer möglichen Inkompatibilität zwischen den geforderten Systemeigenschaften sowie dem beobachteten Systemverhalten werden die nachfolgenden DIBMUK-Phasen angestoßen.

  Aufgaben und Ziele:

  – Ausführliche Meldung und Beschreibung der vorliegenden Störung, Inkonsistenz oder Inkompatibilität des Systems an den Kompatibilitätsverantwortlichen.

  – Identifizierung welche Bauteile, Ebenen und Fachbereiche betroffen und in den Lösungsfindungsprozess einbezogen werden müssen.

  – Absicherung mit Experten der entsprechenden Fachabteilungen, dass es sich bei dem beobachteten Systemverhalten um einen Fehler im Systemmodell oder um eine Inkompatibilität handelt.

  Nach der positiven Identifikation einer Störung bzw. einer Inkompatibilität werden die restlichen vier Phasen des DIBMUK-Prozesses angestoßen. Der Kompatibilitätsverantwortliche steuert und koordiniert dabei die restlichen vier Phasen des Kompatibilitätssicherungsprozesses.

- **Bewertung** der Funktionsstörung bzw. der Inkompatibilität
  Diese Phase beschäftigt sich mit der Bewertung sowie der exakten Untersuchung der vorliegenden Funktionsstörung und dem damit verbundenen Verdachtsmoment auf eine Inkompatibilität. Dabei wird die Kritikalität der gefundenen Störung/Inkompatibilität bewertet und über das weitere Vorgehen zur Wiederherstellung der Funktionalität bzw. der Kompatibilität des Systems entschieden.

  Aufgaben und Ziele:

  – Identifizierung der betroffenen Systembereiche.

  – Gemeinsame und ausführliche Bewertung der Störung/Inkompatibilität durch Experten der entsprechenden Fachabteilungen in Bezug auf Kritikalität, Auswirkungsbereich und Ursachen mit dazugehörigem Verantwortungsbereich. Klärung, ob die Ursache der Störung auf eine Inkompatibilität zurückzuführen ist.

  – Einbezug der beteiligten Entwicklungsabteilungen bzw. der Service- und Wartungsbereiche.

- Definition der **Maßnahmen**
  Die Maßnahmenphase befasst sich mit der Identifikation und Definition von geeigneten Maßnahmen nachdem zuvor festgestellt worden war, dass es sich bei dem beobachteten Systemverhalten um eine (kritische) Inkompatibilität handelt, die es zu kontrollieren bzw. zu beseitigen gilt.

  Aufgaben und Ziele:

  – Die Geschäftsbereiche Kompatibilitätsmanagement, Projektmanagement, Konfigurationsmanagement, Änderungsmanagement und Problemmanagement werden in die Festlegung geeigneter Maßnahmen einbezogen.

  – Gemeinsame Definition von effektiven, rentablen und langfristigen Entscheidungen und Maßnahmen, wie mit der Störung (Fehler/Inkompatibilität) umzugehen ist.

  – Maßnahmen müssen mitsamt Strategie, Produktplattformen, -varianten, -laufzeiten (Abkündigungen) kommuniziert und dokumentiert werden.

  Anmerkung:
  Bei der Festlegung geeigneter Maßnahmen zur Beseitigung der Störung/Inkompatibilität kann es vorkommen, dass entschieden wird, die Störung/Inkompatibilität nicht zu beseitigen da beispielsweise die Beseitigung aus technischen oder aus Kostengründen nicht durchgeführt werden kann.

- **Umsetzung** der Maßnahmen
  Die Umsetzungsphase beschäftigt sich mit der Weiterleitung der zuvor festgelegten Gegenmaßnahmen zur Beseitigung der Inkompatibilität an die verantwortlichen Fachbereiche. Der Kompatibilitätsverantwortliche ist dabei zuständig für die Steuerung und Rückmeldung der Vollständigkeit der Umsetzung bzw. von möglichen Fehlern oder Unklarheiten. Die Änderungsanforderungen am Systemmodell müssen vollständig dokumentiert (vorzugsweise im Änderungs- und Konfigurationsmanagement) und an alle relevanten Entwicklungsabteilungen kommuniziert werden.

  Aufgaben und Ziele:

  – Weiterleitung der Maßnahmenentscheidungen an die entsprechenden (Fach-) Bereiche.

  – Der Kompatibilitätsverantwortliche steuert die Umsetzung sowie die Rückmeldung der Inkompatibilitätsbeseitigung.

  – Vollständige Kommunikation und Dokumentation der Änderungen am System.

- **Kontrolle** der Maßnahmen
  Die letzte Phase des DIBMUK-Prozess befasst sich mit der Kontrolle der vollständigen Umsetzung der zuvor festgelegten Maßnahmen. Dies beinhaltet neben der Kontrolle auch die Rückmeldung der Fachbereiche über den Status bzw. Abschluss der Umsetzung und die Bestätigung des gewünschten Nutzens der Maßnahmen. Erst nach Abschluss der Kontrollphase und der damit verbundenen Beseitigung (oder auch lediglich Beobachtung) der identifizierten Inkompatibilität lässt sich der DIBMUK-Prozess zur Wahrung der Kompatibilität abschließen.

  Aufgaben und Ziele:

  - Der Kompatibilitätsverantwortliche sorgt für Dokumentation aller am System durchgeführten Maßnahmen.
  - Freigabe des Systems durch den Kompatibilitätsverantwortlichen.

Zusätzlich zu dem von F. Bornemann vorgeschlagenen sechs Schritten des DIBMUK-Mikro-Prozesses zur Wahrung und Sicherung der Kompatibilität ist eine intensive Vernetzung sowohl mit den einzelnen Systems Engineering Aufgabenbereichen (vgl. Abbildung: 1.4), als auch mit den Phasen des zu Grunde liegenden Vorgehensmodells (Makro-Prozess) notwendig, um mögliche Inkompatibilitäten während des gesamten Systementwicklungsprozesses zu identifizieren und gegebenenfalls geeignete Gegenmaßnahmen ergreifen zu können. Wird als Vorgehensmodell zum Beispiel das in der Raumfahrt übliche Phasenmodell zur Systementwicklung gewählt, so kann der DIBMUK-Mikro-Prozess vor allem in den Entwicklungsphasen 0 bis D bzw. während der Wartung und Pflege (Phasen E und F) des Systems eingesetzt werden. Dabei ist der konsequente Einsatz des DIBMUK-Mikro-Kompatibilitätssicherungsprozesses besonders während der frühen Systementwicklungsphasen (Phasen 0 bis C) wichtig, da hier im Allgemeinen die „entscheidenden Weichen" für das zu entwickelnde System gestellt werden. Dabei gilt insbesonders: nur während der Phasen 0 bis C wird der gesamte DIBMUK-Prozess durchlaufen. In allen darauf folgenden Phasen des Phasenmodells wird die Definitionsphase des DIBMUK-Prozesses nicht mehr betreten da während dieser Phasen im Allgemeinen keine weiteren kompatibilitätsrelevanten Eigenschaften des Systems identifiziert werden und lediglich die bereits zuvor bekannten kompatibilitätsrelevanten Eigenschaften als Basis für die Identifikations- bzw. die Bewertungsphase des DIBMUK-Prozesses verwendet werden. Insbesondere ist die Integration und Vernetzung der einzelnen Phasen des DIBMUK-Mikro-Prozesses mit den zentralen Aufgabenbereichen des Systems Engineerings, dem Projektmanagement sowie mit der Systemverifikation- und Validierung und dem eigentlichen Systemdesign, von entscheidender Bedeutung. Zusätzlich zu den klassischen Systems Engineering Aufgabenbereichen kommt nun noch das Kompatibilitätsmanagement hinzu, das sich ausschließlich mit der Umsetzung des DIBMUK-Prozesses zur Wahrung und Sicherung der Kompatibilität innerhalb eines Systems beschäftigt. Die Vernetzung der unterschiedlichen Fachbereiche kann zum Beispiel mit Hilfe eines gemeinsamen domänenübergreifenden Systemmodells[15] erreicht werden. Abbildung 1.6 zeigt schematisch die Vernetzung des DIBMUK-Kompatibilitätssicherungsprozesses mit den unterschiedlichen Systems Engineering Aufgabenbereichen sowie die Verbindung sowohl des DIBMUK-Prozesses, als auch der Systems Engineering Aufgabenbereiche mit den unterschiedlichen Phasen des Phasenmodell. Durch diese Darstellung wird deutlich, dass in sämtlichen Phasen des Phasenmodells nahezu alle Systems Engineering Aufgaben inklusive des Kompatibilitätsmanagements durchgeführt werden müssen, um eine schnelle und erfolgreiche Systementwicklung zu unterstützen.

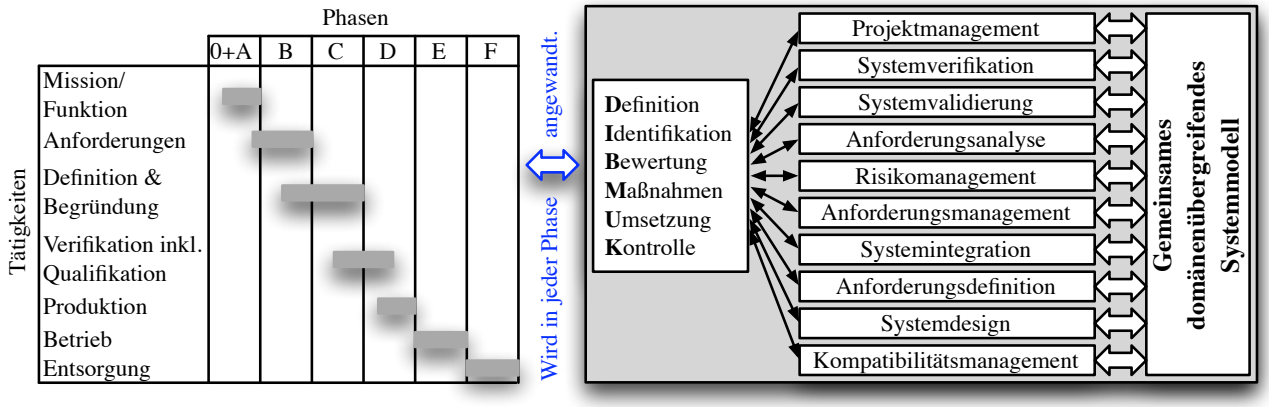


Abbildung 1.6.: Vernetzung zwischen Systems Engineering Aufgaben, dem DIBMUK-Prozess sowie den Projektphasen.

Die Abbildung 1.7 auf der nächsten Seite verdeutlicht die enge Kopplung des DIBMUK-Mikro-Prozesses mit den einzelnen Phasen des Phasenmodells. Dabei kann der DIBMUK-Mikro-Prozess grundsätzlich während sämtlicher Phasen des Phasenmodells eingesetzt werden (vgl. Abbildung: 1.8 auf Seite 10). Ab der Phase D (der Produktion des Systems) kommen im Allgemeinen keine weiteren kompatibilitätsrelevanten Eigenschaften mehr zum Systemmodell hinzu, so dass ab diesem Zeitpunkt die Definitionsphase des DIBMUK-Prozesses nicht mehr durchlaufen wird. Zusätzlich zur engen Kopplung des DIBMUK-Mikro-Prozesses mit dem Makro-Prozess (Vorgehensmodell), verdeutlicht die Abbildung 1.7 die Notwendigkeit eines domänenübergreifenden Systemmodells, in dem sämtliche an der Systementwicklung beteiligten Fachbereiche mitarbeiten. Durch die konsequente Nutzung des gemeinsamen Systemmodells für alle Fachabteilungen kann die Fehlerrate bzw. mögliche Inkompatibilitäten die während der Systementwicklung auftreten können effektiv reduziert bzw. in einzelnen Fällen sogar vollständig verhindert werden. Eine

[15] Nähere Informationen zu Modellen sowie deren besonderen Bedeutung während der Systementwicklung finden Sie in den Kapiteln *„Domänenübergreifendes integriertes Systemmodell"*, *„Modellbildung"* und *„Modellierungssprachen und -techniken für technische Systeme"*.

ausführliche Beschreibung des Aufbaus sowie der Struktur des gemeinsamen domänenübergreifenden Systemmodells finden Sie im Kapitel „2.6 *Domänenübergreifendes integriertes Systemmodell*“ ab Seite 103 dieser Arbeit.

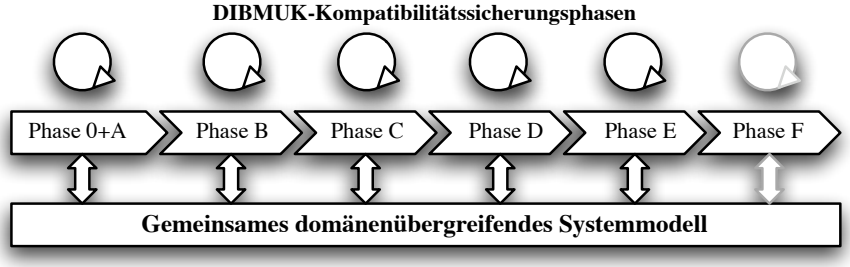


Abbildung 1.7.: Die DIBMUK-Kompatibilitätssicherungsphasen werden während der gesamten Systementwicklung eingesetzt, insbesondere während der frühen Phasen 0 bis D des Phasenmodells bzw. während der Betriebsphase E.

Während der Phase F des Phasenmodells, der Entsorgungsphase, wird im Allgemeinen weder das Kompatibilitätsmanagement noch der DIBMUK-Prozess eingesetzt da in dieser Phase das System außer Dienst gestellt wird und weitere Wartungs- und Instandsetzungsarbeiten nicht mehr durchgeführt werden.

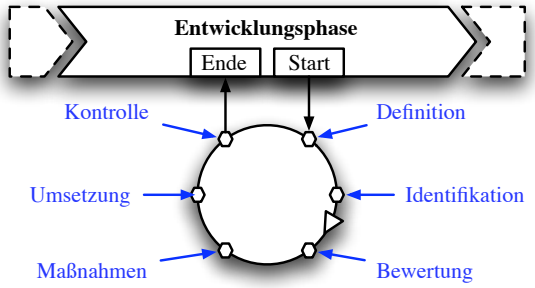


Abbildung 1.8.: Einbettung der DIBMUK-Kompatibilitätssicherungsphasen in eine Entwicklungsphase.

Anmerkung:
Anstatt des aus der Raumfahrt stammenden Phasenmodells kann jedes beliebige andere Vorgehensmodell verwendet werden, das seine Aktivitäten in Phasen unterteilt. Dazu zählen beispielsweise das V-Modell [Roc06], das V-Modell XT [BMI06] sowie das in der Softwareentwicklung weit verbreitete Spiralmodell [Mag06].

Obwohl das Systems Engineering laut Definition eine domänenübergreifende integrierende Wissenschaft ist, wird im weiteren Verlauf dieser Arbeit bewusst auf die Beschreibung aller unterschiedlichen Aufgabenbereiche des Systems Engineerings während der Systementwicklung verzichtet, um den Umfang dieser Arbeit nicht zu sprengen. Aus diesem Grund verweise ich an dieser Stelle ausdrücklich auf Sekundärliteratur zum Thema Systems Engineering, wie zum Beispiel [Hab73], [Ver06a] und [DH02] bzw. die Veröffentlichungen der Gesellschaft für Systems Engineering (GfSE) [GfS07], die sich intensiv mit den einzelnen Aufgabenbereichen des Systems Engineerings beschäftigen.

Systems Engineering – Denken in Modellen
Ein weiterer zentraler Lösungsansatz des Systems Engineerings zur Entwicklung und Beherrschung komplexer Systeme ist das „Denken in Modellen“ bzw. das „Denken in Systemen“, das oft auch als „Systemdenken[16]“ bezeichnet wird. Prof. Dr.-Ing. E. Igenbergs stellt in seiner Vorlesung „Systems Engineering [PDII06, 4]“ die These auf, „Menschen kommunizieren modellbasiert“. Ausgehend von dieser These ist das Denken in Modellen, so wie es das Systems Engineering vorschlägt, lediglich eine Anwendung der jedem Menschen angeborenen Herangehensweise Dinge aus der uns umgebenden Welt als abstrakte Modelle wahrzunehmen. Aufgrund der Tatsache, dass jeder Mensch seine Umwelt als Modell wahrnimmt, ist auch die Kommunikation zwischen Menschen stets modellbasiert. Für die gemeinsame, domänenübergreifende Systementwicklung ist das denken in Modellen eine unerlässliche Technik, da im Allgemeinen ein reales System zu komplex ist, als dass es in allen Details vollständig modelliert und dargestellt werden kann. In einem (abstrakten) Modell eines Systems hingegen sind lediglich diejenigen Informationen enthalten, die für das Verständnis des Systems von Bedeutung sind. Für das Verständnis bzw. die Systementwicklung unwichtige Informationen werden dabei bewusst nicht modelliert, um das Systemmodell nicht zu überfrachten.

Die Abbildung 1.9 auf der nächsten Seite stellt den Auswahl- und Bewertungsprozess von wichtigen und unwichtigen Daten und Informationen bei der Systementwicklung dar. Im Kasten (1) ist die Menge aller Informationen und Daten enthalten, die

[16] Anmerkung:
Im Kapitel „2.2 *Modellbildung*“ ab Seite 31 wird der zentrale Gedanke des Systems Engineerings, das Denken in Modellen, noch einmal aufgegriffen und anhand von einfachen Beispielen vertieft. Zusätzlich werden im Kapitel „3 *Modellierungssprachen und -techniken für technische Systeme*“ ab Seite 109 unterschiedliche Modellierungssprachen und Techniken für (eingebettete) Systeme vorgestellt und ebenfalls anhand eines durchgängigen Beispiels erläutert. Zusätzlich siehe: [DH02, 4,22][PS00][Tho02] für weitere Informationen.

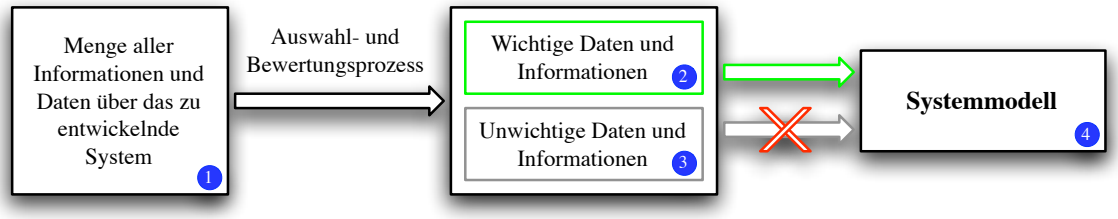


Abbildung 1.9.: Übertragung von wichtigen Daten und Informationen in das Systemmodell.

über das zu entwickelnde System bekannt sind. Dazu zählen zum Beispiel bei einem eingebetteten System Schaltpläne, CAD Zeichnungen und Quellcode. Zusätzlich zu diesen für die Herstellung des Systems notwendigen technischen Daten können in dem Systemmodell auch Managementdaten wie beispielsweise Kostenpläne oder Gantt-Charts[17] enthalten sein (integriertes Modell). Für die Kompatibilitätsmodellierung und -bewertung werden in dieser Arbeit jedoch lediglich die technischen Aspekte eines Systems betrachtet. In (2) sind alle extrahierten Daten enthalten, die für die Systementwicklung notwendig sind, und die in das Systemmodell (4) übernommen werden müssen, um das System entwickeln und herstellen zu können. Der Kasten (3) enthält alle anderen Informationen die nicht direkt für die Systementwicklung notwendig sind. Sie werden nicht in das Systemmodell übernommen, um dessen Komplexität nicht unnötig zu erhöhen.

Das nach dem obigen Auswahlprozess entstandene Systemmodell kann nun sowohl zur Entwicklung des Systems als auch zur Kommunikation zwischen den an der Entwicklung beteiligten Fachabteilungen verwendet werden. Das Systemmodell stellt somit die zentrale Kommunikationsplattform zwischen allen an der Systementwicklung beteiligten Abteilungen dar. Zusätzlich zur Verwendung eines domänenübergreifenden Systemmodells während der Systementwicklung ist ein zentrales fächerübergreifendes Glossar hilfreich, in dem sämtliche Fachbegriffe und Definitionen aus den unterschiedlichen an der Entwicklung beteiligten Domänen enthalten ist. Dieses Glossar dient ausschließlich zur Verbesserung des Verständnisses bzw. der Kommunikation zwischen den einzelnen Projektbeteiligten.

Systems Engineering ist die Grundlage für die modellbasierte Kompatibilitätsbestimmung
Die unterschiedlichen Aufgaben und Tätigkeiten des Systems Engineerings (vgl. Abbildung: 1.7 auf Seite 10) innerhalb eines Projekts bilden zusammen die Grundlage für eine erfolgreiche Systementwicklung. Insbesondere ist dabei der neue Aufgabenbereich, das Kompatibilitätsmanagement, innerhalb des Systems Engineerings zu nennen. Das Kompatibilitätsmanagement kümmert sich ausschließlich um die Sicherung und Wahrung der Kompatibilität während des gesamten Lebenszyklus eines Systems. Um dies zu erreichen, ist eine enge Kopplung zwischen allen Aufgabenbereichen des Systems Engineering Prozesses notwendig. Des Weiteren ist der Einsatz eines domänenübergreifenden Systemmodells hilfreich, das über den gesamten Entwicklungszyklus des Systems eingesetzt und verwendet wird. Dieses Systemmodell muss sämtliche kompatibilitätsrelevanten Eigenschaften des zu entwickelnden Systems enthalten. Aufbauend auf diesem Modell kann dann die Kompatibilität eines Systems bestimmt und bewertet werden.

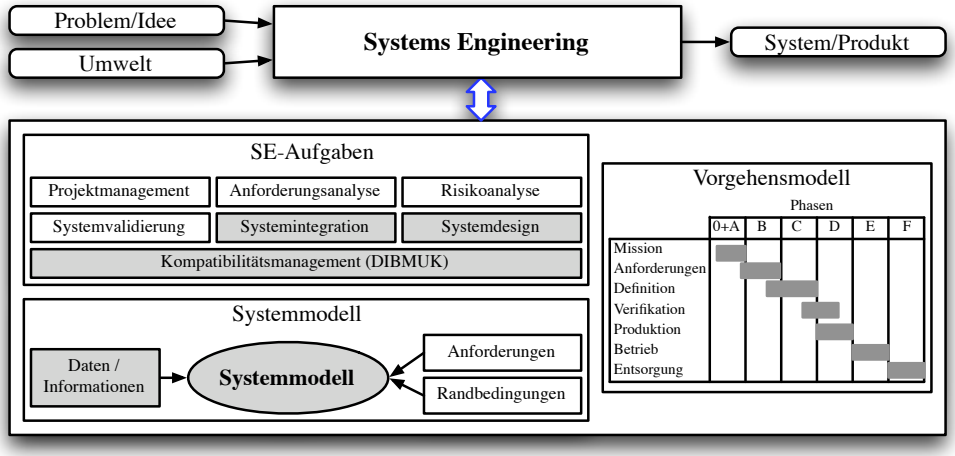


Abbildung 1.10.: Systems Engineering ist die Grundlage für die modellbasierte Kompatibilitätsbestimmung.

Die Abbildung 1.10 illustriert das Zusammenspiel der unterschiedlichen Aspekte des Systems Engineerings. Dabei wird im Verlauf dieser Arbeit explizit auf die grau hinterlegten Bestandteile des Systems Engineerings eingegangen. Die übrigen Systems Engineering Aufgaben, Modelleigenschaften und Vorgehensmodelle werden hier nicht weiter verfolgt[18].

[17] Ein *Gantt-Chart* oder *Balkenplan* ist ein nach dem Unternehmensberater Henry L. Gantt (1861-1919) benanntes Instrument des Projektmanagements, das die zeitliche Abfolge von Aktivitäten graphisch in Form von Balken auf einer Zeitachse darstellt [Wik08j].
[18] Die Integration des Kompatibilitätsmanagements in das ECSS Vorgehensmodell wird insbesondere in [BE08] beschrieben.

Nachdem nun geklärt worden ist, wie das Systems Engineering mit modellbasierter Kompatibilitätsmodellierung und -bewertung verbunden ist, werden im nachfolgenden Kapitel „*Grundlegende Begriffe und Definitionen aus der Systems Engineering Welt*“ die zentralen Begriffe und Definitionen aus dem Systems Engineering eingeführt und anhand von einfachen Beispielen erläutert, die für die Kompatibilitätsmodellierung und -bewertung entscheidend sind.

1.3. Grundlegende Begriffe und Definitionen aus der Systems Engineering Welt

Nachdem im vorangegangenen Kapitel die Frage „*Was hat Systems Engineering mit modellbasierter Kompatibilitätsbewertung zu tun?*“ geklärt worden ist, folgt nun in diesem Kapitel, eine knappe Vorstellung und Erläuterung der wesentlichen Begriffe und Definitionen aus der Systems Engineering Welt, die für die Kompatibilitätsmodellierung sowie für die Bewertung der Kompatibilität von eingebetteten softwarelastigen Systemen von entscheidender Bedeutung sind. Begonnen wird mit dem wohl wichtigsten Begriff, dem „System“. In der Definition 1.2 wird der Begriff System als ein beliebiger, abgegrenzter Gegenstand unseres Denkens definiert[19].

Definition 1.2 ***System***
Ein ***System*** *kann ein beliebiger, abgegrenzter Gegenstand unseres Denkens sein*[20]*. Die* ***Systemgrenze*** *trennt das betrachtete System von der* ***Umwelt****, in die es eingebettet ist [DIN98, 66].*

Diese sehr allgemein gehaltene Systemdefinition von Dr. Negele lässt sich auf unterschiedliche Systeme anwenden. So kann zum Beispiel das *biologische System Mensch* ebenso als System betrachtet werden wie beispielsweise das *technische System Computer*. Für die Modellierung ist der Systembegriff von entscheidender Bedeutung, da er genau festlegt, wo die Grenze zwischen dem zu betrachtenden System, das modelliert und beschrieben werden soll, und der Umwelt liegt. Ohne eine genaue Festlegung der Systemgrenzen des zu entwickelnden Systems ist dessen exakte Beschreibung nicht möglich. Die Abbildung 1.11 zeigt zwei Systeme, *System A* und *System B*, die in der sie umgebenden Umwelt eingebettet sind. Jedes der beiden Systeme hat eine eineindeutige[21] Systemgrenze, die das System gegenüber der Umwelt abgrenzt.

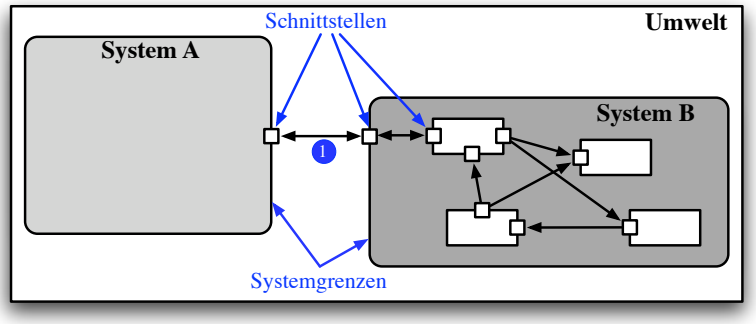


Abbildung 1.11.: System und Umwelt.

Die beiden Systeme *System A* und *System B* sind beide in der sie umgebenden Umwelt eingebettet. Jedes der beiden Systeme hat eine eineindeutige Systemgrenze, die das System gegenüber der Umwelt abgrenzt. Um die Systemgrenze zu überwinden sind spezielle Übergänge, so genannte *Schnittstellen*, notwendig. Nur über diese ausgezeichneten Schnittstellen kann das System mit der Umwelt kommunizieren. Eine anderweitige Kommunikation ist nicht möglich.

Definition 1.3 ***Schnittstelle***[22]
An einer ***Schnittstelle*** *treffen* ***zwei autonome System*** *zusammen, um in* ***geregelter Weise*** *etwa Daten oder Informationen auszutauschen.*

Die Schnittstelle von Systemen bzw. die Schnittstellen zwischen den einzelnen Bestandteilen von Systemen, wie z.B. Subsystemen, ist für die Kompatibilitätsmodellierung und -bestimmung von entscheidender Bedeutung. Die beiden Systeme in der Abbildung 1.11 haben jeweils eine explizit modellierte Schnittstelle, über die sie mit ihre Umwelt interagieren können (1). Nur über diese explizit modellierte Schnittstelle kann das System mit der Umwelt kommunizieren. Eine alternative Kommunikation, die nicht über die Schnittstelle des Systems führt ist, ist nicht möglich. Diese strikte Schnittstellendefinition ist eine zwingende Notwendigkeit sowohl für die Modellierung, als auch für die Bewertung von Kompatibilität.

[19] Weitere Definitionen des Begriffs „System“ finden Sie unter Definition 1.4 und B.4 auf Seite 325.
[20] Anmerkung:
Philosophisch betrachtet,kann es auch Systeme außerhalb unseres Denkens geben. Zum Beispiel solche über die noch niemand nachgedacht hat. Aber auch für diese Systeme gilt die obige Definition, dass sie sich durch die Systemgrenze von ihrer Umwelt abgrenzen lassen.
[21] Im *Systems Engineering* Kontext bedeutet *eineindeutig*, dass ein System einen Namen/Objekt/Element genau ein Mal im System existiert. In der *Mathematik* bedeutet *eineindeutig*: Die Abbildung einer Menge M in eine Menge N, bei der jedem Element von M genau ein Element von N und jedem Element von N genau ein Element von M entspricht.
[22] Siehe auch die erweiterte Schnittstellendefinition nach Prof. Dr. Dr. h.c. M. Broy auf Seite 326 des Anhangs.

Anmerkung:
Die Definition 1.3 gilt insbesondere auch, wenn nur ein System in der Umwelt eingebettet ist.

Beispiel 1: System, Schnittstellen und Umwelt
Jedes System besitzt laut Definition 1.2 eine dezidierte Systemgrenze, die das System von seiner Umwelt abgrenzt. Die Abbildung 1.12 zeigt das System *System A*. Es ist in der Umwelt eingebettet und besitzt zwei ausgezeichnete Schnittstellen, die links und rechts am System angebracht sind. Über diese und nur diese beiden Schnittstellen kommuniziert das System mit seiner Umwelt.

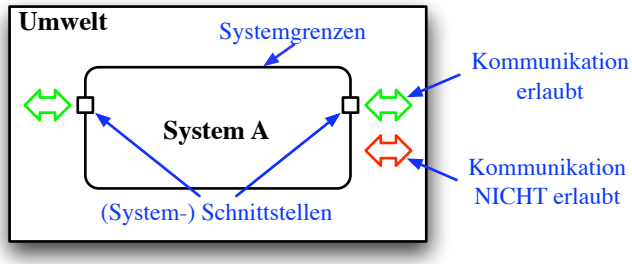


Abbildung 1.12.: Beispiel: System und Umwelt.

Nur über die beiden ausgezeichneten Schnittstellen kann auf das System von außerhalb, also der Umwelt, zugegriffen werden. Das System kann ebenfalls nur über die beiden Schnittstellen Daten an die Umwelt senden. Ein Kommunikationsversuch mit dem System, der nicht über die ausgezeichnete Schnittstelle erfolgt, wird vom System abgelehnt.

Diese strikte „Abschottung“ des Systems gegenüber der Umwelt und insbesondere gegenüber Kommunikationsverbindungen ist für die Kompatibilitätsmodellierung und -bewertung von entscheidender Bedeutung. Nur an den dezidierten Schnittstellen des Systems erfolgt eine Kommunikation des Systems mit der Umwelt. Durch diese strikte Schnittstellendefinition ist eine nicht explizit modellierte Kommunikation bzw. Beeinflussung des Systems von außen ausgeschlossen. Ebenfalls kann so eine unbeabsichtigte Beeinflussung der Umwelt durch das System unterbunden werden. Aus diesem Grund müssen die Eigenschaften der Schnittstellen sehr präzise modelliert und beschrieben werden, damit sie für die Kompatibilitätsbewertung herangezogen werden können. □

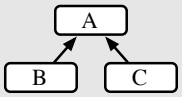
Basierend auf der Systemdefinition (Def.: 1.2) kann ein System weiter in einzelne Teile, so genannte Teil- bzw. Subsysteme, unterteilt werden. In der nachfolgenden Definition wird beschrieben, wie ein System weiter unterteilt werden kann.

Definition 1.4 ***Teil- oder Subsystem***
*Ein **Teil-** bzw. **Subsystem** ist ein abgegrenzter Bereich innerhalb eines Systems, der selbst Merkmale eines **Systems** aufweist.*

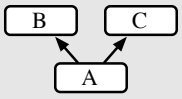
Mit Hilfe der Definition 1.4 ist es möglich, ein komplexes System weiter zu unterteilen bzw. zu strukturieren. Die Abbildung 1.11 auf der vorherigen Seite zeigte, in Anlehnung an die allgemeine Systemdefinition von Dr. Negele (Def.: 1.2), zwei autonome Systeme (*System A* und *System B*), die in der Umwelt eingebettet sind. Das System *System A* enthält keine (sichtbaren) Teil- bzw. Subsysteme. Im Gegensatz dazu enthält das System *System B* vier Teil- bzw. Subsysteme. Jedes der vier Subsysteme besitzt ausgezeichnete Schnittstellen (vgl. Def.: 1.3) über die die Teil- bzw. Subsysteme untereinander und mit dem übergeordneten System kommunizieren können. Aus der Aufteilung eines Systems in Teil- bzw. Subsysteme folgt unmittelbar die nachfolgende Hierarchiedefinition.

Definition 1.5 ***Hierarchie bzw. Systemhierarchie***
*Als **Hierarchie** bezeichnet man ein System von Elementen (Teil- bzw. Subsysteme), die einander **über-** bzw. **untergeordnet** sind. Ist dabei jedem Element höchstens ein anderes Element unmittelbar übergeordnet, so spricht man von einer **Monohierarchie**, während bei einer **Polyhierarchie** auch mehrere übergeordnete Elemente möglich sind. Jede Hierarchie lässt sich mathematisch als Ordnungsrelation[23] beschreiben. Dabei gilt: Ein Baum definiert eine Monohierarchie, während ein gerichteter azyklischer Graph eine Polyhierarchie definiert.*

- ***Monohierarchie – Baumstruktur***
  Jedes System ist genau einem System über- oder untergeordnet.



- ***Polyhierarchie – gerichteter azyklischer Graph***
  Ein System kann mehr als einem System untergeordnet sein.



[23] Siehe hierzu: Kapitel „A.2 *Relationen*“ ab Seite 317.

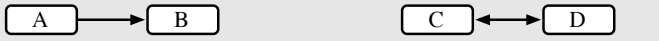
Lässt sich ein System weiter in Teil- bzw. Subsysteme unterteilen, so ist das System hierarchisch strukturiert. Dabei gilt: Gehört jedes Teil- bzw. Subsystem genau einem übergeordneten System, dem Ober- bzw. Vatersystem, an, so ist das System monohierarchisch strukturiert. Daraus folgt unmittelbar, dass die Struktur der meisten technischen Systeme monohierarchisch aufgebaut ist, da ein Teil- bzw. Subsystem in den meisten Fällen nur in einem Vatersystems enthalten sein kann. Sind beispielsweise die beiden Systeme *Kolben* und *Motor* gegeben, so ist das System *Kolben* ein Teil- bzw. Subsystem des Systems *Motor*, da im Allgemeinen ein Kolben in einem Motor verbaut ist. Somit ist das (Teil-) System *Kolben* dem System *Motor* „untergeordnet".

Es gibt jedoch auch technische Systeme die nicht monohierarchisch sondern polyhierarchisch strukturiert sind. So kann beispielsweise ein Subsystem, abhängig von seiner Modellierung, sowohl einem, als zu einem anderen System angehören. Ein Beispiel hierfür ist die Befestigung eines Bildes an der Wand mittels einer Schraube. Die Schraube kann sowohl zum System Bild, zum System Wand oder sogar unabhängig als einzelnes System betrachtet werden. Polyhierarchische Systemstrukturen treten vor allem an den Systemgrenzen auf, wenn ein System nicht eineindeutig einem andern System untergeordnet werden kann.

Nachdem nun die Begriffe System, Teilsystem sowie Hierarchie geklärt worden sind, folgt nun, aufbauend auf diesen Grundbegriffen, die Definition der Verbindungen von Systemen, genauer die Verbindung der Schnittstellen der Systeme. Dabei werden grundsätzlich zwei Arten von Verbindungen unterschieden.

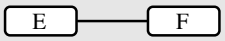
Definition 1.6 ***Verbindung zwischen Systemen***
Systeme können mittels ***Verbindungen****, auch* ***Relationen*** *genannt, an ihren Schnittstellen miteinander verbunden werden. Dabei werden grundsätzlich zwei Verbindungsarten unterschieden:*

- ***Gerichtete Verbindung***
  Mit Hilfe von gerichteten Verbindungen, auch ***Flussrelationen*** *genannt, werden Daten und Informationen stets in Pfeilrichtung vom Sender zum Empfänger hin unverändert übertragen. Des Weiteren werden gerichtete Verbindungen weiter unterschieden in einseitig- und beidseitig gerichtete Verbindungen. Dabei kann jede beidseitig gerichtete Verbindung durch zwei einseitig gerichteten Verbindungen dargestellt werden.*



einseitig gerichtete Verbindung — beidseitig gerichtete Verbindung

- ***Ungerichtete Verbindung***
  Ungerichtete Verbindungen haben keine eindeutige Flussrichtung.



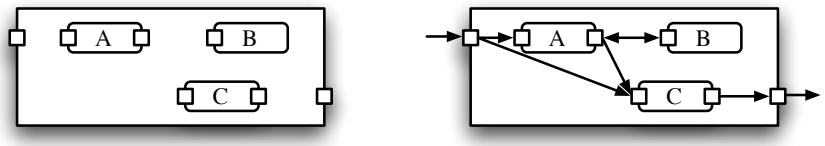
ungerichtete Verbindung

Dabei gilt insbesondere:

- *Jede Verbindung ist stets eine* ***Punkt-zu-Punkt*** *Verbindung.*
- *Verbindungen dürfen* ***nicht reflexiv***[24] *sein.*

Die Abbildung 1.11 auf Seite 12 zeigt zwei autonome Systeme (*System A* und *System B*), die in der Umwelt eingebettet und mittels einer beidseitig gerichteten Punkt-zu-Punkt Verbindung (Flussrelation) an den Schnittstellen miteinander verbunden sind (1). Die Verbindung der beiden Systeme erfolgt ausschließlich an den ausgezeichneten Schnittstellen der beiden Systeme. Das System *System A* besitzt genau eine Schnittstelle auf der rechten Seite, während das System *System B* eine Schnittstelle an der linken Seite hat. Genau an diesen beiden Schnittstellen werden die beiden Systeme miteinander mittels einer beidseitig gerichteten Relation verbunden.

Zusätzlich zu der beidseitig gerichteten Verbindung zwischen den beiden Systemen *System A* und *System B*, besteht das System *System B* aus vier Subsystemen, die ebenfalls Schnittstellen besitzen und mittels Relationen miteinander verbunden sind. Dabei ist zu erkennen, dass ein System auch mehr als eine Schnittstelle besitzen darf. Es ist jedoch laut Definition 1.6 nicht erlaubt, dass ein System eine reflexive Verbindung, also eine Verbindung zu sich selbst besitzt.



a) System ohne explizit modellierte Flussrelationen — b) System mit explizit modellierten Flussrelationen

Abbildung 1.13.: Unterschiedliche Modelle eines Systems.

Für die Modellierung und Bewertung der Kompatibilität eines Systems, genauer der Verbindung zweier autonomer Systeme an der Schnittstelle, ist die explizite Modellierung der Verbindung zwischen zwei Systemen von entscheidender Bedeutung. So kann zum Beispiel die Verbindung zwischen zwei Systemen nur dann überprüft und auf Kompatibilität untersucht werden, wenn bekannt ist, dass die beiden Schnittstellen der Systeme miteinander verbunden sind. Abbildung 1.13 zeigt links (a) ein System bestehend aus drei Teilsystemen *A*, *B* und *C*. Sowohl das System, als auch jedes der drei Teilsysteme besitzt explizit

[24] Siehe hierzu: Kapitel „A.2 *Relationen*" ab Seite 317.

modellierte Schnittstellen, jedoch ist nicht bekannt, wie die Teilsysteme miteinander und mit dem System verbunden sind. Ohne die explizite Modellierung und Darstellung der Verbindungen innerhalb des Systems ist eine Kompatibilitätsprüfung unmöglich, da unbekannt ist, wie die Teilsysteme miteinander verbunden sind. Auf der rechten Seite (b) hingegen sind sämtliche Verbindungen (Flussrelationen) zwischen den einzelnen Teilsystemen explizit dargestellt. Durch die explizite Modellierung der Verbinden zwischen den Schnittstellen der (Teil-) Systeme können diese auf Kompatibilität hin untersucht werden.

Zusätzlich zur expliziten Modellierung der Schnittstellen und der Modellierung der Verbindungen zwischen den (Teil-) Systemen, sind für die Kompatibilitätsmodellierung und -bewertung die Beschreibung der Eigenschaften des Systems und insbesondere die Schnittstelleneigenschaften von entscheidender Bedeutung. Die nachfolgende Definition 1.7 beschreibt sowohl die Eigenschaften eines Systems, als auch die Eigenschaften der Schnittstellen eines Systems.

Definition 1.7 ***Eigenschaften eines Systems***
Ein System besitzt ***Eigenschaften****, auch* ***Attribute*** *genannt, mit deren Hilfe die* ***Merkmale*** *des Systems festgelegt und beschrieben werden. Die Eigenschaften des Systems können dabei sowohl* ***konstanter****, als auch* ***veränderlicher*** *Natur sein.*

Besitzt ein System veränderliche, also dynamische Eigenschaften, so können sich diese über die gesamte Lebenszeit des Systems stetig verändern. Im konstanten (statischen) Fall sind die Eigenschaften während der gesamten Systemlebenszeit unveränderlich. Die Unterscheidung der Systemeigenschaften in statische und dynamische Eigenschaften ist vor allem für die Kompatibilitätsmodellierung und -bewertung von besonderem Interesse, da sich so bereits einfache Inkompatibilitäten zwischen Systemeigenschaften feststellen lassen. Ist zum Beispiel eine dynamische Systemeigenschaft mit einer statischen mittels einer Relation verbunden, so führt dies im Allgemeinen zu einer Inkompatibilität der beiden Eigenschaften.

Beispiel 2: Eigenschaften des Systems Kaffeemaschine
Jedes technische System hat gewisse das System beschreibende Eigenschaften, mit deren Hilfe sich das System bzw. die Schnittstellen des Systems beschreiben lassen. Dabei wird stets zwischen konstanten und veränderlichen Eigenschaften unterschieden[25]. Das System Kaffeemaschine aus der Abbildung 1.14 hat beispielsweise folgende stark vereinfachte Eigenschaften:

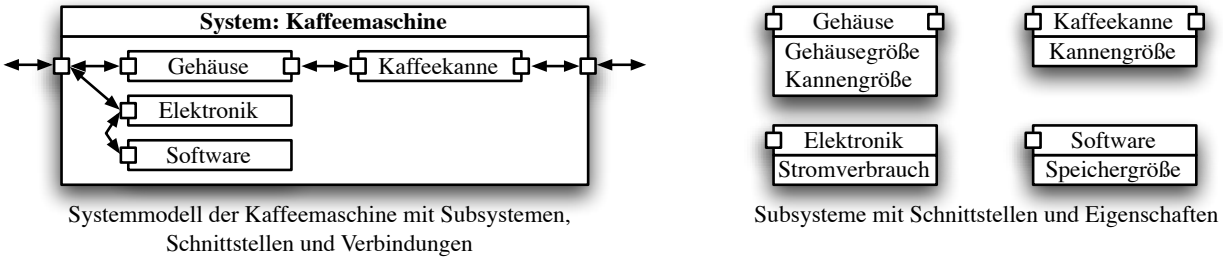


Abbildung 1.14.: Modell des Systems Kaffeemaschine mit Subsystemen, Schnittstellen und Eigenschaften.

- **Konstante Eigenschaften:**
  - Gehäuseform
  - Kannengröße
  - Speichergröße
- **Veränderliche Eigenschaften:**
  - Stromverbrauch

Mittels dieser Eigenschaften kann das System Kaffeemaschine, bestehend aus den vier Subsystemen *Gehäuse, Elektronik, Software* und *Kaffeekanne* beschrieben werden. □

Die exakte Modellierung und Beschreibung der Eigenschaften der unterschiedlichen Schnittstellen eines Systems ist für die Kompatibilitätsmodellierung und -bewertung von entscheidender Bedeutung, da anhand der Eigenschaften überprüft werden kann, ob zum Beispiel die Kommunikation zwischen zwei Systemen kompatibel ist oder nicht. Im obigen Beispiel kann so beispielsweise die maximal mögliche Kannengröße, die in die Kaffeemaschine passt, mit der Größe der zur Verfügung stehenden Kaffeekanne verglichen werden. Ist die maximale Größe der Kaffeekanne kleiner oder gleich der in die Kaffeemaschine passenden Kanne, so ist die Größe der Kanne kompatibel zur Kaffeemaschine.

Aus dem obigen Beispiel folgt unmittelbar, dass für die Beschreibung eines Systems sowohl dessen Ausbau, also die beteiligten Teil- bzw. Subsysteme, deren Schnittstellen, die Systemeigenschaften sowie die Verbindungen zwischen diesen Schnittstellen, die Struktur eines System festlegen. In der Definition 1.8 ist die generische Struktur bzw. der Aufbau eines Systems definiert, als die Menge der Komponenten (Teil- bzw. Subsysteme), die miteinander in Verbindung stehen.

Definition 1.8 ***Struktur eines Systems (nach [FM06] Seite 34)***
Die ***Systemstruktur*** *ist die Menge der Komponenten eines Systems und die Menge der, die Komponenten miteinander verbindenden, Relationen.*

[25] Die Eigenschaften eines Systems werden im Kapitel „2.3.1.2 *Modellierung der Eigenschaften eines Objekts bzw. einer Klasse*“ ab Seite 50 genauer beschrieben.

Die Struktur eines Systems besteht nach obiger Definition aus Teil- bzw. Subsystemen, die mittels Verbindungen (Relationen) miteinander verbunden sind. Genauer: die Teilsysteme sind nach Definition 1.3 auf Seite 12 an ihren dezidierten Schnittstellen mit Hilfe von Verbindungen miteinander verbunden. Das obige Beispielsystem der Kaffeemaschine besteht aus vier Teilsystemen, die an ihren Schnittstellen mittels Relationen miteinander verbunden sind. Nach der Definition 1.8 stellt die Abbildung 1.14 (links) die Struktur des Systems Kaffeemaschine dar.

Zusätzlich zur Struktur eines Systems ist das Verhalten (die Funktionalität) des Systems für das Systemverständnis und die Modellierung des Systems von besonderem Interesse[26]. Um das Verhalten eines Systems beschreiben zu können, sind zunächst die vollständige Struktur des Systems (vgl. Definition: 1.8 auf Seite 15) sowie sämtliche im System enthaltenen Eigenschaften (vgl. Definition: 1.7 auf Seite 15) notwendig um das Verhalten des zu modellierenden Systems beschreiben zu können. Die nachfolgende Definition 1.9 legt fest, dass sich das Verhalten eines Systems als zeitliche Folge von aufeinanderfolgenden Zuständen des Systems beschreiben lässt.

Definition 1.9 ***Verhalten eines Systems (Auszug aus [FM06] ab Seite 34)***
Das ***Systemverhalten*** *ist die Menge der* ***zeitlich aufeinanderfolgenden Zustände*** *eines Systems.*

Besitzt das betrachtete System lediglich statische Eigenschaften, so kann sich das System über die Lebenszeit nicht ändern, aufgrund der Tatsache, dass sich statische Systemeigenschaften laut Definition 1.7 auf der vorherigen Seite nicht ändern dürfen. Das Verhalten des Systems ist somit statischer bzw. konstanter Natur. Hat das System dynamische Systemeigenschaften, also Eigenschaften, die sich über die Lebenszeit des Systems hinweg ändern können, so ist die betrachtete Systemeigenschaft dynamisch und das System hat dynamische Eigenschaften. Bei real existierenden Systemen handelt es sind in der Regel um dynamische Systeme, die jedoch auch statische Eigenschaften aufweisen können. Für die Kompatibilitätsmodellierung und den Kompatibilitätstest ist die Unterscheidung zwischen statischen und dynamischen Systemeigenschaften von besonderer Wichtigkeit, da sich der Kompatibilitätstest dadurch erheblich vereinfachen lässt.

Aufbauend auf der allgemeinen Systemeigenschafts- bzw. der Verhaltensdefinition von Systemen, können diese weiter anhand ihres Verhaltens klassifiziert werden. In der Definition 1.7 auf der vorherigen Seite wurden die Eigenschaften eines Systems definiert. Diese Eigenschaften können sowohl *statisch*, also unveränderlich über die Zeit, als auch *dynamisch* sein. Auch Kombinationen aus beiden Eigenschaftsgruppen sind möglich, so dass es z.B. ein System geben kann, das sowohl statische als auch dynamische Eigenschaften besitzt (vgl. Bsp. auf Seite 15). Die Abbildung 1.15 zeigt die Klassifikation von Systemen anhand ihres zu beobachtenden Verhaltens.

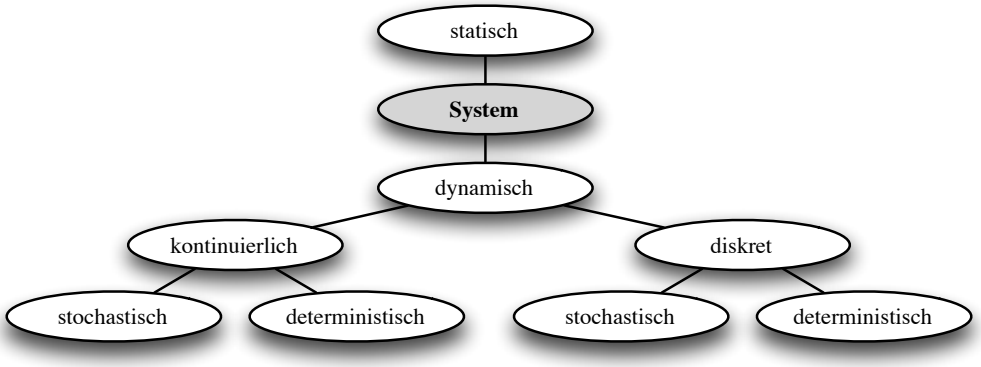


Abbildung 1.15.: Klassifikation von Systemen (nach [FM06, 37]).

Dabei kann das zu beschreibende System bzw. die entsprechende Systemeigenschaft (Bildmitte) entweder *statisch* oder *dynamisch* sein. Nach der Eigenschaftsdefinition 1.7 auf der vorherigen Seite sind jedoch auch beliebige Kombinationen aus statischen und dynamischen Eigenschaften eines Systems möglich. Im Falle einer dynamischen Systemeigenschaft kann diese weiter in die beiden disjunkten Mengen, den so genannten *kontinuierlichen* und *diskreten* Systemeigenschaften unterteilt werden. Unter kontinuierlichen Systemeigenschaften werden solche Eigenschaften verstanden, deren Verhalten sich über die Zeit kontinuierlich, ohne ein fest vorgegebenes Raster verändert. Zwei Beispiele für kontinuierliche Systemeigenschaften wären zum einen die (elektrische) Sinus-Schwingung, und zum anderen die reellen Zahlen der Mathematik $\mathbb{R}$. Im Gegensatz zu den kontinuierlichen Systemeigenschaften können diskreten Systeme, bzw. die diskreten Eigenschaften eines Systems, nur bestimmte Werte innerhalb eines (festen) Rasters einnehmen. Beispielsweise können die natürlichen Zahlen $\mathbb{N}$ nur diskrete Werte annehmen; Zwischenwerte sind nicht vorgesehen.

Sowohl die kontinuierlichen als auch die diskreten Systeme bzw. Systemeigenschaften lassen sich weiter in *stochastische* und *deterministische* Systeme bzw. Systemeigenschaften unterteilen. Dabei kann ein dynamisches, kontinuierliches oder diskretes System bzw. eine Systemeigenschaft entweder stochastisch oder deterministisch sein. Unter stochastischen Systemen bzw. Systemeigenschaften werden solche Systeme verstanden, von denen a priori nicht vorhergesagt werden kann, welchen Zustand sie als nächstes einnehmen werden. Der Folgezustand kann nur mit Hilfe von Wahrscheinlichkeiten angegeben werden. Das System Würfeln oder der Radioaktive Zerfall eines Atomkerns gehört dieser Systemklasse an. Aus einem bekannten Systemzustand kann der Folgezustand nur mit Hilfe einer bestimmten Wahrscheinlichkeit angegeben werden. Die Abbildung 1.16 auf der nächsten Seite zeigt links ein stochastisches System, in dem zwischen den Systemzuständen[27] A und B eine Relation existiert, die mit einer

[26] Siehe hierzu: Kapitel „2.2 *Modellbildung*“ ab Seite 31.
[27] In der Graphentheorie werden die Zustände eines Systems oft auch als *Knoten*, die Relationen zwischen den Knoten als *Transition* bezeichnet.

Wahrscheinlichkeit von 10% genutzt wird ($A \xrightarrow{w=10\%} B$). Zwischen den Knoten A und C, bzw. A und D sind ebenfalls Relationen angegeben, die mit bestimmten Wahrscheinlichkeiten versehen sind ($A \xrightarrow{w=60\%} C$, $A \xrightarrow{w=30\%} D$). Es kann daher a priori nur mit einer bestimmten Wahrscheinlichkeit vorhergesagt werden welche der drei Relationen verwendet wird.

Abbildung 1.16.: Dynamische Systeme – stochastisch vs. deterministisch.

Auf der rechten Seite der Abbildung 1.16 ist das selbe System wie links abgebildet, jedoch wurden hier die drei stochastischen Relationen gegen deterministische ersetzt. Im Gegensatz zu einem stochastischen System ist der nachfolgende Systemzustand eines deterministischen Systems stets a priori bekannt. Aus dem aktuellen Zustand (A) kann der Folgezustand eineindeutig vorhergesagt werden. Im obigen Beispiel wird die Relation zwischen A und B genau dann benutzt, wenn der Wert der Variablen $a = 1$ ist. Ist die Variable $a = 2$, so wird die zweite Relation zwischen dem Knoten A und C benutzt. Ebenso wird die Relation zwischen dem Knoten A und D nur verwendet, wenn der Wert der Variablen $a = 3$ ist. Die meisten technischen Systeme gehören der Klasse der deterministischen Systeme an. Beispiele hierfür sind: Computer, Ampelschaltungen, Getriebe usw..

Nachdem die wesentlichen Systemeigenschaften, ebenso wie die unterschiedlichen Klassen von Systemen, die für die Kompatibilitätsmodellierung und -bestimmung benötigt werden, eingeführt worden sind, folgt nun ein weiterer wichtiger Begriff aus der Systems Engineering Welt, der Begriff der *Sichtbarkeit*. Soll ein komplexes System vollständig modelliert werden, so kann es vorkommen, dass während der Modellierung gewisse Systembestandteile nicht bekannt oder für die Modellierung unwichtig sind. Sind Bestandteile des Systems für die Modellierung unwichtig oder unbekannt, so können diese Teile quasi „ausgeblendet" werden um die Komplexität des Systems zu minimieren bzw. die Übersichtlichkeit zu verbessern. In der Definition 1.10 sind die drei gebräuchlichsten Ansichten eines Systems definiert.

Definition 1.10 ***Ansichten eines Systems***
Im Systems Engineering werden drei Arten von Ansichten auf ein System mit unterschiedlichen Eigenschaften definiert:

- ***Black-Box Ansicht***
  Bei der Black-Box Ansicht auf ein System ist die exakte interne Struktur bzw. der interne Aufbau des Systems für das Verständnis des Systems „uninteressant". Es sind lediglich die äußeren Schnittstellen, sowie das Verhalten des Systems nach außen für dessen Verständnis von Interesse. Die Black-Box Ansicht wird vor allem dann eingesetzt, wenn die innere Struktur eines Systems unbekannt, zu komplex oder nicht von Belang für das Verständnis der Struktur oder Funktionalität des Systems ist.

- ***Glass-Box oder White-Box Ansicht***
  Im Gegensatz zur Black-Box Ansicht, in der die innere Struktur des Systems vollständig verborgen bleibt, ist die interne Struktur bei der Glass-Box Ansicht uneingeschränkt von außerhalb des Systems sichtbar. Die Glass-Box Ansicht wird vor allem dort eingesetzt, wo die interne Struktur des Systems für dessen Erweiterung oder Anwendung von Interesse ist.

- ***Grey-Box Ansicht***
  Die Grey-Box Ansicht stellt einen Kompromiss zwischen den beiden extremen Sichten auf ein System dar. Sie ist eine Kombination aus Black-Box und Glass-Box Ansicht. Hier kann ein wichtiger Systembestandteil vollständig dargestellt sein (Glass-Box), während eine unwichtige Komponente nur als Black-Box dargestellt wird. Die meisten Systeme werden als Grey-Box Modelle dargestellt, da hier ein guter Kompromiss zwischen der Darstellung wichtiger Systembestandteilen und der Abstraktion erreicht wird.

Das *System A*, aus der Abbildung 1.11 auf Seite 12, ist ohne explizit modellierten Inhalt in der so genannten Black-Box Ansicht abgebildet. In dieser Darstellungsvariante wird der komplette Inhalt eines Systems vollständig ausgeblendet, so dass lediglich die Systemgrenze und die Schnittstellen des Systems sichtbar sind. Demgegenüber ist das *System B* mit Inhalt in Glass-Box Ansicht abgebildet. Bei dieser Darstellungsmethode wird der gesamte Inhalt des Systems, also alle Baugruppen, Schnittstellen und Verbindungen zwischen den Baugruppen explizit modelliert und dargestellt. Beide Ansichten auf ein System sind für die Modellierung wichtig, um das Wesentliche vom Unwesentlichen zu trennen. Für die Kompatibilitätsmodellierung und -bewertung ist vor allem die Glass-Box Ansicht wichtig, in der sämtliche Systembestandteile sowie deren Schnittstellen vollständig modelliert und abgebildet sind. Die Black-Box Ansicht ist nur dann für die Kompatibilitätsmodellierung sinnvoll einzusetzen, wenn die Eigenschaften des verborgenen Systems vollständig in dessen Schnittstelle abgebildet sind.

Beispiel 3: Verschiedene Sichten auf ein System/Modell
Die Abbildung 1.17 illustriert die drei möglichen Ansichten des Systems Kaffeemaschine (vgl. Abbildung: 1.14 auf Seite 15). Auf der linken oberen Seite der Abbildung (a) ist die Black-Box Ansicht des Systems Kaffeemaschine dargestellt. Hier ist nur das System und dessen Schnittstellen zur Umwelt hin dargestellt. Der Inhalt des Systems ist vollkommen ausgeblendet.

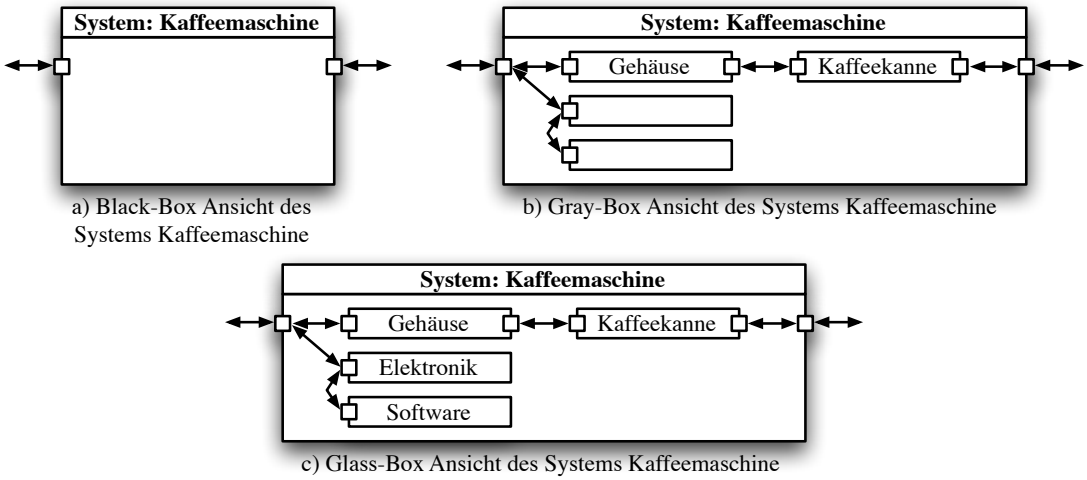


Abbildung 1.17.: Ansichten auf das System Kaffeemaschine.

Auf der rechten oberen Seite (b) ist das gleiche System wie in (a) in der so genannten Grey-Box Ansicht modelliert dargestellt. Das System enthält nun sämtliche Subsysteme sowie die Verbindungen zwischen den einzelnen Systembestandteilen. Jedoch sind nicht alle Subsysteme vollständig dargestellt. In der unteren Bildmitte ist das vollständige System Kaffeemaschine in Glass-Box Ansicht abgebildet, mit allen Subsystemen sowie sämtlichen Verbindungen zwischen den einzelnen Systembestandteilen. □

Nachdem die wichtigsten Begriffe und Definitionen aus dem Systems Engineering Umfeld eingeführt und erläutert wurden, die für das Verständnis von technischen Systemen, und insbesondere für die Modellierung und den Kompatibilitätstest benötigt werden, folgt im nächsten Unterkapitel die Definition des Begriffs Kompatibilität, so wie er im weiteren Verlauf dieser Arbeit verwendet wird.

1.4. Der Kompatibilitätsbegriff

Nachdem im letzten Abschnitt die für das weitere Verständnis dieser Arbeit wesentlichen Begriffe und Definitionen aus dem Systems Engineering Umfeld, mit dem Fokus auf die Beschreibung von technischen Systemen, eingeführt und anhand von einfachen Beispielen erläutert worden sind, wird in diesem Kapitel der zentrale Begriff *Kompatibilität* definiert. Dazu wird zunächst der Kompatibilitätsbegriff aus unterschiedlichen Blickwinkeln beleuchtet und untersucht. Daran anschließend folgt im Unterabschnitt „*Kompatibilitätsarten*“ die Ableitung verschiedener Kompatibilitätsarten aus den beiden zuvor definierten Kompatibilitätsbegriffen. Im Anschluss daran werden die beiden wesentlichen Kompatibilitätsszenarien im Kapitel „*Kompatibilitätsszenarien*“ eingeführt und anhand von Beispielen erläutert.

Als Einstieg in die im weiteren Verlauf dieser Arbeit verwendete Kompatibilitätsbegriffswelt wird zunächst die allgemeinste Definition des Begriffs „Kompatibilität“ vorgestellt. Der Duden definiert den Begriff „Kompatibilität“ wie folgt:

Definition 1.11 ***Allgemeine Kompatibilitätsdefinition nach Duden [Dud05]***
Kompatibilität *die; -, en:*

1. ***Vereinbarkeit*** *(zweier Ämter in einer Person)*
2. ***Austauschbarkeit***, ***Vereinbarkeit verschiedener Systeme*** *(z. B. das Benutzen eines Programms auf einem anderen Computermodell)*
3. *(Sprachwissenschaftlich) syntaktisch-semantische Anschließbarkeit, Kombinierbarkeit von Lexemen (im Satz)*
4. *(Medizinisch) Verträglichkeit verschiedener Medikamente oder Blutgruppen*

Der allgemeine Kompatibilitätsbegriff[28] aus der obigen Definition (Def.: 1.11) enthält im Wesentlichen vier unterschiedliche Teildefinitionen desselben Begriffs. Jede der vier Teildefinitionen ist auf eine spezielle *Domäne* zugeschnitten und kann innerhalb dieser Domäne eingesetzt werden, um die Kompatibilität zwischen zwei Elementen dieser Domäne zu beschreiben. So bedeutet zum Beispiel Kompatibilität innerhalb der Domäne Medizintechnik, dass zwei Medikamente kompatibel sind, wenn ein Medikament gegen ein anderes mit gleicher Wirkung ersetzt werden kann, ohne dass es dabei zu Komplikationen kommt. Innerhalb der Medizin wird der Begriff kompatibel stets im Sinne von Verträglichkeit eingesetzt. In einer anderen Domäne, zum Beispiel der Technik, kann die Kompatibilität jedoch ganz anders definiert sein.

Um eine Domäne weiter zu strukturieren bzw. zu unterteilen kann diese weiter in so genannten *Kontexte* unterteilt werden. Dabei repräsentiert ein Kontext innerhalb einer Domäne eine Verfeinerung bzw. Anpassung des allgemeinen Kompatibilitätsbegriffs an die eingeschränkte Begriffswelt innerhalb dieser einen Domäne. Dabei dürfen sich die Kontexte der Domäne gegenseitig überlappen, wodurch einzelne Begriffe und Definition in mehreren Kontexten gültig sind[29]. Ebenso wie sich Kontexte innerhalb einer Domäne

[28] Weitere Definitionen des Begriffs *Kompatibilität* finden Sie unter [Car], [Sch95], [Bul06], [DAC06], [PDB07] sowie [CKS03].
[29] Mathematisch: Sind D und K Mengen: $\forall x(x \in K \rightarrow x \in D)$, so heißt K Teilmenge von D ($K \subseteq D$). Ein Kontext ist somit eine Teilmenge einer Domäne.

überlappen können, dürfen sich auch ganze Domänen überlappen. Die Abbildung 1.18 zeigt links die generische strukturelle Aufteilung einer Domäne in $1 \ldots n$ (mit $n \in \mathbb{N}$) Kontexte. Auf der rechten Seite der Abbildung ist eine konkrete Domäne, hier die Domäne der eingebetteten Systeme, dargestellt. Diese Domäne wird ihrerseits wiederum in drei Kontexte, *E-Technik*, *Mechanik* und *Software* unterteilt. Jede der drei Kontexte kann ihre eigene Definition des Kompatibilitätsbegriffs verwenden, wodurch es bereits innerhalb der Domäne zu Unstimmigkeiten kommen kann. Aus diesem Grund ist es im Allgemeinen nicht ratsam, den Kompatibilitätsbegriff über Domänen- bzw. Kontextgrenzen hinweg anzuwenden, da es ansonsten leicht zu Missverständnissen bzw. Missinterpretationen kommen kann. So ist zum Beispiel der Kompatibilitätsbegriff, so wie er in der Medizintechnik eingesetzt wird, nicht „kompatibel“ zur Definition im sprachwissenschaftlichen oder Technischen Umfeld.

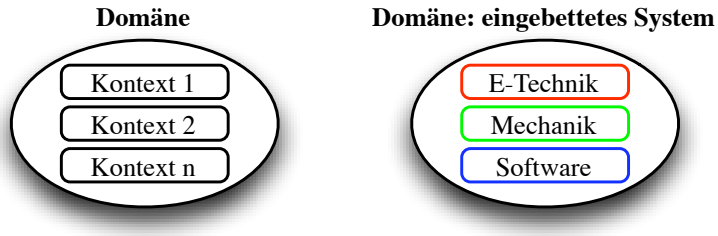


Abbildung 1.18.: Zusammenhang zwischen Domäne und Kontext.

Wie die obige verbale Beschreibung des Kontexts gezeigt hat, ist der Kontext, in dem der Kompatibilitätsbegriff verwendet werden soll, von entscheidender Bedeutung für die Kompatibilitätsmodellierung und -bestimmung. Aus diesem Grund definiert D. Koß in ihrer Dissertation [Koß09] zunächst den Kontext innerhalb dem der Kompatibilitätsbegriff angewendet werden kann.

Definition 1.12 ***Kontext (nach [Koß09])***
*Ein **Kontext** ist eine Menge von **Umgebungseigenschaften**, die relevant für die **Interaktionen** einer Entität (System, Komponente) mit der sie umgebenden Umwelt sind. Der Kontext einer Entität ist im Allgemeinen **frei definierbar**, abhängig von der Sichtweise auf die jeweilige Entität.*

Das folgende Beispiel illustriert den Zusammenhang zwischen dem Kontext und der noch genauer zu definierenden Kompatibilität anhand eines einfachen mechanischen Beispielsystems.

Beispiel 4: Zusammenhang zwischen Kontext und Kompatibilität
Die Abbildung 1.19 zeigt in der Bildmitte ein mechanisches Werkstück mit einem L-förmigen Ausschnitt in der Mitte. Auf der linken Seite der Abbildung ist ein L-Blech (Teil A) dargestellt, das vollständig in die Aussparung des Werkstücks passt. Das L-Blech (Teil B) auf der rechten Seite der Abbildung passt ebenfalls vollständig in die Aussparung des Werkstücks. Ohne den Kontext – die Farbe des Werkstücks in der Bildmitte – zu beachten, passen beide L-Bleche in die Aussparung des Werkstücks – sind also zu diesem kompatibel.

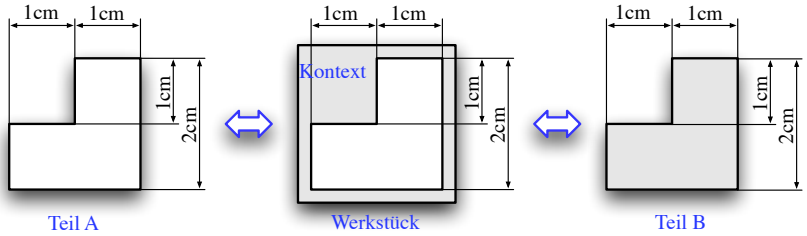


Abbildung 1.19.: Zusammenhang zwischen Kontext und Kompatibilität.

Ist jedoch zusätzlich der Kontext, also nach Definition 1.12 die Menge der *Umgebungseigenschaften* des Werkstücks in der Bildmitte – die graue Farbe – für die Kompatibilitätsbestimmung wichtig, so ist nur noch das L-Blech auf der rechten Seite der Abbildung kompatibel zum Werkstück. Dabei kann der Kontext, nach der Definition 1.12, frei für das Werkstück (Entität) gewählt werden. Im obigen Beispiel wurde die Farbe des Werkstücks zusätzlich zu dessen Abmessungen als kompatibilitätsrelevanter Kontext, also als Menge von kompatibilitätsrelevanten Umgebungseigenschaft des Werkstücks definiert. Das Werkstück könnte jedoch auch weitere Umgebungseigenschaften als seinen kompatibilitätsrelevanten Kontext definieren, wie beispielsweise das Gewicht, die Materialeigenschaft oder beliebige Kombinationen daraus. □

Um diesem „babylonisch“ anmutenden Definitionsdilemma der mannigfaltigen Definitionen des Kompatibilitätsbegriffs zu entkommen, ist es zwingend notwendig, den allgemeinen Kompatibilitätsbegriff aus der Definition 1.11 speziell an die Anforderungen und Bedürfnisse der Domäne „Modellierung von technischen Systemen“, bzw. an die drei darin enthaltenen Kontexte *Elektrotechnik*, *Mechanik* und *Software* anzupassen. Des Weiteren ist es notwendig, dass sich sämtliche Kompatibilitätsbegriffe, aus den drei untergeordneten Kontexten, nahtlos in die übergeordnete Kompatibilitätsbegriffswelt der Domäne einordnen lassen, ohne dass es dabei zu Überschneidungen oder Dissonanzen kommt.

Die beiden nachfolgenden, aus dem allgemeinen Kompatibilitätsbegriff aus der Definition 1.11, abgeleiteten Kompatibilitätsbegriffe wurden von D. Koß speziell an die unterschiedlichen Anforderungen an die Domäne „Kompatibilitätsmodellierung und -bewertung von eingebetteten softwarelastigen Systemen“ sowie die darin enthaltenen drei Kontexte *Elektrotechnik*, *Mechanik* sowie *Software* angepasst. Zusammen bilden sie die Basis des im weiteren Verlauf dieser Arbeit verwendeten Kompatibilitätsbegriffs.

Anwendung des allgemeinen Kompatibilitätsbegriffs auf technische Systeme
Für die Modellierung und Beschreibung von technischen Systemen und insbesondere von eingebetteten softwarelastigen Systemen, werden zwei grundlegende Definitionen des allgemeinen Kompatibilitätsbegriffs aus der Definition 1.11 abgeleitet – *Kompatibilität im Sinne von Verträglichkeit* sowie *Kompatibilität im Sinne von Austausch/Ersetzung*. Auf diese beiden Varianten des allgemeinen Kompatibilitätsbegriffs wird nun genauer eingegangen. Begonnen wird mit der Definition von Kompatibilität im Sinne von Verträglichkeit.

Definition 1.13 ***Kompatibilität im Sinne von Verträglichkeit [Koß09]***
Nach DIN ISO 8402 wird ***Kompatibilität*** *als „****Eignung einer Einheit*** *(eine Einheit kann eine Tätigkeit oder ein Prozess sein ebenso wie ein Produkt, eine Organisation, ein System oder eine Person, oder irgendeine Kombination daraus) unter* ***spezifischen Bedingungen*** *zusammen benutzt zu werden, um* ***relevante Forderungen*** *zu erfüllen“ bezeichnet.*

Kompatibilität im Sinne von Verträglichkeit, angewendet auf technische Systeme, bedeutet, dass zwei (autonome) Systeme/Teilsysteme zueinander kompatibel sind, genau dann wenn sie unter spezifischen Bedingungen (Anforderungen) miteinander interagieren können bzw. die geforderte Funktionalität und das geforderte Verhalten besitzen. Ebenso bedeutet Verträglichkeit, dass zwei Systeme unter spezifizierten Bedingungen miteinander benutzt werden können. Im Sinne von Verträglichkeit kann Kompatibilität jedoch auch bedeuten, dass zwei Eigenschaften in einem System zusammen funktionieren, sich also gegenseitig nicht beeinflussen. Das nachfolgende Beispiel illustriert den oben definierten Begriff der Verträglichkeit anhand eines einfachen Beispiels aus der Domäne Elektrotechnik.

Beispiel 5: Elektromagnetische Verträglichkeit (EMV) von Systemen
Die Elektromagnetische Verträglichkeit [Gmb07][Tec07] beschreibt die Interaktion, genauer die zulässige Beeinflussung, zweier autonomer elektrischer Systeme. Die Abbildung 1.20 zeigt zwei autonome elektrische Systeme, die in der Umwelt enthalten sind. Die beiden Systeme *System A* und *System B* treten dabei sowohl gegenseitig, als auch mit der Umwelt in Interaktion. Die gegenseitigen Beeinflussungen sind dabei von entscheidender Bedeutung für die Funktionalität/Verhalten des jeweiligen Systems.

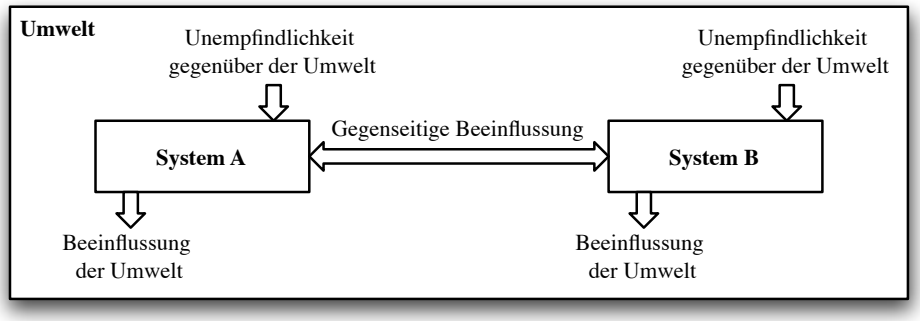


Abbildung 1.20.: Kompatibilität im Sinne von Verträglichkeit.

Die elektromagnetische Verträglichkeit beschreibt, inwieweit sich zwei autonome Systeme gegenseitig beeinflussen dürfen. Außerdem legt die Richtlinie fest, wie hoch die von einem System abgegebene elektromagnetische Strahlung sein darf bzw. wie viel Strahlung ein System von außen aushalten muss, ohne dadurch negativ beeinflusst zu werden. Ein System ist demnach kompatibel, im Sinne von Verträglichkeit, wenn es sämtliche Richtlinien der EMV vollständig erfüllt. □

Im Gegensatz zur Verträglichkeit, beschreibt die *Austauschbarkeit* bzw. die *Ersetzbarkeit* von Teilsystemen, wann ein Teilsystem/Subsystem/Baugruppe eines Systems durch ein anderes Teilsystem/Subsystem/Baugruppe ersetzt/ausgetauscht werden kann, ohne dass die korrekte Funktionsweise dadurch beeinträchtigt wird.

Definition 1.14 ***Kompatibilität im Sinne von Austauschbarkeit/Ersetzbarkeit [Koß09]***
Kompatibilität im Sinne von Austauschbarkeit/Ersetzbarkeit ist die ***Eigenschaft****, in einem System eine (beliebige) Einheit durch eine andere Einheit* ***ersetzen/austauschen*** *zu können,* ***ohne*** *dass die* ***korrekte Funktionsweise*** *(Syntax und Semantik[30]) des Systems* ***beeinträchtigt*** *wird.*

Kompatibilität im Sinne von Austauschbarkeit bzw. Ersetzbarkeit bedeutet, dass in einem System ein Teilsystem (Subsystem, Komponente) durch ein anderes Teilsystem ausgetauscht bzw. ersetzt werden kann, ohne dass die korrekte Funktionsweise des Ausgangssystems dadurch beeinträchtigt wird. Dabei zählt zur korrekten Funktionsweise des Systems sowohl die *Struktur*, also beispielsweise die Anzahl der Anschlüsse, die Größe oder die Form, als auch die *Funktion/Verhalten* des Systems. Insbesondere

[30] Siehe hierzu insbesondere die beiden Definitionen B.5 und B.6 auf Seite 325 des Anhangs.

setzt die Austausch- und Ersetzungskompatibilität *nicht zwingend* die exakt gleiche Funktionsweise (Struktur/Verhalten) des neuen Systems voraus. Es wird lediglich gefordert, dass nach dem Austausch/Ersetzung eines Teilsystems das neu entstandene System die korrekte Funktionsweise des ursprünglichen Systems wiederherstellt. Zusätzlich wird von der Definition 1.14 implizit gefordert, dass das neue Teilsystem nicht nur zum System sondern zum Kontext (Umgebung) des restlichen Systems verträglich und kompatibel ist. Ist die Funktionalität des neuen Systems exakt gleich dem des Ausgangssystems, so spricht man auch von *strikter Kompatibilität*, ansonsten von *nicht strikter Kompatibilität* (vgl. Def.: 1.16 sowie Def.: 1.17). Im nachfolgenden Beispiel wird gezeigt, wie in einem System ein Subsystem durch ein anderes mit veränderter Funktionsweise (Struktur/Verhalten) ersetzt, und das so entstandene neue System auf Kompatibilität hin untersucht werden kann.

Beispiel 6: Kompatibilität im Sinne von austauschbar/ersetzbar
Abbildung 1.21 zeigt auf der linken Seite das Ausgangssystem bestehend aus drei Subsystemen. In diesem System wird das Subsystem *SubSystem 1* gegen das Subsystem *SubSystem 1 neu* ersetzt (Bildmitte). Dabei soll durch den Austausch/Ersetzung des Subsystems *SubSystem 1* durch das Subsystem *SubSystem 1 neu*, die Funktionalität des neuen Systems nicht beeinträchtigt werden. In der Bildmitte ist der Austausch des Subsystems *SubSystem 1* gegen das Subsystem *SubSystem 1 neu* dargestellt. Dabei fällt auf, dass das neue Subsystem (*SubSystem 1 neu*) eine Schnittstelle (Struktur) mehr besitzt, als das ursprüngliche Subsystem (*SubSystem 1*).

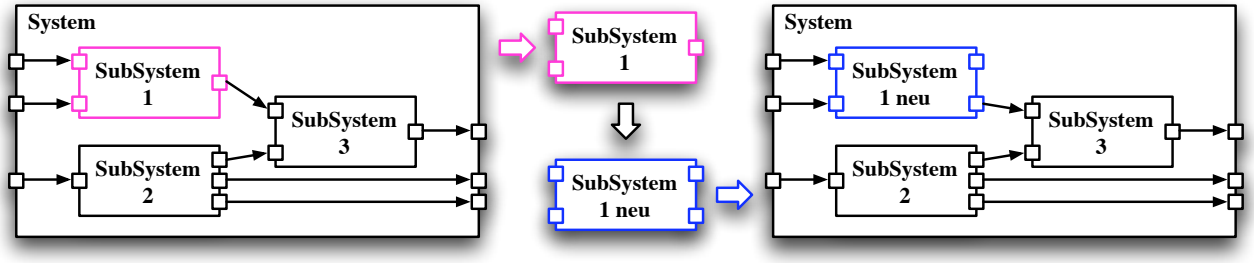


Abbildung 1.21.: Kompatibilität im Sinne von austauschbar/ersetzbar.

Auf der rechten Seite ist das nach dem Ersetzungsvorgang neu entstandene System abgebildet, in dem das modifizierte Subsystem *SubSystem 1 neu* eingebaut worden ist. Obwohl das neue Subsystem eine Schnittstelle mehr aufweist als das ursprüngliche Subsystem darf dadurch die Funktionalität des Systems nicht negativ beeinflusst werden. Das neue Subsystem muss demnach sowohl strukturell, als auch funktional in den Kontext in des Ausgangssystems passen. Es muss demnach mindestens die gleichen Anschlüsse (Schnittstellen) und das gleiche Verhalten aufweisen wie das ursprüngliche Subsystem. Stimmt die Funktionalität des neu entstanden Systems mit der Funktionalität des Ausgangssystem überein, so ist der Austausch des Subsystems kompatibel im Sinne der Definition 1.14.

Ist im Kontext des Systems jedoch festgelegt, dass innerhalb des Systems stets sämtliche Anschlüsse aller enthaltenen Subsysteme angeschlossen sein müssen, so ist die oben dargestellte Ersetzung inkompatibel zum System, aufgrund der Tatsache, dass das neue Subsystem *SubSystem 1 neu* eine Schnittstelle mehr aufweist und somit der Kontext des Systems verletzt wird. Die Abbildung 1.22 verdeutlicht noch einmal den Zusammenhang zwischen dem Kontext und der (Ersetzungs-) Kompatibilität.

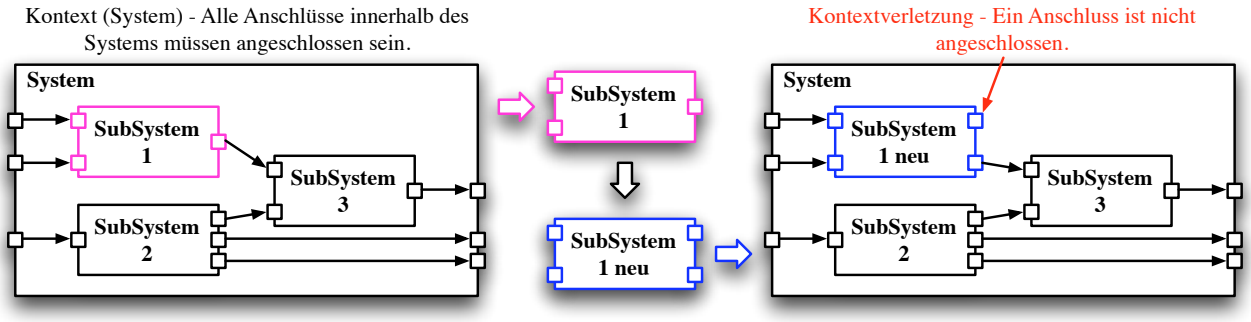


Abbildung 1.22.: Zusammenhang zwischen Ersetzungs-/Austauschkompatibilität und Kontext.

Durch den Kontext, *„Alle Anschlüsse innerhalb des Systems müssen angeschlossen sein.“* des Systems auf der linken Seite der Abbildung, wird verhindert, das die Ersetzung/Austausch des Subsystems *SubSystem 1* gegen das neue Subsystem *SubSystem 1 neu* (Ersetzungs-/Austausch-) kompatibel zum System ist, da nach dem Austausch nicht alle Anschlüsse innerhalb des Systems angeschlossen sind, so wie vom Kontext des Systems gefordert. □

Amerkung:
Im weiteren Verlauf dieser Arbeit, wird stets der Kompatibilitätsbegriff im Sinne der Austauschbarkeit/Ersetzbarkeit aus Definition 1.14 verwendet, außer es wird explizit anders angegeben.

Nachdem nun die beiden zentralen Kompatibilitätsbegriffe eingeführt und anhand von einfachen Beispielen erläutert worden sind, folgt nun, im nachfolgenden Abschnitt *„Kompatibilitätsarten“*, die Beschreibung weiterer wichtiger Kompatibilitätsbegriffe, die speziell für die Modellierung sowie die Bewertung von Kompatibilität von technischen Systemen benötigt werden.

1.4.1. Kompatibilitätsarten

Für die Modellierung, Beschreibung und Bewertung von Kompatibilität innerhalb der Domäne „Modellierung und Bewertung der Kompatibilität von eingebetteten softwarelastigen Systemen“ bzw. der darin enthaltenen drei Kontexte *Elektrotechnik*, *Mechanik* und *Software* reichen die beiden im letzten Abschnitt definierten Kompatibilitätsbegriffe (vgl. Def.: 1.13 und 1.14) bei weitem nicht aus, da sie zu allgemein gehalten sind. Aus diesem Grund werden in diesem Abschnitt weitere Begriffe und Definitionen, aufbauend auf den beiden Grunddefinitionen, eingeführt, die für die Kompatibilitätsbestimmung von technischen Systemen notwendig sind.

Einer der dominierenden Begriffe im Umfeld der Kompatibilitätsmodellierung und -beschreibung sowie der Verifikation von technischen Systemen, ist der Begriff der *Schnittstellenkompatibilität*. Aufbauend auf der allgemein gültigen Schnittstellendefinition aus der Definition 1.3, sowie der Verträglichkeitsdefinition aus Definition 1.13 wird nun der Begriff der Schnittstellenkompatibilität eingeführt.

Definition 1.15 ***Schnittstellenkompatibilität [Koß09]***
Systeme/Teilsysteme/Komponenten sind an einer gemeinsamen Schnittstelle kompatibel, wenn sie ***statisch*** *und* ***dynamisch*** *kompatibel sind*[31].

Dabei ist es laut Definition 1.15 unerheblich, ob es sich um eine Elektrotechnik-, Mechanik- oder Softwareschnittstelle handelt – die Schnittstellenkompatibilität ist demnach domänenunabhängig. Die Schnittstellenkompatibilität fordert lediglich, dass zwei Schnittstellen sowohl statisch als auch dynamisch zueinander passen. Dabei kann die Struktur einer Schnittstelle als *Syntax*, das Verhalten als *Semantik* interpretiert werden[32]. Das nachfolgende Beispiel beschreibt die elektrische Schnittstellenkompatibilität zweier Subsysteme innerhalb eines Systems.

Beispiel 7: Schnittstellenkompatibilität
Das in der Abbildung 1.23 dargestellte System (links) besteht aus drei Subsystemen, die mittels unidirektionaler Verbindungspfeile miteinander an den Schnittstellen verbunden sind. Eine Verbindung ist nur an den dafür vorgesehen ausgezeichneten Schnittstellen der Subsysteme möglich. Auf der rechten Seite der Abbildung sind ohne Beschränkung der Allgemeingültigkeit (o.B.d.A), zwei Subsysteme *SubSystem 1* und *SubSystem 2* aus dem System links herausgegriffen.

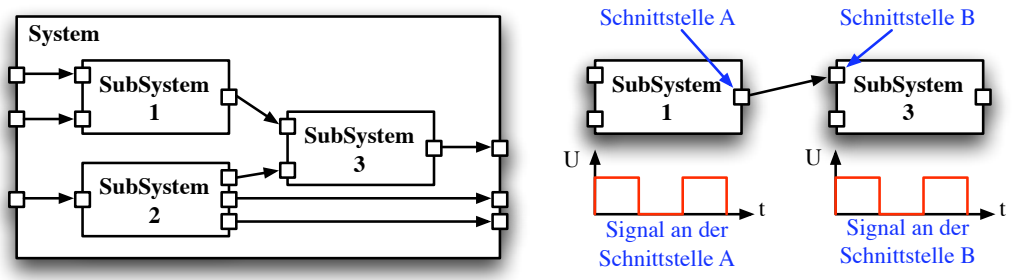


Abbildung 1.23.: Schnittstellenkompatibilität.

Die beiden Subsysteme sind an einer Schnittstellen mittels eines unidirektionalen Verbindungspfeils (gerichteter Flusspfeil/Flussrelation) miteinander verbunden. Über diese Verbindung tauschen die beiden Subsysteme Daten (hier ein elektrisches Rechtecksignal) aus. Wenn das gesendete Signal des Senders (Quelle: Subsystem 1) mit dem vom Empfänger (Senke: Subsystem 3) erwarteten Signal sowohl statisch (unveränderliche Eigenschaften), als auch dynamisch (veränderliche Eigenschaften) übereinstimmt, spricht man von der Kompatibilität der Schnittstellen oder Schnittstellenkompatibilität. Die beiden Signale aus der Abbildung 1.23 (rechts unten) stimmten vollständig (statisch und dynamisch) überein – aus diesem Grund sind die beiden Schnittstellen zueinander kompatibel. □

Aufgrund der generischen Schnittstellen- bzw. der Schnittstellenkompatibilitätsdefinition ist es ebenfalls möglich, zum Beispiel eine elektrische mit einer Softwareschnittstelle zu verbinden, sofern die Verbindung keine der beiden obigen Schnittstellendefinitionen verletzt. Diese ausgezeichnete Schnittstelleneigenschaft ist besonders für die Modellierung eingebetteter softwarelastiger Systeme von Bedeutung, da in einem generischen Systemmodell oft Softwareschnittstellen mit elektrischen Schnittstellen verbunden werden müssen, um den gesamten Modellierungsaufwand reduzieren zu können. Für die Kompatibilitätsbestimmung ist es im Allgemeinen jedoch sinnvoll, die Verbindung von unterschiedlichen Schnittstellenarten zu verbieten, da es sonst zu ungewollten Seiteneffekten, wie zum Beispiel dem „Vergessen“ von notwendigen Verbindungen, kommen kann. So ist zum Beispiel stets eine elektrische Verbindung notwendig auch wenn lediglich ein Softwaresignal zwischen zwei Schnittstellen übertragen werden soll. Wird jedoch nur eine Verbindung modelliert, so kann die andere schnell vergessen werden – eine verstecke Inkompatibilität entsteht.

Weitere Eigenschaften des allgemeinen Kompatibilitätsbegriffs
Die nachfolgenden beiden Definitionen beschäftigen sich speziell mit der Präzisierung des oben eingeführten Kompatibilitätsbegriffs zur Beschreibung der *Austausch- bzw. Ersetzungskompatibilität* (vgl.: Def. 1.14). Sie gelten im Allgemeinen *nicht* für die

[31] Sowohl auf die statische als auch die dynamische Kompatibilität wird in den beiden Definitionen 1.18 und 1.19 genauer eingegangen.
[32] Nähere Informationen zur syntaktischen sowie der semantischen Schnittstelle siehe Definition B.7 auf Seite 326 des Anhangs.

Kompatibilität im Sinne der Verträglichkeit (vgl. Def. 1.13). Die erste Definition schränkt den Kompatibilitätsbegriff weiter ein, während die darauf folgende Definition den Kompatibilitätsbegriff etwas aufweicht und erweitert.

Definition 1.16 ***Strikte Kompatibilität ([Koß09])***
Einheit B ist ***strikt kompatibel*** *zu einem* ***System*** *und einem* ***definierten Kontext****, wenn Einheit A durch Einheit B im System ersetzt wird und B das* ***gleiche Verhalten*** *wie A im System und dem Kontext* ***garantiert****.*

Die strikte Kompatibilität einer Einheit *A* (Komponente/Subsystem) zu einem System *S* und dessen Kontext *K* bedeutet, dass das äußere *beobachtbare Verhalten* des Systems *S vor* und *nach* dem Austausch/Ersetzung der Einheit *A* durch eine andere, o.B.d.A. Einheit *B*, sich nicht verändert, also die neue Einheit *B* dem System *S* und dessen Kontext *K* das *gleiche äußere Verhalten garantiert* wie dies bereits die Einheit *A* getan hat. Dabei wird über das interne Verhalten der Einheit keine Aussage gemacht (vgl. Black-Box Sicht auf ein System). Aus der strikten Kompatibilität folgt jedoch nicht, dass die auszutauschende Komponente identisch der ursprünglich verbauten Komponente sein muss. Beispielsweise kann in einem System eine Komponente *A* durch eine Komponente *B* kompatibel ersetzt/ausgetauscht werden, sofern die Komponente *B* dem System *S* und dessen Kontext *K* das gleiche äußere Verhalten garantiert, wie dies auch die Komponente *A* getan hat. Die neue Komponente *B* kann jedoch ein geändertes internes Verhalten aufweisen und dennoch zum System *S* und dessen Kontext strikt kompatibel sein. Einen Sonderfall der strikten Kompatibilität bildet die Identität.

Satz 1.1 ***Sonderfall der strikten Kompatibilität – Identität***
Wird in einem System S eine Einheit A gegen eine ***identische*** *Einheit B (A ≡ B) ausgetauscht/ersetzt, so ist die neue Einheit* ***strikt kompatibel****, genauer Austausch- bzw. Ersetzungskompatibel, zum System S und dessen Kontext.*

Die strikte Kompatibilität ist die restriktivste Kompatibilitätsart, da sie keine Änderung oder Erweiterung des ursprünglich zu beobachtbaren äußeren Verhaltens des Systems zulässt. Im Gegensatz dazu kann mit Hilfe der nicht strikten Kompatibilität ein System durch den Austausch eines Subsystems sogar erweitert werden.

Definition 1.17 ***Nicht strikte Kompatibilität ([Koß09])***
Einheit B ist ***nicht strikt kompatibel*** *zu einem System, wenn Einheit A durch Einheit B im System ersetzt wird, sich das* ***Verhalten*** *des Systems ändert, dies aber* ***nicht die korrekte*** *(der Spezifikation genügende) Funktionsweise des Systems beeinträchtigt.*

Die nicht strikte Kompatibilität sichert dem System *S* lediglich zu, dass durch den Austausch/Ersetzung der Einheit *A* gegen die neue Einheit *B* mindestens das ursprüngliche, der *Spezifikation des Systems genügende Funktionsweise* (Struktur/Verhalten) nach dem Austausch/Ersetzung der Einheit *A* gegen die neue Einheit *B* wiederhergestellt wird. Dabei muss das Verhalten des neu entstandenen Systems nicht dem des alten entsprechen, es kann demnach erweitert werden, sofern die Erweiterung nicht im Widerspruch zur ursprünglich spezifizierten Funktionalität steht. Des Weiteren darf durch den Austausch/Ersetzung einer Einheit der Kontext des Systems nicht verletzt werden.

Im anschließenden Beispiel werden die beiden Kompatibilitätsarten – strikte und nicht strikte Kompatibilität – anhand eines einfachen Beispielsystems dargestellt und erläutert.

Beispiel 8: Strikte und nicht strikte Kompatibilität
In einem Ausgangssystem soll ein Subsystem aufgrund eines Defekts ausgetauscht werden. Für den Austausch stehen zwei unterschiedliche Varianten des auszutauschenden Subsystems zur Verfügung. Die erste Variante ist strikt kompatibel mit dem auszutauschenden Subsystem, sie besitzt das gleiche äußere Verhalten wie die ursprünglich im System verbaute Variante des Subsystems. Die zweite Variante des Subsystems unterscheidet sich vom ursprünglichen Subsystem durch eine weitere Schnittstelle sowie erweitertem äußeren Verhalten. Die Abbildung 1.24 zeigt auf der linken Seite das Ausgangssystem, in der Mitte das neue System, in dem das defekte Subsystem *SubSystem 1* gegen das neue Subsystem *SubSystem 1 neu A* ersetzt worden ist. Auf der rechten Seite der Abbildung wurde das ursprünglich verwendete Subsystem gegen das Subsystem *SubSystem 1 neu B* ersetzt.

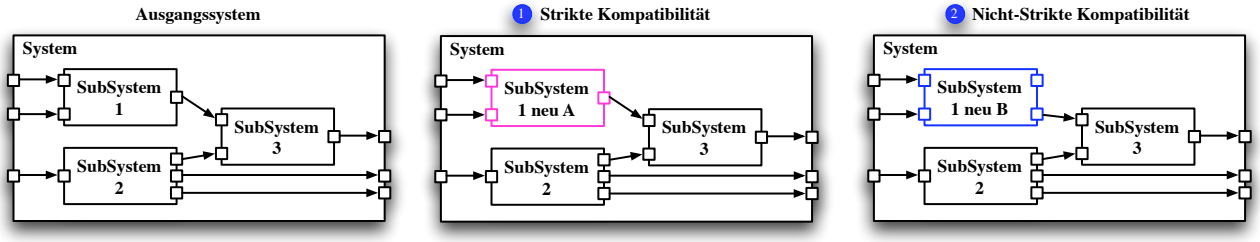


Abbildung 1.24.: Strikte und nicht strikte Kompatibilität.

1. Austausch durch ein *strikt kompatibles* Subsystem:
   Wird das Subsystem *SubSystem 1* gegen das neue strikt kompatible Subsystem *SubSystem 1 neu A* ersetzt/ausgetauscht, so müssen zwei Fälle unterschieden werden.

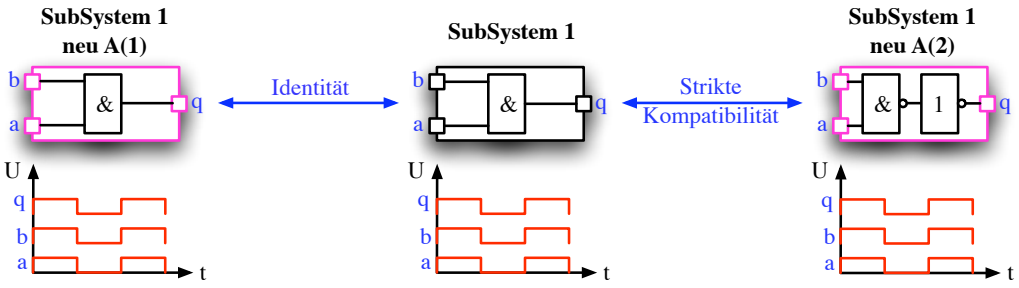


Abbildung 1.25.: Zusammenhang zwischen strikter Kompatibilität und Identität.

- Fall A: Austausch/Ersetzung mit einem identischen Subsystem
  Wird ein Subsystem A eines Systems S mit einem identischen Subsystem B ($A \equiv B$) ersetzt bzw. ausgetauscht, so entspricht dies dem Sonderfall der *strikten Kompatibilität* (vgl. Satz 1.1). In diesem Fall sind keine Kompatibilitätsprobleme zu erwarten, da beide Subsystem identisch, also gleich sind. Die Abbildung 1.25 zeigt links das neue Subsystem *SubSystem 1 neu A(1)*. Dieses Subsystem ist identisch dem ursprünglichen verbauten Subsystem *SubSystem 1*. Sowohl das beobachtbare äußere und innere Verhalten, als auch die innere und äußere Struktur sind absolut identisch.
- Fall B: Austausch/Ersetzung mit einem strikt kompatiblen Subsystem
  Das Subsystem *SubSystem 1 neu A(2)* ist nach Definition 1.16 strikt kompatibel zum System sowie dessen Kontext, weil das neu verbaute Subsystem *SubSystem 1 neu A(2)* sowohl dem System als auch dem Systemkontext das gleiche äußere Verhalten zusichert (garantiert), wie dies bereits das ursprüngliche Subsystem getan hat. Für die strikte Kompatibilitätsbetrachtung wird weder das interne Verhalten noch die interne Struktur des neuen Subsystems benötigt noch für die Kompatibilitätsüberprüfung herangezogen. In der Abbildung 1.25 unterscheidet sich der innere Aufbau des neuen Subsystems *SubSystem 1 neu A(2)* erheblich vom inneren Aufbau des ursprünglich verbauten Subsystems *SubSystem 1*, jedoch das äußere Verhalten ist gleich. Demnach sind die beiden Subsysteme strikt kompatibel zum System und dessen Kontext.

2. Austausch durch ein *nicht strikt kompatibles* Subsystem:
   Das neue Subsystem *SubSystem 1 neu B* unterscheidet sich vom ursprünglich verbauten Subsystem sowohl im beobachtbaren äußeren Verhalten als auch in der äußeren Struktur. Die Abbildung 1.26 zeigt rechts das neue Subsystem *SubSystem 1 neu B*. Das neue Subsystem hat gegenüber dem ursprünglichen Subsystem (links) ein erweitertes äußeres Verhalten und besitzt eine Schnittstelle mehr. Über diese Schnittstelle gibt das Subsystem ein zusätzliches Signal aus. Dieses wird jedoch im System nicht benötigt und beeinflusst das Systemverhalten nicht. Das „restliche Subsystem“ unterscheidet sich weder vom beobachtbaren Verhalten noch von der Struktur vom ursprünglichen Subsystem *SubSystem 1* und genügt weiterhin der ursprünglichen Spezifikation. Insgesamt genügt das neue Subsystem *SubSystem 1 neu B* der ursprünglichen Spezifikation, aufgrund der Tatsache, dass der neu hinzugekommene Anschluss sowie das zusätzliche Verhalten, das System nicht beeinflussen. Aus diesem Grund ist das neue Subsystem nicht strikt kompatibel zum System und dessen Kontext.

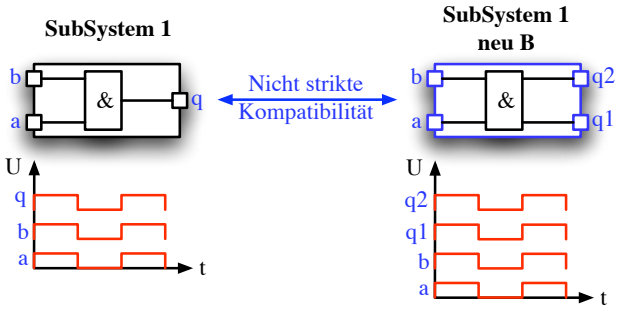


Abbildung 1.26.: Nicht strikte Kompatibilität.

□

Für die Modellierung und die Beschreibung von technischen Systemen sind zwei weitere Kompatibilitätsarten von entscheidender Bedeutung. Zum einen die statische Kompatibilität und zum anderen die dynamische Kompatibilität.

Statische und dynamische Kompatibilität
Jedes technische System kann sowohl statische als auch dynamische Eigenschaften besitzen. So hat zum Beispiel das technische System Kaffeemaschine aus Abbildung 1.14 auf Seite 15 die statischen Eigenschaften *Gehäuseform*, *Kannengröße* und *Speichergröße* sowie die dynamische Eigenschaft *Stromverbrauch*. Um diese Systemeigenschaften auf Kompatibilität mit einer anderen Kaffeemaschine bzw. beim Austausch von Subsystemen der Kaffeemaschine gegen neue untersuchen zu können, müssen die beiden oben eingeführten Kompatibilitätsbegriffe (vgl. Def.: 1.13 und 1.14) auf die beiden Anwendungsfelder – statische und dynamische

Kompatibilität – hin angepasst und verfeinert werden. Die nachfolgende Definition beschreibt zunächst den Begriff der statischen Kompatibilität als die Unveränderlichkeit der Werte einer Eigenschaften des Systems über die gesamte Lebenszeit/Laufzeit des betrachteten Systems hinweg[33].

Definition 1.18 ***Statische Kompatibilität [Koß09]***
Statische Kompatibilität *ist die Verträglichkeit der Schnittstellen der Komponenten zueinander in einem System, ohne dass das* ***System ausgeführt werden muss****.*

Die statische Kompatibilität stützt sich vollständig auf der Verträglichkeitsdefinition aus Definition 1.13 auf Seite 20 ab. Erst wenn die Verträglichkeit zweier Eigenschaften gegeben ist, kann eine Aussage über die statische Kompatibilität dieser Eigenschaften getroffen werden. Sind also zwei Eigenschaften kompatibel im Sinne von Verträglichkeit, so beschreibt die statische Kompatibilität die Unveränderlichkeit der Werte der Eigenschaften über die gesamte Laufzeit/Lebenszeit des Systems hinweg. Daraus folgt unmittelbar, dass der betrachtete Wert der Eigenschaft sich über die Laufzeit/Lebenszeit des Systems nie ändert, also statisch ist. Dabei kann die allgemein gültige, generische Definition der statischen Kompatibilität auf unterschiedliche Domänen wie zum Beispiel auf die Elektrotechnik, Mechanik oder die Softwaretechnik, unverändert angewendet werden. Im Fall der Kaffeemaschine sind sowohl die Gehäuseform, die Kannengröße, als auch die Speichergröße statische Eigenschaften der Kaffeemaschine und können mit Hilfe der statischen Kompatibilitätsdefinition beschrieben und auf Kompatibilität (genauer: statischer Kompatibilität) hin untersucht werden. Beispielsweise hat die Eigenschaft Kannengröße ($S_{x,y,z} = (5, 8, 3)$) oder der Speicherverbrauch ($S = 50$) des Systems Kaffeemaschine einen feste Wert der sich während der gesamten Lebenszeit des Systems niemals ändert.

Hat ein System/Teilsystem zusätzlich zu seinen statischen Eigenschaften dynamische, also solche, die sich über die Zeit verändern, so kann die dynamische Kompatibilität mit Hilfe der nachfolgenden Definition beschrieben werden. Dabei gilt jedoch insbesondere: sind zum Beispiel zwei Eigenschaften statisch inkompatibel, so sind sie ebenfalls dynamisch inkompatibel zueinander. Ist beispielsweise die elektrische Verbindung zwischen dem Stromnetz und der Kaffeemaschine nicht vorhanden oder falsch angeschlossen, so ist die Verbindung ebenfalls dynamisch inkompatibel, da über die Verbindung nicht die notwendigen Signale, die für die korrekte Funktionalität der Maschine notwendig sind, fließen können.

Definition 1.19 ***Dynamische Kompatibilität [Koß09]***
Dynamische Kompatibilität *ist die Verträglichkeit der Komponenten und ihrer Schnittstellen zueinander zur* ***Laufzeit*** *in einem System.*

Für die dynamische Kompatibilitätsbestimmung ist die statische Kompatibilität eine zwingende Voraussetzung. Erst wenn sie gegeben ist, kann das System auf dynamische Kompatibilität hin untersucht werden[34]. Für die dynamische Kompatibilitätsbestimmung ist laut Definition 1.19 die Ausführung des Systems bzw. des Modells des Systems zwingende Voraussetzung, denn nur wenn das System/Modell tatsächlich ausgeführt wird, können dynamische Kompatibilitätsprobleme, wie beispielsweise ein durch Abnutzung eines Lagers zu groß werdendes Spiel, auftreten. Das nachfolgende Beispiel illustriert den oben beschriebenen Zusammenhang zwischen der statischen und der dynamischen Kompatibilität anhand eines einfachen Modells einer Kaffeemaschine.

Beispiel 9: Statische und dynamische Kompatibilität
Gegeben ist das Modell einer einfachen Kaffeemaschine, ähnlich wie in Abbildung 1.14 auf Seite 15, jedoch mit einem zusätzlichen externen Stromanschluss. Die Abbildung 1.27 zeigt das stark vereinfachte Modell einer Kaffeemaschine mit dem externen Stromanschluss *Steckdose* sowie der internen Komponente *Anschlussbuchse* der Kaffeemaschine, die mittels einer Verbindung mit der Steckdose sowohl mechanisch als auch elektrisch verbunden ist.

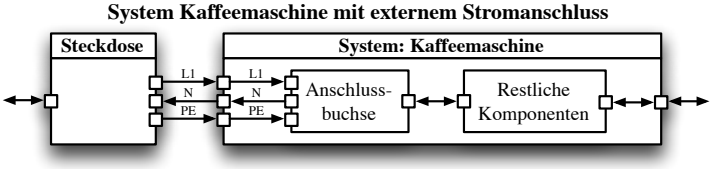
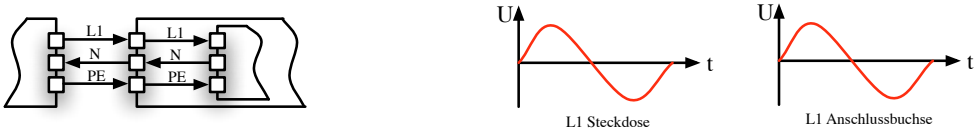


Abbildung 1.27.: Statische und dynamische Kompatibilität einer Kaffeemaschine.

Die Abbildung 1.28 auf der nächsten Seite zeigt einen Ausschnitt aus dem oben dargestellten Systemmodell einer Steckdose sowie der damit verbundenen Kaffeemaschine (Abb.: 1.27), in dem sowohl die mechanische als auch die elektrische Verbindung (Schnittstelle) zwischen der Steckdose auf der einen Seite, und der internen Anschlussbuchse der Kaffeemaschine auf der anderen, statisch und dynamisch kompatibel zueinander sind (vgl. Schnittstellenkompatibilität aus Definition 1.15 auf Seite 22). Im statischen Fall (links) stimmen sämtliche mechanischen und elektrischen Anschlüsse (L1, N, PE) zwischen der Steckdose und der Anschlussbuchse überein. Im dynamischen Fall (rechts) sind sämtliche elektrischen Signale zwischen der Steckdose und der Anschlussbuchse innerhalb des Kaffeemaschine identisch. Dass sämtliche Signale tatsächlich, wie für die Kompatibilität gefordert, identisch, also kompatibel sind, kann erst durch die Ausführung des Systems/Modells verifiziert werden. Durch die „Ausführung" des dargestellten Beispielsystems wird deutlich, dass alle dynamischen Kompatibilitätsanforderungen erfüllt sind. Daraus folgt:

[33] Anmerkung:
Dabei kann sowohl die statische, als auch die dynamische Kompatibilität auf die syntaktische und semantische Kompatibilität zurückgeführt werden. Siehe hierzu [PDB07].
[34] Laut [PDB07] ist die syntaktische Korrektheit eine zwingende Voraussetzung für die semantische Korrektheit.

Die modellierte Verbindung zwischen der Kaffeemaschine sowie der Steckdose aus der Abbildung 1.28 ist sowohl statisch als auch dynamisch kompatibel.

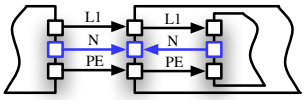


Statische Kompatibilität —
sämtliche Anschlussbelegungen stimmen überein

Dynamische Kompatibilität —
sämtliche zu übertragenden Signalformen zwischen Steckdose und Anschlussbuchse stimmen überein

Abbildung 1.28.: Steckdose und Anschlussbuchse sind statische und dynamisch kompatibel.

Die Abbildung 1.29 zeigt links den gleichen Ausschnitt aus dem obigen Systemmodell. In diesem Teilmodell ist der Anschluss (Schnittstelle) zwischen der Steckdose auf der einen Seite und der Anschlussbuchse der Kaffeemaschine auf der anderen Seite zwar mechanisch kompatibel, da sämtliche Anschlüsse mechanisch verbunden/angeschlossen sind, jedoch elektrisch inkompatibel (Semantik). Das elektrische Signal *N* fließt nun nicht mehr wie vom Anschluss der Kaffeemaschine erwartet, aus dem Anschluss hinaus, sondern wird von der Steckdose geliefert. Das von der Kaffeemaschine erwartete Signal ist jetzt um 180° gegenüber dem, das die Steckdose liefert gedreht. Somit ist die statische Kompatibilität der Steckdose mit der Kaffeemaschine nicht mehr gegeben. Aus diesem Grund ist ein dynamischer Kompatibilitätstest überflüssig; das elektrische Signale zwischen dem Sender (Steckdose) und dem Empfänger (Kaffeemaschine) nicht mehr übereinstimmen und somit inkompatibel sind.



Statische Inkompatibilität —
Anschlussbelegung stimmt NICHT überein

Statische Inkompatibilität --> dynamische Kompatibilität kann nicht geprüft werden!

Abbildung 1.29.: Steckdose und Anschlussbuchse sind statische inkompatibel zueinander.

□

Nachdem nun die für diese Arbeit wesentlichen Kompatibilitätsbegriffe und Definitionen eingeführt und anhand von einfachen Beispielen aus unterschiedlichen Domänen erläutert wurden[35], folgt abschließend die Beschreibung der beiden wesentlichen Kompatibilitätsszenarien, so wie sie bei der Kompatibilitätsbewertung von eingebetteten softwarelastigen Systemen am häufigsten anzutreffen sind.

1.4.2. Kompatibilitätsszenarien

In den vorangegangenen Abschnitten dieses Kapitels wurde zunächst der allgemeine und im Anschluss daran der spezifische Kompatibilitätsbegriff eingeführt und anhand von einfachen Beispielen aus unterschiedlichen Domänen erläutert. In diesem Abschnitt werden – ausgehend von den zuvor eingeführten Begriffen und Definitionen aus dem Kompatibilitätsumfeld – zwei Szenarien eingeführt, die den „Alltag“ bei der Kompatibilitätsmodellierung bzw. bei der Verifikation von Kompatibilität von technischen Systemen widerspiegeln. Beide Szenarien werden anhand eines einfachen Beispiels aus dem Umfeld von eingebetteten softwarelastigen Systemen erläutert.

Als Beispielsystem dient wieder das bereits im Abschnitt *„Grundlegende Begriffe und Definitionen aus der Systems Engineering Welt“* eingeführte System einer einfachen Kaffeemaschine. Anhand dieses Beispielsystems werden zwei unterschiedliche Kompatibilitätsszenarien eingeführt und erläutert. Begonnen wird mit den Szenario 1, in dem die Kaffeemaschine verträglich zu der sie umgebenden Umwelt sein muss, um als kompatibel zu gelten.

- **Szenario 1**: Kompatibilität im Sinne von Verträglichkeit (vgl. Definition: 1.13 auf Seite 20)
  Wenn eine neue Kaffeemaschine entwickelt werden soll, so muss sie laut EMV-Norm verträglich mit der sie umgebenden Umwelt, also anderen Geräten sein. Das heißt, das System Kaffeemaschine darf kein anderes System in irgendeiner Art und Weise negativ beeinflussen. Dieses Szenario entspricht im Wesentlichen der Kompatibilitätsdefinition 1.13, in der die Kompatibilität im Sinne von Verträglichkeit definiert worden ist. Die Abbildung 1.30 auf der nächsten Seite zeigt in der Bildmitte das Black-Box Modell einer Kaffeemaschine mit jeweils zwei Schnittstellen nach außen. Die oberste Schnittstelle beschreibt die elektromagnetische Verträglichkeit der Kaffeemaschine, in der unteren Schnittstelle sind sämtliche übrigen Eigenschaften der Kaffeemaschine hinterlegt. Diese sind für das Beispiel jedoch nicht relevant und werden aus diesem Grund nicht weiter beleuchtet.

[35] Im Anhang, unter „B.1 *Kompatibilitätsarten*“ ab Seite 325 und „B.2 *Weiterführende Begriffe und Definitionen*“ ab Seite 325 finden Sie weiterführende Kompatibilitätsbegriffe und Definitionen. Die hier aufgeführten Begriffe und Definitionen sind ergänzend zu den in diesem Kapitel vorgestellten Kompatibilitätsdefinitionen.

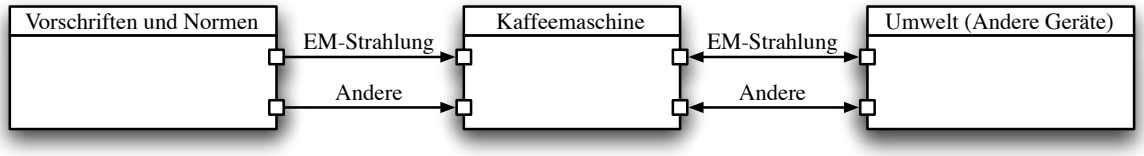


Abbildung 1.30.: Umweltmodell einer Kaffeemaschine.

Die Kaffeemaschine ist auf der linken Seite mittels eines unidirektionalen Flusspfeils mit den unterschiedlichen Vorschriften und Normen verbunden, in denen z.B. die gesetzlichen Vorgaben zur elektromagnetischen Verträglichkeit eines Systems geregelt sind. Diese Vorschriften und Normen muss das System Kaffeemaschine mindestens erfüllen um als kompatibel mit den Vorschriften und Normen zu gelten. Auf der rechten Seite der Abbildung ist stellvertretend für alle weiteren Geräte die Umwelt dargestellt, die diese Geräte enthält. Die Umwelt ist mit der Kaffeemaschine mittels des beidseitig gerichteten Flusspfeils *EM-Strahlung* verbunden. Der beidseitig gerichtete Flusspfeil *EM-Strahlung* zeigt an, dass sowohl die Kaffeemaschine elektromagnetische Strahlung abgibt, als auch dass sie von der Umwelt (anderen Geräten) mit EM-Strahlung bestrahlt wird.

Das System Kaffeemaschine ist dann kompatibel im Sinne von Verträglichkeit, wenn sie zum einen sämtliche Vorgaben und Normen, und zum anderen die elektromagnetische Verträglichkeitsprüfung erfolgreich besteht.

- **Szenario 2**: Kompatibilität im Sinne von austauschbar/ersetzbar (vgl. Definition: 1.14 auf Seite 20)
  Im ersten Szenario wurde die elektromagnetische Verträglichkeit der Kaffeemaschine untersucht und bewertet. Im zweiten Szenario wird nun ein Bestandteil, ein Sub- bzw. Teilsystem, der Kaffeemaschine gegen ein neueres Subsystem aufgrund eines Defekts des alten ersetzt. Nach dem Austausch muss die Kompatibilität des neuen Subsystems mit dem Ausgangssystem der Kaffeemaschine überprüft und bewertet werden. Dies entspricht der zuvor eingeführten Ersetzungskompatibilität aus Definition 1.14.

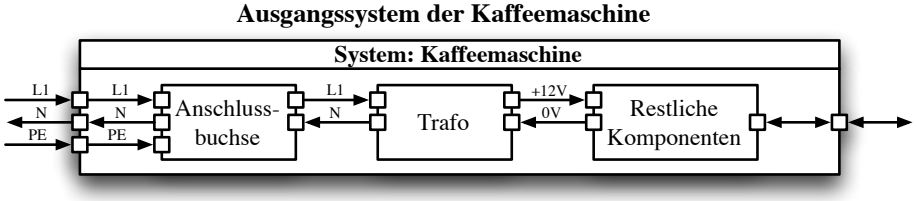


Abbildung 1.31.: Ausgangssystem – Kaffeemaschine.

Die Abbildung 1.31 zeigt das stark vereinfachte Modell einer Kaffeemaschine. In diesem Modell besteht die Kaffeemaschine im Wesentlichen aus drei Blöcken, der *Anschlussbuchse* für den Stromstecker, einem *Trafo*, der die ankommende Wechselspannung aus dem Hausnetz in eine Gleichspannung von $+12\mathrm{V}$ transformiert sowie den nicht näher spezifizierten *restlichen Komponenten* des Systems.

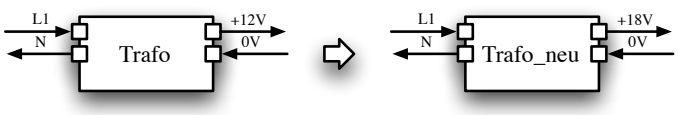


Abbildung 1.32.: Austausch eines defekten Subsystems (Trafo) aus der Kaffeemaschine gegen ein neues.

In diesem Modell der Kaffeemaschine soll nun, im nächsten Schritt, der defekte Trafo gegen einen neuen ersetzt werden. Der neue Trafo liefert jedoch sekundärseitig eine um $6\mathrm{V}$ größere Gleichspannung als der ursprünglich im System verbaute Trafo. Die Abbildung 1.32 zeigt den Ersetzungsvorgang des ursprünglichen Trafos durch den neuen Trafo. Das neu entstandene System muss nun auf Kompatibilität untersucht werden.

Die Abbildung 1.33 zeigt das neu entstandene System der Kaffeemaschine mit dem neuen Trafo *Trafo_neu*. Dieses System soll nun auf Kompatibilität (genauer Ersetzungskompatibilität nach Definition 1.14) hin untersucht und bewertet werden. Jedoch bevor mit der Kompatibilitätsuntersuchung begonnen werden kann, muss spezifiziert werden, ob das restliche System der Kaffeemaschine mit der erhöhten Spannung zurecht kommt oder nicht. Dazu müssen zwei Fälle unterschieden werden:

  - **Strikte Kompatibilität:** (vgl. Definition: 1.16 auf Seite 23)
    Im Fall der strikten Kompatibilitätsforderung ist das neu entstandene System nicht kompatibel zum restlichen System, da laut Definition der strikten Kompatibilität das beobachtbare äußere Verhalten des neue Teilsystems *Trafo_neu* nicht

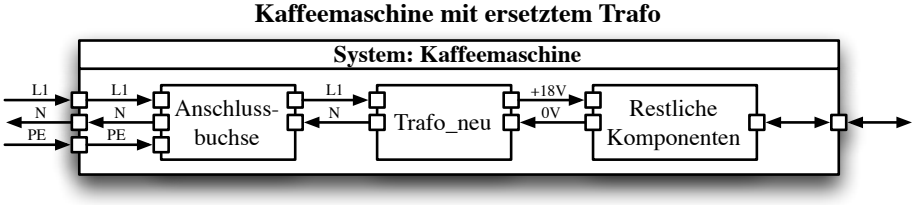


Abbildung 1.33.: Kaffeemaschine mit neuem Trafo.

vom äußeren beobachtbaren Verhalten des ursprünglichen Trafo sowie dem Kontext des Systems abweichen darf. Der neue Trafo besitzt jedoch einen um $6\mathrm{V}$ höheren Ausgangspegel und verletzt somit die strikte Kompatibilität.

– **Nicht strikte Kompatibilität:** (vgl. Definition: 1.17 auf Seite 23)
Im Fall der nicht strikten Kompatibilität muss das restliche System (Kontext) untersucht werden, ob es mit der erhöhten Spannung des Transformators zurecht kommt oder nicht. Liegt die um $6\mathrm{V}$ höhere Ausgangsspannung des neuen Trafos innerhalb der Toleranzen des restlichen Systems, so ist die Ersetzung kompatibel, ansonsten nicht.

Die nicht strikte Kompatibilität ist bei der Instandsetzung von realen Systemen der am häufigsten anzutreffende Kompatibilitätsfall. Dies liegt zum einen daran, dass aufgrund der rasanten technischen Entwicklung moderner Systeme bereits nach kurzer Zeit keine strikt kompatiblen Ersatzteile mehr lieferbar sind, und zum anderen daran dass die meisten Komponenten eines Systems keine Neuentwicklungen, sondern Varianten des ursprünglichen Systems sind. Diese werden jedoch häufig mit veränderter Funktionalität erstellt. In den seltensten Fällen kann eine defekte Baugruppe eines Systems gegen eine identische oder strikt kompatible Variante ersetzt werden. Aus diesem Grund ist die nicht strikte Kompatibilität die in der technischen Anwendung am meisten anzutreffende Kompatibilitätsvariante.

Anmerkung:
Das zweite Szenario kann nicht nur für die Modellierung und Bewertung von Ersetzungskompatibilität von bereits existierenden Systemen eingesetzt werden, sondern lässt sich auch während der Neuentwicklung von komplexen Systemen einsetzen um das neue System möglichst „kompatibel entwickeln“ zu können.

1.4.3. Kompatibilitätsregelwerk

Wie die obigen beiden Kompatibilitätsszenarien gezeigt haben, ist für die Kompatibilitätsbestimmung eines technischen Systems, zusätzlich zu den Kompatibilitätsbegriffen ein „passendes“, auf den allgemein gültigen Kompatibilitätsbegriffen aufbauendes Regelwerk notwendig, in dem genau spezifiziert ist, was in einem System als kompatibel zu bewerten ist und was nicht. Dabei muss das Kompatibilitätsregelwerk sämtliche Regeln aus allen an der Systementwicklung beteiligten Domänen enthalten, um zu gewährleisten, dass das gesamte System auf Kompatibilität und insbesondere auf Kompatibilität zwischen den Domänenschnittstellen, überprüft werden kann. Das Kompatibilitätsregelwerk ist demnach eine Ansammlung domänenspezifischer Kompatibilitätsregeln, mit deren Hilfe die Kompatibilität der unterschiedlichen Schnittstellen zwischen einzelnen Teilsystems eines Systems überprüft und bewerten werden kann.

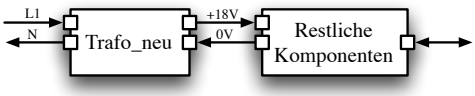
Definition 1.20 ***Kompatibilitätsregelwerk***
Im ***Kompatibilitätsregelwerk*** *sind sämtliche Kompatibilitätsregeln aller an der Systementwicklung beteiligter Domänen hinterlegt. Mit Hilfe dieser Kompatibilitätsregeln kann das zu untersuchende System, genauer die miteinander verbundenen* ***Schnittstellen*** *zwischen den beteiligten Teilsystemen des Systems, domänenübergreifend auf Kompatibilität hin untersucht und bewertet werden.*

Die sehr allgemein gehaltene Definition 1.20 beschreibt das Kompatibilitätsregelwerk als eine Ansammlung von domänenspezifischen Kompatibilitätsregeln, die auf den allgemeinen Kompatibilitätsbegriffen beruhen. Insbesondere wird in der obigen Definition die ausgezeichnete Rolle der Schnittstelle für die Kompatibilitätsbewertung hervorgehoben, da die Kompatibilität eines Systems nur an der Schnittstelle zwischen miteinander verbundenen (Teil-) Systeme untersucht und bewertet werden kann. Aus diesem Grund müssen sämtliche Kompatibilitätsregeln stets so spezifiziert werden, dass sie die Eigenschaften der entsprechenden Schnittstelle exakt beschreiben. Aufgrund der Tatsache, dass Systeme sowohl statische als auch dynamische Eigenschaften haben können, muss auch das Kompatibilitätsregelwerk beide Bereiche abdecken.

Definition 1.21 ***Eigenschaften des Kompatibilitätsregelwerks***
Das ***Kompatibilitätsregelwerk*** *enthält sowohl* ***statische*** *als auch* ***dynamische Kompatibilitätsregeln*** *aus den unterschiedlichen Domänen.*

Beide Definitionen werden nun anhand eines einfachen Beispielsystems genauer erläutert. Die Abbildung 1.34 auf der nächsten Seite zeigt in der Bildmitte einen Auszug aus dem System Kaffeemaschine (vgl. Abbildung: 1.31 auf Seite 27).

Auf der linken Seite der Abbildung ist die Beschreibung der statischen Schnittstelle des Systems abgebildet, so wie sie z.B. im Datenblatt des Herstellers des Teilsystems enthalten ist. Aus dem Datenblatt des Teilsystems *Trafo_neu* sind die beiden statischen Werte für die Versorgungsspannung (Primärseite) des Trafos $L1 = +220\mathrm{V}, N = 0\mathrm{V}$ sowie die beiden Ausgangsspannungen



Statische Eigenschaften der Schnittstelle

Datenblatt: Trafo_neu

Primärseite:
- Eingang: L1 = +220V
- Ausgang: N = 0V

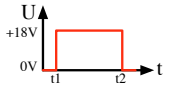
Sekundärseite:
- Eingang: M = 0V
- Ausgang: UB = +18V

Datenblatt: Restliche Komponenten

- Eingang: UB = +12 bis +18V
- Ausgang: M = 0V

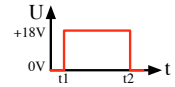
Dynamische Eigenschaften der Schnittstelle

Datenblatt: Trafo_neu



t1 = 1s
t2 = 6s

Datenblatt: Restliche Komponenten



t1 = 1s
t2 = 6s

Abbildung 1.34.: Kaffeemaschine mit neuem Trafo.

(Sekundärseite) des Trafos $U_B = +18\text{V}, M = 0\text{V}$ zu entnehmen. Ebenfalls aus dem Datenblatt des Teilsystems *Restliche Komponenten* stammen die beiden statischen Kennwerte $U_B = +12 \ldots 18\text{V}$ und $M = 0\text{V}$. Damit das Teilsystem *Trafo_neu strikt kompatibel* zum Teilsystem *Restliche Komponenten* bzw. zum alten Transformator *Trafo* ist, müssen sämtliche spezifizierten Werte der beiden Transformatoren sowie die Sekundärwerte der beiden Transformatoren mit den Eingangswerten der Komponente *Restliche Komponenten* vollständig übereinstimmen. Es müssen also die beiden folgende Kompatibilitätsregel (A und B) gelten, damit der Austausch der Komponente *Trafo* gegen die neue Komponente *Trafo_neu* strikt kompatibel ist:

$$\text{A}: \quad L1_{Trafo} = L1_{Trafo_{neu}} \wedge N_{Trafo} = N_{Trafo_{neu}} \wedge M_{Trafo} = M_{Trafo_{neu}} \wedge U_{B_{Trafo}} = U_{B_{Trafo_{neu}}}$$

und

$$\text{B}: \quad M_{Trafo_{neu}} = M_{RestlicheKomponenten} \wedge U_{B_{Trafo_{neu}}} = U_{B_{RestlicheKomponenten}}$$

Ergebnis des strikten statischen Kompatibilitätstests: Das Teilsystem *Trafo_neu* ist nicht kompatibel – im Sinne von strikter Kompatibilität – zum restlichen System der Kaffeemaschine, da der neue Transformator eine um $+6\text{V}$ höhere Ausgangsspannung liefert, als dies der ursprüngliche Trafo gemacht hat (vgl. Abb.: 1.32 (links)). Das beobachtbare äußere Verhalten des neue Transformators *Trafo_neu* unterscheidet sich demnach vom beobachtbaren Verhalten des ursprünglich im System verbauten Transformators *Trafo*. Daraus folgt unmittelbar, dass der neue Transformator nicht strikt kompatibel (genauer: nicht strikt Austausch- bzw. Ersetzungskompatibel) zum System Kaffeemaschine bzw. zum Kontext der Kaffeemaschine ist. Aufgrund der Tatsache, dass der statische Kompatibilitätstest fehlgeschlagen ist, ist die Verifikation der dynamischen Kompatibilität des Systems nicht notwendig.

Um in einem System nicht nur die strikte Kompatibilität modellieren und bewerten zu können, ist es in technischen Systemen üblich, nicht nur einen exakten Wert für eine Eigenschaft anzugeben, sondern einen Bereich in dem eine bestimmte Eigenschaft gültig ist. Dieser Gültigkeitsbereich kann ebenfalls für die Kompatibilitätsbewertung herangezogen werden. Dazu ist allerdings eine spezielle Regelanpassung notwendig, die genau spezifiziert, wann zwei Bereiche kompatibel sind und wann nicht. Solche Regeln werden ebenfalls im Kompatibilitätsregelwerk hinterlegt.

Im obigen Beispiel wurde die strikte Kompatibilität im statischen Fall verletzt und aus diesem Grund der Austausch des alten Teilsystems *Trafo* gegen das neue Teilsystem *Trafo_neu* als inkompatibel zum restlichen System eingestuft. Vergleicht man die einzelnen Werte der beiden Datenblätter (Abb. 1.34 (links)) miteinander, so fällt jedoch auf, dass die Ausgangsspannung des neuen Transformators $U_{B_{Trafo_{neu}}}$ noch innerhalb der Spezifikation des restlichen Systems liegt ($max(U_{B_{Trafo_{neu}}}) <= max(U_{B_{RestlicheKomponenten}})$). Aus diesem Grund ist der Austausch des alten Transformators gegen den neuen zwar inkompatibel im Sinne von *strikter Kompatibilität*, jedoch kompatibel im Sinne der *nicht strikten Kompatibilität*. Um die nicht strikte Kompatibilität verifizieren zu können, muss die obige Kompatibilitätsregel A so modifiziert werden, dass der zuvor fehlgeschlagene statische Kompatibilitätstest gelingt. Dann ist das System kompatibel im Sinne der *nicht strikten* Kompatibilitätsdefinition.

$$\text{A}': \quad L1_{Trafo} \approx L1_{Trafo_{neu}} \wedge N_{Trafo} \approx N_{Trafo_{neu}} \wedge M_{Trafo} \approx M_{Trafo_{neu}} \wedge U_{B_{Trafo}} \approx U_{B_{Trafo_{neu}}}$$

Dabei bedeutet $A \approx B$, dass das Intervall B im Intervall A enthalten sein muss[36].

Ergebnis des nicht strikten statischen Kompatibilitätstests: Durch die Anwendung der modifizierten Kompatibilitätsregel A' auf das obige System Kaffeemaschine, ist der Austausch des Teilsystems *Trafo* gegen das neue Teilsystem *Trafo_neu nicht strikt kompatibel*, aufgrund der Tatsache, dass die um 6V höhere Ausgangsspannung des neuen Transformators innerhalb des Gültigkeitsintervalls des Eingangs U_B des Teilsystems *Restliche Komponenten* sowie dem Kontext des Systems liegt. Nachdem der Austausch des alten Teilsystems gegen das neue durch die Modifikation der ursprünglichen Kompatibilitätsregel A als nicht strikt kompatibel bewertet worden ist, kann nun auch die dynamische Kompatibilität des System Kaffeemaschine anhand der dynamischen Kompatibilitätsregel C verifiziert werden.

$$\text{C}: \quad t_{1_{Trafo}} = t_{1_{Trafo_{neu}}} \wedge t_{2_{Trafo}} = t_{2_{Trafo_{neu}}} \wedge U_{t_{1Trafo}} = U_{t_{1Trafo_{neu}}} \wedge U_{t_{2Trafo}} = U_{t_{2Trafo_{neu}}}$$

Ergebnis des strikten dynamischen Kompatibilitätstests: Die beiden in der Abbildung 1.34 (rechts) dargestellten Graphen bzw. die Werte der beiden angegebenen Punkte t_1 und t_2 sind identisch. Daraus folgt unmittelbar, dass die beiden Teilsysteme *Trafo* und *Trafo_neu strikt dynamisch kompatibel* zueinander sowie zu den restlichen Komponenten des Systems Kaffeemaschine sind.

[36] Nähere Informationen zu Intervallen bzw. zur Schachtelung von Intervallen finden Sie im Kapitel „2.3.2.2.1.3 *Modellierung von Gültigkeitsintervallen in der erweiterten formalen Eigenschaftsdefinition*“ ab Seite 74.

Für die dynamische Kompatibilitätsbestimmung reicht es im Allgemeinen jedoch nicht aus, nur einzelne dezidierte Werte eines Graphen miteinander zu vergleichen. Um zu gewährleisten, dass zwei Graphen identisch bzw. strikt kompatibel oder ähnlich im Sinne der nicht strikten Kompatibilität zueinander sind, müssen die gesamten Graphen miteinander verglichen werden. Im Abschnitt „2.7 *Objektorientierte Modellierung kompatibilitätsrelevanter Eigenschaften von eingebetteten Systemen*“ ab Seite 104 wird exemplarisch der Vergleich zweier Graphen demonstriert.

Anmerkung:
Wie auch bei der statischen Kompatibilitätsbestimmung nicht strikte Kompatibilität mit Hilfe von entsprechenden Kompatibilitätsregeln ausgedrückt werden kann, ist dies auch für die Bestimmung der dynamischen Kompatibilität möglich[37].

Das nächste Kapitel „*Grundlagen der Kompatibilitätsmodellierung und -bestimmung*“ beschäftigt sich mit den Grundlagen der Kompatibilitätsmodellierung sowie der Beschreibung von kompatibilitätsrelevanten Eigenschaften von eingebetteten softwarelastigen Systemen. Dabei werden sämtliche, in diesem Kapitel eingeführten Begriffe und Definitionen aufgegriffen und im Kontext der Modellbildung noch einmal beleuchtet.

[37] Nähere Informationen zur dynamischen Kompatibilität finden Sie unter [Koß09], [Eck07] sowie [Win08].

Kapitel 2.

Grundlagen der Kompatibilitätsmodellierung und -bestimmung

> Wenn ich mich selbst und meine Denkweise analysiere, komme ich zu dem Schluss, dass die Gabe der Fantasie mir mehr bedeutet hat als meine Fähigkeit, mir Wissen anzueignen.
>
> *(Albert Einstein)*

In diesem Kapitel werden sämtliche Grundlagen für die domänenübergreifende modellierungssprachenunabhängige Kompatibilitätsmodellierung und -bestimmung vorgestellt und anhand eines einfachen durchgängigen Beispielsystems – einer Kaffeemaschine – erläutert. Dabei werden die im ersten Kapitel eingeführten grundlegenden Begriffe und Definitionen wieder aufgegriffen und verfeinert.

2.1. Aufbau und Struktur des Kapitels

Das Kapitel „*Grundlagen der Kompatibilitätsmodellierung und -bestimmung*“ gliedert sich in sieben aufeinander aufbauende Hauptbereiche. Begonnen wird das Kapitel mit einer knappen Einführung in die Grundlagen der Modellbildung. Dabei wird zunächst das klassische Modellbildungskonzept und im Anschluss daran das ursprünglich aus der Softwaretechnik stammende, objektorientierte Modellbildungsparadigma vorgestellt. Nach der Einführung des objektorientierten Modellbildungsparadigmas wie auch der damit verbundenen Begriffe, werden die Grundlagen der objektorientierten Modellbildung von eingebetteten softwarelastigen Systemen erläutert. Dabei wird insbesondere auf die graphische Modellierung sowie die formale Beschreibung der kompatibilitätsrelevanten Eigenschaften und der kompatibilitätsrelevanten verhaltensbestimmenden Methoden eines Objekts eingegangen. Außerdem werden hier sämtliche Erweiterungen des ursprünglichen objektorientierten Modellbildungsparadigmas, die speziell für die Kompatibilitätsmodellierung und -bestimmung notwendig sind, vorgestellt und anhand des Beispielsystems Kaffeemaschine erläutert. Nachdem die Grundlagen der objektorientierten Modellbildung gelegt worden sind, folgt die Einführung des Kompatibilitätsregelwerks, bevor im nächsten Abschnitt die objektorientierte Kompatibilitätsprüfung anhand des zuvor beschriebenen Regelwerks erläutert wird. Nach der Vorstellung des Kompatibilitätsregelwerks folgt die Beschreibung weiterer Techniken, die direkt oder indirekt für die Kompatibilitätsmodellierung und -bestimmung hilfreich sind. Dazu zählt im Besonderen die Einführung in die Grundlagen eines domänenübergreifenden integrierten Systemmodells. Abgeschlossen wird dieses Kapitel mit der Anwendung der zuvor eingeführten formalen Notation für die Kompatibilitätsmodellierung anhand von ausgesuchten Beispielen aus den drei Domänen, Mechanik, Elektrik/Elektronik und Software.

2.2. Modellbildung

Im Kapitel „*Was hat Systems Engineering mit modellbasierter Kompatibilitätsbewertung zu tun?*“ wurde das Systems Engineering als eine allumfassende Wissenschaft vorgestellt, deren Ziel es ist, Ingenieure bei der Entwicklung von komplexen (technischen) Systemen zu unterstützen. Zu den wichtigsten dort vorgestellten Paradigmen zählte unter anderem die *Abstraktion und Modellbildung*[38]. In diesem Abschnitt wird nun insbesondere auf die unterschiedlichen Aspekte der Modellbildung bzw. dem Modellbildungsprozess eingegangen, da dieser die zentrale Grundlage für die Kompatibilitätsbestimmung und -bewertung darstellt.

Begonnen wird der Abschnitt „*Modellbildung*“ mit der Vorstellung des allgemeinen Modellbildungsprozesses. Daran anschließend folgt die Spezialisierung des Modellbildungsprozesses auf die Belange der Modellierung und der Bewertung der Kompatibilität von eingebetteten softwarelastigen Systemen.

Der Modellbildungsprozess
Bei der Modellbildung handelt es sich um einen typischen *Abstraktionsprozess*, der in allen technischen und nichttechnischen Disziplinen gleichermaßen mit dem Ziel eingesetzt wird, die Komplexität eines realen oder abstrakten (gedanklichen) Systems[39] auf ein *bewältigbares* Maß zu reduzieren. Im Modell sollen dabei sowohl die entscheidenden Eigenschaften und Funktionen, als auch die Struktur des realen oder gedanklichen Systems erhalten bleiben. Der *Modellbildungsprozess* beschreibt den *Transformationsprozess*, der ein dezidiertes System oder ein Teilstück eines real existierenden oder gedanklichen Systems, wie zum Beispiel eine Kaffeemaschine, ein Auto, eine elektrische Schaltung oder eine chemische Verbindung (reale Systeme), eine mathematische Theorie (gedankliches System) in ein *abstraktes Modell* überführt. Die folgende Definition beschreibt den oben geschilderten Sachverhalt der Modellbildung.

[38] Vgl. Unterpunkt „Abstraktion und Modellbildung“ auf Seite 6.
[39] Vgl. Definition 1.2 eines Systems auf Seite 12.

Definition 2.1 ***Modellbildung (Ursprüngliche Definition [PDIG])***
Unter ***Modellbildung*** *verstehen wir die Festlegung einer* ***formale Darstellung****, die die* ***Struktur*** *und das* ***Verhalten*** *des zu modellierenden realen Systems so genau wie notwendig beschreibt.*

Zusätzlich zur formalen Beschreibung der Modellbildung, kann der generische Modellbildungsprozess auch mit Hilfe der Abbildung 2.1 dargestellt werden. In dieser Abbildung nach [FM06, 99ff] ist der Modellbildungsprozess als ein Zyklus dargestellt, der so oft durchlaufen werden muss, bis das resultierende Modell so exakt wie notwendig das reale System beschreibt, ohne unnötig komplex und unhandlich zu werden.

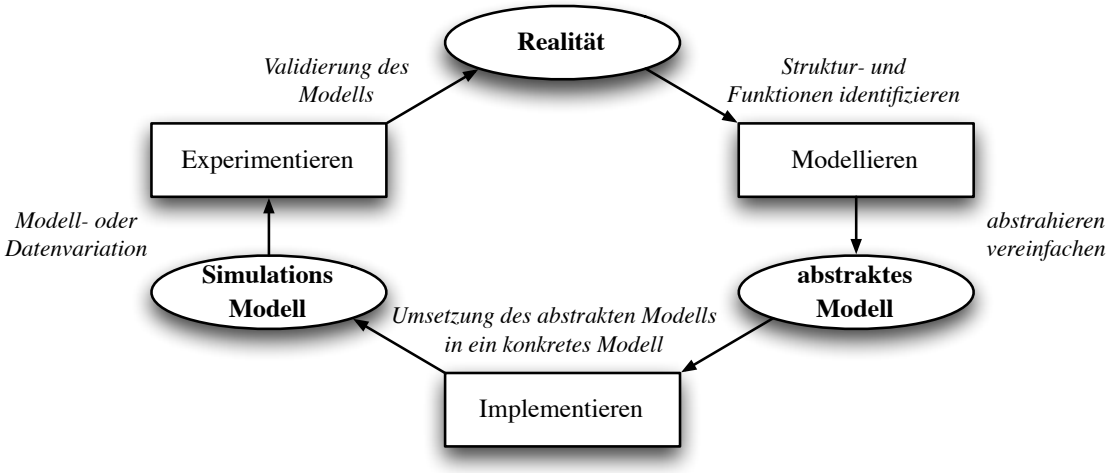


Abbildung 2.1.: Modellbildungszyklus: Übergang von der Realität zum Modell und durch Simulation des Modells zurück in die Realität (nach [FM06, 99]).

Ausgangspunkt für jede Modellbildung ist die *Realität* bzw. ein *abstraktes gedankliches Gebilde*. Während des Modellbildungsprozesses, genauer des Modellbildungszykluses, wird sowohl die grundlegende (System-) Struktur, als auch die Funktionalität und das Verhalten des zu modellierenden realen oder gedanklichen Systems abstrahiert und schrittweise verfeinert. Im nächsten Modellierungsschritt wird aus dem abstrakten generischen Modell (auch qualitatives Modell genannt) ein konkretes quantitatives Modell erzeugt. Das konkrete, mit Daten und Werten befüllte, quantitative Modell wird im nächsten Schritt – dem Simulationsschritt – simuliert. Ziel der Simulation ist die Validierung und Verifikation des Modells durch den Vergleich der Modellmessergebnisse mit realen oder erwarteten Messergebnissen des real existierenden bzw. des gedanklichen Systems. Stimmen die simulierten Ergebnisse mit den erwarteten Ergebnissen überein bzw. liegen sie innerhalb des erwarteten Intervalls (Erwartungswert mit Standardabweichung), so gilt das Modell als valide und gültig, ansonsten muss das Modell weiter verfeinert werden. Dieser Prozess sollte solange wiederholt werden, bis die Simulationsergebnisse mit den realen bzw. erwarteten Ergebnissen übereinstimmen.

Aufbauend auf dem generischen Modellbildungsprozess sowie den beiden Systems Engineering Paradigmen *Abstraktion* und *Modellbildung* stellte Dr. Negele in seiner Dissertation „Systemtechnische Methodik zur ganzheitlichen Modellierung am Beispiel der integrierten Produktentwicklung“ im Kapitel „Vorgehen bei der Modellbildung [DIN98, 79ff]“ eine weitere Verfeinerung des obigen generischen Modellbildungsprozesses, speziell für die Modellierung von technischen Produkten, vor. Die Abbildung 2.2 auf der nächsten Seite zeigt den von Dr. Negele in seiner Dissertation vorgestellten Modellbildungsprozess.

Begonnen wird der Modellbildungsprozess mit der Realität – hier *Realsystem* genannt. Auf der linken Seite der Abbildung ist das Vorgehen bei der Modellbildung eines Systems detailliert dargestellt. Dabei ist zu erkennen, dass die Modellierung in drei großen aufeinander aufbauenden Phasen erfolgt. Während der ersten Modellierungsphase wird die so genannte Systemebene, bestehend aus dem System, den Systemgrenzen, den Inputs und Outputs sowie den Relationen zwischen den Elementen des Systems festgelegt. Dabei wird das Modell des zu modellierenden Systems stetig verfeinert und um weitere Informationen ergänzt. In der zweiten Modellierungsphase werden sämtliche Elementeigenschaften sowie die qualitativen Funktionen des Systems in das im ersten Schritt entstandene Grundgerüst des Modells eingetragen. In der dritten und letzten Modellierungsphase werden die zuvor identifizierten qualitativen Funktionen des Systems mit konkreten Werten befüllt, so dass das Modell des Systems simuliert und gebaut werden kann. Die dritte Modellierungsphase wird auch als Elementebene bezeichnet. Zwischen jeder Phase des Modellbildungsprozesses kann ein Rücksprung zu einer früheren, bereits abgeschlossenen Phase erfolgen, falls während der Modellbildung Fehler oder Inkonsistenzen im Modell festgestellt werden. Die rechte Seite der Abbildung zeigt die Inhalte und Ergebnisse der verschiedenen Modelle, die während des links dargestellten Modellbildungsprozesses entstehen. Aus dem komplexen generischen Modellbildungsprozess lassen sich, ähnlich wie in der Abbildung 2.1 vier Hauptschritte isolieren. Die Abbildung 2.3 auf der nächsten Seite stellt die wesentlichen vier Schritte des in der Abbildung 2.2 auf der nächsten Seite dargestellten allgemeinen generischen Modellbildungsprozesses nach Dr. Negele dar. In diesem vereinfachten Modell wurde bewusst auf die Rücksprünge zwischen den einzelnen Phasen verzichtet um das Modell so einfach wie möglich zu halten. Dennoch enthält es sämtliche wichtige Prozessschritte zur Modellierung eines Systems.

Der vereinfachte Modellbildungsprozess wird nun als Grundlage für die weitere Einführung in die Modellierung von eingebetteten softwarelastigen Systemen verwendet.

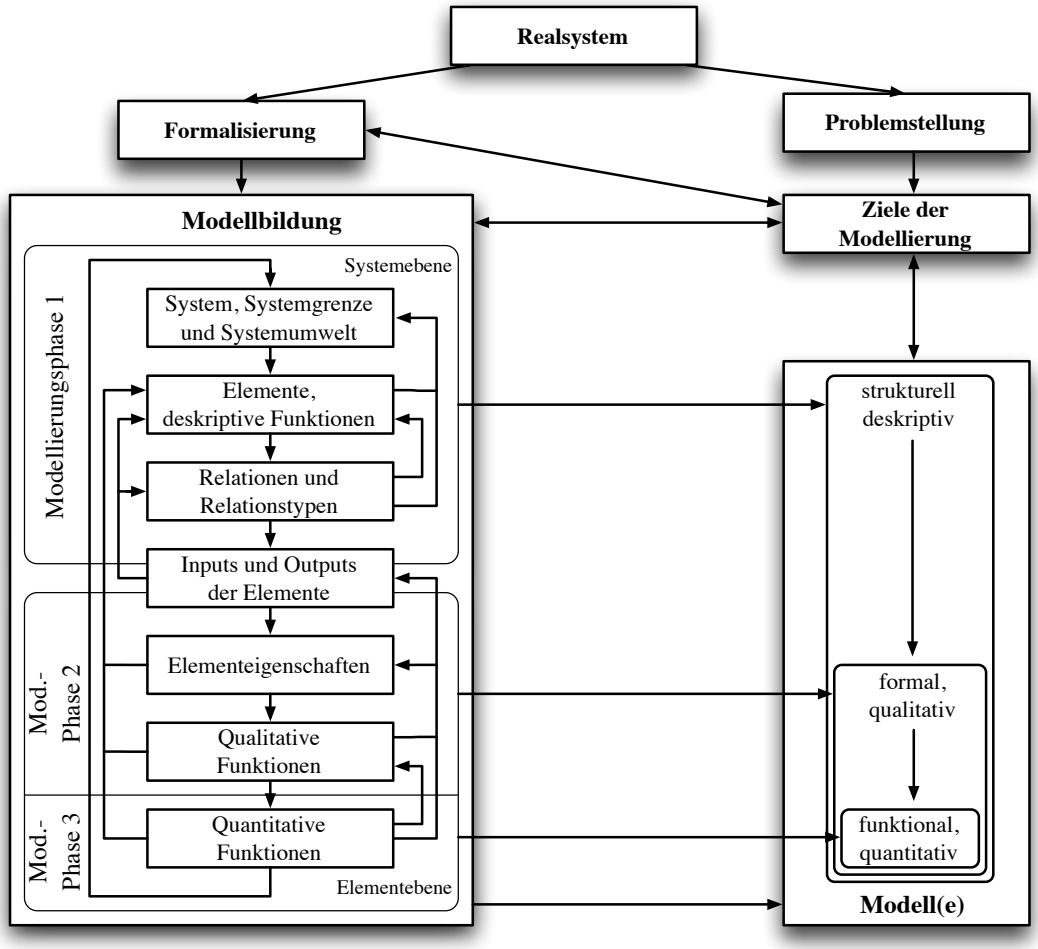


Abbildung 2.2.: Der Modellbildungsprozess – Vorgehensweise bei der Modellbildung ([DIN98, 79]).

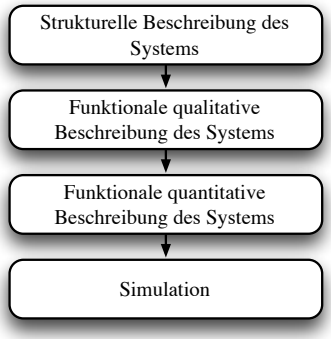


Abbildung 2.3.: Der Modellbildungsprozess – Die wesentlichen Schritte bei der Modellbildung.

Sinn und Zweck der Modellbildung
Eine der meist gestellten Fragen, wenn es um die Modellbildung und Simulation von Systemen geht, lautet: „Wozu brauche ich überhaupt ein Modell?“. „Modelle sind doch nur etwas für Architekten oder Karosseriebauer, ich jedoch erstelle eingebettete Systeme. Dazu benötige ich kein Modell! Außerdem sind Modelle viel zu teuer, zu ungenau und im Allgemeinen nicht realistisch!“

Eine allgemein gültige und verbindliche Antwort auf die obigen Fragen kann und wird es nie geben! Es hat sich jedoch im Laufe der Zeit, vor allem im Systems Engineering- sowie im Informatik-Umfeld, herauskristallisiert, dass für die erfolgreiche Durchführung eines Projekts bzw. für die Entwicklung eines Systems, ein Modell des zu entwickelnden Produkts sehr hilfreich ist. Aus diesem Grund nimmt auch die Modellbildung sowohl im Systems Engineering- als auch im Informatik-Umfeld eine zentrale Rolle ein. Mehr noch, Prof. Dr.-Ing. E. Igenbergs stellt in seiner Vorlesung *Systems Engineering* an der Technischen Universität München die These auf: *„Die menschliche Sprache ist stets modellbasiert! Wir nehmen unsere Umwelt nur modellhaft wahr. Aus diesem Grund können wir uns auch nur über diese Modelle und deren Eigenschaften unterhalten. Niemand hat die „Natur“ bzw. die „Umwelt“ je anders wahrgenommen.“* (Prof. Dr.-Ing. E. Igenbergs). Aus diesem Grund ist alles – auch diese Arbeit – nur ein abstraktes Modell der Realität.

Ein anderer Ansatz um die obigen Fragen nach dem Sinn und Zweck von Modellen bzw. der Modellbildung zu beantworten, ist anhand eines einfachen Beispiels, das den Sinn und Zweck der Modellierung exemplarisch ausführt und das Ergebnis der Modellierung bzw. des Modellbildungsprozesses graphisch dargestellt. Die Abbildung 2.4 illustriert die Anwendung des allgemeinen generischen Modellbildungsprozesses aus der Abbildung 2.1 anhand eines real existierenden Systems (Objekt), hier beispielsweise anhand des IC 7400 [40].

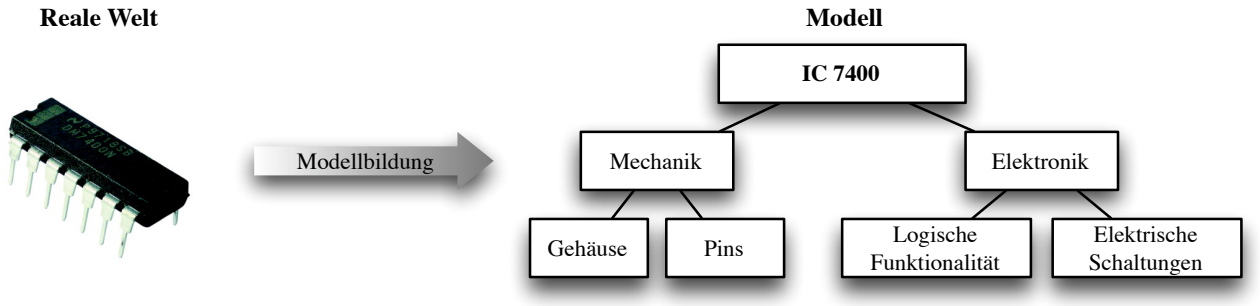


Abbildung 2.4.: Übergang von einem *realen Objekt*, z.B. einem IC 7400, zu einem einfachen Modell des ICs.

Auf der linken Seite der Abbildung ist das real existierende System, der IC 7400, dargestellt. Sowohl die Eigenschaften, als auch die Funktionen des realen ICs werden nun, mit Hilfe des generischen Modellbildungsprozesses, in ein abstraktes Modell des Systems überführt. Auf der rechten Seite der Abbildung ist das Ergebnis des Modellbildungsprozesses abgebildet. Das System *IC 7400* besteht demnach aus zwei Hauptgruppen, einer mechanischen sowie einer elektrischen Gruppe. Beiden Gruppen bestehen ihrerseits wiederum aus zwei weiteren Untergruppen. Jede der Untergruppen kann nun nahezu beliebig weiter verfeinert werden. Eine weitere Darstellungsart des selben IC 7400 zeigt die Abbildung 2.5. Hier ist auf der linken Seite der Abbildung das IC 7400 als abstraktes Modell (technische Zeichnung) des mechanischen Gehäuses inklusive Pin-Nummern und Bezeichnungen dargestellt. Auf der rechten Seite der Abbildung ist die logische Funktion eines Schaltelements des ICs auf drei unterschiedliche Arten aufgeschrieben. Zum einen in der mathematischen Notation $Y = \neg(A \wedge B)$, in einer Bool´schen Funktionstabelle und als DIN[41] Schaltzeichen.

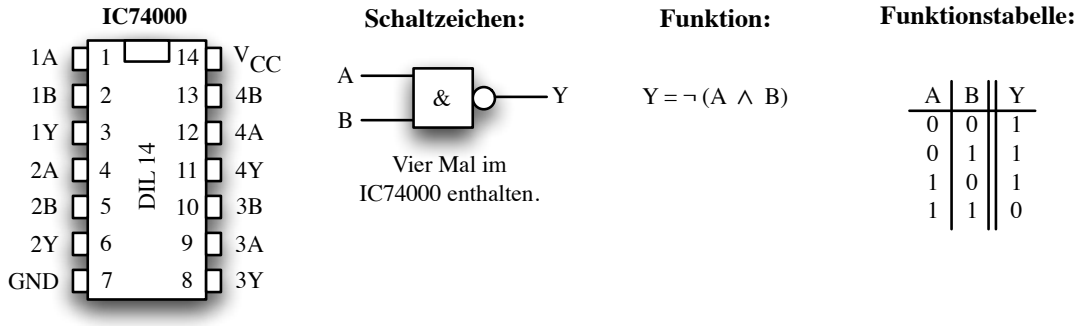


Abbildung 2.5.: IC 7400:
Links: Gehäuse mit Pin-Nummern und Beschaltung
Rechts: Logische Funktion, Funktionstabelle und Schaltzeichen.

Sämtliche, in den beiden Abbildungen dargestellten Modelle bzw. Teilmodelle des IC 7400, beschreiben stets den selben, real existierenden IC 7400, allerdings aus unterschiedlichen Blickrichtungen und aus verschiedenen Modellierungstiefen beleuchtet. Keines der Modelle beschreibt sämtliche Aspekte des realen ICs, da dadurch das abstrakte Modell des ICs zu sehr aufgebläht und unhandlich werden würde. Wie die beiden Modelle des IC 7400 (vgl. Abb.: 2.4 und 2.5) eindrucksvoll demonstrieren, gibt es im Allgemeinen nicht „ein“ oder „das“ Modell eines Systems, sondern nahezu beliebig viele unterschiedliche Modelle, die ein und dasselbe Objekt beschreiben. Jedes Modell eines Systems kann andere Eigenschaften und Belange des Systems beschreiben, die für einen speziellen Zweck in diesem Modell hinterlegt worden sind[42]. Um diesem Dilemma zu begegnen, ist es für die Modellbildung, bzw. den Modellbildungsprozess besonders wichtig, für die Domäne Kompatibilitätsbestimmung von softwarelastigen eingebetteten Systemen, bestimmte Modellierungsrichtlinien vorzugeben, die die Daten, die mindestens in einem Modell enthalten sein müssen, festlegen (siehe hierzu: Kapitel „2.7 *Objektorientierte Modellierung kompatibilitätsrelevanter Eigenschaften von eingebetteten Systemen*“ ab Seite 104).

[40] Das *IC 7400* (*IC* – engl. Integrated Circuit – Integrierter Schaltkreis) ist ein TTL Standard-Baustein (*TTL* – Transistor-Transistor-Logik) mit 4 NAND-Schaltgattern in einem DIL (*DIL* – Dual-in-line) Gehäuse mit 14 Pins.

[41] Das Akronym *DIN* steht für Deutsche Institut für Normen e.V.. Nähere Informationen zu DIN-Normen finden Sie unter [Ne08].

[42] Anmerkung:
„Tatsächlich kann ein Modell niemals vollständig die Realität beschreiben, da nicht jeder Betrachter (z.B. auch dessen Gefühle beim Ansehen der Realität) je modelliert werden können.“ (Dipl. Inf. Carolin Eckl)

Das Ziel der Modellbildung ist die Erzeugung eines Modells, das die Realität möglichst gut abbildet, und dabei nicht zu komplex und vor allem widerspruchsfrei ist. Im nachfolgenden Abschnitt wird zunächst definiert was ein Modell ist und im Anschluss daran, welche Eigenschaften ein „gutes“ Modell haben muss, um für die modellbasierte Kompatibilitätsbestimmung verwendet werden zu können.

Modell
Ein *Modell* ist ein *vereinfachtes Abbild*, eine *Nachbildung* eines *real existierenden* oder *gedanklichen* Objekts[43]. Ziel der Modellierung ist ein *möglichst exaktes Abbild* eines Objekts zu schaffen, das so *nahe an der Realität wie notwendig* aber dennoch so *abstrakt wie nötig/möglich* ist, um das reale oder gedankliche Objekt vollständig, z.B. in einem Computer oder auf einem Blatt Papier, darstellen und beschreiben zu können. Dabei ist die *Informationsreduktion* auf das Wesentliche oberstes Gebot bei der Modellbildung, da sonst das Modell zu komplex und unüberschaubar werden kann.

Der VDI – Verein Deutscher Ingenieure – definiert ein Modell als:

Definition 2.2 ***Modell (nach VDI [VDI07])***
Ein Modell ist eine vereinfachte Nachbildung eines existierenden oder gedachten Systems in einem anderen begrifflichen oder gegenständlichen System. Es wird genutzt, um eine bestimmte Aufgabe zu lösen, deren Durchführung mittels direkter Operationen am Original nicht möglich oder zu aufwendig wäre.

Aufbauend auf der allgemeinen Definition eines Modells fasst Prof. Dr. G. Patzak in seinem Buch „Systemtechnik – Planung komplexer innovativer Systeme“ die wichtigsten Eigenschaften eines „guten“ Modells zusammen [Pat82]:

Die „richtige Modellbildung“ ist die wichtigste Voraussetzung für eine erfolgreiche Untersuchung eines Systems. Das Modell soll mit den Beobachtungen des Systems gut übereinstimmen, d.h. realitätsbezogen sein. Dies soll durch eine gute Übereinstimmung der Systemstruktur sowie der Systemelemente erreicht werden:

- **Formal richtig (exakt)**
  Das Modell soll in sich widerspruchsfrei sein und Ergebnisse möglichst in quantitativer Form liefern. Alle Aussagen des Modells müssen reproduzierbar und nachvollziehbar sein.
- **Produktiv („fruchtbar“)**
  Das Modell soll auf die gestellten Fragen inhaltlich und formal brauchbare Ergebnisse liefern. Anders ausgedrückt: Ein Modell eines Systems kann nur Fragen beantworten, die in seinen Formalisierungskomponenten (Attributen, Funktionen) repräsentiert sind (frei nach Prof. Dr.-Ing. E. Igenbergs).
- **Handhabbar („benutzerfreundlich“)**
  Das Modell soll möglichst leicht anzuwenden und erlernt werden können. Des Weiteren müssen die gelieferten Ergebnisse leicht zu interpretieren und vollständig sein.
- **Effizient („nicht aufwendig, billig“)**
  Der Aufwand für Modellierung und Simulation sollte möglichst niedrig sein.

Die Abbildung 2.5 auf der vorherigen Seite zeigte mehrere Modelle des IC 7400 in unterschiedlichen Ausprägungen. Jedes der Modelle ist für eine spezielle Anwendung spezifiziert und entwickelt worden. So kann zum Beispiel die DIL-Schemazeichnung verwendet werden, um das Aussehen des ICs zu beschreiben, während das DIN-Schaltzeichen in elektrischen Schaltplänen eingesetzt werden kann. Die Bool´sche Funktion sowie die Funktionstabelle sind wiederum für die logische Betrachtung bzw. die Beschreibung der Funktionalität (Verhalten) des ICs wichtig. Alle in der Abbildung dargestellten Modellvarianten genügen den von Prof. Dr. G. Patzak eingeführten Anforderungen an ein gutes, valides Modell.

Zusätzlich zu den unterschiedlichen Anforderungen an ein gutes Modell ist es notwendig, Modelle nach ihrem Zweck bzw. ihrer Anwendung zu klassifizieren.

Klassifikation von Modellen
Für die erfolgreiche Durchführung der Modellbildung ist die Auswahl eines „geeigneten Modells“ von entscheidender Bedeutung für die anschließende Transformation eines realen oder gedanklichen Systems in ein entsprechendes Modell. Wie bereits im Abschnitt „Der Modellbildungsprozess“ ab Seite 31 gezeigt worden ist, gibt es viele unterschiedliche Arten von Systemen. Angefangen von konkreten bis hin zu abstrakten oder gedanklichen Systemen. Um sämtliche Arten von Systemen mit Hilfe von Modellen beschreiben zu können, gibt es nach [FM06] vier Hauptklassen von Modellen, die sich weiter in Subklassen unterteilen lassen. In der nachfolgenden Aufzählung sind die wesentlichen Modellklassen kurz aufgeführt, die speziell für die Modellierung von technischen Systemen verwendet werden. Im Anschluss an die jeweilige Klassifikation der Modelle folgt eine kurze Erläuterung der wesentlichen Eigenschaften der Modellklasse für die modellbasierte Kompatibilitätsbeschreibung und -bewertung, bevor im nachfolgenden Abschnitt noch einmal auf die Details der jeweiligen Modellklasse für die Kompatibilitätsbeschreibung eines Systems eingegangen wird. Begonnen wird die Aufzählung mit der Klassifikation eines Modells nach dessen Verwendungszweck.

- Klassifikation nach dem **Verwendungszweck**:
  Modelle lassen sich nach ihrem Verwendungszweck klassifizieren. Dabei werden im Allgemeinen vier unterschiedliche Arten von Modellen unterschieden:

[43] Ein Objekt kann auch ein ganzes System oder ein Teil eines System (Subsystem) sein. Eine ausführliche Beschreibung und Definition eines *Objekts* finden Sie im nächsten Kapitel 2.3.1.1 ab Seite 46.

  - *Beschreibungsmodelle* um Abläufe innerhalb eines Systems zu veranschaulichen (z.B. graphische Darstellung von (Strom- oder Material-) Flüssen in einem System).
  - *Erklärungsmodelle* um Abläufe innerhalb eines Systems fassbar und begründbar zu machen (z.B. physikalische Gesetze wie beispielsweise das Atommodell).
  - *Entscheidungs- oder Prognosemodelle* um mögliche zukünftige Systementwicklungen oder Eigenschaften vorherzusagen (z.B. Erweiterung eines existierenden Systems um weitere Eigenschaften und Funktionen; das zukünftige Wetter).
  - *Schulungsmodelle* an denen Personen eine besondere Eigenschaft erlernen können, ohne am realen System arbeiten zu müssen (z.B. Schiffs- Flug- und Fahrsimulationsmodelle).

  Die Klassifikation von Modellen anhand ihres Verwendungszwecks ist die gebräuchlichste Klassifikationsart für Modelle von technischen Systemen. Dabei kommen hauptsächlich Beschreibungs- sowie Erklärungsmodelle bzw. Kombinationen aus beiden Modellarten zum Einsatz. Wobei der Übergang zwischen Erklärungsmodellen hin zu Schulungsmodellen eher fließender Natur ist. Beide Modellarten können sowohl für Schulungszwecke als auch für die Erklärung eines Systems verwendet werden. Eine Sonderrolle in der obigen Klassifikation spielen die so genannten Entscheidungs- oder Prognosemodelle. Sie dienen ausschließlich der Entscheidungsfindung bzw. der Prognose von Systemeigenschaften. Im Allgemeinen setzen beide Modellarten auf bereits existierenden Beschreibungs- bzw. Erklärungsmodellen auf.

  Für die Kompatibilitätsmodellierung bzw. die Bewertung der Kompatibilität eines Systems kommen in den meisten Fällen Kombinationen aus Beschreibungs- Erklärungs- sowie Entscheidungs- und Prognosemodellen zu Einsatz.

- Klassifikation nach dem **Detailierungsgrad der Eigenschaften** des zu modellierenden Systems:
  Eine weiter Klassifikationsmöglichkeit für Modelle ist über den Detailierungsgrad der Eigenschaften des Modells. Zum einen gibt es *qualitative Modelle*, bei denen die Eigenschaften des Systems zwar spezifiziert aber nicht quantifiziert sind. Zum anderen Modelle, bei denen die Eigenschaften sowohl qualitativ, als auch quantitativ beschrieben sind – so genannte *quantitative Modelle*.

  Für die Modellierung von technischen Systemen werden im Allgemeinen quantitative Modelle eingesetzt, in denen die Eigenschaften des Modells mit konkreten Werten besetzt sind. Die quantitativen Modelle werden vor allem für die Simulation sowie den späteren Bau des konkreten Systems verwendet, während qualitative Modelle während der frühen Entwicklungsphasen (Phasen 0-A)[44] eingesetzt werden, in denen die konkreten Werte noch nicht spezifiziert sind, bzw. noch nicht vorliegen. Für die Kompatibilitätsmodellierung und -bestimmung sind quantitative Modelle zwingend, da ansonsten die für die Kompatibilitätsbestimmung notwendigen Werte im Modell fehlen.

- Klassifikation nach dem **Verhalten** des zu modellierenden Systems:
  Modelle lassen sich nach dem zu erwartenden Systemverhalten eingruppieren. Dabei wird zwischen *statischen* und *dynamischen Modellen* unterschieden. Insbesondere gilt dabei: Ein Modell eines Systems kann entweder die *statischen* und/oder die *dynamischen* Eigenschaften des Systems enthalten. Eine Kombination aus statischen und dynamischen Modellen ist ebenfalls möglich, jedoch werden kombinierte Modelle in der Praxis eher selten verwendet. Wenn sowohl statische als auch dynamische Aspekte eines Systems modelliert werden müssen, werden meistens zwei Modelle eines Systems erstellt; eines das die statischen Belange eines Systems beschreibt und eines in dem die dynamischen Aspekte modelliert sind. Für die Beschreibung des gesamten Systemverhaltens sind beide Modelle des Systems notwendig. Aus diesem Grund werden in der Praxis die beiden Modellarten so miteinander verbunden („verlinkt"), dass jederzeit sowohl die statischen als auch die dynamischen Eigenschaften des Systems in den beiden Modellen aufgrund der Verlinkung wiedergefunden werden können[45].

  Für die Modellierung bzw. die Bestimmung von Kompatibilität eines Systems sind stets statische und dynamische Modelle notwendig, da jedes System sowohl statische als auch dynamische Eigenschaften besitzt, und diese auch auf Kompatibilität untersucht werden müssen[46].

- Klassifikation in **materielle** sowie **immaterielle Modelle** eines Systems:
  Eine weitere Klassifikationsmöglichkeit für Modelle eines Systems besteht in der Unterteilung in so genannte *materielle* und *immaterielle Modelle*; wobei materielle Modelle eines Systems hauptsächlich, wie bereits in der Einleitung zur Modellbildung erwähnt, in der Architektur, im Flugzeug- sowie im Automobil-Modellbau zur Visualisierung des Systems bzw. für Messzwecke wie zum Beispiel im Windkanal eingesetzt werden. Im Gegensatz dazu werden immaterielle Modelle vorwiegend zur Beschreibung von Systemen eingesetzt. Immaterielle Modelle enthalten somit sämtliche Daten und Informationen um das System fertigen zu können.

  - *Materielle Modelle* sind Modelle, bei denen ein real existierender oder neu zu entwickelnder Gegenstand, z.B. ein Auto oder ein Gebäude, symbolisch/exemplarisch oder maßstäblich dargestellt wird. Materielle Modelle werden vor allem in der Architektur oder der Automobilindustrie (Designstudie, Prototypenbau) eingesetzt um das spätere Produkt bereits während der Entwicklungsphase exemplarisch darzustellen (Prototyp) bzw. Experimente damit durchführen zu können, ohne auf das fertige Produkt warten zu müssen.

    Grundeigenschaften aller materiellen Modelle sind: maßstäblich, analog, symbolisch.

[44] Siehe hierzu: Kapitel „1.2 *Was hat Systems Engineering mit modellbasierter Kompatibilitätsbewertung zu tun?*" ab Seite 3..

[45] Siehe hierzu insbesondere die Kapitel „3.4.3 *Die Unified Modelling Language – UML/UML2*" ab Seite 131, „3.4.4 *Die System Modelling Language – SysML* " ab Seite 144, „3.6 *Einführung in die Kompatibilitätsmodellierungssprache (U)CML*" ab Seite 153 sowie „2.6 *Domänenübergreifendes integriertes Systemmodell*" ab Seite 103.

[46] Anmerkung:
Alle realen Systeme sind dynamisch, da sie sich während ihrer Lebenszeit ständig verändern. Ein Beispiel dafür ist der Turm von Pisa. Innerhalb eines „kurzen" Beobachtungszeitraum ist das System „Turm von Pisa" statisch. Über einen längeren Beobachtungszeitraum z.B. über mehrere Jahre, „bewegt/neigt" sich der Turm jedoch (nach [FM06, 39]).

– *Immaterielle Modelle* werden im Gegensatz zu materiellen Modellen vor allem zur abstrakten graphischen Darstellung von Systemen benutzt, die zu groß oder zu klein, zu komplex oder „nicht greifbar“ sind, um sie als materielle Modelle darstellen zu können. Beispielsweise werden immaterielle Modelle in der Schaltungstechnik oder der Chemie verwendet um Systeme zu beschreiben.

  In der Chemie beispielsweise kann eine real existierende Verbindung wie z.B. Ethen[47], in drei unterschiedlichen Modellklassen repräsentiert sein:

  * Formelschreibweise (immateriell, mathematisch): Formelschreibweise von Ethen C_2H_4
  * Strukturschreibweise (immateriell, graphisch, symbolisch): Darstellung der Struktur von Ethen in der chemischen Strukturschreibweise.

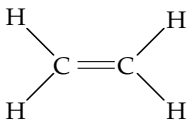


Abbildung 2.6.: Strukturformel der chemischen Verbindung Ethen (C_2H_4).

  * Atommodell (materiell und immateriell): Das Atommodell der Ethen-Verbindung nimmt eine Sonderstellung ein, da ein Atommodell sowohl materiell (als z.B. maßstäbliches Modell) als auch immateriell durch einen Rechner dargestellt werden kann.

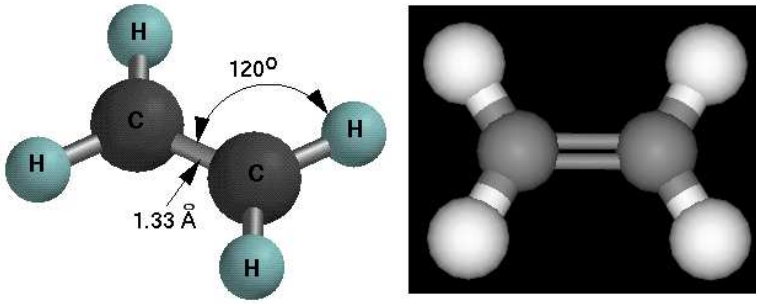


Abbildung 2.7.: Zwei Atommodelle der Ethen-Verbindung.

Grundeigenschaften immaterieller Modelle sind: verbal, formal, mathematisch, algorithmisch, graphisch.

In letzter Zeit sind die Grenzen zwischen den materiellen und immateriellen Modellen jedoch verwischt worden. Da aufgrund der verbesserten Visualisierungsmöglichkeiten von modernen Rechnern auch rein mathematische Modelle, wie z.B. ein simulierter Crashtest in der Automobilindustrie nun mit einem Rechner visualisiert werden können, ohne ein reales Modell (materielles Modell) bauen und zerstören zu müssen.

Für die Modellierung von Kompatibilität, sowie die anschließende Bewertung, können sowohl materielle wie auch immaterielle Modelle eingesetzt werden. Im Allgemeinen werden jedoch mehr immaterielle als materielle Modelle für die Kompatibilitätsbestimmung herangezogen. Dies liegt zum einen daran, dass immaterielle Modelle bereits sehr früh im Entwicklungszyklus eingesetzt werden können um sowohl das statische als auch das dynamische Systemverhalten zu beschreiben. Und zum anderen daran, dass durch die Nutzung von immateriellen Modellen während der Systementwicklung Zeit und Kosten gespart werden können, da keine aufwendigen materiellen Modelle hergestellt werden müssen. In der Fertigungsphase eines Systems werden jedoch sehr häufig materielle Modelle für die Kompatibilitätsbestimmung verwendet. So werden zum Beispiel so genannte Lehren in der Automobilfertigung eingesetzt, um die Form eines gepressten Kotflügels nach dessen Pressung mit einer Referenzpressung vergleichen und im Falle einer Abweichung geeignete Gegenmaßnahmen ergreifen zu können.

In naher Zukunft werden sich die Grenzen zwischen materiellen und immateriellen Modellen in der Kompatibilitätsmodellierung und der -bestimmung von Kompatibilität ebenfalls weiter auflösen, so wie es beispielsweise schon heute in der Chemie oder der Automobilindustrie der Fall ist. Dies liegt vor allem an der stetig wachsenden rechnergestützten Entwicklung und Fertigung von Systemen, so dass bereits die Modelle (Pläne) eines Systems auf mögliche Inkompatibilitäten hin untersucht werden können, ohne dafür ein reales Modell des Systems zu fertigen.

[47] Ethen (auch Äthen; früher Ethylen beziehungsweise Äthylen) ist ein farbloses, süßlich riechendes Gas. Es ist das einfachste Alken (ungesättigter Kohlenwasserstoff mit einer Kohlenstoffdoppelbindung) und in der chemischen Industrie sowie als Phytohormon von hoher Bedeutung [Wik07b].

Wie die obige Aufzählung der unterschiedlichen Modellklassen zeigt, sind die Grenzen zwischen den unterschiedlichen Klassen nicht strikt im mathematischen Sinne, sondern fließend. So kann zum Beispiel ein Beschreibungsmodell sowohl für die Prognose eines Systems als auch für die Schulung eingesetzt werden. Vorraussetzung dafür ist lediglich, dass das Modell von allen Beteiligten verstanden und richtig interpretiert werden kann. Dies ist jedoch nicht bei allen Modellen bzw. den mit den Modellen verbundenen Modellierungssprachen möglich da die meisten Modelle sehr stark auf eine Domäne bzw. ein Zielpublikum zugeschnitten sind. So kann beispielsweise das immaterielle Modell eines chemischen Elements lediglich von Chemikern bzw. geschulten Personen richtig interpretiert werden. Ohne chemisches Hintergrundwissen ist dies nicht möglich. Jedoch kann das immaterielle Modell einer chemischen Verbindung auch zu Schulungszwecken verwendet werden, wodurch sich der Bogen von den immateriellen Modellen hin zu den Beschreibungs- bzw. Schulungsmodellen schließt.

Eine weitere Klassifikationsmöglichkeit für Modelle beschreibt Prof. A. T. Bahill (Ph.D.), mit neun weiteren Autoren, in der Veröffentlichung „The Design-Methods Comparison Project [PDB98a]“. Darin werden elf unterschiedliche Modellklassen bzw. Modellierungssprachen kurz vorgestellt, mit deren Hilfe technische Systeme modelliert und beschrieben werden können. In der Veröffentlichung werden die unterschiedlichen Modellierungssprachen anhand eines übergreifenden Beispielsystems (eines Ampelsystems) erläutert. Dabei wird sowohl die statische Struktur, als auch das Verhalten des Systems modelliert. In der Veröffentlichung wird das Hauptaugenmerk auf die Modellierung des dynamischen Verhaltens gelegt. Das Ergebnis der Veröffentlichung ist eine grobe Übersicht über die unterschiedlichen Fähigkeiten der vorgestellten Modellierungssprachen sowie deren besondere Einsatzmöglichkeiten für bestimmte Problemklassen.

Für die Modellierung und Beschreibung sowie den Kompatibilitätstest von eingebetteten softwarelastigen Systemen ist eine Kombination aus einigen der oben vorgestellten Modellklassen sowie den grundsätzlichen Systems Engineering Paradigmen notwendig. Die Verknüpfung von Eigenschaften existierender Modelle zu einem für den Kompatibilitätstest geeigneten Modell wird im nachfolgenden Abschnitt erläutert.

Grundeigenschaften eines Modells zur Modellierung und Beschreibung der Kompatibilität von eingebetteten softwarelastigen Systemen
Wie die obige Klassifikation der unterschiedlichen Modelle gezeigt hat, existieren für fast alle Anwendungsfälle eigene Modellklassen, die sich zum Teil überschneiden bzw. gegenseitig ergänzen. Für die modellbasierte Kompatibilitätsbewertung ist eine Kombination aus mehreren existierenden Modellen bzw. Modellierungssprachen notwendig, da keine existierende Modellklasse exakt auf das Anwendungsfeld Kompatibilitätsmodellierung und -bewertung passt. Für die domänenübergreifende Modellierung und Bewertung von Kompatibilität von eingebetteten softwarelastigen Systemen ist eine Kombination aus Beschreibungs- und Prognosemodellen notwendig, in denen sowohl statische als auch dynamische Aspekte eines Systems qualitativ und quantitativ modelliert und beschrieben werden können. Zusätzlich zur Beschreibung der Systemeigenschaften sowie der Verbindungen zwischen den Teilsystemen des Systems ist eine Bewertung des Kompatibilitätstests notwendig. Aus diesem Grund muss das Kompatibilitätsmodell zusätzlich zu den Beschreibungseigenschaften bestimmte Prognoseeigenschaften besitzen, wodurch das Modell anhand von Kompatibilitätsregeln bewertet werden kann. In der nachfolgenden Aufzählung sind einige der grundsätzlichen Anforderungen an ein Modell zur Kompatibilitätsbewertung aufgelistet.

- Das Modell muss **domänenübergreifend** eingesetzt werden können.
- Das Modell muss sowohl den **klassischen** als auch den **objektorientierten Modellierungsansatz** unterstützen[48].
- Das Modell muss in der Lage sein **hierarchische** Systemstrukturen zu modellieren.
- Das Modell muss die Modellierung von **Komponenten, Schnittstellen** und **Verbindungen** zwischen den Schnittstellen unterstützen.
- Sowohl **statische** als auch **dynamische** Eigenschaften müssen modelliert werden können.
- Das Modell muss sowohl die Modellierung von **qualitativen** als auch **quantitativen** Eigenschaften unterstützen.
- Im Modell müssen **Kompatibilitätsregeln** hinterlegt werden können.
- Aufbauend auf den Kompatibilitätsregeln muss das Modell auf **Kompatibilität** untersucht werden können.

Aus diesen unterschiedlichen Anforderungen an ein Modell (bzw. an eine Modellierungssprache) für Kompatibilität von eingebetteten softwarelastigen Systemen folgt unmittelbar, dass ein generisches Kompatibilitätsmodell bzw. eine Kompatibilitätsmodellierungssprache eine Kombination aus unterschiedlichen Modellklassen sein muss. Am besten ist eine Kombination aus Beschreibungs- Erklärungs- sowie Entscheidungs- und Prognosemodellen für die Kompatibilitätsmodellierung und -bestimmung geeignet, in denen sowohl statische als auch dynamische Aspekte sowie quantitative Werte des Systems enthalten sind. Zusätzlich müssen im Modell die Kompatibilitätsregeln hinterlegt werden können[49].

2.2.1. Unterschiedliche Modellbildungsparadigmen

Im Kapitel „*Modellbildung*“ wurde sowohl der generische Modellbildungsprozess nach Dr. Negele, als auch der vereinfachte, daraus abgeleitete Modellbildungsprozess vorgestellt, mit deren Hilfe sich sowohl die Struktur eines Systems, als auch das Systemverhalten systematisch in ein Modell transferieren lassen. Diese Transformation der Systemdaten eines realen Systems in ein – im Allgemeinen abstraktes – Systemmodell ist vor allem für die Kompatibilitätsbestimmung eines Systems von entscheidender Bedeutung, da in einem Modell des Systems sowohl dessen statische Struktur, als auch das dynamische Verhalten des Systems so hinterlegt werden

[48] Siehe hierzu: Kapitel „2.2.1 *Unterschiedliche Modellbildungsparadigmen*“ ab Seite 38..
[49] Weitere Anforderungen an eine Kompatibilitätsmodellierungssprache sind im Kapitel „3.2 *Anforderungen an eine Kompatibilitätsmodellierungssprache*“ ab Seite 109 zusammengefasst.

muss, dass es alle Anforderungen an die Kompatibilitätsbestimmung erfüllt[50]. Im Systemmodell müssen demnach sämtliche statischen Eigenschaften des zu modellierenden Systems enthalten sein, wie beispielsweise alle Teilsysteme (Komponenten), die statischen Eigenschaften der Teilsysteme des Systems sowie deren Schnittstellen und Verbindungen untereinander. Für die dynamische Kompatibilitätsbestimmung muss darüber hinaus das Verhalten jedes einzelnen Teilsystems genau (zum Beispiel mittels Zustandsautomaten oder *Message Sequence Charts* (MSCs)) spezifiziert und im Modell des Systems hinterlegt werden.

Um sowohl das statische als auch das dynamische Systemmodell erstellen zu können, gibt der allgemein gültige Modellbildungsprozess die einzelnen Schritte exakt vor, ohne jedoch vorzuschreiben, auf welche Art und Weise sowohl das statische als auch das dynamische Systemmodell erzeugt werden soll. In der folgenden Aufzählung werden aus diesem Grund zwei unterschiedliche Arten der Zerlegung eines Systems vorgestellt.

Logische und physikalische Systemdekomposition
Sehr eng mit dem Modellbildungsprozess bzw. der Hierarchiebildung[51] ist die Dekomposition eines komplexen Systems in handhabbare Teile verwandt. Bereits Gaius Julius Cäsar stellte das Paradigma „teile und herrsche (Divide et impera!)[52]“ auf, da er erkannt hatte, dass sich kleine Einheiten besser kontrollieren und leiten lassen als große. Das selbe Prinzip lässt sich auch auf technische Systeme anwenden. Um ein komplexes technisches System beherrschbar zu gestalten, muss dieses in kleinere Teilstücke zerlegt werden. Dieser Vorgang wird solange iterativ wiederholt, bis das gesamte System in handhabbare Teilsysteme unterteilt worden ist, die mit den Mitteln der Modellbildung beherrscht werden können. Diese Unterteilung eines technischen Systems kann zum einen logisch, zum anderen physikalisch geschehen.

- **Logische Systemdekomposition**
  Bei der logischen Dekomposition eines Systems werden logisch zusammengehörigen Systembestandteile in einer größeren Einheit (Pakete) zusammengefasst. Dabei spielt der physikalische Aufbau oder die Struktur des Systems eine untergeordnete Rolle. Ein Beispiel für die logisch-hierarchische Dekomposition eines Systems ist die funktionale Dekomposition eines Systems. Bei der funktionalen Dekomposition wird ein System als „Funktionsbaum [ZSL, 22ff]“ dargestellt, in dem, ausgehend von der Wurzel, die Funktionen des Systems immer feiner spezifiziert werden.

- **Physikalische Systemdekomposition**
  Im Gegensatz zur logischen Dekomposition eines Systems, werden bei der physikalischen Unterteilung des Systems die physikalischen Zusammenhänge und Eigenschaften des Systems in den Vordergrund der Modellierung gestellt. Die hierarchische Dekomposition eines physikalische Systems wird vor allem bei der hardwarenahen Modellierung eines Systems eingesetzt.

Beide Systemdekompositionsmethoden lassen sich sowohl in der klassischen als auch in der objektorientierten Modellbildung einsetzen, um die hierarchische Strukturierung eines Systems vorzunehmen. Auch bei der Dekomposition von Systemen sind die Übergänge zwischen den beiden Extremen wieder fließend, wie schon bei der Klassifikation der Modelle. Dies liegt zum einen in der Tatsache begründet, dass moderne Systeme meistens aus Hard- und Software bestehen, die von unterschiedlichen Entwicklungsteams unabhängig voneinander und mit unterschiedlichen Methoden entwickelt werden. Und zum anderen daran, dass es manchmal vorkommen kann, dass bestimmte Systembestandteile besser logisch manche besser physikalisch unterteilt und gruppiert werden können. Daher sind in der Praxis meistens Kombinationen aus beiden Paradigmen anzutreffen.

Nach der Vorstellung der beiden unterschiedlichen Systemdekompositionsmethoden folgt nun die Beschreibung zweier zentraler Modellbildungsparadigmen: Die klassische Modellbildung, so wie sie das Systems Engineering vorschlägt, sowie das aus der Softwareentwicklung stammende objektorientierte Modellierungskonzept.

Vorgehen bei der klassischen Modellbildung
Beim klassischen Modellbildungsprozess wird ein System schrittweise, ausgehend vom gesamten System, von oben nach unten („top-down“) in seine einzelnen Funktionen bzw. Funktionsgruppen heruntergebrochen. Dieser Vorgang wird auch als *funktionale Dekomposition* bzw. als *funktionale Abstraktion* eines Systems bezeichnet[53]. Nachdem die einzelnen Funktionen des zu modellierenden Systems identifiziert worden sind, werden diese logisch gruppiert und einzelnen physikalischen Teilsystemen zugeordnet. In der nachfolgenden Aufzählung sind die wesentlichen vier Schritte des Vorgehens bei der klassischen Modellbildung aufgeführt und anhand des Beispielsystems Kaffeemaschine erläutert[54].

1. *Identifikation der Funktionen und Eigenschaften des zu modellierenden Systems*
   Der wohl wichtigste Schritt bei der (klassischen) Modellbildung ist die Identifikation der wesentlichen *Eigenschaften* und *Funktionen* des zu modellierenden Systems. Dabei wird das Hauptaugenmerk auf die Identifikation der Eigenschaften und Funktionen des Systems gelegt, die für die Modellierung sowie die anschließende Simulation des Systems notwendig sind. Nach der Identifikationsphase der Eigenschaften und Funktionen des Systems wird versucht, diese logisch, zum Beispiel in einer Baumstruktur (Funktionsbaum), zu gruppieren. Nachdem sämtliche Eigenschaften und Funktionen des Systems identifiziert und eingeordnet worden sind, folgt im nächsten Schritt die Modellierung der Struktur des Systems.

   Das bereits im Kapitel „*Grundlegende Begriffe und Definitionen aus der Systems Engineering Welt*“ eingeführte Beispielsystem der Kaffeemaschine besitzt vier Eigenschaften, sowie die nicht explizit dargestellten Funktionen *Kaffee kochen* und *Kaffee warmhalten*.

[50] Siehe hierzu auch „Grundeigenschaften eines Modells zur Modellierung und Beschreibung von Kompatibilität von eingebetteten softwarelastigen Systemen“ auf Seite 38.
[51] Vgl. „Definition: 1.5“ auf Seite 13..
[52] Nach [Tho08], die verkürzte Version von „Teile deine Gegner, und es fällt dir leichter, sie zu beherrschen!“.
[53] Siehe hierzu auch [PDB98a].
[54] Eine ausführliche Beschreibung des Vorgehens im klassischen Modellbildungsprozess finden Sie unter [Eng07], [DIQ01], [DIS99] sowie [Büc83] und [KS03].

2. *Aufbau eines Strukturmodells*
   Nachdem im ersten Schritt die Eigenschaften und Funktionen des Systems identifiziert wurden, folgt in diesem Schritt der Aufbau des Strukturmodells. Dazu muss zunächst eine geeignete Modellierungssprache ausgewählt werden, mit deren Hilfe das System modelliert werden soll. Eine Auswahl an unterschiedlichen Modellklassen, Modellierungssprachen und -methoden wird sowohl in den beiden Kapiteln „2.2 *Modellbildung*“ ab Seite 31 und „3 *Modellierungssprachen und -techniken für technische Systeme*“ ab Seite 109, als auch in der Veröffentlichung „The Design-Methods Comparison Project [PDB98a]“ abgehandelt. Nachdem eine geeignete Modellierungssprache ausgewählt worden ist, werden in diesem Modellierungsschritt sämtliche Strukturelemente des zu modellierenden Systems identifiziert und in das Systemmodell aufgenommen. In dem Modell, genauer im Strukturmodell, werden sämtliche strukturbildenden Teilsysteme des zu modellierenden Systems eingetragen, die später die Eigenschaften und Funktionen des Systems beinhalten sollen. Im Allgemeinen wird dabei top-down vorgegangen, also ausgehend vom Gesamtsystem wird das System in seine (logischen/physikalischen) Teilsysteme heruntergebrochen, bis sämtliche Teilsysteme des zu beschreibenden Systems im Modell enthalten sind. Dabei wird versucht, die endgültige Struktur des gesamten Systems vollständig zu beschreiben.

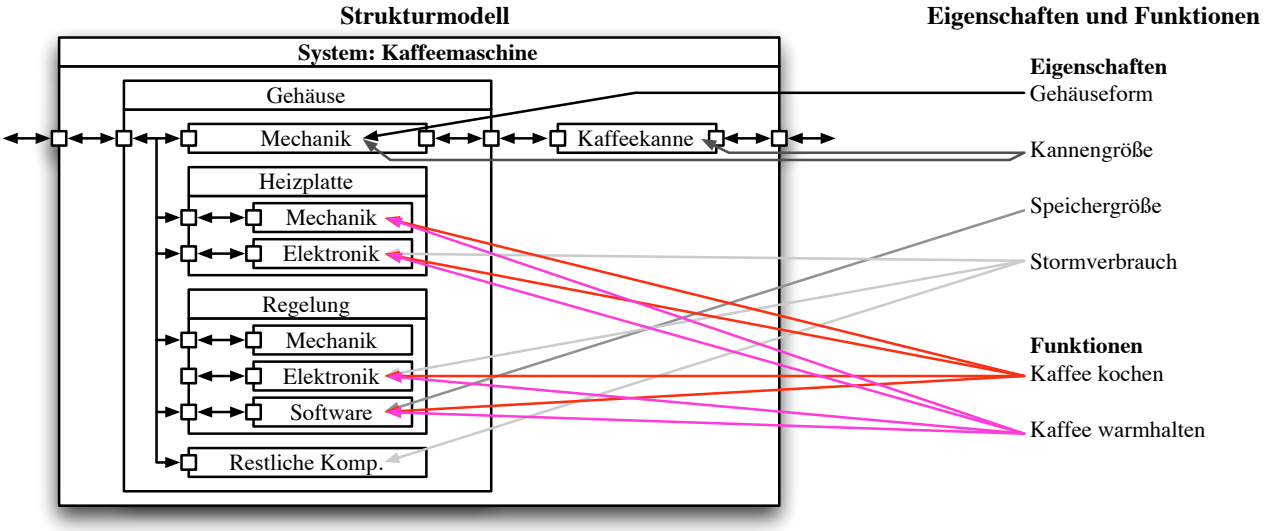


Abbildung 2.8.: Zuweisung der Eigenschaften und Funktionen an die Strukturelemente des Systems Kaffeemaschine.

Die Abbildung 2.8 zeigt links das stark vereinfachte Strukturmodell des Beispielsystems Kaffeemaschine. In diesem Modell ist eine Auswahl der wesentlichen Strukturelemente des Systems Kaffeemaschine enthalten. Zusätzlich zu den Elementen des Systems sind in dieser Abbildung sowohl die Schnittstellen, als auch die Verbindungen zwischen den einzelnen Schnittstellen der Elemente des Systems eingetragen. Zusammen bilden sie das Strukturmodell des Systems Kaffeemaschine.

3. *Zuweisen der in 1 gefundenen Funktionen und Eigenschaften an Strukturelemente*
   Im dritten Schritt werden die unter 1 identifizierten Eigenschaften sowie die (logischen) Funktionen des Systems konkreten Strukturelementen, die unter 2 identifiziert worden sind, zugewiesen. Dabei kann es vorkommen, dass eine Eigenschaft oder eine Funktion über mehrere Strukturelemente verteilt wird. Die Abbildung 2.8 zeigt, auf der linken Seite der Abbildung das bereits bekannte Strukturmodell des Systems Kaffeemaschine. Rechts daneben sind sämtliche identifizierten Eigenschaften und Funktionen der Kaffeemaschine aufgelistet und mit Hilfe von farbigen Pfeilen bestimmten Teilsystemen des Strukturmodells zugewiesen worden. Dabei fällt auf, dass beide Funktionen des Systems von jeweils zwei Teilsystemen gemeinsam wahrgenommen werden. Im Allgemeinen wird jedoch versucht, eine dezidierte Systemeigenschaft, oder eine bestimmte Funktion lediglich einem Element des Strukturmodells zuzuweisen, um dadurch die Flexibilität bzw. die Erweiterbarkeit des Systems einfacher zu gestalten.

   Für die Kompatibilitätsmodellierung ist es von besonderem Interesse, dass einzelne Systemeigenschaften und Funktionen jeweils nur von einem dezidierten Teilsystem wahrgenommen werden, da es ansonsten zu großen Problemen beim Austausch einer Komponente (Teilsystems) gegen eine andere kommen kann[55].

4. *Simulation des Modells des Systems*
   Der letzte Schritt der klassischen Modellbildung besteht in der Simulation des Modells. Dabei wird das Modell des Systems durch einen Simulator simuliert und die Simulationsergebnisse mit den vorher festgelegten Werten verglichen. Stimmen beide überein, so gilt das Modell des Systems als valide.

Eine weitere Verfeinerung des klassischen Modellbildungsprozesses stellen A. Kossiakoff und W. N. Sweet in ihrem Buch „Systems Engineering – Principles and Practice –“ vor. Im zweiten Kapitel ([KS03, 31ff]) ihres Buches beschreiben sie eine einfache Erweiterung des klassischen Modellbildungsansatzes, bei dem ein System sowohl in *funktionale* als auch *physikalische Blöcke* unterteilt wird. Diese Unterteilung eines technischen Systems soll es allen, an der Systementwicklung beteiligten Ingenieuren ermöglichen, sämtliche Aspekte eines Systems zu erfassen und diese in einem abstrakten Systemmodell abbilden zu können. Das abstrakte Systemmodell dient dann als Grundlage für die anschließende konkrete Implementierung in den einzelnen Fachabteilungen.

[55] Für weitere Informationen zur Identifikation und Zuordnung von Funktionen zu Strukturelementen siehe [DIP08].

- **Functional Building Blocks: Functional Elements** [KS03, 36]
  Begonnen wird die Modellierung eines technischen Systems mit der Identifikation der funktionalen Blöcke des zu beschreibenden Systems. Dabei werden die einzelnen funktionalen Blöcke weiter nach ihrem Verwendungszweck innerhalb des Systems unterteilt und eingruppiert. Die drei wesentlichen Gruppen nach denen sich sämtliche Eigenschaften und Funktionen eines Systems einteilen lassen sind:

  – *Information* (Information): Hierunter wird sowohl die Informationsverarbeitung, als auch der Informationsfluss innerhalb eines Systems zusammengefasst.

  – *Material* (Material): Unter Material werden sämtliche physikalischen Objekte zusammengefasst, die für die Modellierung des Systems notwendig sind.

  – *Energie* (Energy): Zusammenfassung sämtlicher Energiearten, die innerhalb eines Systems verwendet werden.

  Dabei gilt insbesondere, dass sämtliche identifizierten funktionalen Elemente des Systems folgenden Anforderungen genügen müssen:

  – *Aussagekraft/Stellenwert* (Significance): Jede Eigenschaft/Funktion des Systems muss eine bestimmte Bedeutung innerhalb des Systems aufweisen um im Modell aufgenommen zu werden.

  – *Einzigartigkeit* (Singularity): Jede Eigenschaft/Funktion sollte möglichst nur innerhalb einer Fachdisziplin verwendet werden. Überlappungen sollten vermieden werden.

  – *Allgemeingültigkeit* (Commonality): Jede Eigenschaft/Funktion sollte möglichst so verwendet/benutzt werden, so wie sie auch innerhalb anderer Systeme verwendet/benutzt wird.

  Das Ergebnis der functional blocks Analyse wird in eine Tabelle eingetragen, in der die identifizierten Eigenschaften und Funktionen des Systems nach ihrer Kategorie gruppiert werden. Zusätzlich sollte die Tabelle eine Beschreibung jeder Eigenschaft bzw. jeder Funktion enthalten. Diese Tabelle wird im zweiten Schritt, der physikal building blocks Analyse, als Grundlage für das physikalische Strukturmodell verwendet.

- **Physical Building Blocks: Components** [KS03, 38]
  Nachdem die Elemente und Funktionen des Systems im letzten Schritt identifiziert und in tabellarische Form gebracht wurden, folgt nun die Zuordnung der Eigenschaften und Funktionen zu physikalischen Blöcken (Teilsystemen/Komponenten) des Systems. Dabei repräsentieren die physikalischen Blöcke die entsprechenden Eigenschaften und Funktionen des Systems. Daraus folgt, dass die physikalische Struktur des Systems den zuvor festgelegten Gruppen – Information, Material und Energie – nachempfunden wird. Die Zuordnung der zuvor identifizierten Eigenschaften und Funktionen zu physikalischen Strukturen geschieht wiederum in tabellarischer Form.

Ergebnis: Sowohl einfache, als auch komplexe technische Systeme können sowohl mit Hilfe des klassisches Vorgehen bei der Modellbildung als auch mittels des erweiterten Modellbildungsprozesses modelliert und dargestellt werden. Dabei treten jedoch häufig Probleme auf, sobald Systeme, die aus Hard- und Software bestehen, modelliert werden müssen. Dies kommt daher, dass in der Informatik ein grundsätzlich anderes Modellbildungsparadigma, die objektorientierte Modellierung, verwendet wird, bei der nicht die Eigenschaften und Funktionen eines Systems im Vordergrund stehen, sondern das Objekt.

Ein weiteres grundsätzliches Problem der klassischen Modellbildung liegt in der Art der Zuordnung zwischen Eigenschaften und Funktionen sowie der Abbildung dieser auf reale Komponenten des späteren Systems. Dabei kann es, wie oben gezeigt, vorkommen, dass eine Eigenschaft oder Funktion des Systems von mehr als einer realen Komponente umgesetzt wird. Diese „Zerstückelung“ von Systemeigenschaften bzw. -funktionen führt im Allgemeinen dazu, dass sowohl die Erweiterbarkeit, als auch die Wartbarkeit des Systems erheblich in Mitleidenschaft gezogen werden. Aus diesem Grund hat, in den vergangenen Jahren das objektorientierte Modellbildungsparadigma ebenfalls Einzug in die klassische funktionsorientierte Modellbildung gehalten. Bereits Dr. Quirmbach [DIQ01, 46ff] hat in seiner Dissertation das objektorientierte Modellbildungskonzept eingeführt und an die speziellen Anforderungen des Systems Engineerings, und insbesondere an das Systems Engineering Element-Konzept[56] angepasst. Im Anschluss daran hat Dr. L. Schrepfer [DIS99, 20ff], das von Dr. Quirmbach vorgeschlagene Modellbildungskonzept weiter verfeinert und um weitere Konzepte aus der objektorientierten Programmierung angereichert.

Vorgehen bei der objektorientierten Modellbildung
Nachdem – im letzten Jahrzehnt – das objektorientierte Vorgehen bei der Softwareentwicklung an Fahrt aufgenommen und in den vergangenen zwei bis drei Jahren schließlich die Oberhand gewonnen hat, ist die objektorientierte Modellbildung von Systemen jetzt auch im eher klassischen Systems Engineering Umfeld angekommen. Dies liegt zum einen daran, dass beispielsweise immer mehr eingebettete Systeme, bestehend aus mechanischen, elektrischen sowie Softwarekomponenten in modernen Produkten eingesetzt werden, bei denen an der Systementwicklung Ingenieure aus den unterschiedlichsten Fachdisziplinen beteiligt sind, um das Produkt gemeinsam zu entwickeln. Zum anderen werden moderne Produkte selten neu entwickelt, sondern sind Varianten bereits existierender Produkte. Um diese Art der Produktentwicklung möglichst effizient unterstützen zu können, ist das klassische funktionsorientierte Vorgehensmodell bei der Systementwicklung denkbar ungeeignet, da es nicht in der Lage ist, Informationen und Daten aus Vorgängerprodukten ohne große Anpassungen in ein neues Produkt zu überführen.

Eine einfache und kostengünstige Lösung der beiden oben beschriebenen Probleme bietet das, ursprünglich aus der Softwareentwicklung stammende, objektorientierte Vorgehen auch bei der Systementwicklung. Zum einen können dadurch Ingenieure aus unterschiedlichen Fachbereichen an ein und demselben Modell des Systems arbeiten, ohne ständig die eigene Sichtweise auf ein Teilsystem dem Gesamtmodell anpassen zu müssen. Und zum anderen werden die Ingenieurteams durch die objektorientierte

[56] Nähere Informationen zum Systems Engineering Element-Konzept finden Sie im Kapitel „3.4.2 *Das Systems Engineering Element-Konzept – „Die Münchner Schule““* ab Seite 116.

Modellbildung bei der Weiterentwicklung von Produkten maßgeblich unterstützt. Aus diesen beiden Gründen erfreut sich die *Object-Oriented Systems Engineering Method* (OOSEM)[57] auch im Systems Engineering Umfeld immer größerer Akzeptanz und Beliebtheit. Die Object-Oriented Systems Engineering Method vereinigt dabei das klassische top-down Vorgehen bei der klassischen Systembeschreibung, so wie es im Systems Engineering üblich ist, mit den modernen Methoden der Objektorientierung. Dabei nutzt die OOSEM unter anderem die Konzepte und Techniken der *Unified Modelling Language* (UML/UML2)[58] sowie der speziell für das Systems Engineering entwickelte *System Modelling Language* (SysML)[59].

Ziele der objektorientierten Systemgestaltung
In der nachfolgenden Aufzählung sind die wesentlichen Ziele der objektorientierten Systemgestaltung aufgelistet, die sowohl für die Softwareentwicklung, als auch unverändert für das objektorientierte Systems Engineering gültig sind.

- **Modularität**
  Ein System besteht aus unterschiedlichen Modulen (Teilsystemen). Die Summe aller Module bildet das System.
- **Uniformität und Systematik**
  Die Prinzipien der Uniformität und Systematik besagen, dass ein System aus gleichartigen Grundbausteinen (Elementen, Objekten, Klassen, Komponenten etc.) aufgebaut ist. Jedes Modell eines Systems besteht demnach aus den gleichen Bestandteilen, wodurch die Flexibilität und Wiederverwendbarkeit von Modulen des Systems erheblich erleichtert wird.
- **Änderbarkeit und Flexibilität**
  Ein System sollte stets so strukturiert und aufgebaut sein, dass es sich einfach ändern und anpassen lässt. Wenn ein System leicht änderbar bzw. wartbar ist, so ist es auch flexibel.
- **Wiederverwendbarkeit**
  Alle Elemente (Klassen, Methoden etc.) eines Systems sollten stets so ausgelegt werden, dass sie auch in einem anderen System ohne Änderungen wiederverwendet werden können. Durch die konsequente Nutzung der Wiederverwendbarkeit von Elementen des Systems wird nicht nur die Qualität des Systems, sondern auch dessen Entwicklung bzw. Anpassung an neue Anforderungen erheblich verbessert.

Durch die konsequente Anwendung der obigen Ziele, wird es den Systemingenieuren aus den unterschiedlichen Domänen ermöglicht, schnell und effizient komplexe Systeme zu entwickeln. Dabei geht die objektorientierte Systementwicklung stets in den folgenden Phasen vor[60].

Objektorientierte Analyse (OOA – Object-Oriented Analysis)
Der erste Schritt bei der objektorientierten Modellbildung ist die *objektorientierte Analyse*. Dabei wird zuerst ein Informationsmodell aufgebaut, in dem sämtliche realen Objekte des Systems, deren Kardinalität, die Zusammenhänge zwischen den Objekten des Systems (Systemhierarchie) sowie die Verbindungen zwischen den Objekten identifiziert und eingetragen werden. Zusätzlich zu den realen Objekten des Systems werden die Anforderungen an das zu modellierende System in das Informationsmodell aufgenommen. Das Informationsmodell wird oft auch als „Problembeschreibung“ oder „Lastenheft“ bezeichnet. Nachdem das Informationsmodell fertig gestellt worden ist, folgt im nächsten Modellierungsschritt die Umsetzung der Informationen in ein konkretes (Klassen-) Modell. Dabei wird versucht, aus den realen Objekten Klassen zu bilden, die miteinander in logischen Beziehungen stehen.

Nachdem sämtliche Objekte des Systems in Klassen überführt worden sind, wird für jede nichttriviale Klasse des Systems ein Zustandsautomat angefertigt, mit dessen Hilfe das dynamische Verhalten der Klasse beschrieben wird. Zusätzlich zur Beschreibung des Verhaltens der Klassen, wird die Interaktion zwischen den einzelnen Klassen des Systems mittels Zustandsautomaten modelliert.

- **Identifikation der Objekte eines Systems**
  Der erste Schritt bei der objektorientierten Modellierung eines Systems besteht in der Identifikation sämtlicher (realer) Objekte des Systems. Dabei wird stets versucht, die gefundenen Objekte in eine Klassenstruktur einzuordnen. Zusätzlich werden Klassen, die bestimmte Gemeinsamkeiten besitzen in Verbindung zueinander gebracht. Diese Verbindung kann zum Beispiel eine „enthalten sein Relation“ oder eine „Vererbungsbeziehung“ sein[61].

  Das Beispielsystem Kaffeemaschine besteht auf der obersten Ebene aus zwei physikalischen Objekten: Dem Gehäuse der Kaffeemaschine, sowie einer Kanne. Dabei kann die Kaffeemaschine weiter unterteilt werden in eine Regelung, eine Heizplatte sowie die restlichen Komponenten die nicht weiter betrachtet werden sollen. Dieser hierarchische Aufbau der Kaffeemaschine kann im Klassendiagramm mit Hilfe von enthalten sein Beziehungen modelliert und dargestellt werden. Die Abbildung 2.9 auf der nächsten Seite zeigt das stark vereinfachte Klassenmodell der Kaffeemaschine.
- **Identifikation der Eigenschaften und Funktionen einer Klasse/Objekts**
  In diesem Modellierungsschritt wird für jede zuvor identifizierte Klasse/Objekt die Eigenschaften und Funktionen der Klasse/Objekts festgelegt.

  Für die Kaffeemaschine werden nun exemplarisch, für die beiden Objekte *Regelung* und *Heizplatte*, Eigenschaften und Funktionen festgelegt. Die Regelung hat eine Eigenschaft Speichergröße für die Regelungssoftware, eine Eigenschaft Stromverbrauch,

[57] Siehe hierzu die Veröffentlichung [LFPM00], sowie die Web-Seite [Tur08].
[58] Eine knappe Einführung in die Modellierungssprache UML/UML2 finden Sie im Kapitel „3.4.3 *Die Unified Modelling Language – UML/UML2*“ ab Seite 131, sowie unter [CS04], [PDB05], sowie [BRJ98].
[59] Eine knappe Einführung in die Modellierungssprache SysML finden Sie im Kapitel „3.4.4 *Die System Modelling Language – SysML* “ ab Seite 144, sowie unter [Wei06] und [Sys06].
[60] Das Vorgehen bei der objektorientierten Modellbildung wird manchmal auch als Modellgetriebene Architektur – Modell-driven-architecture (MDA)[CS04, 33,208] bezeichnet.
[61] In den beiden Kapiteln „2.3.1.1 *Klassen und Objekte eines Systems*“ ab Seite 46 und „3.4.3 *Die Unified Modelling Language – UML/UML2*“ ab Seite 131 werden die einzelnen Beziehungsarten genauer beschrieben.

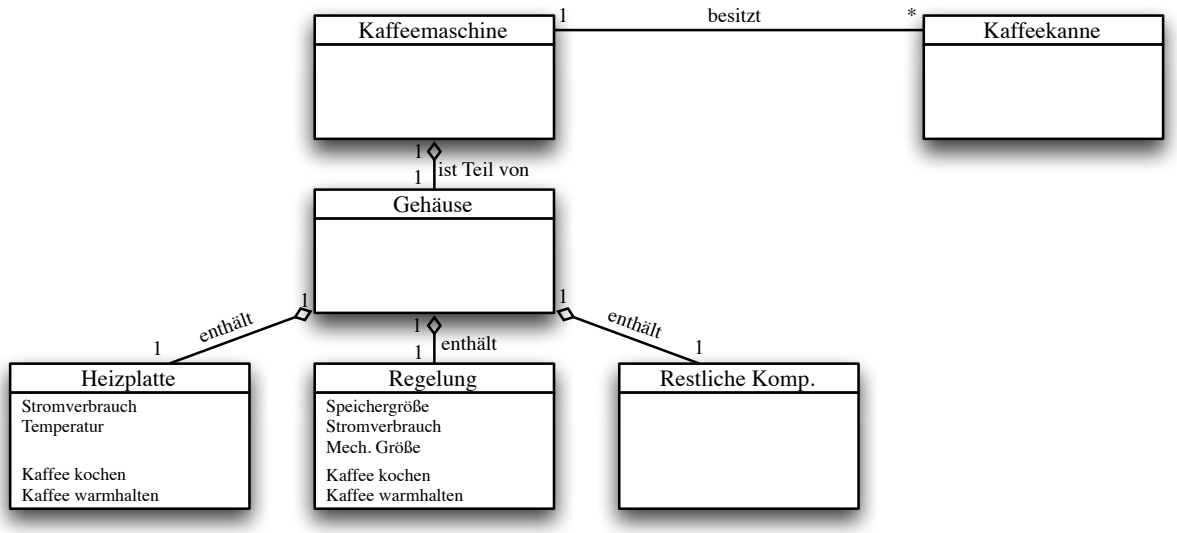


Abbildung 2.9.: Vereinfachtes Klassenmodell der Kaffeemaschine.

sowie eine mechanische Größe. Außerdem hat das Objekt Regelung die beiden Funktionen Kaffee kochen und Kaffee warmhalten. Die Heizplatte hat die Eigenschaften mechanische Größe, Stromverbrauch und Temperatur sowie die beiden Funktionen Kaffee kochen und Kaffee warmhalten.

- **Verbindungen zwischen den Klassen/Objekten des Systems**
  Im vorletzten Modellbildungsschritt wird versucht, Verbindungen zwischen den einzelnen Objekten/Klassen des zu modellierenden Systems zu identifizieren und diese in das zuvor erstellte Klassenmodell einzutragen. Darüber hinaus wird im Allgemeinen jeder modellierten Verbindung eine Kardinalität zugewiesen, anhand derer die Häufigkeit einer Klasse innerhalb der Beziehung angegeben wird.

  Im obigen Klassenmodell der Kaffeemaschine wurden verschiedene Beziehungen zwischen den unterschiedlichen Klassen des Systems eingetragen. Zum Beispiel ist zwischen der Klasse *Kaffeemaschine* und der Klasse *Gehäuse* eine „ist Teil von“ Beziehung modelliert, die ausdrücken soll, dass das Gehäuse ein Teil der Kaffeemaschine ist. Zwischen der Kaffeemaschine und der Kaffeekanne ist eine „besitzt“ Beziehung modelliert, die zum Ausdruck bringt, dass zu einer Kaffeemaschine mindestens eine Kaffeekanne gehört.

- **Verhaltensbeschreibung der Klassen**
  In den vorangegangenen drei Schritten wurde ausgehend vom realen System ein Klassenmodell erstellt, in dem sämtliche Eigenschaften und Funktionen jeder betrachteten Klasse hinterlegt worden sind. Zusätzlich wurden die Verbindungen zwischen den Klassen modelliert. In diesem Modellierungsschritt wird das statische Klassenmodell um eine Verhaltenskomponente erweitert. Dazu wird die gewünschte Funktionalität jeder nichttrivialen Klasse mit Hilfe von Zustandsautomaten modelliert. Zusätzlich werden in diesem Schritt die Interaktionen zwischen den Klassen des Systems modelliert und ebenfalls mittels Zustandsdiagrammen beschrieben.

  Für das System Kaffeemaschine wird nun ein stark vereinfachter Zustandsautomat angegeben, mit dessen Hilfe das Verhalten des Systems auf oberster Ebene beschrieben wird. Die Abbildung 2.10 zeigt den Zustandsautomaten der Kaffeemaschine.

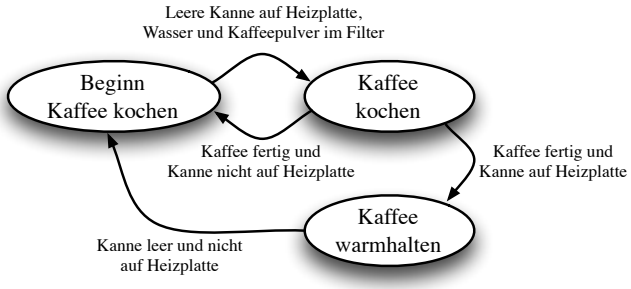


Abbildung 2.10.: Stark vereinfachter Zustandsautomat der Kaffeemaschine.

Der obige Automat zeigt graphisch, wie die einzelnen Zustände der Kaffeemaschine zusammenhängen. Dabei repräsentieren die Ellipsen die Zustände der Kaffeemaschine während die Pfeile (Transitionen) den Übergang von einem Zustand in einen anderen anzeigen. Zusätzlich ist an jedem Pfeil eine (logische bzw. Bool´sche) Bedingung angegeben, die festlegt, unter welcher Bedingung die Kaffeemaschine von einem Zustand in einen anderen übergeht. Beispielsweise führt ein Pfeil vom

Zustand *Kaffee kochen* in den Zustand *Kaffee warmhalten*. Diese Transition wird ausgelöst, wenn die Bedingung *Kaffee fertig und Kanne auf Heizplatte* erfüllt ist. Ist die Bedingung nicht erfüllt, so verbleibt der Automat solange im Zustand Kaffee kochen, bis die Bedingung erfüllt ist.

Nachdem die objektorientierte Analyse des Systems abgeschlossen ist, folgt im nächsten Schritt die Verfeinerung bzw. die Umsetzung des Klassenmodells in ein reales System.

Objektorientiertes Design (OOD – Object-Oriented Design) und objektorientierte Programmierung (OOP – Object-Oriented Programming)
Während der objektorientierten Analyse wurde das zu modellierende System analysiert und das Ergebnis der Analyse in ein Klassendiagramm eingetragen. Aufbauend auf diesem Klassendiagramm wird während des *objektorientierten Designs* das Klassenmodell weiter verfeinert und um zusätzliche Basisklassen ergänzt, die für die anschließende Implementierungsphase benötigt werden. Nach Abschluss des objektorientierten Designs folgt die Umsetzung des erweiterten Klassenmodells in der *objektorientierten Programmierung*. Dabei werden zum Beispiel Teile des Klassenmodells aus einer objektorientierten Programmiersprache in ein lauffähiges Programm übersetzt, während andere Systembestandteile in Hardware ausgeführt werden. Abgeschlossen wird die objektorientierte Programmierung mit der Integration aller Systembestandteile zu einem vollständigen realen System.

Vergleich von klassischer und objektorientierter Modellbildung
Für die abstrakte Modellierung von komplexen technischen Systemen kann sowohl das klassische als auch das objektorientierte Modellbildungsparadigma gleichermaßen eingesetzt werden. Das objektorientierte Modellbildungsparadigma ist dabei, sowohl laut Dr. L. Schrepfer ([DIS99, 10]), als auch Prof. A. T. Bahill (Ph.D.) ([PDB98a]) besser für die Modellierung von technischen Systemen, die aus Hard- und Software bestehend, geeignet, da sich aus objektorientierten Modellen leichter Quellcode bzw. das reale physikalische System erzeugen/bauen lässt, als dies bei klassischen funktionsorientierten Modellen der Fall ist. Darüber hinaus „denkt“ der Mensch stets in Objekten. Daraus folgt unmittelbar, dass die objektorientierte Modellierung nicht erst erlernt werden muss, sondern ohne große Übung auf unterschiedliche Anwendungsgebiete angewendet werden kann.

Die Tabelle 2.1 zeigt kompakt die Gemeinsamkeiten und Unterschiede zwischen der klassischen funktionsorientierten und der objektorientierten Modellbildung von Systemen anhand von Schlagworten auf.

	Funktionale Modellbildung	**Objektorientierte Modellbildung**
Modellierung von Hard- und Software	Ja	Ja (mit Einschränkungen)
Einfache Wiederverwendbarkeit von Modellen	Teilweise (modellabhängig)	Ja
Durchgänge Modellierung	Ja	Ja
Kapselung von Daten und Funktionen	Nein	Ja
Flexibilität der Modelle	Teilweise (modellabhängig)	Ja
Modellierung von Kompatibilität	Ja	Ja

Tabelle 2.1.: Gegenüberstellung von klassischer und objektorientierter Modellbildung

Für die Kompatibilitätsmodellierung und -bestimmung kann ebenso wie für die Modellierung von technischen Systemen sowohl das klassische funktionsorientierte Vorgehen, als auch das objektorientierte Modellbildungsparadigma angewendet werden. Auch hier gilt wieder der obige Sachverhalt: Technische Systeme bestehend aus Hard- und Software, und insbesondere eingebettete Systeme, lassen sich einfacher mit Hilfe des objektorientierten Vorgehens beschreiben. Aus diesem Grund wird im weiteren Verlauf dieser Arbeit, sofern nicht explizit anderweitig angegeben, das objektorientierte Modellbildungsparadigma für die Modellbildung von eingebetteten softwarelastigen Systemen angewendet. Dies gilt insbesondere für das Kapitel „3 *Modellierungssprachen und -techniken für technische Systeme*“ ab Seite 109, in dem vier unterschiedliche Modellierungssprachen zur Modellierung und Bewertung der Kompatibilität von eingebetteten softwarelastigen Systemen vorgestellt werden.

2.2.2. Simulation von Modellen

Der letzte Schritt bei der Modellbildung, unabhängig ob das Modell mittels des klassischen funktionsorientierten oder der objektorientierten Modellbildung entstanden ist, ist die Validierung des Modells gegen die zuvor festgelegten Anforderungen an das Modell. Dies geschieht in der Regel durch die Simulation des entstandenen Modells. Dabei beschreibt der VDI, wie ein Modell simuliert werden sollte, damit es den Anforderungen an ein valides Modell genügt.

Definition 2.3 ***Simulation von Modellen nach VDI 3633***
In der Simulationstechnik versteht man unter ***Experiment*** *die* ***gezielte empirische Untersuchung*** *des Modellverhaltens durch wiederholte Simulationsläufe mit systematischen Parametervariationen.*

Laut der obigen Definition 2.3 ist die Simulation eines Modells eine wiederholte empirische Untersuchung sowohl der Struktur als auch des Verhaltens dieses Modells. Dabei werden die systematischen Parameter (Systemeigenschaften) des zu untersuchenden Modells mit den zuvor spezifizierten Anforderungen (Anforderungsparameter) befüllt und das Modell mit diesem Parameterset simuliert. Nach jedem Simulationslauf wird das Simulationsergebnis mit den Anforderungen, bzw. dem zu erwartenden Ergebnis verglichen. Tritt bei einer Simulation eine Abweichung von den zu erwartenden Ergebnissen oder ein Fehler auf, so wird der aktuelle Simulationsschritt abgebrochen und das Modell des Systems untersucht, um die Abweichung oder den Fehler im Modell zu identifizieren und zu beseitigen. Nach der Beseitigung der Abweichung bzw. des Fehlers wird der zuvor abgebrochene

Simulationsschritt noch einmal ausgeführt. Sind nun keine Abweichungen oder Fehler vom zu erwartenden Ergebnis mehr zu erkennen, so werden sämtliche Simulationen noch einmal ausgeführt, um eine unabsichtliche Fehlerausbreitung durch die Änderung am Modell zu überprüfen.

Zusätzlich zur Simulation eines Modells kann dieses durch (formale) Validierung weiter auf Korrektheit bzw. Übereinstimmung mit den Anforderungen an das Modell untersucht werden. Die nachfolgende Definition 2.4 beschreibt die wesentlichen Punkte, welche nach VDI bei der Validierung eines Modells nachgewiesen werden müssen.

Definition 2.4 ***Validierung von Modellen nach VDI 3633***
Überprüfen der ***hinreichenden Übereinstimmung*** *von* ***Modell*** *und* ***System****. Es ist sicherzustellen, dass das Modell das* ***Verhalten des Originalsystems*** *im Hinblick auf die* ***Untersuchungsziele genau genug und fehlerfrei widerspiegelt****.*

Durch die Validierung eines Modells, so wie sie der VDI in der Definition 2.4 vorschlägt, wird nachgewiesen, dass ein Modell eines Systems hinreichend genau mit dem realen System bzw. den Anforderungen an das Modell dieses Systems übereinstimmt. Das Modell bildet sowohl die Struktur als auch das Verhalten des Systems so genau nach, dass eine hinreichende Übereinstimmung zwischen dem Systemmodell und den Anforderungen an das Modell bzw. das reale System, erreicht wird. Dabei kommt es besonders auf die Festlegung der „Übereinstimmungsparameter“ an. Mit Hilfe dieser Parameter wird für jede Systemeigenschaft sowohl eine obere, als auch eine untere Schranke festgelegt, die angibt, welche Abweichung zwischen dem ermittelten Modellwert und dem zuvor festgelegten Sollwert akzeptiert wird. Über- bzw. unterschreitet der ermittelte Modellparameter diese Grenze, so liegt ein Fehler im Modell vor. Liefert beispielsweise der Modellparameters $L1$ des Transformators der Kaffeemaschine (vgl. Abbildung: 1.34 auf Seite 29) den Wert $L1 = +220\text{V}$ und wurde zuvor ein oberer bzw. unterer Grenzwert für die Abweichung der Eigenschaft $L1$ spezifiziert und festgelegt, zum Beispiel $+210 < L1 < +240\text{V}$, so kann durch die Validierung überprüft werden, ob der aktuelle Wert für die Eigenschaft $L1$ innerhalb der beiden Grenzen liegt. In diesem Fall liegt die Eigenschaft $L1$ innerhalb der zuvor spezifizierten Abweichung.

Die beiden VDI Definitionen lassen sich auch für die Kompatibilitätsprüfung eines Modells eines Systems in leicht modifizierter Art und Weise anwenden.

Zusammenhang zwischen Simulation und Kompatibilitätsprüfung
Für die Bestimmung der Kompatibilität eines Systems kommt ebenfalls eine Art Simulation und Validierung des Modells zum Einsatz. Dabei wird im ersten Schritt das statische System auf Kompatibilität untersucht, während im zweiten Schritt – ähnlich der Simulation des Systemmodells – die dynamische Kompatibilität des Modells untersucht wird. Dabei werden ständig sämtliche Simulationsergebnisse mit den zu erwartenden Ergebnissen der Simulation verglichen. Auch für die Kompatibilitätsprüfung gilt: Alle (Simulations-) Ergebnisse des Modells müssen hinreichend genau mit den zu erwartenden Ergebnissen übereinstimmen. Dabei kann mit der dynamischen Kompatibilitätsuntersuchung eines Modells erst begonnen werden, wenn die statische Kompatibilität des Modells gewährleistet ist[62]. Enthält das statische Modell Fehler, so kann die dynamische Kompatibilitätsuntersuchung nicht durchgeführt werden, da eine statische Inkompatibilität in der Regel auch eine dynamische Inkompatibilität nach sich zieht.

Ist zum Beispiel bei der Kaffeemaschine eine Verbindung zwischen zwei Teilsystemen nicht kompatibel, weil beispielsweise die Spannung des Senders größer ist, als die vom Empfänger erwartete Spannung, so liegt ein statischer Kompatibilitätsfehler vor. In diesem Fall ist eine Simulation des Systems nicht ratsam, weil aufgrund der unterschiedlichen Spannungen das Systemmodell bereits einen „virtuellen“ Schaden aufweiset, der die Simulationsergebnisse verfälschen würde und somit eine scheinbare dynamische Kompatibilität vorspielt, die jedoch nicht vorliegt, da die Ausgangsbasis, also die statische Kompatibilität nicht gegeben ist. Hier gilt der Grundsatz: Aus etwas Falschem kann ein beliebiges Ergebnis gefolgert werden[63].

Zusätzlich zur Simulation eines Modells kommt bei der Kompatibilitätsüberprüfung eines Modells eine formale Verifikation des Systems zu Einsatz. Dabei werden, im Gegensatz zur Simulation, nicht die Werte eines Simulationslaufs zueinander in Beziehung gesetzt, sondern es werden speziell die Eigenschaften miteinander verglichen. Die Überprüfung der Eigenschaften kann zum Beispiel die Schnittstelle zwischen zwei Komponenten/Teilsystemen des Modells betreffen. Sendet beispielsweise der Sender ein Datum zum Empfänger, so wird bei der Simulation lediglich überprüft, ob der übertragene Wert den Erwartungen entspricht. Bei der Kompatibilitätsprüfung hingegen wird nicht nur der Wert, sondern auch der Typ in die Prüfung mit einbezogen. Schickt der Sender beispielsweise die Werte (a) 1, 4, 5 vom Typ **`long`** an den Empfänger. In diesem Besipiel erwartet der Empfänger der Nachricht ebenfalls die Werte 1, 4, 5. Seine interne Repräsentation des empfangenden Datums ist jedoch vom Typ **`int`** (vgl. Abbildung 2.11 auf der nächsten Seite). Für die „kleinen“ zuvor spezifizierten Werte 1, 4 und 5 ist dies vollkommen in Ordnung. Ist der gesendete Wert jedoch größer, als dieser mit Hilfe des Empfänger-Datentyps **`int`** repräsentiert werden kann, zum Beispiel (b) 255 bei 8 Bit Genauigkeit eines **`int`**, so kommt es zu einer Inkompatibilität des Systemmodells, die im Allgemeinen durch die Simulation des Modells nicht detektiert werden kann, aufgrund der Tatsache, dass sonst sämtliche Definitionsbereiche jedes Datums vollständig überprüft werden müssten. Dies führt jedoch zu einer Explosion der Komplexität des zu untersuchenden Systemmodells, und schließlich dazu, dass das Modell nicht mehr simuliert werden kann.

Bei der formalen Kompatibilitätsüberprüfung des Modells hingegen wird die Typinkompatibilität im Modell gefunden, ohne dass dafür sämtliche gültigen Werte eines Datentyps überprüft werden müssen. Dies liegt vor allem daran, dass – im Gegensatz zur Simulation – nicht nur der zu übertragende Wert, sondern auch der Datentyp des Wertes für die Kompatibilitätsprüfung zwischen Sender und Empfänger einer Nachricht herangezogen wird, um zu prognostizieren, ob die Datenübertragung kompatibel, also ohne Wertverlust oder mit Wertverlust erfolgt (Inkompatibilität)[64].

[62] Siehe hierzu insbesondere das Kapitel „1.4.1 *Kompatibilitätsarten*“ ab Seite 22.

[63] Die Aussage: „Ex falso sequitur quodlibet“ (lat.: aus Falschem folgt Beliebiges) stammt aus der Aussagenlogik und beschreibt den Sachverhalt, dass aus einer falschen Aussage beliebiges gefolgert werden kann. Siehe hierzu insbesondere [AHS08], [DJ08] und [Wik08h].

[64] Nähere Informationen zur Typüberprüfung finden Sie im Kapitel „2.3.2.2.1 *Modellierung von Einheiten, dekadischen und nichtdekadischen Präfixen sowie von Gültigkeitsintervallen in der erweiterten formalen Eigenschaftsnotation für die Modellierung und Bestimmung von Kompatibilität*“ ab Seite 69.

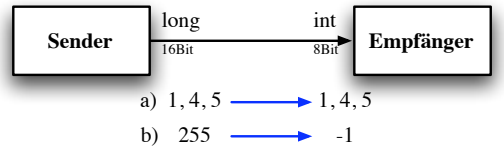


Abbildung 2.11.: Zusammenhang Simulation und Kompatibilitätsprüfung.

2.3. Objektorientierte Modellbildung von eingebetteten softwarelastigen Systemen

In diesem Kapitel werden zunächst die wesentlichen Begriffe und Definitionen aus der objektorientierten Modellbildung kurz vorgestellt sowie anhand des bereits bekannten Beispielsystems Kaffeemaschine erläutert. Dabei wird insbesondere auf die Modellierung von eingebetteten softwarelastigen Systemen eingegangen.

Zunächst werden im folgenden Kapitel die Grundlagen der objektorientierten Modellbildung Schritt für Schritt eingeführt und anhand von Beispielen erläutert. Dabei wird sowohl auf die Modellierung der Struktur, als auch auf die Verhaltensmodellierung und -beschreibung von technischen Systemen eingegangen. Im Anschluss an die allgemeine Einführung in die objektorientierte Modellbildung folgt im Kapitel „*Erweiterung des objektorientierten Modellbildungsparadigmas auf die objektorientierte Kompatibilitätsmodellierung und -bestimmung*“ die Anpassung und Erweiterung des generellen objektorientierten Modellbildungsparadigmas an die speziellen Anforderungen der Modellierung und Bestimmung von Kompatibilität.

2.3.1. Grundkonzepte der objektorientierten Modellbildung

Nachdem bereits im Kapitel 2.2.1 unterschiedliche Modellbildungsparadigmen und insbesondere das Vorgehen bei der objektorientierten Modellbildung vorgestellt wurden, wird in diesem Unteranschnitt explizit auf die objektorientierte Modellbildung von technischen Systemen eingegangen. Dazu wird zunächst das Konzept der *Klassen* und *Objekte* sowie deren ausgezeichnete Rolle bei der objektorientierten Modellbildung vorgestellt. Nach der Vorstellung der Klassen und Objekte folgt zunächst die Einführung in die Modellierung der unterschiedlichen *Eigenschaften* eines Objekts / einer Klasse und im Anschluss daran die *Verhaltensmodellierung* eines Objekts/Klasse. Dabei wird eine vereinfachte formale Notation, basierend auf der UML/UML2-Sprachspezifikation, für sämtliche beschriebenen objektorientierten Sprachelemente eingeführt. Aufbauend auf den Klassen und Objekten werden im darauf folgenden Abschnitt die besonders für die Kompatibilitätsmodellierung und -bestimmung wichtigen Grundeigenschaften der Schnittstellen eines Objekts/Klasse erläutert. Abgeschlossen wird dieses Grundlagenkapitel mit der Erläuterung der Überführung (Instantiierung) einer Klasse in ein Objekt.

Sämtliche hier eingeführten objektorientierten Sprachelemente werden wieder anhand des bereits bekannten Beispielsystems Kaffeemaschine erläutert.

2.3.1.1. Klassen und Objekte eines Systems

Der zentrale Begriffe der objektorientierten Modellbildung, unabhängig davon ob Software oder Hardware modelliert und beschrieben werden soll, ist der Begriff *Objekt*. Dabei kann ein Objekt ein beliebiger Gegenstand unseres Denkens oder Handelns sein. Die folgende sehr allgemein gehaltene Definition 2.5 stellt dabei die unterschiedlichen Ausprägungen des Begriffs Objekt sowie die zugehörigen Domänen heraus.

Definition 2.5 ***Objekt (Allgemein)***
Ein Objekt bezeichnet (Auszug aus [Wik07p] bzw. Zusammenfassung aus [Boo94b, 54, 57, 59, 109ff]):

1. ***Allgemein etwas unspezifiziertes*** *(Sache, Gegenstand oder Ding)*
2. *Etwas greifbares und/oder sichtbares Ding*
3. *Im Sinne der Dialektik das, worauf ein Subjekt seine beobachtende, sinnliche, empirische und praktisch-verändernde Aktivität richtet (Philosophisches Objekt)*
4. *Etwas worauf sich das Denken oder Handeln bezieht*
5. ***Ein Gegenstand mathematischer Untersuchungen*** *(Mathematisches Objekt)*
6. *Ein Satzglied (Grammatikalisches Objekt)*
7. *Einen Himmelskörper (Astronomisches Objekt)*
8. *Eine Einheit in einem Geoinformationssystem (Geoobjekt)*
9. ***Eine Einheit in der Programmierung*** *(Objekt in der Programmierung)*
10. *Eine bestimmte Art künstlerischer Werke (Objektkunst)*

Dabei sind für die objektorientierte Modellbildung von eingebetteten softwarelastigen Systemen vor allem die drei Subdefinitionen 1, 5 und 9 von besonderem Interesse. Die restlichen Unterpunkte der Definition sind für die hier untersuche Domäne nicht entscheidend. Sie sind jedoch der Vollständigkeit halber ebenfalls angegeben worden um die unterschiedlichen Anwendungsfelder des Begriffs aufzuzeigen. Die obige Definition ist jedoch für die objektorientierte Modellierung von Systemen und insbesondere für die Kompatibilitätsmodellierung und -bewertung zu allgemein gehalten.

Objekte
Ausgehend von der Allgemeinen Objektdefinition definiert H. Neumann in seinem Buch „Objektorientierte Entwicklung von Software-Systemen“ [Neu95] ein Objekt als einen Gegenstand, der durch Abstraktion aus einem realen Objekt entsteht.

Definition 2.6 ***Objekt nach [Neu95] auf Seite 574***
Ein Objekt ist zunächst ein ***Gegenstand****, eine* ***Rolle****, ein* ***Konzept*** *oder ein* ***Prozess*** *im realen Problem- und Anwendungsbereich eines Softwaresystems (reales Objekt). Durch einen geeigneten* ***Abstraktionsprozess*** *wird daraus ein* ***Objekt*** *abgeleitet, das als Basismodul in die Ablaufstruktur eines Softwaresystems eingebunden wird.*

Die obige Definition von H. Neumann beschreibt ein reales Objekt als einen Gegenstand, eine Rolle, einen Prozess oder ein Konzept aus dem realen Problem- bzw. Anwendungsbereich eines Systems. Dieses reale Objekt wird dann, mittels eines geeigneten Abstraktionsprozesses, in ein, im Allgemeinen abstraktes Objekt überführt (reales Objekt $\xrightarrow{Abstraktion}$ abstraktes Objekt). Dabei hat jedes (abstrakte) Objekt sowohl einen *Status* (Zustand), ein *Verhalten* sowie eine *Identität*, die es gegenüber allen anderen Objekten auszeichnet. Insbesondere ist durch die flexible Art der Objektdefinition ein Objekt nicht nur auf die Modellierung eines bestimmten Sachverhalts eines Systems beschränkt, sondern der Begriff Objekt kann auf unterschiedlichste Aspekte eines Systems gleichermaßen angewendet werden. So kann zum Beispiel ein Objekt ein Gegenstand sein, beispielsweise eine Kaffeemaschine, oder ein Objekt kann einen Prozess innerhalb eines Systems beschreiben, zum Beispiel den Prozess „Kaffee kochen“. Der Objektbegriff kann so flexibel eingesetzt werden, dass sich mit Hilfe eines Objekts sämtliche Aspekte eines Systems vollständig modellieren und beschreiben lassen.

Um ein Objekt „greifbar“ zu machen, werden Objekte im Allgemeinen als Rechtecke, mit einem eineindeutigen unterstrichenen Namen dargestellt. Die Abbildung 2.12 zeigt eine mögliche Darstellungsform eines Objekts, so wie sie in der Unified Modelling Language (UML/UML2) benutzt wird.

Abbildung 2.12.: Darstellung eines Objekts (in Anlehnung an die UML/UML2-Darstellung von Objekten [CS04, 71]).

Aufgrund der besonderen Eigenschaften des Objekts, ist es möglich, die objektorientierte Sichtweise für die Modellierung von komplexen technischen Systemen, bestehend aus Hard- und Software, gleichermaßen einzusetzen, ohne dass es dabei zu einem „Bruch“ in der Modellbildung bzw. der Modellierung zwischen den an der Systementwicklung beteiligten Domänen kommt. Um diesen lückenlosen Übergang zwischen den unterschiedlichen Domänen erreichen zu können, ist lediglich der Begriff „Softwaresystem“ durch die allgemeine Systemdefinition (vgl. „Definition: 1.2“ auf Seite 12) in der obigen Definition 2.6 zu ersetzen. Nach der Substitution des Begriffs Softwaresystem durch System im Systems Engineering Kontext lässt sich die objektorientierte Modellierung auf beliebige technische Systeme anwenden. Auf die Anwendung des objektorientierten Modellbildungsparadigmas auf technische Systeme wird unter anderem in [PDB98a, 93ff] und [KS03, 150ff, 379ff] ausführlich eingegangen.

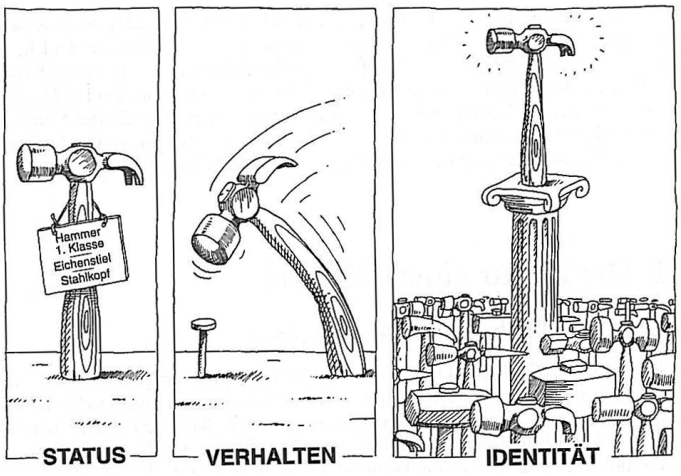


Abbildung 2.13.: Ein Objekt hat einen Status (Zustand), weist ein wohldefiniertes Verhalten auf und besitzt eine eindeutige Identität (aus [Boo94b, 110]).

Die Abbildung 2.13 illustriert den Begriff des Objekts, ausgehend von der Definition 2.6, anhand der grundsätzlichen Eigenschaften jedes Objekts. Jedes Objekt hat eine einzigartige Identität, anhand derer es in einem System zweifelsfrei identifiziert werden kann. Zusätzlich besitzt jedes Objekt genau einen Status (Zustand) und weist ein bestimmtes Verhalten auf. Mittels dieser drei Objekteigenschaften lassen sich sämtliche Objekte charakterisieren, unabhängig ob es sich dabei um ein Software- oder ein Hardwareobjekt handelt.

Klassen
In einem realen technischen System befinden sich im Allgemeinen sehr viele unterschiedliche Objekte, die die verschiedensten Teilsysteme eines Systems, wie beispielsweise das Teilsystem Energieversorgung oder die Steuerung der Kaffeemaschine, repräsentieren. Dabei sind jedoch nicht alle Objekte eines Systems grundverschieden, sondern weisen bestimmte charakteristische Gemeinsamkeiten in Struktur und Verhalten auf. Diejenigen Objekte des Systems, die beispielsweise eine bestimmte Eigenschaft, oder ein gemeinsames Verhalten haben, können zu einer größeren Struktur – den Klassen – zusammengefasst werden. Die nachfolgende Definition 2.7 – von G. Booch – beschreibt genau diesen Sachverhalt.

Definition 2.7 ***Klasse nach [Boo94b] auf Seite 136***
Eine ***Klasse*** *ist eine Menge von Objekten, die eine* ***gemeinsame Struktur*** *und ein* ***gemeinsames Verhalten*** *aufweisen.*

Zur Verdeutlichung der besonderen Eigenschaften einer Klasse, zeigt die Abbildung 2.14 eine Menge von Objekten – hier Autos –. Jedes Objekt Auto, unabhängig ob es sich dabei um einen PKW oder LKW handelt, hat bestimmte gemeinsame Eigenschaften und/oder ein gemeinsames Verhalten. Aus diesem Grund lassen sich aus dem Objekt Auto Klasse(n) ableiten, in denen die abstrakten grundlegenden Eigenschaften sowie das Verhalten jedes Autos zusammengefasst sind. Im Beispielbild sind dies die beiden Eigenschaften Steuerung (Lenkrad) und Schaltung sowie das Verhalten fahren.

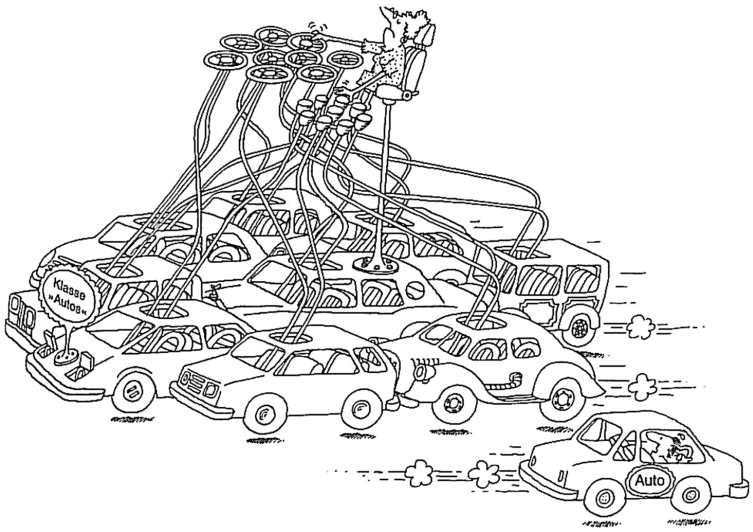


Abbildung 2.14.: Eine Klasse ist eine Menge von Objekten, die eine gemeinsame Struktur und ein gemeinsames Verhalten aufweisen (aus [Boo94b, 137]).

Durch die Einführung des Klassenkonzepts der objektorientierten Modellbildung wird das Konzept der Wiederverwendbarkeit von Teilen eines Systems erheblich verbessert. Ist beispielsweise die Klasse *Steuerung*, aus dem obigen Beispiel, identifiziert und entsprechend modelliert worden, so kann diese, ohne große Änderungen in jedem weiteren Fahrzeug z.B. als Teilsystem verwendet werden. Zusätzlich kann die Klasse *Steuerung* auch in anderen Klassen von Fahrzeugen verwendet werden, wie beispielsweise in der Klasse Motorrad oder der Klasse der Flugzeug.

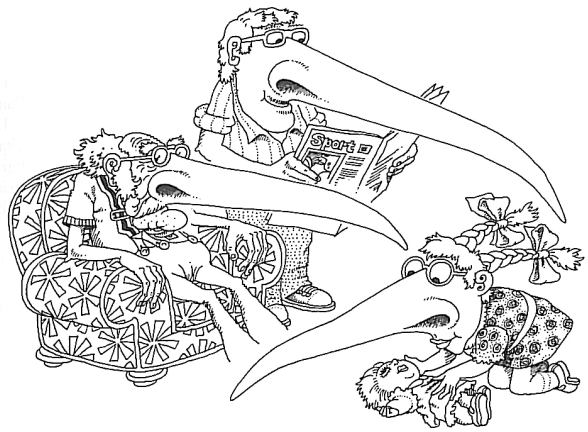


Abbildung 2.15.: Eine Unterklasse kann die Struktur und das Verhalten seiner Oberklasse erben (aus [Boo94b, 145]).

Ebenso wie Objekte können auch Klassen mit Hilfe von Rechtecken mit einem eineindeutigen Namen in der Mitte dargestellt werden. Im Gegensatz zur Darstellung von Objekten wird der Name der Klasse nicht unterstrichen. Die Abbildung 2.16 zeigt die graphische Darstellung einer Klasse, so wie sie in der UML/UML2-Notation üblich ist.

Abbildung 2.16.: Darstellung einer Klasse (in Anlehnung an die UML/UML2-Darstellung von Klassen [CS04, 71]).

Untrennbar mit dem Konzept der Wiederverwendbarkeit von Klassen ist das Konzept der *Vererbung* bzw. *Erweiterung* der Klassen verbunden. So kann beispielsweise die Klasse *Steuerung*, wie oben erwähnt, in unterschiedlichen Bereichen eingesetzt werden. Dazu sind im Allgemeinen jedoch Änderungen sowohl der Eigenschaften als auch des Verhaltens der Klasse notwendig. Um jedoch die Klasse *Steuerung* nicht ständig an die unterschiedlichen Anforderungen anpassen zu müssen, bietet die Objektorientierung das Konzept der Vererbung an. Dabei wird die ursprüngliche Klasse als Basisklasse/Vaterklasse der neuen benutzt. Dadurch erhält die neue Klasse, auch Kindklasse genannt, sämtliche (öffentlichen/geschützten[65]) Eigenschaften sowie das (öffentliche/geschützte) Verhalten der Vaterklasse. Das Konzept der objektorientierten Vererbung von einer Vaterklasse hin zu einer Kindklasse wird auch als Verfeinerung der Klasse bezeichnet. In umgekehrter Richtung wird von *Generalisierung* gesprochen. Die Abbildung 2.17 zeigt links eine Vererbungsbeziehung zwischen der Vaterklasse *Vaterklasse* sowie den beiden davon abgeleiteten Kindklassen *Kind A* und *Kind B*. Dabei erben die beiden Kindklassen sämtliche öffentlichen/geschützten Eigenschaften, genauso wie das Verhalten der Vaterklasse. Besitzt beispielsweise die Vaterklasse die öffentliche Eigenschaft $maxGeschwindigkeit = 100\frac{\text{km}}{\text{h}}$, so haben auch die beiden Kindklassen die Eigenschaft $maxGeschwindigkeit = 100\frac{\text{km}}{\text{h}}$ ohne dass diese in der Kindklasse explizit aufgeführt werden muss[66].

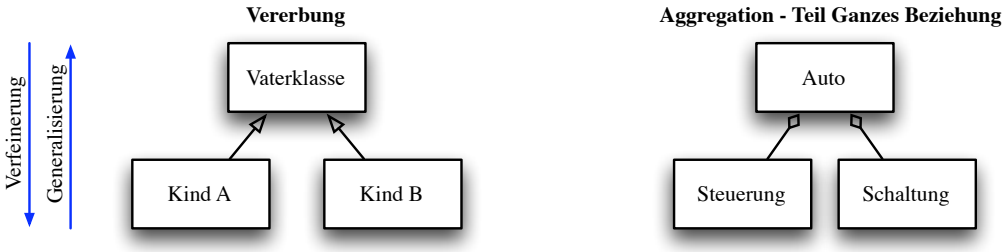


Abbildung 2.17.: Beziehungen zwischen Klassen eines Systems.

In objektorientierten Modellierungssprachen existieren neben der Vererbungsbeziehung noch weitere drei Verbindungsarten[67] um verschiedene Arten von Beziehungen zwischen Klassen ausdrücken und modellieren zu können. Die Abbildung 2.17 zeigt beispielsweise rechts eine *Aggregations-Beziehung* zwischen Klassen. Mit Hilfe der Aggregations-Beziehung zwischen zwei Klassen eines Systems wird besonders zum Ausdruck gebracht, dass die eine Klasse ein *Teil der anderen Klasse* ist. So ist beispielsweise die Steuerung eines Autos sowie die Schaltung ein Teil des Fahrzeugs. Im Allgemeinen existiert kein Auto ohne eine Steuerung da ohne diese eine Lenkung des Fahrzeugs unmöglich ist. Der gleiche Sachverhalt gilt auch mit bestimmten Einschränkungen für die Schaltung eines Autos. Nicht jedes Auto hat eine explizite Schaltung, wie beispielsweise manche Elektroautos, die direkt also ohne Schaltung, von einem Elektromotor angetrieben werden.

Eine weitere Kerneigenschaft fast aller modernen objektorientierten Modellierungssprachen ist das Konzept des „Information Hiding“. Dabei werden beim Information Hiding sämtliche für den Anwender unwichtigen Implementierungsdetails der Klasse vor ihm verborgen. Alle für den Anwender wichtigen Bestandteile, wie beispielsweise die öffentlichen Eigenschaften und die öffentlichen Methoden – allgemein die Schnittstelle der Klasse – sind sichtbar und können vom Anwender jederzeit ohne weitere Einschränkungen eingesehen und verwendet werden. Die konkrete Umsetzung (die Implementierung) zum Beispiel einer Methode der Klasse wird jedoch stets vor dem Anwender der Klasse verborgen und kann somit nicht eingesehen oder gar verändert werden.

Definition 2.8 ***Information Hiding***
Das Verbergen von Implementierungsdetails vor dem Benutzer wird als ***Information Hiding*** *bezeichnet. Der Benutzer eines Systems muss nichts über dessen inneren Aufbau oder Struktur wissen, um es benutzen/bedienen zu können. Für den Benutzer ist lediglich die äußere Schnittstelle des Systems von Interesse.*

Durch die konsequente Anwendung des Information Hidings bei der Modellbildung und insbesondere bei der Wiederverwendung von Klassen eines komplexen Systems, kann die Komplexität einer Klasse aus Anwendersicht erheblich verringert werden, da er nicht mit für ihn unwichtigen internen Details der Klasse konfrontiert wird, sondern nur die wesentlichen Bestandteile, d.h. die Schnittstelle der Klasse zu sehen bekommt.

[65] Nähere Informationen zu den unterschiedlichen Ausprägungen der Eigenschaften einer Klasse finden Sie im Kapitel „3.4.3 *Die Unified Modelling Language – UML/UML2*“ ab Seite 131.

[66] Nähere Informationen zu den Eigenschaften bzw. dem Verhalten einer Klasse finden Sie in den nachfolgenden beiden Abschnitten bzw. im Kapitel „3.4.3 *Die Unified Modelling Language – UML/UML2*“ ab Seite 131.

[67] Im Kapitel „3.4.3.1.1 *UML/UML2-Strukturdiagramme*“ ab Seite 132 werden die verbleibenden drei Verbindungsarten zwischen Klassen vorgestellt und erläutert.

Anmerkungen:

- Das Konzept des Information Hidings ist in erster Näherung mit dem bereits im Kapitel „*Grundlegende Begriffe und Definitionen aus der Systems Engineering Welt*“ (Definition 1.10) vorgestellten Sichtenkonzept – Black-Box, Gray-Box und Glass-Box – verwandt. Auch hier wurden unwichtige Bestandteile, zum Beispiel bei Gray- oder Black-Box Darstellung eines Teilsystems, vor dem Anwender verborgen. Im Gegensatz zum Sichtenkonzept werden beim Information Hiding stets sämtliche Implementierungsdetails (Methodenrümpfe) versteckt und nur die Schnittstelle (Signatur der Methoden) der Klasse ist sichtbar[68].
- Weiterführende Begriffe und Definitionen aus dem Umfeld der „objektorientierte Modellierung“ finden Sie im Anhang unter „C.1 *Weiterführende Begriffe und Definitionen aus der objektorientierten Modellierung*“ ab Seite 327.

Beispiel 10: Objektorientierte Modellierung des Systems Kaffeemaschine
Bereits im Beispiel auf Seite 15, sowie der Abbildung 2.8 auf Seite 40, wurde ein einfaches Modell des Systems Kaffeemaschine erstellt. Dieses Modell wird nun schrittweise unter Anwendung des objektorientierten Modellbildungsparadigmas[69]in ein objektorientiertes Systemmodell überführt. Laut dem Vorgehen bei der objektorientierten Modellbildung werden nun zunächst die Objekte und Klassen des Systems identifiziert.

Das zu modellierende System Kaffeemaschine besteht aus den folgenden Teilsystemen:

- Einem Gehäuse,
- einer Heizplatte,
- einer Regelung,
- einer Kaffeekanne,
- sowie den restlichen, nicht weiter spezifizierten Komponenten des Systems.

Jedes Teilsystem besteht wiederum aus einzelnen Bestandteilen. Beispielsweise besteht das Teilsystem *Regelung* aus den drei Untersystemen *Mechanik, Elektrotechnik* und *Software*. Die Abbildung 2.18 zeigt das (UML/UML2) Klassenmodell des Systems Kaffeemaschine.

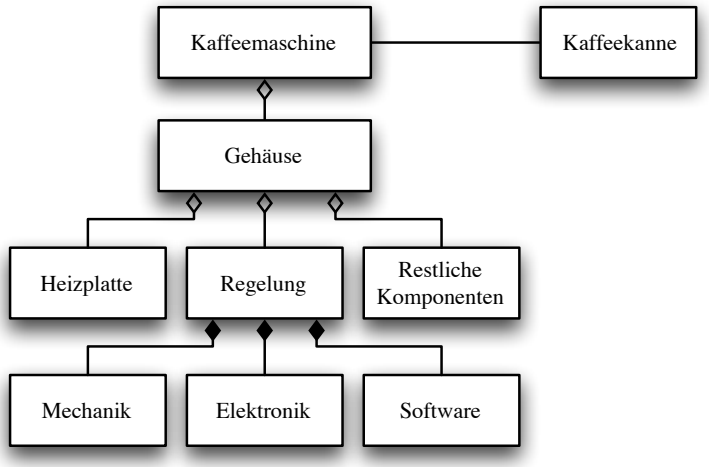


Abbildung 2.18.: Vereinfachtes Klassenmodell des Systems Kaffeemaschine.

Im Klassenmodell sind die unterschiedlichen Klassen des Systems eingetragen, die aus den identifizierten Objekten des Systems entstanden sind. Zusätzlich zur Darstellung der Klassen wurden in der Abbildung 2.18 zusätzlich die Verbindungen zwischen den Klassen eingetragen, die jedoch im Allgemeinen erst im dritten Schritt der objektorientierten Modellbildung in das Klassenmodell eingetragen werden. □

Nachdem nun die wesentlichen Eigenschaften der Klassen und Objekte eingeführt worden sind, folgt nun im nächsten Kapitel die Beschreibung der Eigenschaften (Attribute) einer Klasse.

2.3.1.2. Modellierung der Eigenschaften eines Objekts bzw. einer Klasse

Im ersten Schritt der objektorientierten Modellbildung wurden die Objekte und Klassen eines Systems identifiziert und beschrieben. Im zweiten Schritt werden nun die Eigenschaften, auch Attribute eines Objekts/Klasse genannt, eingeführt und anhand des bereits bekannten Beispielsystems Kaffeemaschine erläutert. Dabei ist es für die Modellierung der Eigenschaften eines Objekts/Klasse unerheblich, ob es sich um eine mechanische, elektronische oder Software-Eigenschaft handelt. Sämtliche Eigenschaften eines Objekts/Klasse lassen sich mit Hilfe einer einheitlichen formalen Syntax beschreiben.

Begonnen wird zunächst mit der Einführung des allgemeinen Eigenschaftsbegriffs. Dieser wird im zweiten Schritt auf die speziellen Belange der objektorientierten Modellbildung angepasst.

[68] Weitere Informationen zum Thema „Information Hiding“ entnehmen Sie bitte [Str90, 29, 778], [PDB95, 164, 188] bzw. [Wik07g].
[69] Siehe hierzu „Vorgehen bei der objektorientierten Modellbildung“ auf der Seite 41.

Unterschiedliche domänenabhängige Ausprägungen des Begriffs Eigenschaft
In der Literatur ist der Begriff „Eigenschaft“ (Attribut), ebenso wie schon zuvor der Begriff des Objekts (vgl. die Definition 2.5), nicht exakt definiert. Vielmehr hat jede Domäne ihre eigene Definition und Umschreibung des Begriffs *Eigenschaft* die im Allgemeinen nicht mit der Definition des Begriffs in anderen Domänen übereinstimmt oder diesen Definitionen sogar widerspricht. Die folgende allgemeine Definition beschreibt drei mögliche Ausprägungen des Begriffs *Eigenschaft*.

Definition 2.9 ***Unterschiedliche Ausprägungen des Begriffs Eigenschaft eines Objekts/Klasse (nach [Wik08e])***
Der Begriff ***Eigenschaft*** *bezeichnet:*

1. *Allgemein ein* ***realisiertes Merkmal***, ***eine Funktion***, ***ein Attribut*** *oder* ***eine Qualität***, *die einer Klasse von Objekten, Prozessen, Relationen, Ereignissen, einer Handlung, einer Person oder Personengruppe gemeinsam ist bzw. sie von anderen unterscheidet.*
2. *In vielen* ***objektorientierten Modellierungssprachen*** *bezeichnet der Begriff Eigenschaft ein* ***Attribut*** *eines Objekts.*
   - ***UML/UML 2:*** *Ein Attribut beschreibt eine Eigenschaft, z.B. einer Klasse. Attribute sind zusammengesetzt aus Sichtbarkeit, Variablenname und Wert (Objekt) der Variablen [Fow00].*
   - ***SysML:*** *Das Attribut definiert eine Struktureigenschaft der Klasse bzw. eines Objekts. Die Beschreibung besteht aus Sichtbarkeit, Name, Type und einer Multiplizität [Wei06, 225ff].)*
3. *Physikalische Eigenschaften oder chemische Eigenschaften.*

Wie die obige Definition zeigt, kann der Begriff der Eigenschaft in den unterschiedlichsten Domänen eingesetzt werden, um die ausgezeichneten Merkmale eines Objekts bzw. einer Klasse zu beschreiben. Im weiteren Verlauf dieser Arbeit wird eine, speziell auf die Domäne Modellierung und Bestimmung von Kompatibilität von eingebetteten softwarelastigen Systemen angepasste Version, ähnlich der Unterdefinition 2, für die Beschreibung der Eigenschaften eines Objekts bzw. einer Klasse verwendet.

Eigenschaften einer Klasse bzw. eines Objekts
Bereits im Kapitel „*Grundlegende Begriffe und Definitionen aus der Systems Engineering Welt*“ wurde der Begriff der *Eigenschaft* von Systemen eingeführt. Aufbauend auf der dort vorgestellten allgemeinen Eigenschaftsdefinition für technische Systeme (vgl. Definition 1.7 auf Seite 15) sowie der obigen Unterdefinition 2, werden in diesem Abschnitt die Eigenschaften der Objekte und Klassen formal beschrieben und definiert.

Laut Definition 1.7 besitzt jedes System bestimmte, das System charakterisierende konstante und veränderliche Eigenschaften. Mittels dieser Systemeigenschaften werden die das System bestimmenden Merkmale des Systems festgelegt. Diese Aussage kann o.B.d.A. auf Objekte bzw. Klassen angewendet werden. Also hat jedes reale oder abstrakte Objekt/Klasse bestimmte, das Objekt/Klasse charakterisierende Eigenschaften. Mit Hilfe dieser Eigenschaften wird sowohl die Struktur, als auch zu einem gewissen Teil das Verhaltens des Objekts/Klasse bestimmt.

Definition 2.10 ***Eigenschaften eines Objekts/Klasse***
Ein Objekt/Klasse besitzt ***Eigenschaften****, auch Attribute genannt, mit deren Hilfe die* ***Merkmale*** *eines Objekts/Klasse festgelegt und beschrieben werden. Die Eigenschaften des Objekts/Klasse können dabei sowohl* ***konstanter-****, als auch* ***veränderlicher*** *Natur sein.*

Das folgende Beispiel illustriert die unterschiedlichen Eigenschaften des Beispielsystems Kaffeemaschine.

Beispiel 11: Eigenschaften der Objekte des Systems Kaffeemaschine
Im vorherigen Kapitel (vgl. Beispiel 2.3.1.1) wurde das Klassenmodell, bestehend aus den identifizierten Objekten des Systems Kaffeemaschine, unter Anwendung des objektorientierten Modellbildungsparadigmas[70]erstellt. Aufbauend auf dem dort erarbeiteten Klassenmodell der Kaffeemaschine werden, im zweiten Schritt der objektorientierten Modellbildung, die das Objekt charakterisierenden Eigenschaften des Objekts/Klassen identifiziert und in die Rümpfe der entsprechenden Klassen (Objekten) eingetragen.

Beispielsweise hat das Objekt *Gehäuse* die beiden Eigenschaften *Gehäusegröße* sowie *Kannengröße*. Das Objekt *Kaffeekanne* besitzt ebenfalls die Eigenschaft *Kannengröße*. Sowohl die beiden Eigenschaften des Objekts *Gehäuse*, als auch die des Objekts *Kaffeekanne* sind dabei unveränderliche Eigenschaften der Objekte. Sie sind während der gesamten Lebensdauer der Objekte konstant. Im Gegensatz dazu hat das Objekt *Elektronik* eine über die Zeit veränderliche Eigenschaft, nämlich die Eigenschaft *Stromverbrauch*. Dieser ändert sich während der gesamten Lebenszeit des Objekts ständig. Außerdem besitzt das Objekt *Regelung* die drei Eigenschaften *Speichergröße, Stromverbrauch* sowie *Mech. Größe*.

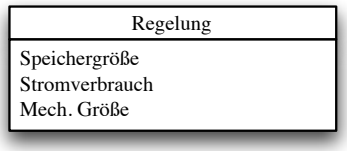


Abbildung 2.19.: Auszug aus dem Klassenmodell des Systems Kaffeemaschine – Klasse *Steuerung*.

Sämtliche identifizierten Eigenschaften der Objekte werden nun den entsprechenden Klassen zugewiesen. Die Abbildung 2.19 auf der vorherigen Seite zeigt einen Ausschnitt aus dem entsprechenden Klassenmodell des Systems Kaffeemaschine. □

Zusätzlich lassen sich die Eigenschaften eines Objekts/Klasse, gemäß Definition 2.10 mit einem so genannten *Sichtbarkeitsattribut* versehen, das regelt, welche Eigenschaften und Methoden[71] eines Objekts/Klasse von der das Objekt/Klasse umgebenden Umwelt aus erreichbar sind. Die folgende Definition (Def.: 2.11) beschreibt drei Sichtbarkeitsarten für die Eigenschaften und Methoden eines Objekts/Klasse.

Definition 2.11 ***Sichtbarkeit von Eigenschaften und Methoden eines Objekts/Klasse***
Jede ***Eigenschaft*** *sowie jede* ***Methode*** *eines Objekts/Klasse können mit einem* ***Sichtbarkeitsattribut*** *versehen werden, das den Zugriff auf die Eigenschaft bzw. die Methode regelt. Es stehen drei Sichtbarkeitsarten zur Verfügung:*

- **+** *Öffentlich, ohne Einschränkung sichtbar.*
- **-** *Privat, nur innerhalb des Objekts/Klasse sichtbar.*
- **#** *Geschützt, nur vom eigenen Objekt/Klasse bzw. von davon abgeleiteten Objekten/Klassen sichtbar.*

Durch die Einführung von speziellen Sichtbarkeiten auf die Eigenschaften und Methoden eines Objekts/Klasse wird insbesondere das objektorientierte Paradigma des Information Hiding[72] erfüllt.

Formale Beschreibung der Eigenschaften eines Objekts/Klasse
Um die unterschiedlichen Eigenschaften eines Objekts/Klasse modellieren bzw. beschreiben zu können, ist eine einheitliche wohlgeformte formale Notation von Vorteil, die es allen an der Systementwicklung beteiligten Personen ermöglicht, die Eigenschaftsdefinitionen eines Objekts/Klasse lesen und verstehen zu können. Um dies zu erreichen, bietet sich die formale syntaktische Beschreibung der Eigenschaften eines Objekts/Klasse, in Anlehnung an die in der UML/UML2 übliche Attributbeschreibung an[73]. Die nachfolgende Syntaxbeschreibung zeigt einen knappen Auszug aus der stark vereinfachte Attributdefinition für Objekte/Klassen in der so genannten Backus-Naur-Form (BNF)[74]:

```
<AttributDeklaration> ::= <Sichtbarkeit> <Attributname> ":" <Datentyp> ("=" <Ausdruck> | ... )?

<Sichtbarkeit>        ::= "+" | "-" | "#".
<Datentyp>            ::= "void" | "bool" | "int" | "long" | "float" | "string"...
<Ausdruck>            ::= (<Buchstabe> | <Ziffer>)*.
<Zuweisung>           ::= <Eigenschaftsname> "=" <Ausdruck>.
<Attributname>        ::= <Buchstabe> (<Buchstabe> | <Ziffer>)*.
<Eigenschaftsname>    ::= <Buchstabe> (<Buchstabe> | <Ziffer>)*.
<Buchstabe>           ::= "A" | "B" | "C" | "D" ... "Z" | "a" | "b" | "c" | "d" ... "z".
<Ziffer>              ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "0".
```

Listing 2.1: Eigenschaftsdeklaration

Definition 2.12 ***Formale Beschreibung der Eigenschaften eines Objekts/Klasse***
Sämtliche ***Eigenschaften*** *eines* ***Objekts/Klasse*** *lassen sich mit Hilfe der im Listing 2.1 beschriebenen formalen BNF-artigen Syntax formal modellieren und beschreiben.*

Mit Hilfe der obigen Notation (Listing 2.1) können die Eigenschaften eines Objekts/Klasse formal beschrieben werden. Im nachfolgenden Beispiel werden sämtliche Eigenschaften der Klasse *Regelung* des Beispielsystems Kaffeemaschine in der oben eingeführten formalen Notation dargestellt.

Beispiel 12: Modellierung der Eigenschaften der Klasse „Regelung" der Kaffeemaschine
Die Abbildung 2.20 zeigt die Klasse *Regelung* inklusive der Eigenschaften der Klasse. Aus der Abbildung geht hervor, dass die Klasse *Regelung* drei nicht öffentliche Eigenschaften *Speichergroesse*, *Stromverbrauch* und *Mech_Groesse* besitzt.

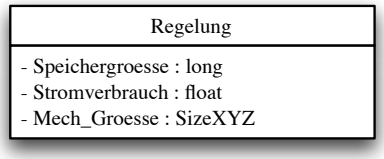


Abbildung 2.20.: Modellierung der Eigenschaften der Klasse *Regelung* des Beispielsystems Kaffeemaschine.

[70] Siehe hierzu „Vorgehen bei der objektorientierten Modellbildung" auf der Seite 41.
[71] Methoden eines Objekts/Klassen werden im Kapitel „2.3.1.3 *Modellierung des Verhaltes eines Objekts/Klasse*" ab Seite 53 erläutert.
[72] Siehe hierzu die Definition 2.8 auf Seite 49.
[73] Siehe hierzu auch [CS04][OMG07c][Gro06a][Gro06b].
[74] Die Backus-Naur-Form (BNF) bzw. die erweiterte Backus-Naur-Form (EBNF) dient zur formalen Beschreibung einer (Programmier-) Sprache. Nähere Informationen zur BNF finden Sie unter [Wik06b][DL][Ram06b][Ram06a], EBNF unter [Sch06] bzw. im Glossar.

Auf die drei Eigenschaften der Klasse kann von außerhalb der Klasse aufgrund der eingeschränkten Sichtbarkeit der Eigenschaften nicht direkt zugegriffen werden. Sollen die drei Eigenschaften von außerhalb zugreifbar gemacht werden, so kann dies entweder durch Änderung der Sichtbarkeit (von *privat* (-) auf *öffentlich* (+)) oder durch die Einführung von speziellen Zugriffsmethoden[75] geschehen.

Die Eigenschaft `Speichergroesse` ist beispielsweise vom Datentyp **`long`**, kann also ganze Zahlen in Abhängigkeit von der Speicherbreite des Datentyps speichern. Sollte eine Eigenschaft zusätzlich mit einem Wert versehen werden, so kann dies ebenfalls mittels der oben eingeführten Notation erfolgen. So kann beispielsweise der Eigenschaft *Speichergroesse* der Wert 500 zugewiesen werden (`Speichergroesse = 500`). □

Nachdem in diesem Kapitel die objektorientierte Modellierung der Eigenschaften eines Objekts/Klasse eingeführt wurde, folgt im nächsten Kapitel die Modellierung des Verhaltens eines Objekts bzw. einer Klasse.

2.3.1.3. Modellierung des Verhaltes eines Objekts/Klasse

Das Verhalten eines Objekts/Klasse lässt sich auf unterschiedliche Arten formal beschreiben. Beispielsweise kann das Verhalten eines kompletten Systems oder einzelner Objekte des Systems mittels *Zustandsautomaten*, in denen sämtliche Zustände, in denen sich das Objekt/Klasse befinden kann, mit den dazugehörigen Übergängen zwischen den Zuständen, modelliert und beschrieben werden (vgl. Abbildung: 2.10 auf Seite 43). Eine weitere Art der Verhaltensmodellierung stellen die so genannten *Message Sequence Charts* (MSC), eine alternative Ausprägung der allgemeinen *Sequenzdiagramme* dar. Mit ihrer Hilfe werden insbesondere der Signal- bzw. der Nachrichtenaustausch zwischen Objekten/Klassen eines Systems formal festgelegt. Eine weitere Möglichkeit das Verhalten eines Systems zu spezifizieren erfolgt durch eine objektorientierte Programmiersprache in der sowohl die Signatur der Methode als auch der Methodenrumpf angegeben wird. Auch Kombinationen aus den unterschiedlichen Modellierungsarten sind möglich, werden in der Praxis jedoch selten verwendet.

In den folgenden beiden Unterabschnitten diese Kapitels werden nun zwei unterschiedliche Arten der Verhaltensbeschreibung eines Objekts/Systems vorgestellt und anhand von Beispielen erläutert, die speziell für die spätere Kompatibilitätsmodellierung und Bestimmung verwendet werden können. Begonnen wird dabei mit der Verhaltensmodellierung mittels programmiersprachlicher Konstrukte.

2.3.1.3.1. Verhaltensbeschreibung eines Objekts/Klasse mittels programmiersprachlicher Konstrukte

Das Verhalten eines Objekts/Klasse lässt sich mit Hilfe von programmiersprachlichen Konstrukten bzw. objektorientierten Programmiersprachen, wie beispielsweise der objektorientierten Programmiersprache C++, formulieren. Dabei werden grundsätzlich zwei unterschiedliche Arten von verhaltensbeschreibenden Methoden eines Objekts/Klasse unterschieden.

Definition 2.13 ***Verhaltensbeschreibung eines Objekts/Klasse***
Mit Hilfe der ***Methoden****, auch Funktionen der Operationen genannt, wird das* ***Verhalten*** *der Klasse bzw. eines Objekts beschrieben und bestimmt. Die Methoden der Klasse lassen sich dabei in zwei Gruppen unterteilen:*

- ***Zugriffsmethoden***
  Mit Hilfe der Zugriffsmethoden, auch Accessoren genannt, kann auf die internen privaten Datenstrukturen (Eigenschaften) des Objekts/Klasse kontrolliert von außen zugegriffen werden.
- ***Verhaltensbeschreibende Methoden***
  Die verhaltensbeschreibenden Methoden dienen zur Modellierung des Verhaltens der Klasse.

Jede Methode besteht aus zwei Teilen. Zum einen der *Signatur* der Methode, in der genau festgelegt ist, wie die Methode heißt und welche Parameter sie besitzt. Und zum anderen dem *Methodenrumpf* in dem das eigentliche Verhalten der Methode beschrieben ist. Die folgende Abbildung 2.21 zeigt sowohl die Signatur der Methode **`void`** `Regelung::SetSpeichergroesse(`**`long`** `theSize)`, als auch deren Methodenrumpf in C++ Notation.

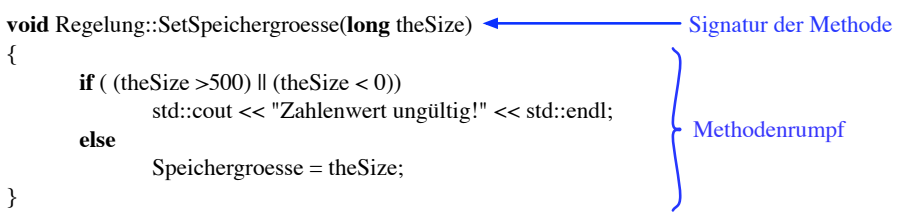


Abbildung 2.21.: Methodensignatur und Methodenrumpf.

Die Signatur einer Methode wird auch als Schnittstelle der Methode bezeichnet, da sie genau festlegt, wie die Methode heißt und welche Parameter sie erwartet bzw. als Funktionsergebnis zurück liefert. Aus diesem Grund muss für die Beschreibung der Methoden eines Objekts/Klasse, wieder eine einheitliche wohl geformte formale Notation verwendet werden, um Missverständnisse zwischen unterschiedlichen, an der Entwicklung beteiligten Personen zu vermeiden.

[75] Die Zugriffsmethoden werden im Kapitel „2.3.1.3 *Modellierung des Verhaltes eines Objekts/Klasse*“ ab Seite 53 erläutert.

Formale Beschreibung der Methoden eines Objekts/Klasse
Für die formale Beschreibung der Signatur der Methoden eines Objekts/Klasse bietet sich wieder die stark vereinfachte, an die UML/UML2 angelehnte BNF-Notation an. Die folgende Definition beschreibt den Aufbau und die Struktur der Signatur einer Methode.

```
<FunktionsDeklaration> ::=
    <Sichtbarkeit> (<Eigenschaftstyp>)? <Datentyp> <Funktionsname> "(" ("void" | (<Parameter> ","?)*) ")"

<Parameter>       ::= (<Eigenschaftstyp>)? <Datentyp> <Parametername>?.

<Sichtbarkeit>    ::= "+" | "-" | "#".
<Eigenschaftstyp> ::= "const" | "var".
<Datentyp>        ::= "void" | "bool" | "int" | "long" | "float" | "string"...
<Funktionsname>   ::= <Buchstabe> (<Buchstabe> | <Ziffer>)*.
<Parametername>   ::= <Buchstabe> (<Buchstabe> | <Ziffer>)*.
<Buchstabe>       ::= "A" | "B" | "C" | "D" ... "Z" | "a" | "b" | "c" | "d" ... "z".
<Ziffer>          ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "0".
```

Listing 2.2: Methodendeklaration

Definition 2.14 ***Formale Beschreibung der verhaltensbeschreibenden Methoden eines Objekts/Klasse***
*Sämtliche **verhaltensbeschreibenden Methoden** eines **Objekts/Klasse** lassen sich mit Hilfe der im Listing 2.2 beschriebenen, formalen BNF-artigen Syntax formal modellieren und beschreiben.*

Das nachfolgende Beispiel illustriert die Modellierung des Verhaltens der Klasse *Regelung* (genauer eines Objekts der Klasse) anhand des bereits bekannten Systems Kaffeemaschine.

Beispiel 13: Modellierung des Verhaltens der Klasse „Regelung“ der Kaffeemaschine
Die Abbildung 2.22 zeigt das vereinfachte Modell der Klasse *Regelung* des Beispielsystems Kaffeemaschine inklusive der zuletzt eingeführten Eigenschaften und der neu hinzugekommenen Methoden der Klasse sowohl für den Eigenschaftszugriff, als auch zur Verhaltensbeschreibung der Klasse.

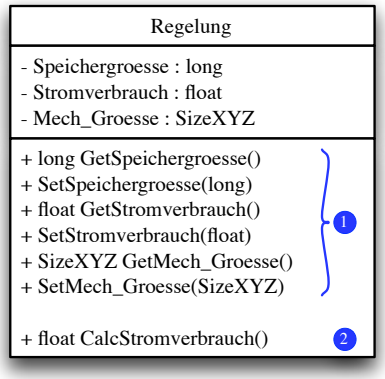


Abbildung 2.22.: Modellierung des Verhaltens der Klasse *Regelung* des Beispielsystems Kaffeemaschine.

Wie bereits oben erwähnt (vgl. „Definition: 2.13“ auf Seite 53) lassen sich die unterschiedlichen Methoden jeder Klasse in zwei Gruppen unterteilen. Zum einen die Zugriffsmethoden für die privaten internen Eigenschaften der Klasse, zum anderen die verhaltensbeschreibenden Methoden.

- **Zugriffsmethoden** der Klasse *Regelung*
  Mit Hilfe der sechs Zugriffsmethoden (1) kann kontrolliert auf die internen Datenstrukturen der Klasse zugegriffen werden. Beispielsweise kann mit Hilfe der Methode `long GetSpeichergroesse()` die Speichergröße der Klasse abgefragt werden. Im Gegensatz dazu kann mittels der Methode `SetSpeichergroesse(long)` die Speichergröße der Klasse gesetzt werden. Insbesondere kann die Methode `SetSpeichergroesse(long)` den zu setzenden Wert überwachen. Wird beispielsweise der Wert 1000 der Methode `SetSpeichergroesse(long)` als Parameter übergeben und diese akzeptiert jedoch nur Werte zwischen 0...500, so wird der übergebene Wert nicht gesetzt und ein Fehlertext ausgegeben.

  Das folgende Listing zeigt die vollständige Methode `SetSpeichergroesse(long)` in C++ Notation:

```
1 void Regelung::SetSpeichergroesse(long theSize)
2 {
3   if ( (theSize >500) || (theSize < 0))
4     std::cout << "Zahlenwert␣ungültig!" << std::endl;
5   else
6     Speichergroesse = theSize;
7 }
```

- **Verhaltensbeschreibende Methoden** der Klasse *Regelung*
  Unter (2) ist die verhaltensbeschreibende Methode `float CalcStromverbrauch()` in den Rumpf der Klasse *Regelung* eingetragen worden. Mittels dieser Methode wird das Verhalten der Klasse modelliert und beschrieben. Die Methode `float CalcStromverbrauch()` berechnet beispielsweise den Stromverbrauch der Klasse *Regelung* inklusive aller davon abgeleiteten Klassen des Systems Kaffeemaschine.

□

Wie das obige Beispiel aufzeigt, kann mit Hilfe der Methoden (genauer der Methodenrümpfe) eines Objekts/Klasse das Verhalten des Objekts/Klasse beschrieben und modelliert werden, unabhängig davon ob die zu modellierende Methode (Verhalten) aus der Domäne Informatik, Elektrotechnik oder Mechanik stammt. Für die Methodenbeschreibung ist lediglich eine eineindeutige formale domänenunabhängige Notation notwendig.

Im weiteren Verlauf dieser Arbeit werden die Signaturen der Methoden eines Objekts/Klasse entweder in der objektorientierten Programmiersprache C++ oder in der hier eingeführten formalen BNF-artigen Notation (vgl. „Definition: 2.14“ auf Seite 54) angegeben. Im Gegensatz dazu werden die Methodenrümpfe stets als programmiersprachliche Konstrukte, in Anlehnung an die objektorientierte Programmiersprache C++, formuliert.

2.3.1.3.2. Verhaltensbeschreibung eines Objekts mittels MSCs

Das Verhalten eines Objekts/Klasse kann auch, wie bereits in der Einleitung zu diesem Abschnitt erwähnt, mittels *Message Sequence Charts* beschrieben werden. Dabei wird, im Gegensatz zur Verhaltensmodellierung durch Methodenrümpfe, der Schwerpunkt auf die exakte Beschreibung der Sequenz der zu sendenden bzw. empfangenden Nachrichten eines Objekts/Klasse gelegt. Die Abbildung 2.23 zeigt den grundsätzlichen Aufbau eines einfachen MSCs.

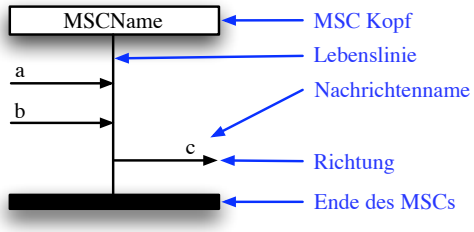


Abbildung 2.23.: Grundsätzlicher Aufbau eines MSCs.

Im Kopf des MSCs ist der eineindeutige Name des MSCs eingetragen. Zwischen dem MSC Kopf und den Ende des MSCs ist eine Lebenslinie eingezeichnet. Das Beispiel MSC hat zwei ankommende Nachrichten (a,b) sowie eine abgehende Nachricht (c). Innerhalb des MSCs ist die Reihenfolge der Nachrichten sowie die Richtung der Nachricht eindeutig festgelegt, also $a \rightarrow b \rightarrow c$. Nicht definiert hingegen ist die Zeit die von einer Nachricht zur nächsten vergeht[76].

Definition 2.15 ***Verhaltensbeschreibung eines Objekten/Klassen mittels Message Sequence Charts***
Das ***Verhalten****, genauer der* ***Nachrichtenaustausch****, zwischen Objekten/Klassen eines Systems kann mittels* ***Message Sequence Charts*** *(MSCs) vollständig modelliert und beschrieben werden.*

Beispiel 14: Modellierung des Verhaltens einer Klasse mit Hilfe eines MSCs
Die Abbildung 2.24 auf der nächsten Seite zeigt in der Bildmitte das Klassendiagramm *Rechner*, mit der Signatur der verhaltensbeschreibenden Methode `+ int add(int a, int b)` sowie dem dazugehörigen, in C++ Notation (Bildmitte unten), ausgeführten Methodenrumpf (Quellcode). Auf der linken Seite der Abbildung ist der Zustandsautomat angegeben mit dem das Verhalten der Klasse ebenfalls modelliert werden kann. Rechts ist das zur Klasse gehörige MSC dargestellt. Alle drei Arten der Verhaltensbeschreibung der Klasse *Rechner* sind dabei gleichwertig.

Wie die Abbildung 2.24 zeigt, kann das Verhalten des Objekts/Klasse *Rechner* sowohl mittels eines Zustandsautomaten (links), Quellcode (Bildmitte) bzw. eines Message Sequence Charts (rechts) beschrieben werden[77]. Dabei gilt: Für die Verhaltensbeschreibung eines Objekts und insbesondere die Kompatibilitätsmodellierung und -bewertung muss zunächst die Signatur der Methoden mittels der oben eingeführten formalen BNF-artigen Notation beschrieben werden (vgl. Kapitel 2.3.1.3.1). Um das eigentliche Verhalten der Klasse zu beschreiben kann entweder der Methodenrumpf in einer Programmiersprache notiert, oder das Verhalten mittels MSCs bzw. Zustandsautomaten oder Sequenzdiagrammen beschrieben werden. Das MSC der Klasse *Rechner* beispielsweise legt fest, dass zuerst das Signal *a* dann das Signal *b* anliegen muss. Nachdem beide Signale angelegt sind berechnet die Klasse das Ergebnis und gibt es als Signal *c* aus. Sowohl der Zustandsautomat, als auch der in C++ notierte Methodenrumpf beschreibt genau das selbe Verhalten der Klasse in einer anderen Darstellungsart. Die drei unterschiedlichen Notationsarten zur Verhaltensbeschreibung der Klasse *Rechner* unterscheiden sich dabei lediglich durch die Art der Notation bzw. durch die darin abgebildete Informationstiefe. Beispielsweise stellt die C++ Notation sowohl die Signatur der Methode als auch den Methodenrumpf sehr kompakt dar. Im Gegensatz dazu ist die Signalreihenfolge sowohl im MSC als auch im Zustandsautomaten besser dargestellt. □

[76] Eine ausführliche Beschreibung der unterschiedlichen MSCs finden Sie unter [Krü00], [Wik07m], [Ker05] sowie [Eck07].
[77] Anmerkung: Im Allgemeinen ist die Mächtigkeit einer Programmiersprache wesentlich größer als die Mächtigkeit von MSCs bzw. von Zustandsautomaten (vgl. Chomsky Hierarchie [PDB94], [Wik08d] und [PDM08]).

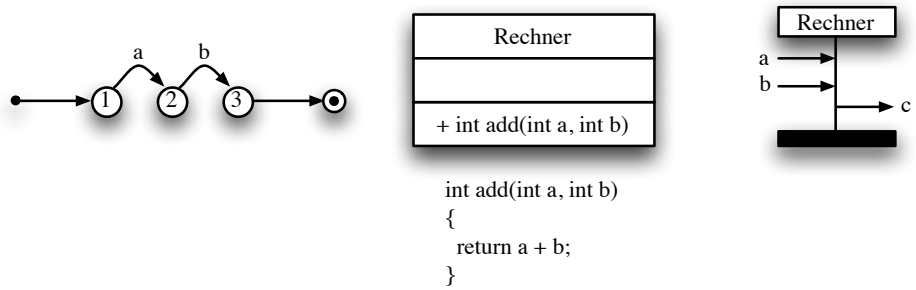


Abbildung 2.24.: Verhaltensbeschreibung eines Objekt eines Systems mit Hilfe von Zustandsautomaten (links) bzw. Message Sequence Charts (rechts).

Nachdem nun sowohl die Eigenschaften, als auch die Methoden eingeführt worden sind, folgt im kommenden Kapitel die Beschreibung der Schnittstellen zwischen Objekten/Klassen eines Systems.

2.3.1.4. Modellierung von Klassenschnittstellen

Damit eine Klasse mit ihrer Umwelt interagieren kann, muss sie zum einen Dienste anbieten, die andere Klassen des Systems benutzen können, und zum anderen kann die Klasse (genauer die Methoden der Klasse) auch Dienste (Eigenschaften und Methoden) anderer Klassen in Anspruch nehmen, um ihre Aufgaben erfüllen zu können. Aus diesem Grund hat jede Klasse zwei getrennte Schnittstellenteile. Eine, in der diejenigen Dienste (Eigenschaften und Methoden) eingetragen sind, die sie anderen Klassen des Systems anbietet, die so genannte „anbietende Schnittstelle“, und einen Schnittstellenteil, in der sämtliche dafür benötigten Eingangsgrößen eingetragen sind. Dieser Schnittstellenteil der Klasse wird auch als „benötigte Schnittstelle“ der Klasse bezeichnet. Zusammen bilden die beiden Schnittstellenteile die Schnittstelle der Klasse. Die Abbildung 2.25 zeigt beispielhaft, wie die Schnittstelle einer Klasse mit Hilfe der so genannten „Lollipop“ Notation modelliert und dargestellt werden kann.

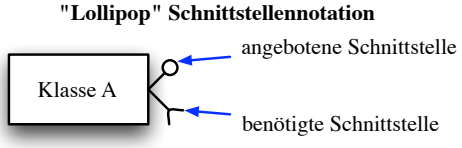


Abbildung 2.25.: Modellierung der Schnittstelle einer Klasse.

Die Klasse *Klasse A* hat demnach zwei (öffentliche) Schnittstellenteile nach außen. Zum einen benötigt die Klasse Information von der Umwelt der Klasse um ihren Aufgabe verrichten zu können. Dies zeigt die Klasse mittels eines offenen Lollipops an. Zum anderen bietet die Klasse Dienste der sie umgebenden Umwelt an. Dies signalisiert sie mittels des geschlossenen Lollis. Welche Daten die Klasse für die Ausführung ihrer Arbeit benötigt bzw. welche Dienste sie ihrer Umwelt zur Verfügung stellt, geht aus dieser einfachen Schnittstellenbeschreibung jedoch nicht hervor.

Das nachfolgende Beispiel zeigt sowohl die anbietende als auch die benötigte Schnittstelle der Klasse *Regelung* des Systems Kaffeemaschine.

Beispiel 15: Modellierung der Schnittstelle der Klasse „Regelung“ der Kaffeemaschine
Die Klasse *Regelung* des Beispielsystems Kaffeemaschine besitzt drei private Eigenschaften sowie sechs öffentliche Zugriffsmethoden auf die privaten Eigenschaften der Klasse. Die sechs öffentlichen Methoden der Klasse bilden zusammen die Schnittstelle der Klasse. Die Abbildung 2.26 auf der nächsten Seite zeigt das Modell der Klasse *Regelung* des Beispielsystems Kaffeemaschine inklusive der Schnittstelle der Klasse.

Dabei ist weder in der anbietenden noch in der benötigten Schnittstelle eingetragen, welche Eigenschaften und Methoden die Klasse genau anbietet/benötigt. Nur aus dem Zusammenhang mit der Methodendeklaration innerhalb der Klasse ist ersichtlich, dass auf jede der sechs Methoden der Klasse von außen zugegriffen werden kann. Welche Daten die Klasse von ihrer Umwelt benötigt, um ihren Dienst zu verrichten, ist nur aus den Methodenrümpfen ersichtlich. Die drei Eigenschaften der Klasse sind nicht direkt von außerhalb der Klasse über die Schnittstelle zu erreichen. Auch diese Information ist nicht in der Schnittstelle sondern lediglich aus dem Deklarationsteil der Klasse ersichtlich. □

Modellierung von Verbindungen zwischen den Schnittstellen von Klassen
Um die Interaktion zwischen zwei Klassen eines Systems modellieren und graphisch darstellen zu können, werden zwischen den Schnittstellen der beteiligten Klassen gestrichelte Linien eingezeichnet. Die Abbildung 2.27 auf der nächsten Seite zeigt die Verbindung zwischen den Schnittstellen der Klassen *Klasse A*, *Klasse B* und *Klasse C*.

Dabei bedeutet beispielsweise die Verbindung der beiden Schnittstellen zwischen den Klassen *Klasse A* und *Klasse B*, dass diese beiden Klassen miteinander Daten austauschen bzw. die Dienste der jeweils anderen Klasse in Anspruch nehmen. Die Klasse

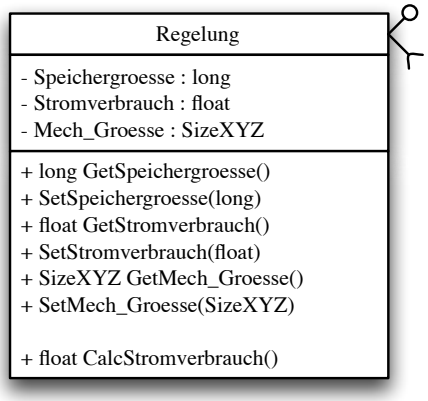


Abbildung 2.26.: Modellierung der Schnittstelle der Klasse *Regelung* des Beispielsystems Kaffeemaschine.

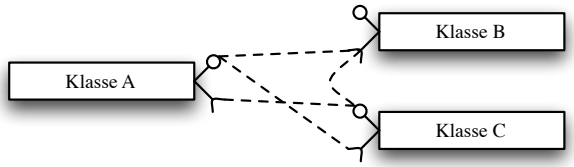


Abbildung 2.27.: Modellierung der Verbindung zwischen den Schnittstellen von Klassen.

Klasse A bietet dabei Daten und Informationen über ihre „anbietende Schnittstelle“ an. Die Klasse *Klasse B* kann ohne die Daten der Klasse *Klasse A* ihre Aufgaben nicht erfüllen. Aus diesem Grund ist die „benötigte Schnittstelle“ der Klasse *B* mit der angebotenen Schnittstelle der Klasse *Klasse A* verbunden.

Das folgende Beispiel zeigt die Verbindung zwischen den Schnittstellen der Klassen des Beispielsystems Kaffeemaschine.

Beispiel 16: Modellierung der Verbindungen zwischen den Schnittstellen der Klassen des Beispielsystems Kaffeemaschine
Die Abbildung 2.28 zeigt einen kleinen Ausschnitt aus dem ursprünglichen Klassendiagramm des Beispielsystems Kaffeemaschine. In diesem Diagramm sind beispielhaft für alle anderen die vier Klassen *Steuerung, Mechanik, Elektronik* und *Software* inklusive der Schnittstellen herausgegriffen worden. Zusätzlich zu den Klassen sind in dieser Abbildung die Verbindungen zwischen den Schnittstellen der Klassen modelliert und dargestellt.

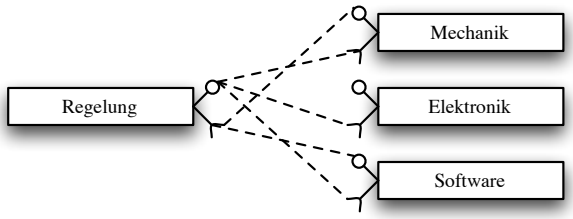


Abbildung 2.28.: Modellierung der Verbindung zwischen den Schnittstellen der Klassen *Regelung, Mechanik, Elektronik* und *Software* des Beispielsystems Kaffeemaschine.

Aus dem Interaktionsmodell der Klassen geht, wie oben erläutert, zwar die Existenz einer Verbindung und der damit verbundene Datenaustausch hervor, jedoch ist nicht genauer spezifiziert, welche Daten ausgetauscht werden. Diese Information kann nur aus den Klassen extrahiert werden. □

Anmerkung:
Nähere Informationen zu den unterschiedlichen Ausprägungen der Schnittstellen von Klassen finden Sie im Kapitel „3.4.3 *Die Unified Modelling Language – UML/UML2*“ ab Seite 131, sowie unter [CS04] und [PDB05].

2.3.1.5. Instantiierung einer Klasse

Nachdem nun die wichtigsten Begriffe und Definitionen der Objektorientierung in den vorangegangenen Kapiteln eingeführt worden sind, ist es an der Zeit, den Zusammenhang zwischen einem *Objekt* und einer *Klasse* zu erläutern. Bei der objektorientierten Modellbildung – genauer während des objektorientierten Modellbildungsprozesses (siehe hierzu den Abschnitt „Vorgehen bei der objektorientierten Modellbildung“ ab Seite 41) – wurde aus einem realen Objekt ein abstraktes Objekt erzeugt. Anschließend

wurde versucht, die Gemeinsamkeiten aller identifizierten Objekte zu finden und diese zu Klassen zusammenzuführen. Wobei die Klassen eine abstrakte Beschreibung des realen Objekts darstellten. Bei der *Instantiierung* wird aus einer Klasse ein (konkretes) Objekt erzeugt.

Definition 2.16 ***Instantiierung einer Klasse***
Als ***Instantiierung*** *wird der Prozess bezeichnet, der aus einer (allgemeinen)* ***Klasse*** *ein konkretes* ***Objekt*** *generiert.*

Bei der Instantiierung einer Klasse wird aus einer Klasse ein konkretes Objekt erzeugt. Dabei werden sämtliche internen Datenstrukturen des Objekts mit konkreten/quantitativen Werten vorbelegt. Die Methoden einer Klasse werden dabei einmal für alle Objekte, die aus der Klasse generiert werden, angelegt. Außerdem werden bei der Instantiierung sämtliche mit der Klasse verbundenen Klassenstrukturen ebenfalls instantiiert[78]. Das folgende Beispiel zeigt die Instantiierung der Klasse *Regelung* aus dem Beispielsystem Kaffeemaschine.

Beispiel 17: Instantiierung der Klasse *Regelung* des Beispielsystems Kaffeemaschine
Angenommen innerhalb des Systems Kaffeemaschine gäbe es zwei unabhängige Regelungen, dann könnten die beiden in der Abbildung 2.29 dargestellten Regelungen, aus der Klasse *Regelung* generiert werden. Die Generierung der beiden Regelungen *myRegelung01* und *myRegelung02* wird in der Abbildung 2.29 exemplarisch dargestellt.

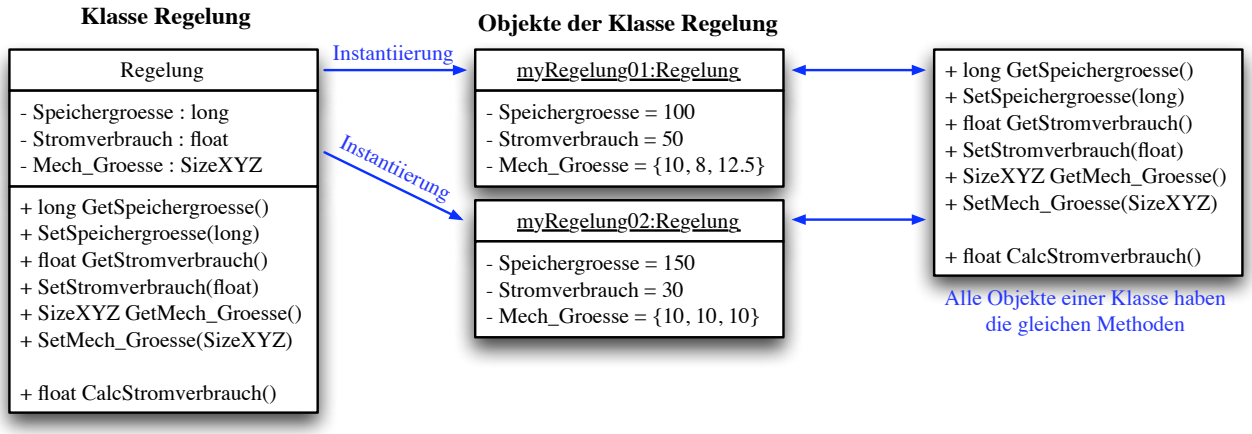


Abbildung 2.29.: Instantiierung der Klasse *Regelung* aus dem Beispielsystem Kaffeemaschine.

Auf der linken Seite der Abbildung ist die Klasse *Regelung* abgebildet. Aus dieser Klasse werden nun die beiden Objekte *myRegelung01* und *myRegelung02* mittels Instantiierung erzeugt (Bildmitte). Dabei werden sämtliche internen Datenstrukturen (Eigenschaften) der Klasse mit konkreten Werten belegt. Außerdem wird für alle Objekte, die aus der Klasse *Regelung* gebildet werden, der Methodenteil der Klasse *Regelung* einmal erzeugt und mit den Objekten „verlinkt“ (rechts). □

Nachdem nun sämtliche grundlegenden Konzepte und Eigenschaften der objektorientierten Modellbildung vorgestellt worden sind, folgt im nächsten Kapitel die Erweiterung und Anpassung des allgemeinen objektorientierten Modellbildungsparadigmas auf die Modellierung und Bewertung von Kompatibilität.

2.3.2. Erweiterung des objektorientierten Modellbildungsparadigmas auf die objektorientierte Kompatibilitätsmodellierung und -bestimmung

Für die objektorientierte Kompatibilitätsmodellierung und -bestimmung reichen die im letzten Kapitel vorgestellten grundlegenden objektorientierten Modellbildungskonzepte bei weitem nicht aus. Aus diesem Grund werden hier, aufbauend auf dem obigen Grundlagenkapitel, sämtliche notwendigen Erweiterungen des allgemeinen objektorientierten Modellbildungsparadigmas vorgestellt, die für die objektorientierte Kompatibilitätsmodellierung und -bestimmung notwendig sind.

Notwendigkeit der Erweiterung des allgemeinen objektorientierten Modellbildungsparadigmas für die objektorientierte Kompatibilitätsmodellierung und -bestimmung
Bei der Planung und Herstellung von komplexen technischen Systemen, die im Allgemeinen aus mehreren hundert unterschiedlichen Komponenten (Objekten/Klassen/Baugruppen) bestehen können, treten immer wieder Inkompatibilitäten an den Schnittstellen zwischen den einzelnen Baugruppen des Systems auf. Dies liegt zum einen an der oft mangelhaften Dokumentation der einzelnen Baugruppen des Systems und zum anderen an einer fehlenden fächerübergreifenden Validierungsmethode für die Schnittstellen des Systems. Das nachfolgende Beispiel illustriert die fatale Verkettung von Inkompatibilitäten zwischen zwei Baugruppen eines Satelliten.

[78] Für die Vorbelegung/Initialisierung der internen Datenstrukturen eines Objekts bei der Instantiierung der Klasse bieten die meisten objektorientierten Programmiersprachen eine spezielle Methode, den so genannten *Konstruktor* an. Mit dessen Hilfe werden die internen Datenstrukturen des Objekts mit bestimmten, im Konstruktor gespeicherten oder übergebenen Werten bei der Konstruktion des Objekts initialisiert.

Beispiel 18: Einführendes Beispiel
Im Jahre 1999 zerschellte die NASA Raumsonde *Mars Climate Orbiter* nach einer Flugzeit von ca. neun Monaten beim Landeversuch auf der Marsoberfläche. Wie sich nach der Rekonstruktion der empfangenen Telemetriedaten herausgestellt hat, kam es bereits bei der Planung und Konstruktion der Sonde zu einer Verkettung schwerwiegenden Fehlern.

Als Hauptfehlerursache, die unmittelbar zum Absturz und somit zum Verlust der Sonde geführt hat, wurde ein *Umrechnungsfehler* zwischen *zwei Einheiten* und der damit verbundenen *Maßstäbe* identifiziert. So berechnete das Programm „SM_Force“ der Firma Lockheed Martin Astronautics[79] alle auftretenden Kräfte und Drehmomente in der englischen Maßeinheit „Pfund“, während das Computermodell am Jet Propulsion Laboratory[80] alle Angaben im metrischen System und in „Newton“ erwartete. Durch diesen fatalen Umrechnungs- bzw. Einheitenfehler der Ingenieure wurde bei der Simulation der Flugbahn der Sonde statt mit einer Abweichung von $1\text{Pfund} * \text{Sekunde}$ nur mit einer Abweichung von $1\text{Newton} * \text{Sekunde}$ gerechnet. Richtig wäre jedoch $4,45\text{Newton} * \text{Sekunden}$ gewesen. Aufgrund dieses Rechenfehlers erreichte die Sonde nicht die vorherberechneten Bahnkoordinaten um in eine Umlaufbahn um den Mars einzuschwenken. Statt dessen zerschellte sie auf der Oberfläche des Mars.

Anmerkung:
Eine genauere Beschreibung der Ursachen des *Mars Climate Orbiter* Absturzes finden Sie unter [BPDN02, 6-8]. Weiterführende Informationen zum Themenbereich *Softwarefehler* finden Sie unter [Gie06]. □

Das obige Beispiel zeigt eindrucksvoll, dass die einzelnen Baugruppen (Subsysteme) des Systems *Mars Climate Orbiter* korrekt funktioniert haben, jedoch trat bei der Interaktion zwischen den einzelnen Subsystemen des Satelliten ein Kommunikationsfehler an der Schnittstelle zwischen den beteiligten Subsystemen, aufgrund einer nicht exakt spezifizierten Schnittstelle, auf. Die Schnittstelleninkompatibilität führte schließlich zum Scheitern der Mission.

Um Fehler, wie sie im obigen Beispiel geschildert wurden, zu vermeiden werden in den nachfolgenden Unterabschnitten dieses Kapitels die grundlegenden Konzepte vorgestellt, die verhindern sollen, dass es zu Kompatibilitätsfehlern an den Schnittstellen zwischen einzelnen Baugruppen (Objekten/Klassen) eines technischen Systems kommen kann. Dazu wird das im letzten Kapitel eingeführte objektorientierte Modellbildungsparadigma so erweitert, dass sämtliche kompatibilitätsrelevanten Aspekte aller Schnittstellen zwischen Objekten des Systems modelliert und getestet werden können.

2.3.2.1. Identifikation der kompatibilitätsrelevanten Systemeigenschaften

Der wohl wichtigste Schritt bei der objektorientierten Kompatibilitätsmodellierung und -bewertung von technischen- und insbesondere von softwarelastigen eingebetteten Systemen, besteht in der Identifikation sämtlicher kompatibilitätsrelevanter Eigenschaften sowie des kompatibilitätsrelevanten Verhaltens eines Objekts/Teilsystems/Systems. Werden bei der Identifikation der kompatibilitätsrelevanten Eigenschaften bzw. des Verhaltens Fehler gemacht, so können sich diese unter Umständen auf das gesamte System negativ auswirken. So geschehen beim oben vorgestellten Absturz der NASA Sonde *Mars Climate Orbiter*. Hier wurden offensichtlich nicht alle kompatibilitätsrelevanten Eigenschaften und Funktionen der Subsysteme des Systems Satellit identifiziert, so dass es durch einen Datenfehler in der Software bei der Kommunikation zwischen zwei Subsystemen zu diesem folgenschweren Absturz der Sonde gekommen ist. Aus diesem Grund ist sowohl eine saubere Definition der kompatibilitätsrelevanten Eigenschaften als auch des kompatibilitätsrelevanten Verhaltens des Systems notwendig.

Zusätzlich zur formalen Beschreibung des kompatibilitätsrelevanten Verhaltens als auch der Eigenschaften eines Objekts / einer Klasse ist ein dezidierter Identifikationsprozess, ähnlich dem im Kapitel „*Was hat Systems Engineering mit modellbasierter Kompatibilitätsbewertung zu tun?*“ vorgestellten DIBMUK-Prozess, zur sicheren Identifikation der kompatibilitätsrelevanten Eigenschaften und des Verhaltens notwendig.

Anmerkung:
Wenn im weiteren Verlauf dieser Arbeit von den *kompatibilitätsrelevanten Eigenschaften* eines Objekts / einer Klasse gesprochen wird, so beinhaltet diese implizit auch das kompatibilitätsrelevante Verhalten eines Objekts / einer Klasse[81].

Definition der kompatibilitätsrelevanten Eigenschaften und des kompatibilitätsrelevanten Verhaltens eines Objekts/Klasse
Bereits im Kapitel „2.3.1.2 *Modellierung der Eigenschaften eines Objekts bzw. einer Klasse*“ ab Seite 50 wurde der Prozess zur Identifikation der ausgezeichneten Eigenschaften eines Objekts/Klasse gezeigt. Ausgehend von allen identifizierten Eigenschaften sowie der verhaltensbeschreibenden Methoden eines Objekts/Klasse, werden nun diejenigen Eigenschaften und Methoden bestimmt, die für die Modellierung und Bestimmung der Kompatibilität von besonderem Interesse sind. Dabei kann sich die Identifikation sämtlicher kompatibilitätsrelevanter Eigenschaften sowie des kompatibilitätsrelevanten Verhaltens eines Objekts/Klasse als sehr schwierig zu bewältigende Aufgabe herausstellen. Denn bereits innerhalb einer Fachdomäne ist es im Allgemeinen sehr schwierig, die kompatibilitätsrelevanten Eigenschaften des zu modellierenden Objekts/Klasse zweifelsfrei zu identifizieren. Dies liegt zum einen an der Tatsache, dass zum einen an der Entwicklung eines Objekts/Klasse sehr viele unterschiedliche, unabhängig voneinander arbeitende Ingenieure beteiligt sind und es meistens keine ausgezeichnete Person innerhalb des Entwicklungsteams gibt, die die gesamte Entwicklung überschauen und die kompatibilitätsrelevanten Eigenschaften aufzeigen kann. Und zum anderen, daran, dass im Großen und Ganzen in einem Systemmodell nicht zwischen allen voneinander direkt oder indirekt abhängigen Eigenschaften/Methoden der Objekte/Klassen Verbindungen im Modell hinterlegt sind.

Umso schwieriger gestaltet sich die Systementwicklung und die Kompatibilitätsbewertung, wenn an der Entwicklung eines Objekts/Klasse Ingenieure unterschiedlicher Domänen beteiligt sind. Dies liegt zum einen daran, dass bei den meisten Entwicklungen kein gemeinsames, konsistentes Systemmodell existiert in dem sämtliche domänenspezifische Aspekte des zu modellierenden

[79] Lockheed Martin Astronautics war zuständig für den Bau und Test des Satelliten.
[80] Zu den Verantwortungsbereichen der Firma Jet Propulsion Laboratory gehörte unter anderem die Projektleitung, Softwareentwicklung, Simulation und Überwachung der Mission.
[81] Im weiteren Verlauf dieser Arbeit wird stets vereinfacht „eines Objekts/Klasse“ geschrieben, anstatt „eines Objekts / einer Klasse“.

Systems/Objekts hinterlegt werden können. Zum anderen an den zum Teil sehr unterschiedlichen Modellen, die für die Modellierung in den unterschiedlichen Domänen verwendet werden. Beispielsweise werden in der Mechanik CAD Zeichnungen zur Beschreibung der Struktur eines Objekts verwendet, während in der Elektrotechnik Stromlaufpläne genutzt werden. In einem modernen eingebetteten System existieren jedoch sehr viele Schnittpunkte zwischen den an der Entwicklung beteiligten Welten. Und gerade an den Schnittpunkten zwischen den unterschiedlichen, an der Entwicklung beteiligten Gruppen, kommt es immer wieder zu immensen Kompatibilitätsproblemen, da die Schnittstellen nicht „sauber" definiert sind bzw. die kompatibilitätsrelevanten Eigenschaften eines Objekts/Klasse nicht an die anderen Ingenieure weitergegeben werden. Um diesem Problem zu begegnen, ist ein gemeinsames domänenübergreifendes Systemmodell notwendig, in dem sämtliche kompatibilitätsrelevanten Eigenschaften jeder Domäne enthalten sind[82].

Jedoch bevor zweifelsfrei geklärt werden kann, welche Eigenschaften und Methoden für die Kompatibilitätsmodellierung und -bestimmung ausschlaggebend sind, muss zunächst definiert werden, welche Merkmale eines Objekts/Klasse kompatibilitätsrelevant sind. Die folgende Aufzählung fasst die wichtigsten Merkmale der kompatibilitätsrelevanten Eigenschaften eines Objekts/Klasse kompakt zusammen.

- **Sämtliche kompatibilitätsrelevanten Eigenschaften eines Systems stehen direkt oder indirekt miteinander in Beziehung**
  Alle Eigenschaften eines Systems, die direkt oder indirekt miteinander in Beziehung stehen, werden als kompatibilitätsrelevante Eigenschaften des Systems bezeichnet. Stehen beispielsweise zwei Eigenschaften nicht in Beziehung miteinander, so beeinflussen sie sich im Allgemeinen nicht gegenseitig, weil sie keine gemeinsamen „Berührungspunkte" haben. Dabei ist jedoch besonders zu beachten, dass aufgrund der domänenübergreifenden Modellierung (zum Beispiel von eingebetteten Systemen) nicht alle tatsächlich vorhandenen Verbindungen (Berührungspunkte) zwischen zwei kompatibilitätsrelevanten Eigenschaften des Systems auch in einem der Modelle des Systems enthalten sind. Die Abbildung 2.30 zeigt den geschilderten Sachverhalt anhand eines einfachen Beispiels.

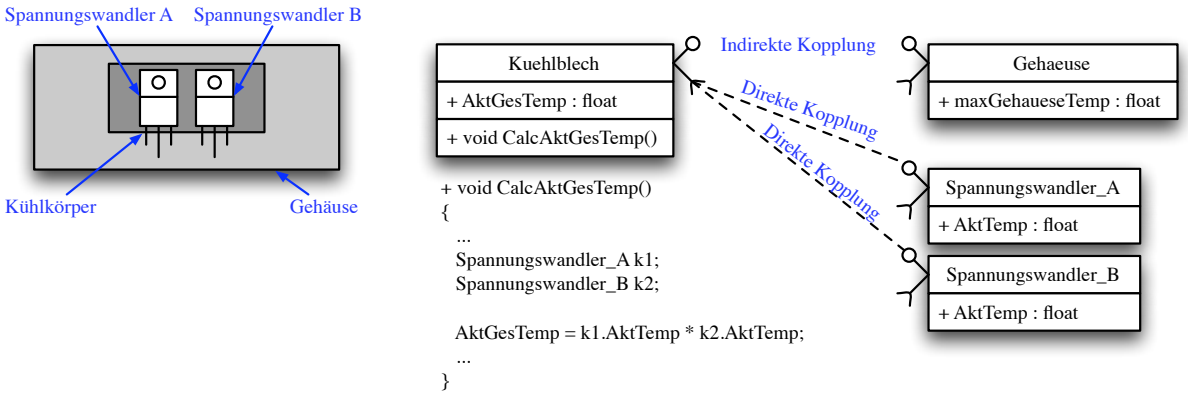


Abbildung 2.30.: Eigenschaften und Methoden müssen direkt oder indirekt miteinander in Beziehung stehen um kompatibilitätsrelevant zu sein.

Dabei ist auf der linken Seite der Abbildung eine stark vereinfachte CAD Zeichnung eines Gehäuses, in dem sich ein Kühlkörper befindet, an den wiederum zwei Spannungswandler angeschlossen sind. Im Modell des Systems (rechts), ist die thermische Verbindung der beiden Spannungswandler mit dem Kühlkörper explizit modelliert worden. Jedoch weder zwischen dem Kühlkörper noch den beiden Spannungswandlern ist eine thermische Verbindung mit dem sie umgebenden Gehäuse modelliert worden. Aufgrund der Tatsache, dass sich der Kühlkörper innerhalb des Gehäuses befindet, besteht jedoch eine indirekte Verbindung zwischen dem Kühlkörper auf der einen und dem Gehäuse auf der anderen Seite. Aus diesem Grund ist zusätzlich eine Verbindung zwischen dem Gehäuse und dem Kühlkörper zu modellieren, da es ansonsten zu einer Inkompatibilität (Temperaturüberschreitung) kommen kann.

Ergebnis: Sind in einem System Eigenschaften vorhanden, die nicht direkt, sondern indirekt miteinander in Beziehung stehen, wie im obigen Beispiel, so muss die indirekte Beziehung explizit in das Modell aufgenommen werden. Geschieht dies nicht, kommt es unweigerlich zu einer Inkompatibilität.

- **Eigenschaften, die sowohl für die Struktur, als auch das Verhalten des Objekts/Klasse relevant sind, werden als kompatibilitätsrelevante Eigenschaften des Objekts/Klasse bezeichnet**
  Sämtliche Eigenschaften und Methoden eines Objekts/Klasse, die sowohl für die Modellierung der Struktur, als auch für die Beschreibung des Verhaltens eines Objekts/Klasse ausschlaggebend sind, werden als kompatibilitätsrelevant eingestuft. Eigenschaften oder Methoden eines Objekts/Klasse die unwesentlich für die Modellierung und Beschreibung des Objekts/Klasse sind, sind demnach nicht direkt für die Kompatibilitätsbestimmung eines Objekts/Klasse verantwortlich. Dabei kann es jedoch vorkommen, dass eine Eigenschaft eines Objekts/Klasse erst durch eine indirekte Verbindung (s.o.) innerhalb des Objekts/Klasse oder zwischen zwei Objekten/Klassen eines Systems zu einer kompatibilitätsrelevanten Eigenschaft wird, da jetzt beispielsweise die Eigenschaft für die Struktur des Systems ausschlaggebend geworden ist.

  Die Abbildung 2.31 auf der nächsten Seite zeigt links eine Ziegelmauer mit einem fehlenden Stein in der Mitte sowie der Beschreibung des fehlenden Steins. Das Loch in der Mauer besitzt die beiden, für die Modellierung der Struktur wesentlichen, Eigenschaften *Größe* und *Gewicht*. Auf der rechten Seite der Abbildung sind zwei alternative Steine dargestellt, sowie deren Eigenschaften tabellarisch aufgelistet.

[82] Siehe hierzu insbesondere [BE08] und [Eck12].

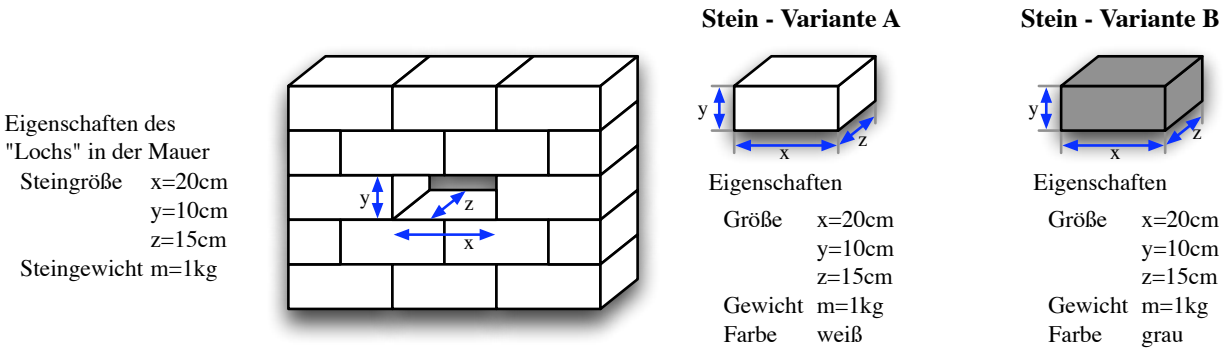


Abbildung 2.31.: Weiße Ziegelmauer mit einem fehlenden Stein in der Mitte sowie zwei alternativen Ersatzsteinen.

Für die Kompatibilitätsbewertung zwischen der Ziegelmauer und den beiden Steinvarianten müssen zwei unterschiedliche Fälle betrachtet werden, obwohl für die Kompatibilität zwischen der Ziegelmauer und den beiden Varianten der Steine sämtliche mechanischen kompatibilitätsrelevanten Eigenschaften der Ziegelmauer von beiden Steinvarianten voll erfüllt werden.

1. Für die Kompatibilität ist die Eigenschaft *Farbe* des Steins **unwichtig**:
   Die Ziegelwand besitzt die beiden, für die Strukturbeschreibung wichtigen Eigenschaften, *Größe* und *Gewicht*. Beide kompatibilitätsrelevanten Eigenschaften der Ziegelwand werden sowohl von den wesentlichen Eigenschaften vom Stein der Variante A, als auch von der Variante B des Steins voll erfüllt. Aufgrund der Tatsache, dass die Ziegelwand keine Eigenschaft *Farbe* besitzt, können beide Steinvarianten ohne Einschränkung verwendet werden. Es wurden sämtliche kompatibilitätsrelevanten Eigenschaften sowohl der Ziegelwand, als auch des Steins voll erfüllt.
2. Für die Kompatibilität ist die Eigenschaft *Farbe* des Steins **wichtig**:
   Besitzt die Ziegelmauer zusätzlich zu den beiden kompatibilitätsrelevanten Eigenschaften *Größe* und *Gewicht*, die auf den ersten Blick unwichtige Eigenschaft *Farbe*, so kann die Eigenschaft *Farbe* in Verbindung mit der ebenfalls unwichtigen Farbeigenschaft der beiden Steinvarianten ebenfalls zu einer kompatibilitätsrelevanten Eigenschaft der Ziegelwand werden. Dies passiert genau dann, wenn die Gesamtfarbe der Ziegelwand kompatibilitätsrelevant ist, weil beispielsweise die Wandfarbe einheitlich sein soll.

Ergebnis: Die wesentlichen struktur- bzw. verhaltensbestimmenden Eigenschaften eines Objekts/Klasse sind in der Regel kompatibilitätsrelevant. Es kann jedoch vorkommen, dass durch die direkte oder indirekte Verbindung zwischen zwei unwichtigen Eigenschaften eine kompatibilitätsrelevante Eigenschaft entsteht (vgl. Kontext).

- **Sämtliche öffentlichen und geschützten Eigenschaften bzw. Methoden eines Objekts/Klasse sind kompatibilitätsrelevant**
  Sowohl für alle öffentlichen und geschützten Eigenschaften eines Objekts/Klasse, als auch für alle öffentlichen und geschützten verhaltensbeschreibenden Methoden eines Objekts/Klasse (die zusammen die Schnittstelle eines Objekts/Klasse bilden) gilt: Diese Eigenschaften und Methoden sind kompatibilitätsrelevant aufgrund der Tatsache, dass sie potentiell von anderen Objekten/Klassen uneingeschränkt verwendet werden können.

  Für alle privaten Eigenschaften und Methoden des Objekts/Klasse gilt: Auch diese können unter Umständen für die Kompatibilität innerhalb eines Objekts/Klasse oder mit einem anderen, verbundenen Objekt/Klasse von Bedeutung sein. Sie können jedoch im Allgemeinen nicht ohne direkten Eingriff in die interne Struktur des Objekt/Klasse verändert werden. Aus diesem Grund werden sie meistens nicht als kompatibilitätsrelevant betrachtet, obwohl sie es sein können.

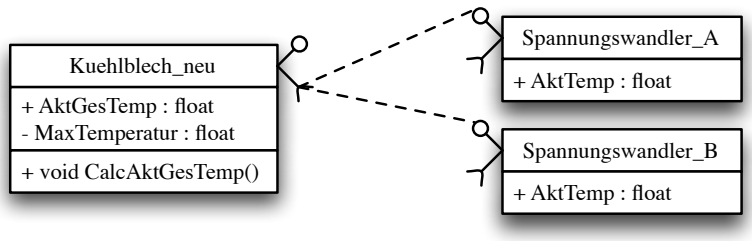


Abbildung 2.32.: Erweiterung des Beispiels aus Abbildung 2.30 um eine private Eigenschaft des Objekts/Klasse *Kuehlblech_neu*.

Die Abbildung 2.32 zeigt links die modifizierte Klasse *Kuehlblech_neu*, die gegenüber der ursprünglichen Klasse aus der Abbildung 2.30 die zusätzliche private Eigenschaft *MaxTemperatur* besitzt. Diese interne Eigenschaft legt fest, bis zu welcher maximalen Temperatur das Kühlblech einwandfrei arbeitet. Obwohl die Eigenschaft *MaxTemperatur* eine interne Eigenschaft der Klasse *Kuehlblech_neu* ist, ist sie für die Kompatibilität sehr wichtig weil sie implizit festlegt, wie viele Spannungswandler an dem Kühlblech befestigt werden können[83].

[83] Bei diesem einfachen Beispiel handelt es sich absichtlich um kein „gutes“ Modell eines realen Kühlblechs. Jedoch um zu zeigen, dass auch private interne Eigenschaften sehr wohl kompatibilitätsrelevant sein können ist es sehr gut geeignet.

Ergebnis: Sämtliche öffentlichen und geschützten Eigenschaften und Methoden eines Objekts/Klasse, die die Schnittstelle des Objekts/Klasse bilden, sind für die Kompatibilität des Objekts/Klasse relevant. Bei der Kompatibilitätsbetrachtung dürfen jedoch die privaten Eigenschaften und Methoden eines Objekts/Klasse nicht vollständig außer Acht gelassen werden.

Aus den obigen drei Voraussetzungen an eine kompatibilitätsrelevante Eigenschaft/Methode eines Objekts/Klasse lässt sich die folgende Definition ableiten:

Definition 2.17 ***Kompatibilitätsrelevante Eigenschaften eines Objekts/Klasse***
*Als **kompatibilitätsrelevante Eigenschaften** und **Methoden** eines Objekts/Klasse werden diejenigen ausgezeichneten Eigenschaften und Methoden des Objekts/Klasse verstanden, die **direkt** oder **indirekt** mit anderen Eigenschaften oder Methoden des selben oder eines damit verbundenen Objekts/Klasse in Beziehung stehen und sowohl für die Struktur, als auch das Verhalten des Objekts/Klasse von **entscheidender Bedeutung** sind. Des Weiteren sind insbesondere diejenigen Eigenschaften und Methoden des Objekts/Klasse als kompatibilitätsrelevant einzustufen, die direkt über die ausgezeichnete **Schnittstelle** des Objekts/Klasse mit anderen Objekten/Klassen in Beziehung stehen.*

In der obigen Definition 2.17 wurden die privaten Eigenschaften und Methoden eines Objekts/Klasse nicht explizit als kompatibilitätsrelevant definiert, obwohl das obige Beispiel gezeigt hat, dass auch private Eigenschaften und Methoden sehr wohl kompatibilitätsrelevant sein können. Dies liegt vor allem daran, dass die interne Struktur eines Objekts/Klasse im Allgemeinen dem Benutzer unbekannt ist (vgl. Information Hiding) und aus diesem Grund auch nicht für die Kompatibilitätsbestimmung herangezogen werden kann. Aus der allgemeinen Definition der kompatibilitätsrelevanten Eigenschaften eines Objekts/Klasse sowie aus der Eigenschaftsdefinition 1.7 auf Seite 15, kann die folgende Abgrenzung zwischen den kompatibilitätsrelevanten- und den nicht kompatibilitätsrelevanten Eigenschaften gezogen werden.

Definition 2.18 ***Menge der kompatibilitätsrelevanten Eigenschaften eines Objekts/Klasse***
Sei A_O die Menge aller Eigenschaften/Methoden eines Objekts/Klasse O. Außerdem sei K_A die Menge der kompatibilitätsrelevanten Eigenschaften/Methoden des Objekts/Klasse mit $K_A \subseteq A_O$. Dann gilt: Die Menge der kompatibilitätsrelevanten Eigenschaften/Methoden ist kleiner oder gleich der Menge aller Eigenschaften/Methoden des Objekts/Systems.

Nach Definition 2.18 sind nicht alle Eigenschaften und Methoden eines Objekts/Klasse für die Kompatibilitätsbestimmung heranzuziehen. Insbesondere ist jedoch darauf zu achten, dass, wie das obige Beispiel gezeigt hat, sehr viele Eigenschaften eines Objekts/Klasse als kompatibilitätsrelevant eingestuft werden müssen, die auf den ersten Blick unwichtig für die Kompatibilität des Objekts/Klasse sind. In Kombination mit anderen Objekten/Klassen eines Systems können jedoch zuvor als unwichtig eingestufte Eigenschaften und Methoden eines Objekts/Klasse kompatibilitätsrelevant werden.

Nachdem nun festgelegt worden ist, was unter den kompatibilitätsrelevanten Eigenschaften eines Objekts/Klasse verstanden wird, folgt die Vorstellung eines Prozesses, mit dessen Hilfe die kompatibilitätsrelevanten Eigenschaften eines Systems identifiziert und bewertet werden können.

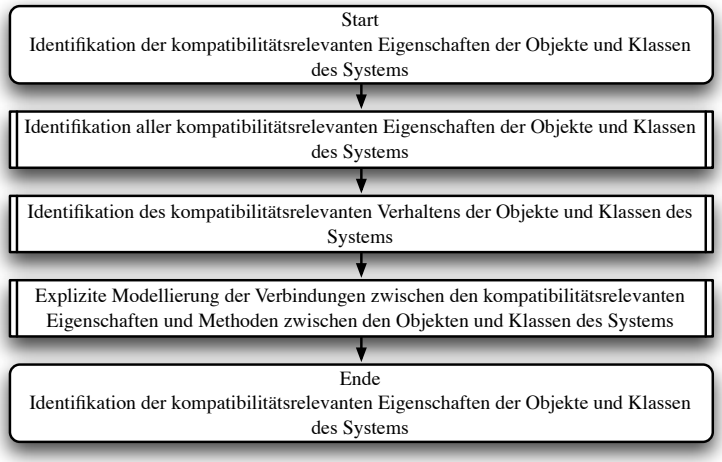


Abbildung 2.33.: Identifikation der kompatibilitätsrelevanten Eigenschaften eines Objekts/Klasse.

Grundsätzliches Vorgehen bei der objektorientierten Identifikation der kompatibilitätsrelevanten Eigenschaften und Methoden eines Objekts/Klasse
Nachdem im vorangegangen Abschnitt die Schwierigkeit der Identifikation der kompatibilitätsrelevanten Eigenschaften und Methoden eines Objekts/Klasse anhand mehrerer einfacher Beispiele aufgezeigt wurden, wird in diesem Abschnitt der generische Identifikationsprozess vorgestellt, mit dessen Hilfe die kompatibilitätsrelevanten Eigenschaften und Methoden eines Systems identifiziert werden können. Insbesondere kann der hier vorgestellte Prozess innerhalb der ersten Phase des DIBMUK-Mikro-Kompatibilitätssicherungsprozesses[84], der Definitionsphase, eingesetzt werden, um die kompatibilitätsrelevanten Eigenschaften,

[84] Siehe hierzu den Abschnitt „Integration von Kompatibilitätsmanagement in die Systems Engineering Welt“ auf Seite 7.

sowie die für das Verhalten des Objekts/Klasse relevanten Methoden sicher zu identifizieren. Nach dem DIBMUK-Prozess bilden die identifizierten kompatibilitätsrelevanten Eigenschaften, mit den kompatibilitätsrelevanten verhaltensbeschreibenden Methoden die Grundlage für den Ablauf des gesamten DIBMUK-Prozess.

Die Abbildung 2.33 auf der vorherigen Seite zeigt das grundsätzliche Vorgehen bei der objektorientierten Identifikation der kompatibilitätsrelevanten Eigenschaften der Objekte/Klassen eines Systems. Grundlage für den hier vorgestellten generischen Identifikationsprozess für die kompatibilitätsrelevanten Eigenschaften eines Objekts/Klasse ist der bereits im Kapitel *„Unterschiedliche Modellbildungsparadigmen“* besprochene allgemeine objektorientierte Modellbildungsprozess. Während des allgemeinen objektorientierten Modellbildungprozesses wurden sämtliche Objekte, Eigenschaften und Methoden des Systems identifiziert und anschließend den einzelnen Klassen des Systems zugewiesen. Aufbauend auf dem durch den objektorientierten Modellbildungsprozess entstanden Klassenmodell, mit den darin enthaltenen Klassen, Schnittstellen und Verbindungen, und insbesondere den Eigenschaften und Methoden der Klassen, folgt nun der Identifikationsprozess der kompatibilitätsrelevanten Eigenschaften und Methoden der einzelnen Objekte/Klassen des zu modellierenden Systems.

Die folgende Aufzählung beschreibt die einzelnen Schritte des in der Abbildung 2.33 graphisch dargestellten generischen objektorientierten Identifikationsprozesses für die kompatibilitätsrelevanten Eigenschaften und Methoden eines Objekts/Klasse. Begonnen wird der generische Identifikationsprozess mit der Identifikation der kompatibilitätsrelevanten Eigenschaften der Objekte/Klassen des Systems.

1. **Identifikation aller kompatibilitätsrelevanten Eigenschaften der Objekte und Klassen des Systems**
   Im ersten Schritt des generischen Identifikationsprozesses werden sämtliche kompatibilitätsrelevanten Eigenschaften jedes Objekts/Klasse des Systems identifiziert. Dabei werden zunächst die ausgezeichneten öffentlichen Eigenschaften, die in der Schnittstelle des Objekts/Klasse stehen, für jedes einzelne Objekts/Klasse des Systems separat untersucht und anhand der Notwendigkeit einer Kompatibilitätsprüfung bewertet. Nachdem diese Prüfung für alle Objekte/Klassen des Systems abgeschlossen worden ist, folgt der zweite Schritt: Die Identifikation derjenigen Eigenschaften des Systems, die aufgrund von Beziehungen zwischen Objekten/Klassen des Systems kompatibilitätsrelevant sind. Die Abbildung 2.34 zeigt den hier beschriebenen Prozess im Detail.

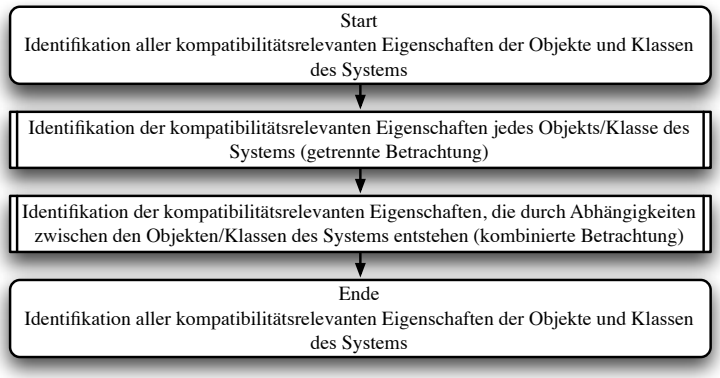


Abbildung 2.34.: Identifikation der kompatibilitätsrelevanten Eigenschaften.

Zunächst werden die öffentlichen Eigenschaften, die in der Schnittstellenbeschreibung jedes Objekts/Klasse stehen, einzeln untersucht und dann eingehend bewertet, ob sie für die Kompatibilitätsmodellierung und -bestimmung herangezogen werden müssen. Dabei werden sämtliche öffentlichen Eigenschaften des Objekts/Klasse einzeln und isoliert, ohne Beziehungen zu anderen Klassen, betrachtet. Im Allgemeinen werden bei der isolierten Betrachtung der Eigenschaften eines Objekts/Klasse sämtliche öffentlichen Eigenschaften des Objekts/Klasse als kompatibilitätsrelevant eingestuft da jede öffentlich zugängliche Eigenschaft eines Objekts/Klasse von allen anderen Objekten/Klassen des Systems beliebig verändert werden kann, ohne dass die Klasse diese Änderung beeinflussen kann. Die Abbildung 2.35 illustriert das Vorgehen bei der isolierten Identifikation der Eigenschaften einer Klasse. Dabei wurden die beiden öffentlichen Eigenschaften der Klasse *K1* – *A* und *B* – als kompatibilitätsrelevant identifiziert.

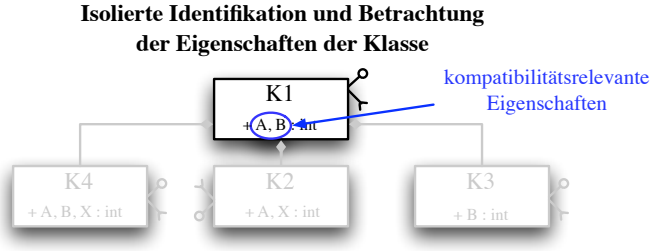


Abbildung 2.35.: Isolierte Identifikation und Betrachtung der kompatibilitätsrelevanten Eigenschaften einer Klasse.

Zusätzlich zu den öffentlichen Eigenschaften einer Klasse können, wie die obige Aufzählung (Einführung Punkt 3) gezeigt hat, auch private Eigenschaften eines Objekts/Klasse für die Kompatibilitätsmodellierung und -bewertung wichtig sein. Wenn die interne Struktur eines Objekts/Klasse bekannt ist, sollte auch diese auf kompatibilitätsrelevante Eigenschaften hin untersucht werden.

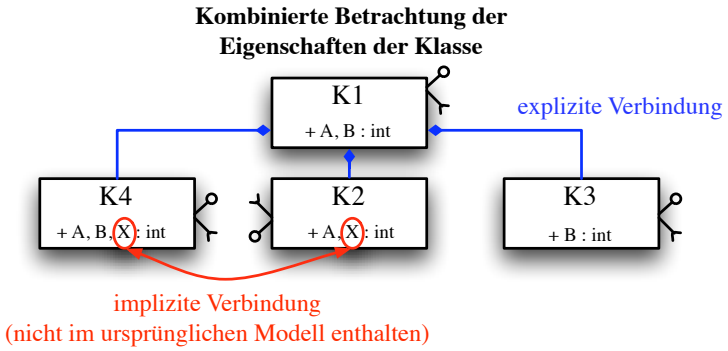


Abbildung 2.36.: Identifikation der kompatibilitätsrelevanten abhängigen Eigenschaften eines Systems.

Nachdem die isolierte Betrachtung der kompatibilitätsrelevanten Eigenschaften jedes Objekts/Klasse des Systems abgeschlossen worden ist, folgt im zweiten Schritt die Untersuchung der abhängigen Eigenschaften der einzelnen Objekte/Klassen des zu untersuchenden Systems. Dabei werden sämtliche expliziten und impliziten Verbindungen zwischen den einzelnen Objekten/Klassen des Systems auf ihre Kompatibilitätsrelevanz hin untersucht und bewertet, über die Eigenschaften ausgetauscht werden. Dabei ist insbesondere darauf zu achten, dass unter Umständen nicht alle kompatibilitätsrelevanten Verbindungen zwischen den einzelnen (kompatibilitätsrelevanten) Eigenschaften der Objekte/Klassen des Systems während der objektorientierten Analyse identifiziert und im Systemmodell hinterlegt worden sind (vgl. Abbildung: 2.36 auf Seite 64 Eigenschaft „X“). Ein weiterer Effekt, der bei der vernetzten Identifikation und Betrachtung der kompatibilitätsrelevanten Eigenschaften des Systems auftreten kann, ist dass eine Klasse neue öffentliche Eigenschaften hinzubekommt, die speziell für die Kompatibilitätsmodellierung und -bestimmung bestimmt sind, jedoch im ursprünglichen Modell nicht enthalten waren. Die folgende Abbildung 2.37 zeigt den soeben geschilderten Fall, dass eine Klasse eine neue, nur für die Kompatibilitätsbestimmung notwendige Eigenschaft hinzubekommt, die im ursprünglichen Modell nicht enthalten war.

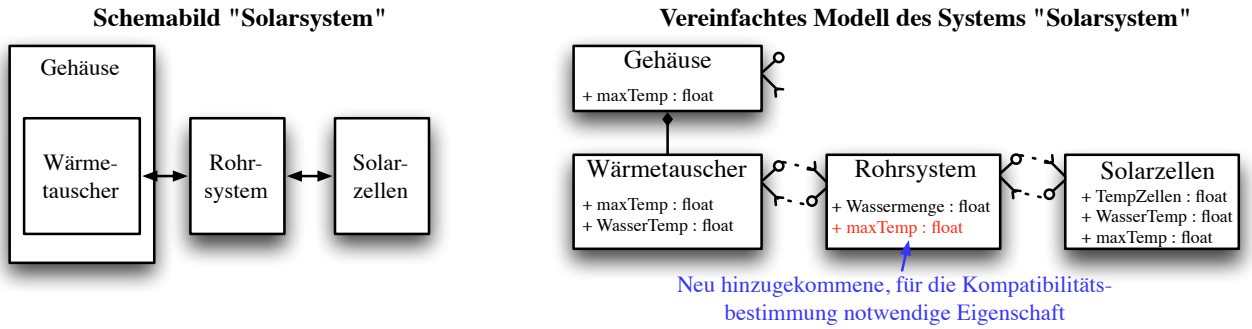


Abbildung 2.37.: Erweiterung einer Klasse um eine weitere Eigenschaft für die Kompatibilitätsbestimmung.

Das in der Abbildung 2.37 dargestellte Solarsystem besteht aus drei Klassen die miteinander in Verbindung stehen. Dabei fällt auf, dass sowohl die Klasse *Wärmetauscher* als auch die Klasse *Solarzellen* die öffentliche Eigenschaft *maxTemp* besitzen, in der sie die maximal zulässige Temperatur der jeweiligen Klasse speichern. Verbunden sind die beiden Klassen *Wärmetauscher* und *Solarzelle* über die Klasse *Rohrsystem*. Das Rohrsystem hat jedoch keine Eigenschaft *maxTemp*. Diese ist jedoch für die Kompatibilitätsbestimmung wichtig, so dass die Klasse *Rohrsystem* um die öffentliche kompatibilitätsrelevante Eigenschaft *maxTemp* erweitert werden muss, um die Kompatibilität des Systems überprüfen zu können.

Um mögliche Fehlerquellen bei der Identifikation und Modellierung der kompatibilitätsrelevanten Eigenschaften eines Systems zu minimieren, sollte die Identifikation der kompatibilitätsrelevanten Eigenschaften stets von Repräsentanten aus allen an der Systementwicklung beteiligten Domänen gemeinsam durchgeführt werden. Dies gilt insbesondere bei der interdiziplinären Systementwicklung, aufgrund der Tatsache, dass in den meisten Modellen des Systems gerade die Schnittstellen zwischen den einzelnen Domänen nicht explizit modelliert werden[85] und es somit an dieser Stelle meistens zu Kompatibilitätsproblemen kommt, die nur von erfahrenen Entwicklern identifiziert werden können (vgl. Abbildung: 2.30 auf Seite 60).

Das folgende Beispiel illustriert das Vorgehen bei der Identifikation der kompatibilitätsrelevanten Eigenschaften am Beispielsystem Kaffeemaschine.

[85] Siehe hierzu insbesondere [BE08].

Beispiel 19: Identifikation der kompatibilitätsrelevanten Eigenschaften des Beispielsystems Kaffeemaschine
Bereits in der Abbildung 1.14 auf Seite 15 wurde ein einfaches Modell einer Kaffeemaschine vorgestellt. Dieses wurde in den folgenden Kapiteln mit Hilfe der objektorientierten Analyse in ein Klassenmodell der Kaffeemaschine überführt (vgl. Abbildung: 2.18 auf Seite 50). Ausgehend von diesen beiden Modellen wird nun der Identifikationsprozess für die kompatibilitätsrelevanten Eigenschaften des Systems Kaffeemaschine exemplarisch ausgeführt.

Dabei wird zunächst, wie im oben eingeführten Verfahren beschrieben, mit der isolierten Identifikation und Betrachtung der kompatibilitätsrelevanten Eigenschaften der Klassen des Systems begonnen.

- *Isolierte Betrachtung der Objekte/Klassen des Systems*
  In diesem Schritt werden sämtliche öffentlichen Eigenschaften jeder Klasse des Systems einzeln auf ihre Relevanz für die Kompatibilitätsmodellierung und -bestimmung untersucht. Dabei werden im Allgemeinen sämtliche öffentlichen Eigenschaften der Klasse als kompatibilitätsrelevant eingestuft. Dies liegt vor allem daran, dass sämtliche öffentlichen Eigenschaften einer Klasse beliebig von allen anderen Klassen des Systems modifiziert werden können[86]. Die Abbildung 2.38 zeigt die isolierte Betrachtung der Klasse *Kaffeekanne*, sowie die identifizierte kompatibilitätsrelevante Eigenschaft der Klasse.

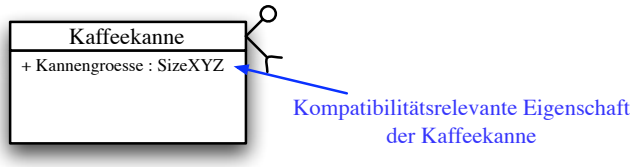


Abbildung 2.38.: Isolierte Betrachtung der Klasse *Kaffeekanne* des Systems Kaffeemaschine.

Bei der isolierten Betrachtung der Klasse *Kaffeekanne* wird lediglich die bereits vorhandene öffentliche Eigenschaft *Kannengroesse* als kompatibilitätsrelevante Eigenschaft der Klasse *Kaffeekanne* eingestuft. Aufgrund der Tatsache, dass die Klasse keine weiteren öffentlichen Eigenschaften besitzt, kann die Untersuchung der Klasse abgebrochen werden.

Wie oben bereits mehrfach erwähnt, können auch interne private Eigenschaften einer Klasse für die Kompatibilität wichtig sein. Im Fall der Kaffeekanne ist jedoch die interne Struktur der Klasse unbekannt, so dass keine weiteren Eigenschaften der Klasse als kompatibilitätsrelevant identifiziert werden können.

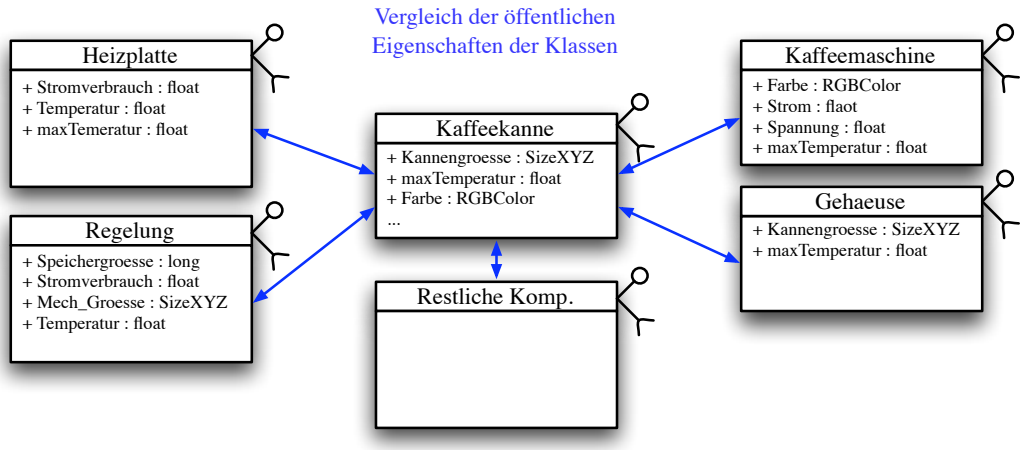


Abbildung 2.39.: Identifikation der abhängigen kompatibilitätsrelevanten Eigenschaften der Klasse *Kaffeekanne* des Beispielsystems Kaffeemaschine.

- *Kombinierte Betrachtung der Objekte/Klassen des Systems*
  Im zweiten Schritt der objektorientierten Identifikation der kompatibilitätsrelevanten Eigenschaften eines Objekts/Klasse, werden sowohl die explizit als auch die implizit modellierten Verbindungen zwischen den einzelnen Objekten/Klassen des Systems betrachtet. Die Abbildung 2.39 zeigt das grundsätzliche Vorgehen bei der Identifikation der abhängigen Eigenschaften der Klassen des Systems. Dabei werden sämtliche Verbindungen, die zwischen den einzelnen Klassen des Systems existieren untersucht, ob eine öffentliche Eigenschaft von einer anderen Klasse des Systems benutzt wird. Ist dabei eine kompatibilitätsrelevante Eigenschaft einer Klasse mit einer nicht kompatibilitätsrelevanten Eigenschaft einer anderen Klasse verbunden, so werden beide Eigenschaften als kompatibilitätsrelevant eingestuft. Haben zwei Klassen eine implizite Verbindung, wie beispielsweise die Kaffeekanne mit der Kaffeemaschine, so kann

[86] Anmerkung:
In einem „guten“ Modell eines Systems, werden nur sehr selten öffentliche Eigenschaften verwendet. Dies liegt vor allem daran, dass öffentliche Eigenschaften von jedem verändert werden können und die Klasse darüber keine Kontrolle hat. In guten Modellen werden aus diesem Grund stets alle Eigenschaften als privat oder protected modelliert und sind nur über so genannte Accessoren von außerhalb der Klasse zugreifbar.

es vorkommen, dass eine Klasse eine weitere Eigenschaft hinzubekommt, die nur für die Kompatibilitätsbestimmung notwendig ist. Sämtliche so identifizierten für die Kompatibilitätsbestimmung notwendigen Eigenschaften müssen in das Modell des Systems aufgenommen werden, um den Kompatibilitätstest durchführen zu können. In der Abbildung 2.39 wird der Identifikationsprozess der abhängigen kompatibilitätsrelevanten Eigenschaften exemplarisch für die Klasse *Kaffeekanne* ausgeführt und beschrieben.

Begonnen wird die Untersuchung mit der öffentlichen Eigenschaft *Kannengroesse* der Klasse *Kaffeekanne*. Dabei stellt sich heraus, dass die Klasse *Gehaeuse* ebenfalls die kompatibilitätsrelevante Eigenschaft *Gehauesegroesse* besitzt. Außerdem ist bekannt, dass die Kaffeekanne innerhalb der Kaffeemaschine sein muss (vgl. Abbildung: 2.9 auf Seite 43). Aus diesem Grund werden beide Eigenschaften *Kannengroesse* und *Gehaeusegroesse* als kompatibilitätsrelevant für das System Kaffeemaschine eingestuft. Des Weiteren wird diese, für die Kompatibilität wichtige Verbindung zwischen den beiden Eigenschaften im dritten Schritt des Identifikationsprozesses explizit modelliert.

Bei der weiteren Betrachtung der expliziten und impliziten Verbindungen der Klasse *Kaffeekanne* mit den restlichen Klassen des Systems zeigt sich, dass aufgrund der Tatsache, dass sich die Kaffeekanne innerhalb des Gehäuses der Kaffeemaschine befindet, die Klasse *Kaffeekanne* eine weitere Eigenschaft – die Eigenschaft *maxTemperatur* – hinzubekommt. Dies liegt daran, dass die Klasse *Gehäuse* eine maximal zulässige Temperatur besitzt, über die das Gehäuse nicht erwärmt werden darf. Aus diesem Grund muss die Klasse *Kaffeekanne* ebenfalls die Eigenschaft *maxTemperatur* enthalten, um zu verifizieren, dass die maximal zulässige Temperatur der Kaffeekanne nicht größer als die maximal zulässige Temperatur des Gehäuses der Kaffeemaschine wird. Auch diese kompatibilitätsrelevante Verbindung wird im dritten Schritt explizit modelliert.

Die als unwichtig für die Kompatibilitätsbestimmung eingestufte Eigenschaft *Farbe* der Klasse *Kaffeemaschine* kann nun ebenfalls kompatibilitätsrelevant werden, sofern die Farbe der Kaffeekanne gleich der Farbe des Systems Kaffeekanne sein muss. Zum gegenwärtigen Zeitpunkt ist jedoch unbekannt, ob die Farbe der Kaffeemaschine gleich der Farbe der Kanne sein muss. Aus diesem Grund wird die Eigenschaft *Farbe* weiterhin als nicht kompatibilitätsrelevant eingestuft.

□

Wie das obige Beispiel zeigt, ist für die sichere Identifikation und Bestimmung der kompatibilitätsrelevanten Eigenschaften eines Objekts/Klasse Hintergrundwissen notwendig, um zu entscheiden, ob eine Eigenschaft kompatibilitätsrelevant ist oder nicht.

2. **Identifikation des kompatibilitätsrelevanten Verhaltens der Objekte und Klassen des Systems**
   Der Identifikationsprozess der kompatibilitätsrelevanten verhaltensbeschreibenden Methoden der Objekte/Klassen verläuft ähnlich dem oben beschriebenen Identifikationsprozess der kompatibilitätsrelevanten Eigenschaften der Objekte/Klassen des Systems. Aus diesem Grund wird er hier nicht noch einmal im Detail ausgeführt. Die Abbildung 2.40 zeigt lediglich das Vorgehen bei der Identifikation der kompatibilitätsrelevanten verhaltensbestimmenden Methoden der Objekte/Klassen des zu untersuchenden Systems.

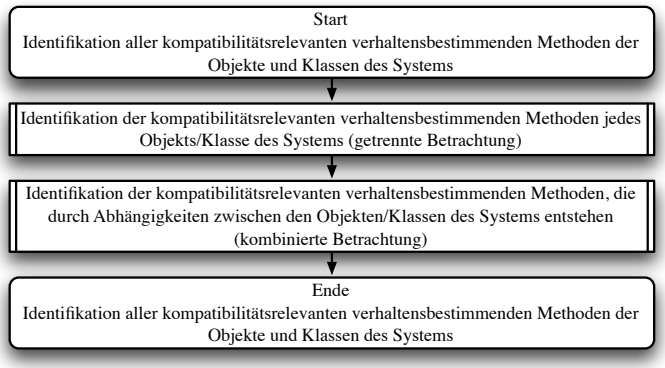


Abbildung 2.40.: Identifikation der kompatibilitätsrelevanten verhaltensbestimmenden Methoden der Objekte/Klassen.

3. **Explizite Modellierung der Verbindungen zwischen den kompatibilitätsrelevanten Eigenschaften und Methoden der Objekte und Klassen des Systems**
   Nachdem nun sämtliche kompatibilitätsrelevanten Eigenschaften und die kompatibilitätsrelevanten verhaltensbeschreibenden Methoden der einzelnen Objekte/Klassen des Systems identifiziert worden sind, folgt in diesem Schritt die explizite Modellierung sämtlicher Verbindungen zwischen den einzelnen öffentlichen Eigenschaften und Methoden der Klassen des Systems. Grundlage für die explizite Modellierung der Verbindungen der kompatibilitätsrelevanten Eigenschaften und Methoden bildet dabei der erste Schritt (1) der objektorientierten Identifikation der kompatibilitätsrelevanten Eigenschaften des Systems sowie die Identifikation der kompatibilitätsrelevanten verhaltensbeschreibenden Methoden des Systems (2).

   Die Abbildung 2.41 auf der nächsten Seite zeigt das Vorgehen bei der expliziten Modellierung der Verbindungen zwischen den einzelnen kompatibilitätsrelevanten Eigenschaften und Methoden des Systems an den Klassenschnittstellen.

   Das folgende Beispiel illustriert die explizite Verbindung der kompatibilitätsrelevanten Eigenschaften des Beispielsystems Kaffeemaschine.

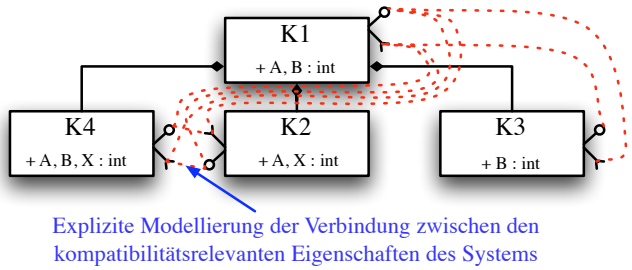


Abbildung 2.41.: Explizite Modellierung der Verbindungen zwischen den kompatibilitätsrelevanten Eigenschaften des Systems.

Beispiel 20: Explizite Modellierung der Verbindungen zwischen den kompatibilitätsrelevanten Eigenschaften des Beispielsystems Kaffeemaschine
Das ursprüngliche Klassenmodell der Kaffeemaschine aus der Abbildung 2.9 auf Seite 43 enthält das stark vereinfachte Modell der Kaffeemaschine, in dem jedoch nicht alle Eigenschaften des Systems enthalten sind. Gegenüber dem ursprünglichen Modell sind nun sämtliche Klassen mit den zuvor identifizierten kompatibilitätsrelevanten Eigenschaften versehen worden, mit Ausnahme der Klasse *Restliche Komp.*, über die keine weiteren Informationen vorhanden sind. Die Abbildung 2.42 zeigt das Klassenmodell des Systems Kaffeemaschine inklusive der explizit modellierten kompatibilitätsrelevanten Verbindungen zwischen den Eigenschaften der einzelnen Klassen des Systems.

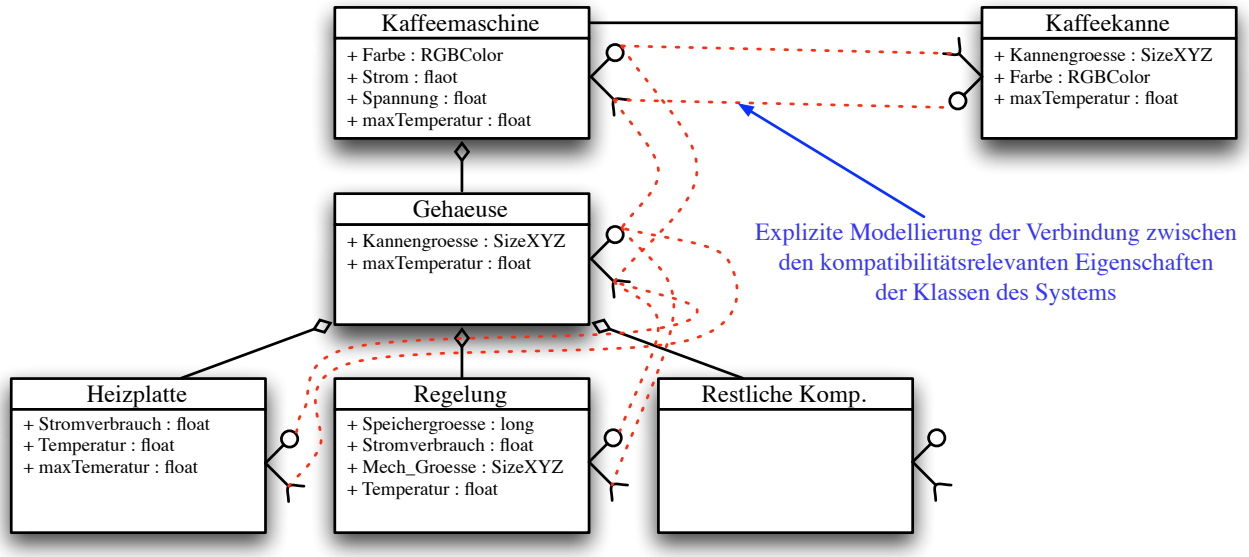


Abbildung 2.42.: Explizite Modellierung aller Verbindungen zwischen den kompatibilitätsrelevanten Eigenschaften des Systems.

□

Nachdem nun sämtliche kompatibilitätsrelevanten Eigenschaften sowie die kompatibilitätsrelevanten verhaltensbeschreibenden Methoden identifiziert wurden, folgt in den folgenden Kapiteln eine ausführliche Beschreibung der unterschiedlichen Erweiterungen der allgemeinen objektorientierten Modellierung der Eigenschaften und Methoden der Objekte und Klassen eines Systems. Begonnen wird dabei mit der Vorstellung der erweiterten Syntax zur Beschreibung der Eigenschaften eines Objekts/Klasse.

2.3.2.2. Erweiterung der Eigenschaftsdeklaration von Objekten/Klassen für die Modellierung von Kompatibilität

Wie das einführende Beispiel des Verlustes der Raumsonde *Mars Climate Orbiter* aufgrund eines Datenfehlers („Einheitenverwechselung“) bei der Übertragung von einem Subsystem des Satelliten zu einem anderen gezeigt hat, ist es besonders wichtig, die Eigenschaften und Methoden jedes einzelnen Objekts/Klasse eines Systems so genau wie möglich formal zu beschreiben. Um dies zu erreichen, reicht die im Kapitel „2.3.1.2 *Modellierung der Eigenschaften eines Objekts bzw. einer Klasse*“ ab Seite 50 vorgestellte BNF-artige formale Notation für die Modellierung der Eigenschaften der Objekte/Klassen eines Systems jedoch nicht aus. Aus diesem Grund wird in diesem Kapitel zunächst die Erweiterung der allgemeinen BNF-artigen formalen Notation der Eigenschaften der Objekte/Klassen eines Systems eingeführt. Im Anschluss daran folgt in den nächsten Unterabschnitten eine ausführliche Beschreibung der einzelnen Aspekte der erweiterten formalen Notation der Eigenschaften der Objekte/Klassen für die Modellierung und Bewertung von Kompatibilität.

Erweiterung der formalen Beschreibung der Eigenschaften eines Objekts/Klasse für die Kompatibilitätsmodellierung und -bewertung
Bereits mehrfach wurde gezeigt, dass für die Durchführung der Kompatibilitätsbestimmung und -bewertung eine eineindeutige formale Notation der Eigenschaften eines Objekts/Klasse zwingend erforderlich ist. Das nachfolgende Listing (2.3) beschreibt die Erweiterung der formalen Notation der Eigenschaften eines Objekts/Klasse, so dass sie für die Kompatibilitätsmodellierung und -bestimmung eingesetzt werden kann. Dabei baut die hier vorgestellte erweiterte formale Notation auf der allgemeinen Eigenschaftsdefinition 2.12 auf.

```
<AttributDeklaration> ::=
    <Sichtbarkeit> (<Eigenschaftstyp>)? <Eigenschaftsname> ":" <Datentyp>
    ("[" "in" (<Teiler>)? <Einheit> ("," <Intervall>)? "]")?
    ("=" <Wert> ("[" "in" (<Teiler>)? <Einheit> ("," <Intervall>)?"]")?)?.

<Sichtbarkeit>              ::= "+" | "#" | "-".
<Eigenschaftstyp>           ::= "const" | "var".
<Datentyp>                  ::= "void" | "bool" | "int" | "long" | "float" | "string"...
<Eigenschaftsname>          ::= <Buchstabe> (<Buchstabe> | <Ziffer>)*.
<Intervall>                 ::= "[" | "]" <Untergrenze> "..." <Obergrenze> "[" | "]".
<Obergrenze>                ::= ("0.")? <Ziffer> (<Ziffer>)*.
<Untergrenze>               ::= ("0.")? <Ziffer> (<Ziffer>)*.
<Teiler>                    ::= <SITeiler> | <SoftwareTeiler>.
<SITeiler>                  ::= "p" | "n" | "m" | "m" | "K" | "M" | "G" | "T".
<SoftwareTeiler>            ::= "Ki" | "Kibi" | "Mi" | "Mebi" | "Gi" | "Gibi" | "Ti" | "Tibi".
<Einheit>                   ::= <SIEinheiten> | <MechanikEinheiten> | <ElektrotechnikEinheiten> | <SoftwareEinheiten>.
<SIEinheiten>               ::= "m" | "kg" | "s" | "A" | "K" | "mol" | "cd".
<MechanikEinheiten>         ::= "m^2" | "m^3" | "Kg" | "°C" | "N" | "p".
<ElektrotechnikEinheiten>   ::= "V" | "A" | "Ohm" | "C" | "L" | "Hz" | "W".
<SoftwareEinheiten>         ::= "bit" | "b" | "Byte".
<Wert>                      ::= (<Buchstabe> | <Ziffer>)+
<Buchstabe>                 ::= "A" | "B" | "C" | "D" ... "Z" | "a" | "b" | "c" | "d" ... "z".
<Ziffer>                    ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "0".
```

Listing 2.3: Erweiterte formale Notation von kompatibilitätsrelevanten Eigenschaften eines Objekts/Klasse

Definition 2.19 ***Formale Beschreibung der erweiterten Eigenschaftsdeklaration eines Objekts/Klasse***
Sämtliche ***kompatibilitätsrelevanten Eigenschaften*** *eines* ***Objekts/Klasse*** *lassen sich mit Hilfe der, im Listing 2.3 beschriebenen, formalen BNF-artigen Syntax formal modellieren und beschreiben.*

Neu gegenüber der ursprünglichen Definition (Definition: 2.12) ist der *Eigenschaftstyp*, mit dessen Hilfe festgelegt werden kann, ob eine Eigenschaft konstant (**`const`**), also sich während der gesamten Lebenszeit des Objekts/Klasse nicht ändert, oder ob die Eigenschaft sich über die Zeit variabel (**`var`**) verhält. Ist eine Eigenschaft mit dem Eigenschaftstyp **`const`** versehen worden, so führt ein Änderungsversuch dieser Eigenschaft zu einem Fehler. Mit Hilfe des Eigenschaftstyps **`const`** kann somit eine invariante Eigenschaft erzeugt werden. Um unnötige „Schreibarbeit" zu vermeiden, kann das Schlüsselwort **`var`** weggelassen werden, denn per Definition sind sämtliche nicht explizit als **`const`** deklarierten Eigenschaften variabel. Zusätzlich kann jede Eigenschaft eines Objekts/Klasse mit einer explizit modellierten *Einheit*, einem *Teiler*, sowie einem *Gültigkeitsintervall* versehen werden[87], um die Eigenschaft möglichst exakt beschreiben zu können.

Wie die obige BNF-artige formale Notation für die Modellierung der kompatibilitätsrelevanten Eigenschaften zeigt, ist es auch weiterhin zulässig, die Eigenschaften eines Objekts/Klasse ohne die zusätzlichen Merkmale, wie beispielsweise die Einheiten und Teiler, zu modellieren. Ohne die hier vorgestellten Erweiterungen kann die Kompatibilität dieser Eigenschaften jedoch nicht verifiziert werden. Aus diesem Grund wird empfohlen, die obige formale Notation für sämtliche Eigenschaften eines Objekts/Klasse zu verwenden, auch wenn diese in erster Linie nicht für die Kompatibilitätsbestimmung verwendet werden. Wie die obigen Beispiele gezeigt haben können auch im Moment „unwichtige" Eigenschaften in Verbindung mit neuen Objekten/Klassen kompatibilitätsrelevant werden. Deshalb sollten stets alle Eigenschaften eines Objekts/Klasse in der erweiterten Notation beschrieben werden.

Ein weiterer Vorteil der durchgängigen Verwendung der erweiterten formalen Notation für die Modellierung und Beschreibung aller Eigenschaften eines Objekts/Klasse liegt in der damit verbundenen Uniformität der Notation. Wodurch sämtliche, an der Entwicklung eines Systems beteiligte Ingenieure, stets die gleiche leicht verständliche Notation verwenden, so dass es zu weniger Missverständnissen bzw. Ungereimtheiten im Modell kommen kann.

Das nachfolgende Beispiel illustriert die Verwendung der erweiterten formalen Notation der Eigenschaften eines Objekts/Klasse anhand des bereits mehrfach angesprochenen Beispielsystems Kaffeemaschine.

Beispiel 21: Modellierung der kompatibilitätsrelevanten Eigenschaften des Beispielsystems Kaffeemaschine
In diesem Beispiel wird die Klasse *Heizplatte* des Klassenmodells der Kaffeemaschine aus der Abbildung 2.39 auf Seite 65 exemplarisch für alle anderen Klassen des Systems mit Hilfe der oben eingeführten erweiterten formalen Notation zur Kompatibilitätsmodellierung und -bestimmung dargestellt. Die Abbildung 2.43 auf der nächsten Seite zeigt die Klasse *Heizplatte* mit der neu eingeführten formalen Notation der Eigenschaften. Von den drei kompatibilitätsrelevanten Eigenschaften der Klasse *Heizplatte* sind die beiden Eigenschaften *Stromverbrauch* und *Temperatur* als variabel deklariert worden, weil sie sich während der gesamten Lebenszeit einer Instanz der Klasse ändern können. Die Eigenschaft *maxTemperatur* hingegen wurde als konstant deklariert. Sollte jemand versuchen, diese Eigenschaft während der Lebenszeit des Objekts zu verändern, wird ein Fehler geworfen.

[87] Anmerkung:
Siehe hierzu die folgenden drei Unterkapitel: „2.3.2.2.1 *Modellierung von Einheiten, dekadischen und nichtdekadischen Präfixen sowie von Gültigkeitsintervallen in der erweiterten formalen Eigenschaftsnotation für die Modellierung und Bestimmung von Kompatibilität*" ab Seite 69, „2.5 *In der Softwaretechnik übliche Präfixe (auch Binärpräfixe genannt)*" ab Seite 72 sowie „2.3.2.2.1.3 *Modellierung von Gültigkeitsintervallen in der erweiterten formalen Eigenschaftsdefinition*" ab Seite 74.

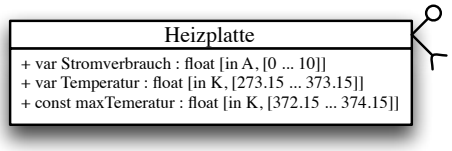


Abbildung 2.43.: Modellierung der Eigenschaften der Klasse *Heizplatte* mit Hilfe der erweiterten formalen Notation für die Kompatibilitätsbestimmung von Systemen.

Neben der expliziten Modellierung der Eigenschaftsart sind sämtliche Eigenschaften der Klasse zusätzlich mit einer Einheit und einem Gültigkeitsintervall versehen worden. Für die Eigenschaft *maxTemeratur* (`+ const maxTemperatur : float [in K, [372.15 ... 374.15]]`) beispielsweise wurde somit festgelegt, dass sie stets die Temperatur in der Einheit K (Kelvin) innerhalb eines Bereichs von 372.15...374.15 erwartet. Stimmt die Einheit, der Datentyp oder der Gültigkeitsbereich nicht, kann der Wert nicht zugewiesen werden da es ansonsten zu einem Kompatibilitätsfehler kommt. Beispielsweise kann der Eigenschaft *maxTemperatur* nicht der Wert 400, also `+ const maxTemperatur : float [in K, [372.15 ... 374.15]] = 400 [in K]` zugewiesen werden, da der Wert 400 größer als die obere Schranke des zulässigen Gültigkeitsintervalls ist. Ebenfalls würden die beiden Zuweisungen `... = 373 [in °C]` oder `... = 373` zu einem Kompatibilitätsfehler (1. Fall), bzw. zu einer Warnung (2. Fall) führen[88]. □

Die oben eingeführte formale Notation zur Modellierung und Beschreibung der (kompatibilitätsrelevanten) Eigenschaften eines Objekts/Klasse kann sowohl für die Modellierung von Elektrotechnik, Mechanik als auch Software gleichermaßen eingesetzt werden.

Sämtliche Erweiterungen gegenüber der allgemeinen formalen Notation der Eigenschaften der Objekte/Klassen, die im obigen Beispiel exemplarisch eingeführt wurden, werden in den folgenden drei Unterkapiteln ausführlich erläutert.

2.3.2.2.1. Modellierung von Einheiten, dekadischen und nichtdekadischen Präfixen sowie von Gültigkeitsintervallen in der erweiterten formalen Eigenschaftsnotation für die Modellierung und Bestimmung von Kompatibilität

In diesem Kapitel werden, aufbauend auf der erweiterten formalen Notation für die Modellierung der kompatibilitätsrelevanten Eigenschaften eines Objekts/Klasse, die international genormten Einheiten zusammen mit den dekadischen und nichtdekadischen Präfixen und Gültigkeitsintervallen eingeführt und erläutert. Eine saubere Definition sowohl der zulässigen Einheiten und Teiler, als auch der Gültigkeitsintervalle ist für die Kompatibilitätsmodellierung und -bestimmung von entscheidender Bedeutung, wie das einführende Beispiel des Absturzes der NASA Raumsonde *Mars Climate Orbiter*[89] eindrucksvoll gezeigt hat.

Außerdem bildet dieses Kapitel die Grundlage für die im Kapitel „*Modellierungssprachen und -techniken für technische Systeme*“ eingeführten Modellierungssprachen für technische Systeme und insbesondere für das Kapitel „*Einführung in die Kompatibilitätsmodellierungssprache (U)CML*“.

2.3.2.2.1.1. Modellierung von Einheiten in der erweiterten formalen Eigenschaftsdefinition

In diesem Abschnitt werden sowohl das international genormten SI-Einheitensystem[90] als auch die gebräuchlichsten Einheiten aus den Fachbereichen Elektrotechnik, Mechanik und Softwaretechnik aufgelistet und erläutert, die sowohl für die Modellierung und Beschreibung aber auch für die Durchführung des Kompatibilitätstests benötigt werden. Die in diesem Abschnitt eingeführten Hard- und Softwareeinheiten bilden außerdem die Grundlage für die Beschreibung der Eigenschaften der in dieser Arbeit verwendeten Hard- und Software Beispielsysteme. Abgeschlossen wird das Kapitel mit einem Beispiel, in dem eine kleine Auswahl der hier vorgestellten Einheiten, in der erweiterten formalen Notation für die Modellierung und Bewertung von kompatibilitätsrelevanten Eigenschaften eines Objekts/Klasse, exemplarisch gezeigt wird.

Das SI-Einheitensystem

Das international genormte SI-Einheitensystem wurde im Jahre 1960 mit sieben verbindlich festgelegten Basiseinheiten veröffentlicht. Mit Hilfe der sieben Basiseinheiten lassen sich alle anderen Einheiten ausdrücken.

Die sieben SI-Basiseinheiten (nach [Bun06a]):

- das **Meter** (`m`) als Einheit der Länge
- das **Kilogramm** (`kg`) als Einheit der Masse
- die **Sekunde** (`s`) als Einheit der Zeit
- das **Ampere** (`A`) als Einheit der elektrischen Stromstärke
- das **Kelvin** (`K`) als Einheit der thermodynamischen Temperatur
- das **Mol** (`mol`) als Einheit der Stoffmenge
- die **Candela** (`cd`) als Einheit der Lichtstärke

[88] Nähere Informationen hierzu finden Sie in den Kapiteln „2.3.2.2.1 *Modellierung von Einheiten, dekadischen und nichtdekadischen Präfixen sowie von Gültigkeitsintervallen in der erweiterten formalen Eigenschaftsnotation für die Modellierung und Bestimmung von Kompatibilität*“ ab Seite 69, „2.3.2.2.1.3 *Modellierung von Gültigkeitsintervallen in der erweiterten formalen Eigenschaftsdefinition*“ ab Seite 74 bzw. „2.5 *Objektorientierte Kompatibilitätsprüfung*“ ab Seite 91.

[89] Siehe hierzu das Beispiel auf Seite 58

[90] Das *Internationale Einheitensystem* kurz *SI* ist eine Abkürzung für frz.: *Système international d'unités*. Im SI werden physikalische und technische Einheiten und Teiler (Präfixe) definiert und genormt. Nähere Informationen zum international genormten Einheitensystem „SI-Einheiten“ finden Sie unter [Bun06a][Bun06b][Wik06l] und 2.3.2.2.1.2 auf Seite 72.

Größe	Formel-zeichen	Einheit	Einheiten-zeichen	Definition
Länge	l	Meter	m	Länge der Strecke, die das Licht im Vakuum während der Dauer von $\frac{1}{299792458}$ Sekunden durchläuft.
Masse	m	Kilo-gramm	kg	Das Kilogramm ist die Einheit der Masse; es ist gleich der Masse des internationalen Kilogrammprototyps[91]. Ein originaler, nicht präfixierter Name für das Kilogramm war das Grave.
Zeit	t	Sekunde	s	Das 9 192 631 770-fache der Periodendauer der dem Übergang zwischen den beiden Hyperfeinstrukturniveaus des Grundzustandes von Atomen des Nuklids Cs_{133} entsprechenden Strahlung.
Stromstärke	I	Ampere	A	Stärke eines konstanten elektrischen Stromes, der durch zwei parallele, geradlinige, unendlich lange und im Vakuum im Abstand von 1 Meter voneinander angeordnete Leiter von vernachlässigbar kleinem kreisförmigem Querschnitt fließend, zwischen diesen Leitern je 1 Meter Leiterlänge die Kraft $2 \cdot 10^{-7}$ Newton hervorrufen würde.
Thermody-namische Temperatur	T	Kelvin	K	Der 273.16 Teil der thermodynamischen Temperatur des Tripelpunktes des Wassers[92].
Stoffmenge	n	Mol	mol	Die Stoffmenge eines Systems, das aus genau so viel Einzelteilchen besteht, wie Atome in 0,012 Kilogramm des Kohlenstoffnuklids C_{12} enthalten sind. Bei Benutzung des Mol müssen die Einzelteilchen spezifiziert sein und können Atome, Moleküle, Ionen, Elektronen und andere Teilchen oder Gruppen solcher Teilchen genau angegebener Zusammensetzung sein.
Lichtstärke	I_V	Candela	cd	Die Lichtstärke in einer bestimmten Richtung einer Strahlungsquelle, die monochromatische Strahlung der Frequenz[93] $540 \cdot 10^{12}$ Hz aussendet und deren Strahlstärke in dieser Richtung $\frac{1}{683}$ Watt durch Steradiant beträgt.

Tabelle 2.2.: SI-Basiseinheiten [Wik06l].

Mit Hilfe dieser sieben Basiseinheiten können sämtliche physikalisch technischen Eigenschaften eines Systems vollständig beschrieben werden. Für die Kompatibilitätsmodellierung und -bestimmung von eingebetteten softwarelastigen Systemen werden jedoch nicht nur die sieben Basiseinheiten verwendet sondern, je nach Fachgebiet, aus den Basiseinheiten abgeleitete Einheiten. Ein großer Teil der aus den SI-Basiseinheiten abgeleiteten Einheiten ist ebenfalls in zahlreichen internationalen verbindlichen Normen zusammengefasst, so dass diese ebenfalls für die Beschreibung von Systemeigenschaften verwendet werden können.

Anmerkung:
Das internationale Büro für Maß und Gewicht (Kürzel BIPM; französisch Bureau international des poids et mesures [Mes06]) ist eine internationale Organisation mit der Aufgabe, ein weltweit einheitliches und eindeutiges System von Maßen auf Basis des SI-Einheitensystems zur Verfügung zu stellen.

Elektrotechnik- und Mechanikeinheiten
In der Elektrotechnik und der Mechanik ist es üblich, nicht nur die von der SI genormten Basiseinheiten für die Beschreibung der Eigenschaften von einzelnen Bauteilen oder komplexen Baugruppen zu verwenden, sondern die zum Teil national oder international genormten domänenspezifischen Einheitenzeichen. Beispielsweise wird in der Elektrotechnik der elektrische Widerstand nicht mit Hilfe der SI-Basiseinheit $1\mathrm{kg} \cdot \frac{\mathrm{m}^2}{A^2 \cdot \mathrm{s}^3} = 1\frac{\mathrm{W}}{\mathrm{A}^2} = \frac{1}{\mathrm{S}} = 1\frac{\mathrm{V}}{\mathrm{A}} = 1\Omega$, sondern mit Hilfe des ebenfalls genormten Einheitenzeichens Ω beschrieben.

Die Tabelle 2.3 enthält einige aus den SI-Basiseinheiten abgeleitete Einheiten aus den beiden Fachbereichen Elektrotechnik und Mechanik.

Softwaretechnikeinheiten
Für die Modellierung von softwaretechnischen Systemen sind die im SI definierten Basiseinheiten nicht einsetzbar da die SI-Einheiten ursprünglich für die Beschreibung von physikalischen Einheiten konzipiert worden sind. Um Software bzw. die Größe von Softwaredatentypen dennoch eindeutig modellieren und beschreiben zu können, wurde das Einheitensymbol *Bit*[94] in der IEC 60027-2 als Basiseinheit aller Software-Datentypen festgelegt. Als Einheitensymbol wurde von der IEC 60027-2 „bit“ festgelegt.

[93] Zurzeit wird an einer neuen Definition der Masseneinheit gearbeitet, die auf der Atommasse und nicht mehr auf einem Prototyp beruhen soll (siehe hierzu den Artikel Kilogramm).
[93] Die Beschreibung des Normals erfolgt durch die Internationale Temperaturskala aus dem Jahr 1990 (ITS-90). Zwischen den Zahlenwerten der thermodynamischen Temperatur T und der Celsiustemperatur ϑ besteht der Zusammenhang: $\{\vartheta_{\mathrm{Grad,Celsius}}\} = \{\mathrm{T_{Kelvin}}\} - 273.15$ ([PDN08], [Wik08l] und [PDIG08]).
[93] Wellenlänge: ca. 555 nm
[94] Das Akronym *bit* stammt aus dem englischen **b**inary dig**it**.

Mechanik-Einheiten		**Elektronik-Einheiten**	
Zeichen	*Beschreibung*	*Zeichen*	*Beschreibung*
m	Meter	V	Volt
m^2	Quadratmeter	A	Ampere
m^3	Kubikmeter	Ω	Widerstand
Kg	Kilogramm	C	Kapazität
°C	Grad Celsius	L	Induktivität
N	Newton	Hz	Frequenz
p	Pascal	W	Leistung

Tabelle 2.3.: Auszug einiger aus den SI-Basiseinheiten abgeleiteten Mechanik- und Elektronikeinheiten.

Im Standard IEEE 1541 hingegen ist „b“ als Einheitensymbol festgelegt worden. Beide Normen basieren zwar auf der gleichen Basisgröße, jedoch verwenden sie unterschiedliche Einheitensymbole. Für die Kompatibilitätsmodellierung und -bestimmung sollte entweder die IEC 6027-2 oder die IEEE 1541 Norm verwendet werden.

Formale Notation der Einheiten für die Kompatibilitätsmodellierung und -bestimmung
Um eine Eigenschaft eines Objekts/Klasse genauer spezifizieren zu können, als dies mit der ursprünglichen formalen Notation der Eigenschaften eines Objekts/Klasse möglich ist (vgl. Def.: 2.12 auf Seite 52), wurde im Kapitel *„Erweiterung der Eigenschaftsdeklaration von Objekten/Klassen für die Modellierung von Kompatibilität“* (Definition 2.19 bzw. Listing 2.3 auf Seite 68) die Erweiterung der allgemeinen Notation der Eigenschaften eines Objekts/Klasse vorgestellt. Nachdem nun sowohl die SI-, als auch die Softwareeinheiten eingeführt wurden, folgt nun die exakte Formulierung der Einheiten in der erweiterten formalen Notation für die Modellierung und Bestimmung von Kompatibilität.

Nach der erweiterten Eigenschaftsdefinition kann jede Eigenschaft eines Objekts/Klasse um exakt eine Einheit erweitert werden. Das folgende Listing zeigt noch einmal eine Auswahl an möglichen Einheiten, die in der erweiterten formalen Notation verwendet werden können.

```
<Einheit>                 ::= <SIEinheiten> | <MechanikEinheiten> | <ElektrotechnikEinheiten> | <SoftwareEinheiten>.
<SIEinheiten>             ::= "m" | "kg" | "s" | "A" | "K" | "mol" | "cd".
<MechanikEinheiten>       ::= "m^2" | "m^3" | "Kg" | "°C" | "N" | "p".
<ElektrotechnikEinheiten> ::= "V" | "A" | "Ohm" | "C" | "L" | "Hz" | "W".
<SoftwareEinheiten>       ::= "bit" | "b" | "Byte".
```

Aus der Definition 2.19 auf Seite 68 bzw. dem dazugehörigen Listing folgt unmittelbar, dass jede Eigenschaft eines Objekts/Klasse mit genau einer Einheit versehen werden kann. Des Weiteren ist auch zukünftig möglich, Eigenschaften ohne eine explizit modellierte Einheit zu beschreiben, jedoch wird dies für die Modellierung von kompatibilitätsrelevanten Eigenschaften nicht empfohlen, weil die so formulierte Eigenschaft nicht für die Kompatibilitätsbestimmung verwendet werden kann. Aus Kompatibilitätsgründen zur ursprünglichen formalen Definition (vgl. Definition 2.1 auf Seite 52) der Eigenschaften sind jedoch beide Schreibweisen erlaubt. Dabei gilt insbesondere: bei Zuweisungen müssen die verwendeten Einheiten exakt übereinstimmen. Eine implizite Konvertierung zwischen Einheiten findet nicht statt.

Das folgende Beispiel zeigt sowohl die Verwendung der SI konformen Einheiten, als auch die Benutzung der Softwareeinheiten in der im Kapitel *„Erweiterung der Eigenschaftsdeklaration von Objekten/Klassen für die Modellierung von Kompatibilität“* eingeführten formalen Notation für Einheiten eines Objekts/Klasse.

Beispiel 22: Kompatibilitätskonforme Modellierung der Eigenschaften eines Objekts/Klasse
In diesem Beispiel werden einige der in diesem Kapitel eingeführten Einheiten anhand von Beispielen in der erweiterten formalen Notation für die kompatibilitätsrelevanten Eigenschaften eines Objekts/Klasse modelliert.

- **Modellierung von SI-Einheiten:**
  Die Eigenschaftsdeklaration

  ```
  + const maxTemperatur : float [in K, [372.15 ... 374.15]] = 373 [in K]
  ```

  legt fest, dass die Eigenschaft *maxTemperatur* eine öffentliche (**+**), konstante (**const**) Eigenschaft von Datentyp **float** ist, deren Gültigkeitsintervall von 372.15 bis einschließlich 374.15 reicht. Außerdem muss die zugewiesene Temperatur stets in Kelvin angegeben werden. Im obigen Beispiel wird der Eigenschaft *maxTemperatur* der Wert 373 zugewiesen. Der Wert 373 ist ebenfalls in Kelvin angegeben. Somit ist die obige Zuweisung korrekt.

  Im Gegensatz zur obigen Zuweisung ist die folgende unzulässig:

  ```
  + const maxTemperatur : float [in K, [372.15 ... 374.15]] = 373 [in °C]
  ```

  In diesem Beispiel ist zwar der zuzuweisende Wert von 373 innerhalb des zulässigen Gültigkeitsbereichs, jedoch ist die Einheit falsch. Die Zuweisung wird nicht ausgeführt, aufgrund der Tatsache, dass die beiden Einheiten nicht übereinstimmen und eine implizite Konvertierung von einer Basiseinheit in eine andere nicht vorgenommen wird.

- **Modellierung von Softwareeinheiten:**
  Die folgende Zuweisung soll ebenfalls auf Kompatibilität überprüft werden:

  ```
  + var Speichergroesse : long [in b, [0 ... 8192]] = 8192 [in bit]
  ```

Bei dieser Zuweisung tritt ein Fehler auf, obwohl laut Definition die beiden Softwareeinheiten *b* und *bit* eigentlich identisch sind. Trotzdem wird die Zuweisung nicht ausgeführt, weil nicht festgelegt wurde, dass die beiden Einheiten identisch, also kompatibel zueinander sind.

Wie die beiden Teilbeispiele gezeigt haben, können mit Hilfe der oben eingeführten formalen Notation für die Modellierung und Beschreibung von (kompatibilitätsrelevanten) Eigenschaften sowohl elektrotechnische-, mechanische- als auch Softwareeigenschaften modelliert und beschrieben werden. Außerdem wurde vor allem im Softwarebeispiel verdeutlicht, dass ein Regelwerk[95], in dem die Kompatibilitätsregeln hinterlegt sind, zwingend für die Kompatibilitätsbestimmung notwendig ist. □

2.3.2.2.1.2. Modellierung von dekadischen und nicht dekadischen Präfixen mit Hilfe der erweiterten formalen Eigenschaftsdefinition

Nachdem im vorangegangen Abschnitt sowohl die SI-, als auch die Softwaretechnikeinheiten eingeführt wurden, folgt nun die Definition der Präfixe. Im Allgemeinen werden Präfixe verwendet um sehr große Zahlenwerte verkürzt, ohne viele „Nullen“ schreiben zu müssen. Die SI definiert zu diesem Zweck so genannte *dekadische Präfixe*, die vor allem in der Elektrotechnik und der Mechanik verwendet werden. Zur verkürzten Schreibweise von Softwaregrößen gibt es so genannte *binäre Präfixe*, da Softwaregrößen nicht auf dem dekadischen Zahlensystem beruhen.

Elektrotechnik- und Mechanikpräfixe

In der Tabelle 2.4 sind sämtliche vom SI-Konsortium definierten *dekadischen Präfixe* (laut [Bun06a] auch SI-Vorsätze genannt) aufgelistet, die für die Beschreibung von elektrischen bzw. mechanischen Eigenschaften verwendet werden können.

Vorsatzzeichen	Name	Potenz	Vorsatzzeichen	Name	Potenz
d	Dezi	10^{-1}	da	Deka	10^{1}
c	Zenti	10^{-2}	h	Hekto	10^{2}
m	Milli	10^{-3}	K	Kilo	10^{3}
μ	Mikro	10^{-6}	M	Mega	10^{6}
n	Nano	10^{-9}	G	Giga	10^{9}
p	Piko	10^{-12}	T	Tera	10^{12}
f	Femto	10^{-15}	P	Peta	10^{15}
a	Atto	10^{-18}	E	Exa	10^{18}
z	Zepto	10^{-21}	Z	Zetta	10^{21}
y	Yokto	10^{-24}	Y	Yotta	10^{24}

Tabelle 2.4.: Dekadische Präfixe nach IEEE- bzw. SI-Standard ([Bec92] und [Bun06b]).

Softwaretechnikteiler (Binärpräfixe)

Im Gegensatz zu den dekadischen Präfixen, die stets auf der Basis 10 beruhen, werden in der Softwaretechnik stets Präfixe mit der Basis 2 verwendet. Die Tabelle 2.5 zeigt die in der Softwaretechnik gebräuchlichsten Präfixe.

Grundeinheit	Potenzschreibweise 2^x	Name
1 Bit 4 Bit = 1 Nibble 1 Byte = 8Bit	$2^0 = 1$	
1 KB = 1024 Bytes	2^{10} Bytes	*Kilo*
1 MB = 1024 KB	2^{20} Bytes	*Mega*
1 GB = 1024 MB	2^{30} Bytes	*Giga*
1 TB = 1024 GB	2^{40} Bytes	*Tera*
1 PB = 1024 TB	2^{50} Bytes	*Peta*
1 EB = 1024 PB	2^{60} Bytes	*Exa*

Tabelle 2.5.: In der Softwaretechnik übliche Präfixe (auch Binärpräfixe genannt).

Um Verwechslungen zwischen den dekadischen Präfixen und den Binärpräfixen zu vermeiden, wurde von der IEC (International Electrotechnical Commission) folgende Abänderung der in der Tabelle 2.5 definierten Binärpräfixe vorgeschlagen ([Zim06],[ita06] und [com06]):

- **Kibi [Ki]** ist laut der IEC-Norm 60027-2 das Präfix für 1.024 (2 hoch 10) im Binärsystem. Nach wie vor sind Kilo-Angaben auch noch mit binärem Bezugssystem verbreitet, obwohl dies zu Verwirrung führt. Korrekt wäre zum Beispiel 1 KibiByte = 1 KiByte = 1.024 Byte = 1.024 KiloByte = 1.024 kByte.
- **Mebi [Mi]** ist laut der IEC-Norm 60027-2 das Präfix für 1.048.576 (2 hoch 20) im Binärsystem. Dies hat sich aber noch nicht durchgesetzt. Nach wie vor sind Mega-Angaben auch noch mit binärem Bezugssystem verbreitet. Korrekt wäre zum Beispiel: 1 MebiByte = 1 MiByte = 1.048.576 Byte = ca. 1,05 MegaByte = ca. 1,05 MByte.

[95] Siehe hierzu insbesondere die beiden Kapitel „1.4.3 *Kompatibilitätsregelwerk*“ ab Seite 28 und „2.4 *Kompatibilitätsregelwerk für die Modellierung und Bewertung von Kompatibilität*“ ab Seite 82.

- **Gibi [Gi]** ist laut der IEC-Norm 60027-2 das Präfix für 1.073.741.824 (2 hoch 30) im Binärsystem. Nach wie vor sind Giga-Angaben auch noch mit binärem Bezugssystem verbreitet. Korrekt wäre zum Beispiel: 1 GibiByte = 1 GiByte = 1.073.741.824 Byte = ca. 1,07 GigaByte = ca. 1,07 GByte.
- **Tebi [Ti]** ist laut der IEC-Norm 60027-2 das Präfix für 1.099.511.627.776 (2 hoch 40) im Binärsystem. Dies hat sich aber noch nicht durchgesetzt. Nach wie vor sind Tera-Angaben auch noch mit binärem Bezugssystem verbreitet. Korrekt wäre zum Beispiel: 1 TebiByte = 1 TiByte = 1.099.511.627.776 Byte = ca. 1,1 TeraByte = ca. 1,1 TByte.

Für die Modellierung und Bestimmung der Kompatibilität zwischen zwei Eigenschaften eines Systems wird ebenfalls auf die Präfixschreibweise zurückgegriffen. Dabei wird strikt zwischen der von der SI definierten dekadischen Schreibweise und der binärschreibweise der Präfixe unterschieden. Aus diesem Grund ist es beispielsweise in der objektbasierten Kompatibilitätsmodellierungssprache (U)CML[96] vorgeschrieben, Softwareeinheiten nur mit Hilfe der von der IEC definierten Präfixe zu benennen, da es ansonsten zu schwerwiegenden versteckten Inkompatibilitäten im Modell kommen kann. Ist zum Beispiel die Eigenschaft `+ var Speichergroesse : long [in Kb, [0 ... 8]]` deklariert und wird ihr später der Wert `8 [in KibiByte]` zugewiesen, so kommt es zu einem Kompatibilitätsfehler, aufgrund der Tatsache, dass die beiden Präfixe nicht kompatibel zueinander sind.

Beispiel 23: Anwendung der Präfixschreibweise für sehr große/kleine Zahlen
Soll beispielsweise der elektrische Widerstand eines Isolators angegeben werden, so kann dies entweder ohne Präfixschreibweise als *10.000.000* Ω erfolgen, oder mit Hilfe der Präfixschreibweise als *10* $M\Omega$. Es lassen sich jedoch nicht nur sehr große Zahlen mit Hilfe der Präfixschreibweise schreiben, sondern auch sehr kleine, wie z.B. $0,000.01\Omega = 10\mu\Omega$

Die Präfixschreibweise kann auch in der Softwaretechnik angewendet werden. So kann z.B. der Speicherverbrauch eines Systems von *1.073.741.824 Byte* mit Hilfe der Präfixschreibweise als *1 GibiByte* oder ca. *1,07 GByte* geschrieben werden. Im Falle von Softwaretechnikgrößen ist unbedingt auf die korrekte Präfixnotation zu achten, da *1 GibiByte* ungleich *1 GByte* ist. □

Umrechnung zwischen unterschiedlichen Präfixen
In der praktischen Anwendung wird meistens stillschweigend zwischen den Präfixen von unterschiedlichen Eigenschaften hin- und hergewandelt. So wird beispielsweise der Ohmsche Widerstand $R_1 = 1000\Omega$ gleich dem Ohmschen Widerstand $R_2 = 1K\Omega$ angesehen, obwohl die angegebene Stellengenauigkeit nicht übereinstimmt und im strikten Sinne die beiden Widerstände R_1 und R_2 nicht gleich sind. Um das Problem der Stellengenauigkeit zu vermeiden, findet bei der Kompatibilitätsmodellierung und -bewertung keine implizite Konvertierung zwischen unterschiedlichen Präfixen statt. Demnach sind die beiden Widerstände R_1 und R_2 nur dann gleich groß, wenn sowohl ihre Präfixe, als auch die angegebenen gültigen Stellen gleich sind.

Formale Notation der Präfixe für die Kompatibilitätsmodellierung und -bestimmung
Für die Modellierung und Beschreibung der Präfixe stehen unter anderem die im folgenden Listing aufgeführten Präfixe (Teiler) zur Verfügung. Dabei wird wieder strikt zwischen den SI- und Softwarepräfixen unterschieden, um Kompatibilitätsprobleme zu vermeiden. Aus diesem Grund wird die Modellierung der Softwarepräfixe nur nach dem IEC Standard erlaubt.

```
<Teiler>          ::= <SITeiler> | <SoftwareTeiler>.
<SITeiler>        ::= "p" | "n" | "m" | "m" | "K" | "M" | "G" | "T".
<SoftwareTeiler>  ::= "Ki" | "Kibi" | "Mi" | "Mebi" | "Gi" | "Gibi" | "Ti" | "Tibi".
```

Dabei gilt insbesondere, wie oben gezeigt, dass eine implizite Umrechnung zwischen unterschiedlichen Präfixen nicht stattfindet.

Das folgende Beispiel illustriert sowohl die Verwendung der SI konformen Präfixe, als auch der Softwarepräfixe in der im Kapitel *„Erweiterung der Eigenschaftsdeklaration von Objekten/Klassen für die Modellierung von Kompatibilität“* eingeführten formalen Notation für Einheiten eines Objekts/Klasse.

Beispiel 24: Kompatibilitätskonforme Modellierung der Eigenschaften eines Objekts/Klasse II
In diesem Beispiel wird die Verwendung sowohl der SI konformen Präfixe, als auch die Anwendung der Softwarepräfixe demonstriert.

- **Modellierung von SI-Präfixen:**
  Die Eigenschaftsdeklaration

  `+ var Stromverbrauch : float [in A, [0 ... 10]] = 800 [in mA]`

  legt fest, dass die öffentliche, variabel (**var**) deklarierte Eigenschaft *Stromverbrauch* Stromwerte vom Datentyp **float** von 0 bis einschließlich 10A (Ampere) akzeptiert. Der Eigenschaft *Stromverbrauch* wird auf der rechten Seite der Wert 800mA zugewiesen. Diese Zuweisung führt unmittelbar zu einem Kompatibilitätsfehler, weil weder die gültige Stellenzahl noch die Präfixe übereinstimmen. Soll die obige Zuweisung ohne Kompatibilitätsfehler erfolgen so kann entweder die Deklaration der Eigenschaft *Stromverbrauch* oder die Wertezuweisung angepasst werden.
- **Modellierung von Softwarepräfixen:**
  In der Softwaremodellierung werden ebenfalls Präfixe verwendet. Diese stammen jedoch nicht von der SI, sondern wurden von der IEC genormt. Im folgenden Beispiel wird die Eigenschaft *Speichergroesse* als öffentliche, veränderbare Eigenschaft vom Datentyp **long**, mit einem Gültigkeitsintervall von 0 bis einschließlich 8KiBiByte deklariert. Dieser Deklaration wird auf der rechten Seite der Wert 8 mit dem Präfix KiBi zugewiesen.

  `+ var Speichergroesse : long [in KiBiByte, [0 ... 8]] = 8 [in KiBiByte]`

  Da sowohl die Stellengenauigkeit und das verwendete Präfix stimmen, als auch der übergebene Wert innerhalb des Gültigkeitsintervalls liegt, ist diese Zuweisung kompatibel.

□

[96] Siehe hierzu: Kapitel *„3.6 Einführung in die Kompatibilitätsmodellierungssprache (U)CML“* ab Seite 153.

2.3.2.2.1.3. Modellierung von Gültigkeitsintervallen in der erweiterten formalen Eigenschaftsdefinition

Zusätzlich zur Deklaration von Einheiten und Präfixen ist es für die Kompatibilitätsmodellierung und -bestimmung eines Objekts/Klasse von entscheidender Bedeutung, zusätzlich zum exakten Wert einer Eigenschaft einen Gültigkeitsbereich (Gültigkeitsintervall) anzugeben, in dem ein bestimmter Wert liegen muss um als kompatibel angesehen zu werden. Dieses Vorgehen hat sich in der Elektrotechnik bzw. der Mechanik schon seit sehr langer Zeit etabliert, da Werte wie zum Beispiel die Länge eines Gegenstands niemals exakt gefertigt oder gemessen werden können. Aus diesem Grund werden Längen stets mit einer Toleranz bzw. einer tolerierbaren Abweichung angegeben. In der Informatik hingegen hat sich das Konzept der Gültigkeitsbereiche noch nicht durchgesetzt. Hier werden sämtliche Größenangaben stets exakt angegeben ohne eine bestimmte Toleranz.

Das Konzept der Gültigkeitsintervalle ist vor allem für die Modellierung und die Bestimmung von Kompatibilität von entscheidender Bedeutung da technische physikalische Größen nicht exakt bestimmt bzw. gefertigt werden können und es stets zu Abweichungen kommen kann. Abweichungen sind nicht die Ausnahme sondern die Regel. Für die Kompatibilitätsmodellierung und -bestimmung ist es daher entscheidend, diese Abweichungen vom Standardmaß so genau wir möglich zu quantifizieren. Um die Abweichung zu modellieren, bietet sich die aus der Mathematik stammende Intervallschreibweise an. Die folgende Abbildung 2.44 (links, „Teil A“) zeigt eine vereinfachte CAD Zeichnung eines Rechtecks mit einer Kantenlänge von $l = 30\text{mm}$ und einer Breite von $b = 10\text{mm}$. Sowohl für die Länge, als auch die Breite sind jeweils Gültigkeitsbereiche angegeben.

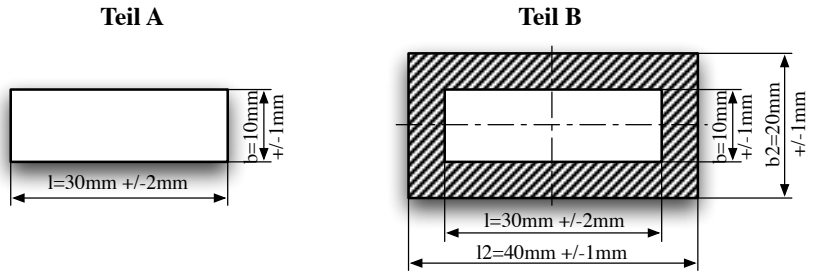


Abbildung 2.44.: CAD Zeichnung mit Bemaßung.

Demnach darf die tatsächliche Länge (l) des Rechtecks innerhalb von minimal 28mm bis maximal 32mm liegen. Diese Angabe kann, wie oben erwähnt, mittels eines aus der Mathematik stammenden Intervalls beschrieben werden, also $l = [28 \dots 32]\text{mm}$. Ebenso kann die Breite des Rechtecks $b = [9 \dots 11]\text{mm}$ in der Intervallschreibweise angegeben werden. Soll das Bauteil *Teil A* in die Aussparung des *Teils B* passen, so müssen die Gültigkeitsintervalle entweder deckungsgleich oder das *Teil A* muss sowohl in der Länge als auch in der Breite kleiner sein als die Aussparung des *Teils B*.

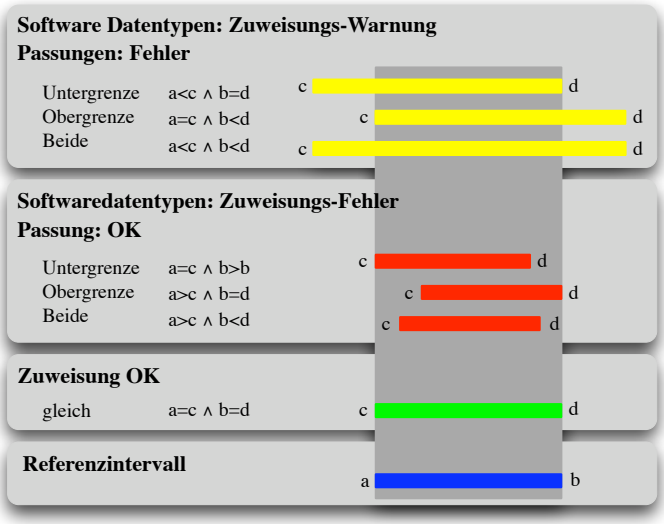


Abbildung 2.45.: Intervallschachtelung

Auch für die Modellierung von Softwaregrößen kann die Intervallschreibweise eingesetzt werden. Beispielsweise kann der aktuelle Speicherverbrauch eines Objekts mit Hilfe eines Gültigkeitsintervalls angegeben werden (`+ var AktuellerSpeicherverbrauch : long [in KiBiByte, [0 ... 8]]`). In diesem Beispiel kann die Eigenschaft *AktuellerSpeicherverbrauch* beispielsweise Werte von 0 bis maximal 8KiBiByte annehmen. Eine Zuweisung von größeren oder kleineren Werte wird abgelehnt. Durch die Einführung von Gültigkeitsintervallen in der Programmierung kann beispielsweise verhindert werden, dass es zu einem so genannten „buffer underrun“ oder „stack overflow“ kommt, der in den meisten Fällen zu einem unkontrollierten Systemverhalten oder zum Absturz der Software führt.

Die Abbildung 2.45 zeigt graphisch, wann zwei Intervalle ineinander passen und wann nicht. Dabei wird stets zwischen zwei unterschiedlichen Interpretationen der Intervallschachtelung unterschieden.

- **Szenario 1: Modellierung von Softwaredatentypen**
  Wird bei der Modellierung von Softwaredatentypen das Referenzintervall mit den beiden Grenzen a und b dem Zielintervall c, d zugewiesen, so kommt es im dritten Abschnitt weder zu einem Informationsverlust, noch zu einer Warnung, aufgrund der Tatsache, dass beide Grenzen der Intervalle exakt gleich sind. Im zweiten Abschnitt hingegen ist stets eine oder beide Grenzen des Zielintervalls kleiner als das Referenzintervall, so dass hier ein Informationsverlust bei der Zuweisung des Referenzintervalls auf das Zielintervall droht. Wird beispielsweise der Eigenschaft *A* vom Datentyp **`int`** die Eigenschaft *B* vom Datentyp **`long`** zugewiesen, so kommt es im Allgemeinen zu einem Datenverlust, weil der Datentyp **`int`** häufig nur Werte von 0 bis 2^{16} speichern kann. Der Datentyp **`long`** hingegen Werte von 0 bis 2^{32}. Im ersten Abschnitt der Abbildung ist mindestens eine Grenze des Zielintervalls größer als das Referenzintervall, so kann es zu keinem Informationsverlust kommen. Aus diesem Grund wird hier eine Warnung ausgegeben.
- **Szenario 2: Modellierung von Passungen**
  Soll mit Hilfe der Intervallschreibweise entschieden werden, ob zwei Intervalle gleich sind, so muss das Zielintervall stets innerhalb des Referenzintervalls enthalten sein. Ansonsten passen die beiden Intervalle nicht ineinander. Im ersten Abschnitt der Abbildung 2.45 ist stets eine Grenze größer als die entsprechenden Grenzen des Referenzintervalls. Daraus folgt unmittelbar, dass das Zielintervall – und somit die Passung – nicht kompatibel sind. Im zweiten Abschnitt der Abbildung hingegen ist mindestens eine Grenze kleiner als die entsprechenden Grenzen des Ausgangsintervalls. In diesem Fall passt das Zielintervall vollständig in das Referenzintervall.

Formale Notation der Gültigkeitsintervalle für die Kompatibilitätsmodellierung und -bestimmung
In der im Kapitel „*Erweiterung der Eigenschaftsdeklaration von Objekten/Klassen für die Modellierung von Kompatibilität*“ eingeführten erweiterten formalen Notation für die Modellierung von (kompatibilitätsrelevanten) Eigenschaften eines Objekts/Klasse (vgl. Definition 2.19 bzw. Listing 2.3) wurde bereits die Intervallschreibweise zur detaillierten Beschreibung der Gültigkeitsintervalle eingeführt. Das folgende Listing zeigt noch einmal den entsprechenden Deklarationsteil.

```
<Intervall>     ::= "[" | "]" <Untergrenze> "..." <Obergrenze> "[" | "]".
<Obergrenze>    ::= ("0.")? <Ziffer> (<Ziffer>)*.
<Untergrenze>   ::= ("0.")? <Ziffer> (<Ziffer>)*.
```

Demnach hat jedes Intervall eine eineindeutige Unter- sowie eine eineindeutige Obergrenze. Zwischen der Unter- und der Obergrenze sind drei Punkte eingetragen. Sowohl die Unter- als auch die Obergrenze wird mittels einer eckigen Klammer symbolisiert. Dabei zeigt die Öffnungsrichtung der Klammer an, ob die Grenze zum Intervall gehört oder nicht. Beispielsweise gehört die Obergrenze 10 dazu, während die Untergrenze 3 nicht zum Intervall $]3 \dots 10]$ gehört. Außerdem muss stets gelten, dass die Obergrenze eines Intervalls größer oder gleich der Untergrenze ist, also $Untergrenze <= Obergrenze$. Sind die beiden Grenzen identisch ($Untergrenze = Obergrenze$) so kann dem Intervall nur exakt ein Wert zugewiesen werden.

Beispiel 25: Kompatibilitätskonforme Modellierung der Eigenschaften eines Objekts/Klasse III
Mit Hilfe der Intervallschreibweise kann einer Eigenschaft eines Objekts/Klasse ein bestimmter Gültigkeitsbereich zugewiesen werden, innerhalb dessen sämtliche akzeptierten Werte liegen müssen. Beispielsweise kann die Eigenschaft *Stromverbrauch* aus dem obigen Beispiel wie folgt deklariert werden:

```
+ var Stromverbrauch : float [in A, [0 ... 10]]
```

Dann können der Eigenschaft *Stromverbrauch* Werte zwischen 0 und maximal 10A zugewiesen werden. Außerdem müssen die Werte vom Datentyp **`float`** sein, um akzeptiert zu werden. Wird das Gültigkeitsintervall so abgeändert, dass lediglich Werte größer 3 bis maximal 7A erlaubt sind, so muss das obige Intervall genauso abgeändert werden `... [in A, ]3 ... 7]]`. Soll lediglich ein bestimmter Wert, beispielsweise 5A akzeptiert werden, so lautet das entsprechende Intervall `... [in A, [5 ... 5]]`. □

Nachdem nun sämtliche Erweiterungen gegenüber der ursprünglichen formalen Notation der Eigenschaften eines Objekts/Klasse eingeführt und beschrieben wurden, folgt im nächsten Kapitel die Verknüpfung der Präfixe und Einheiten zu einem zusammengesetzten neuen Datentyp, der speziell für die Kompatibilitätsmodellierung verwendet werden sollte.

2.3.2.2.2. Verknüpfung von Datentyp, Präfix, Einheit und Gültigkeitsintervall

Nachdem im letzten Kapitel „*Modellierung von Einheiten, dekadischen und nichtdekadischen Präfixen sowie von Gültigkeitsintervallen in der erweiterten formalen Eigenschaftsnotation für die Modellierung und Bestimmung von Kompatibilität*“, sowie den dazugehörigen drei Unterabschnitten, die Notwendigkeit einer eineindeutigen Spezifikation der Einheiten, Präfixen und Intervalle für die formale Beschreibung der Eigenschaften eines Objekts/Klasse gezeigt wurde, folgt in diesem Kapitel die Verknüpfung von Datentypen, Präfixen, Einheiten und Gültigkeitsintervallen zu einem *zusammengesetzten Datentyp*. Dieser Schritt ist für eine erfolgreiche Kompatibilitätsmodellierung zwingend notwendig, um Fehler wie z.B. die Verwendung von unterschiedlichen Datentypen, Präfixen, Einheiten oder Intervallen zu vermeiden (vgl. Beispiel auf Seite 58: Absturz der NASA Sonden „Mars Climate Orbiter“). Zwar lässt sich die Modellqualität für die Kompatibilitätsmodellierung und -bestimmung durch jede der vorher vorgestellten Einzelmaßnahmen entscheidend verbessern, jedoch erst durch die Verknüpfung wird das volle Potential ausgeschöpft. In der nachfolgenden Definition 2.20 ist die Verknüpfung von Datentyp, Präfix, Einheit und Intervall zu einem zusammengesetzten Datentyp beschrieben.

Definition 2.20 ***Verknüpfung von Datentyp, Einheit, Präfix und Gültigkeitsintervall zu einer zusammengesetzten Datentyp***
*Ein **zusammengesetzter Datentyp** besteht stets aus genau einem **Datentyp**, einem optionalen **Präfix**, einer **Einheit** sowie einem **Gültigkeitsintervall**. Beschrieben wird der zusammengesetzte Datentyp mittels der in der Definition 2.19 eingeführten formalen Notation.*

Anmerkung zur Definition 2.20:
Die obige Definition definiert lediglich die Verknüpfung von genau einem Datentyp, Präfix mit einer Einheit und einem Gültigkeitsintervall. Ob diese Verknüpfung „sinnvoll“ ist oder nicht wird hier nicht festgelegt. Zum Beispiel ist der zusammengefasste Datentyp `... float [in KiBiByte V, [0 ... 5]]` nicht sinnvoll, da im Allgemeinen das Präfix *KiBi* nur für Software und nicht für elektrische/elektronische Einheiten verwendet wird. Damit nur noch „sinnvolle“ zusammengesetzte Datentypen modelliert werden können, ist zusätzlich ein Regelwerk notwendig, in dem hinterlegt ist, welche zusammengesetzten Datentypen gültig sind[97].

Das folgende Beispiel illustriert die Verwendung der in diesem Kapitel neu eingeführten zusammengesetzten Datentypen anhand zweier Beispielsysteme.

Beispiel 26: Verknüpfung von Datentyp, Einheit und Teiler zu einem zusammengesetzten Datentyp
Um die Inkompatibilität zwischen zwei miteinander verbundenen Eigenschaften eines Systems zu vermeiden, können zusammengesetzte Datentypen in einem Modell verwendet werden. Die Zusammenfassung von Datentyp, Präfix, Einheit und Gültigkeitsintervall ist dabei nicht nur auf die Softwaretechnik beschränkt, sondern kann in allen Bereichen gleichermaßen angewendet werden.

- *Szenario 1:* Datenaustausch zwischen dem *Sender A* und dem *Empfänger B*.
  Laut Dokumentation schickt die Baugruppe *Sender A* das Datum 1KV, dessen Wert in den Datentyp **`int`** verpackt ist (`int sp = 1`) an die Baugruppe *Empfänger B*. Dabei wird weder der Datentyp, das Präfix, die Einheit noch das Gültigkeitsintervall vom Sender an den Empfänger übermittelt. Der Empfänger erwartet jedoch laut Datenblatt, dass das Datum das Präfix m und die Einheit V hat. Ohne die Verknüpfung von Datentyp, Präfix, Einheit und Gültigkeitsintervall zu einem zusammengesetzten Datentyp kann auf Modellebene nicht festgestellt werden, dass eine Inkompatibilität zwischen dem Sender und dem Empfänger vorliegt (vgl. Abb. 2.46 oben), da nicht alle dafür notwendigen Informationen im Modell des Systems enthalten sind. Der Sender sendet den „nackten“ Wert 1 und der Empfänger empfängt den Wert 1 und interpretiert ihn ohne über die unterschiedlichen Präfixe und Einheiten Bescheid zu wissen. Es kommt zu einer Inkompatibilität die im Allgemeinen nur sehr schwer in einem komplexen System entdeckt werden kann.

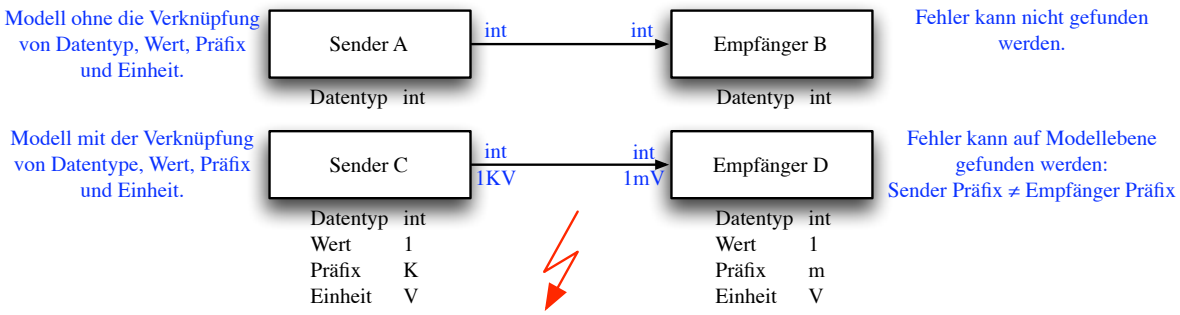


Abbildung 2.46.: Verknüpfung von Datentyp, Präfix, Einheit und Gültigkeitsintervall zu einer zusammengesetzten Einheit.

Im unteren Teil der Abbildung 2.46 ist das selbe Modell wie oben dargestellt, mit dem Unterschied, dass hier die Verbindung von Datentyp, Wert, Präfix und Einheit explizit im Modell mittels der neu eingeführten zusammengesetzten Datentypen hinterlegt wurde. Durch die Kombination ist es möglich, bereits auf Modellebene die Inkompatibilität der beiden Präfixe im Modell festzustellen. Der Sender C sendet die Eigenschaft `+ var Spannung : float [in KV, [0 ... 10]] = 1 [in KV, [0 ... 10]]` an den Empfänger der Nachricht (Empfänger D: `+ var Spannung : float [in mV, [0 ... 10]] = 1 [in mV, [0 ... 10]]`). Dabei wird festgestellt, dass die Zuweisung nicht ausgeführt werden kann, da eine „Präfixinkompatibilität“ vorliegt.

- *Szenario 2:* Elektrische Verbindung zwischen zwei Baugruppen.
  Zwei Baugruppen (Stromquelle und Glühbirne) eines Systems sind mittels einer Leitung miteinander verbunden. Der Sender (hier die Stromquelle) liefert laut Datenblatt 1.5V und 50mA. Im Modell des Systems sind sowohl beide Werte als auch die dazugehörigen Präfixe und Einheiten vollständig modelliert (vgl. Abb.: 2.47).

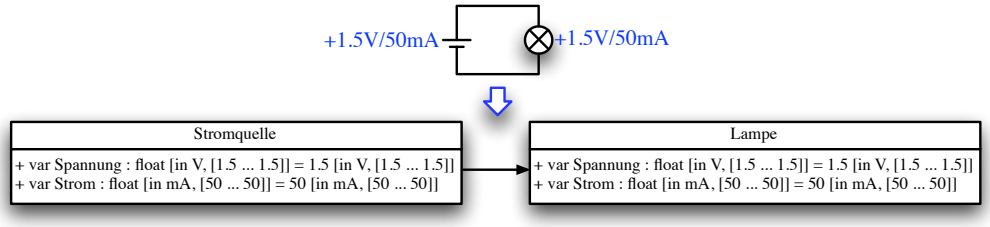


Abbildung 2.47.: Verknüpfung von Datentyp, Präfix, Einheit und Gültigkeitsintervall zu einem zusammengesetzten Datentyp.

[97] Nähere Informationen zum Kompatibilitätsregelwerk finden Sie im Grundlagenkapitel „1.4.3 *Kompatibilitätsregelwerk*“ ab Seite 28, sowie in den beiden Kapiteln „2.4 *Kompatibilitätsregelwerk für die Modellierung und Bewertung von Kompatibilität*“ ab Seite 82 und „3.6.5.1 *(U)CML-Sprachregelwerk*“ ab Seite 236.

Aus der Abbildung 2.47 ist zu entnehmen, dass das Objekt/Klasse *Stromquelle* zum Objekt/Klasse *Lampe* kompatibel ist, aufgrund der Tatsache, dass sowohl der Wert, der Datentyp, das Präfix, die Einheit und das Gültigkeitsintervall übereinstimmen.

Durch die Verknüpfung von Datentyp, Präfix, Einheit und einem Gültigkeitsintervall ist es möglich, sowohl mechanische, elektrische/elektronische als auch Softwareeigenschaften so zu beschreiben, dass sie auf Kompatibilität untersucht werden können. □

Nachdem nun die kompatibilitätskonforme formale Notation der Eigenschaften eines Objekts/Klasse eingeführt und anhand verschiedener Beispiele aufgezeigt worden ist, folgt im nächsten Kapitel die Erläuterung der kompatibilitätskonformen formalen Beschreibung der verhaltensbeschreibenden Methoden eines Objekts/Klasse.

2.3.2.3. Erweiterung der Methodendeklaration eines Objekts/Klasse für die Modellierung von Kompatibilität

Ebenso wie die allgemeine formale Notation der Eigenschaften eines Objekts/Klasse an die besonderen Anforderungen der Kompatibilitätsmodellierung und -prüfung angepasst werden musste, müssen auch die verhaltensbestimmenden Methoden eines Objekts/Klasse für die Kompatibilitätsmodellierung und -bestimmung angepasst werden. Dabei wird die allgemeine Methodendeklaration (vgl. „Definition: 2.14“ auf Seite 54) um die im letzten Kapitel eingeführte erweiterte Eigenschaftsdefinition (vgl. „Definition: 2.19“ auf Seite 68) erweitert. Die folgende Definition beschreibt die Erweiterung der allgemeinen formalen Notation für die Modellierung der verhaltensbeschreibenden Methoden eines Objekts/Klasse.

Erweiterung der formalen Beschreibung der verhaltensbeschreibenden Methoden eines Objekts/Klasse für die Kompatibilitätsmodellierung und -bewertung
Die folgende BNF-artige formale Syntaxbeschreibung zeigt sämtliche notwendigen Erweiterungen gegenüber der in der Definition 2.14 eingeführten Standardnotation für die Modellierung der verhaltensbeschreibenden Methoden einer Klasse. Dabei wird für die Modellierung der Parameter der Methoden eines Objekts/Klasse auf die formale erweiterte Notation der Eigenschaften eines Objekts/Klasse zurückgegriffen.

```
<FunktionsDeklaration> ::=
    <Sichtbarkeit> (<Eigenschaftstyp>)? <Datentyp> ("[" "in" (<Teiler>)? <Einheit> ("," <Intervall>)? "]")? <Methodenname>
    "(" ("void" | (<Parameter>) ","? *) ")" <Eigenschaftstyp>;

<Parameter> ::=
    (<Eigenschaftstyp>)? <Datentyp> <Parametername>
    ("[" "in" (<Teiler>)? <Einheit> ("," <Intervall>)? "]")?.

<Sichtbarkeit>              ::= "+" | "#" | "-".
<Eigenschaftstyp>           ::= "const" | "var".
<Datentyp>                  ::= "void" | "bool" | "int" | "long" | "float" | "string"...
<Funktionsname>             ::= <Buchstabe> (<Buchstabe> | <Ziffer>)*.
<Parametername>             ::= <Buchstabe> (<Buchstabe> | <Ziffer>)*.
<Intervall>                 ::= "[" | "]" <Untergrenze> "..." <Obergrenze> "[" | "]".
<Obergrenze>                ::= ("0.")? <Ziffer> (<Ziffer>)*.
<Untergrenze>               ::= ("0.")? <Ziffer> (<Ziffer>)*.
<Teiler>                    ::= <SITeiler> | <SoftwareTeiler>.
<SITeiler>                  ::= "p" | "n" | "m" | "m" | "K" | "M" | "G" | "T".
<SoftwareTeiler>            ::= "Ki" | "Kibi" | "Mi" | "Mebi" | "Gi" | "Gibi" | "Ti" | "Tibi".
<Einheit>                   ::= <SIEinheiten> | <MechanikEinheiten> | <ElektrotechnikEinheiten> | <SoftwareEinheiten>.
<SIEinheiten>               ::= "m" | "kg" | "s" | "A" | "K" | "mol" | "cd".
<MechanikEinheiten>         ::= "m^2" | "m^3" | "Kg" | "°C" | "N" | "p".
<ElektrotechnikEinheiten>   ::= "V" | "A" | "Ohm" | "C" | "L" | "Hz" | "W".
<SoftwareEinheiten>         ::= "bit" | "b" | "Byte".
<Wert>                      ::= (<Buchstabe> | <Ziffer>)+
<Buchstabe>                 ::= "A" | "B" | "C" | "D" ... "Z" | "a" | "b" | "c" | "d" ... "z".
<Ziffer>                    ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "0".
```

Listing 2.4: Erweiterte formale Notation von kompatibilitätsrelevanten verhaltensbeschreibenden Methoden eines Objekts/Klasse

Definition 2.21 ***Modellierung des Verhaltens eines Objekts/Systems/Teilsystems***
Sämtliche ***kompatibilitätsrelevanten verhaltensbeschreibenden Methoden*** *eines* ***Objekts/Klasse*** *lassen sich mit Hilfe der, im Listing 2.4 beschriebenen, formalen BNF-artigen Syntax formal modellieren und beschreiben.*

Anmerkung zur Definition 2.21:
Das Verhalten eines Objekts kann beispielsweise auch mit Hilfe von Zustandsautomaten oder Message Sequence Charts (MSCs) beschrieben werden. In dieser Arbeit wird jedoch nur die Beschreibung mittels Methoden bzw. MSCs erläutert.

Mit Hilfe der in der Definition 2.21 bzw. dem dazugehörigen Listing eingeführten, BNF-artigen Erweiterung der ursprünglichen Methodendeklaration lassen sich sämtliche Methoden einer Klasse vollständig formal für die Kompatibilitätsmodellierung und -bewertung spezifizieren. Dabei wurde die ursprüngliche Methodendeklaration (vgl. „Definition: 2.14“ auf Seite 54) hauptsächlich um die formale Spezifikation der *Parameter* der Methode erweitert. Die Parameter der Methode werden nun ebenfalls um einen optionalen *Ergebnistyp*, sowie einen optionalen *Teiler*, eine *Einheit* und ein *Gültigkeitsintervall* erweitert. Zusätzlich kann jede Methode mit einem *Eigenschaftstyp* versehen werden. Dabei bedeutet der *Eigenschaftstyp* am Methodenanfang (**`+ var float [...] foo(...)`**), dass die Methode *foo* bei jedem Methodenaufruf ein bestimmtes Ergebnis vom Type **`float`** zurück liefert. Dieses Ergebnis kann sich bei jedem Aufruf der Methode ändern. Aus diesem Grund ist es als **`var`** spezifiziert. Am Ende einer Methodendeklaration wird ebenfalls ein Eigenschaftstyp festgelegt. Ist dieser Typ **`const`** bedeutet dies, dass die so spezifizierte Methode keine internen Eigenschaften der Klasse verändert – die Methode ist somit invariant gegenüber den internen Datenstrukturen (Eigenschaften) der Klasse.

Das folgende Beispiel illustriert die Anwendung der oben eingeführten erweiterten formalen Notation zur Beschreibung der verhaltensbeschreibenden Methoden eines Objekts/Klasse anhand des Beispielsystems Kaffeemaschine.

Beispiel 27: Modellierung von kompatibilitätsrelevanten verhaltensbeschreibenden Methoden des Beispielsystems Kaffeemaschine
Auf der Seite 54 (Abbildung 2.22) wurde bereits die verhaltensbeschreibende Methode **+ float** `CalcStromverbrauch()` der Klasse *Regelung* des Beispielsystems Kaffeemaschine vorgestellt. Diese Methode wird nun mit Hilfe der oben eingeführten formalen Notation so angepasst und erweitert, dass sie für die Kompatibilitätsbestimmung verwendet werden kann. Außerdem wird die Zugriffsmethode **+** `SetSpeichergroesse(`**long**`)` für die Kompatibilitätsbestimmung erweitert.
Die Abbildung 2.48 zeigt links einen Auszug aus der ursprünglichen allgemeinen Klassendeklaration der Klasse *Regelung*. Rechts ist die modifizierte, für die Kompatibilitätsbestimmung erweiterte Klassendeklaration abgebildet.

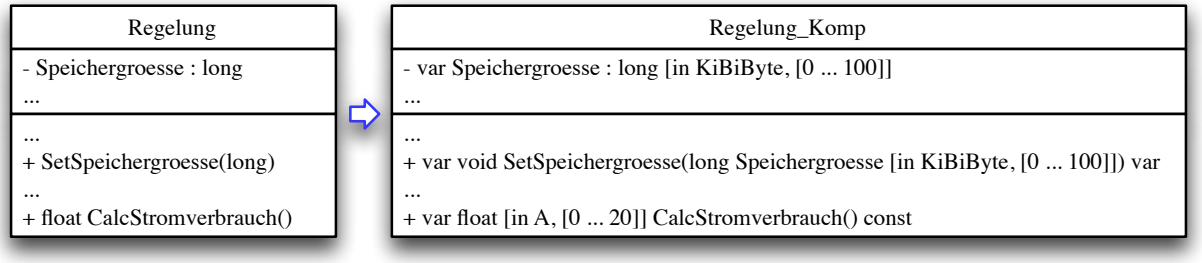


Abbildung 2.48.: Vergleich der ursprünglichen Klassendeklaration mit erweiterter Methodendeklaration für die Modellierung und Bewertung von Kompatibilität.

Die ursprüngliche verhaltensbeschreibende Methode **+ float** `CalcStromverbrauch()` der Klasse *Regelung* wird nun mit Hilfe der in der Definition 2.21 beschriebenen Erweiterungen für die Kompatibilitätsmodellierung und -bewertung angepasst. Zunächst wird der Ergebnisdatentyp **float** um eine explizit modellierte Einheit und einen Gültigkeitsbereich erweitert. Beide hierfür notwendigen Informationen stammen aus den entsprechenden Datenblättern der Baugruppe. Aufgrund der Tatsache, dass der Ergebniswert der Methode nicht konstant ist, sondern sich bei jedem Aufruf der Methode ändern kann, wird er als **var** deklariert. Das Schlüsselwort **const** am Ende der Methodendeklaration deutet an, dass die Methode `CalcStromverbrauch()` keine internen Strukturen der Klasse verändert. Die vollständige Methodendeklaration lautet demnach **+ var float** `[`**in** `A, [0 ... 20]] CalcStromverbrauch()` **const**.
Die ursprüngliche Zugriffsmethode **+** `SetSpeichergroesse(`**long**`)` für die interne private Eigenschaft *Speichergroesse* wird nun ebenfalls für die Kompatibilitätsmodellierung angepasst. Das Ergebnis der Anpassung ist die erweiterte Methode **+ var void** `SetSpeichergroesse(`**long** `Speichergroesse [`**in** `KiBiByte, [0 ... 100]])` **var**. Diese Methode hat genau einen Parameter, dessen Datentyp als **long** spezifiziert ist. Dieser Datentyp wird nun ebenfalls mit einer explizit modellierten Einheit und einem Gültigkeitsintervall versehen. Zusätzlich wird die Methode als **var** spezifiziert, da sie die interne Eigenschaft *Speichergroesse* verändert. □

Transformation der BNF-artigen formalen Notation der verhaltensbeschreibenden Methoden eines Objekts/Klasse in die Syntax der Programmiersprache C++
Um eine in der BNF-artigen erweiterten formalen Notation angegebene Methode einer Klasse in der objektorientierten Programmiersprache C++ programmieren zu können, muss die formale BNF-Spezifikation der Methode in eine C++ konforme Notation transferiert werden. Dies ist jedoch nicht ohne die Anwendung einiger „Tricks“ möglich, da keine Programmiersprache die in den beiden Definitionen 2.19 und 2.21 vorgestellten Erweiterungen für die Eigenschafts- bzw. Methodenmodellierung beherrscht. Die Abbildung 2.49 zeigt eine mögliche Art und Weise, wie eine für die Kompatibilitätsmodellierung und -bestimmung spezifizierte Methode dennoch nach C++ transferiert werden kann.

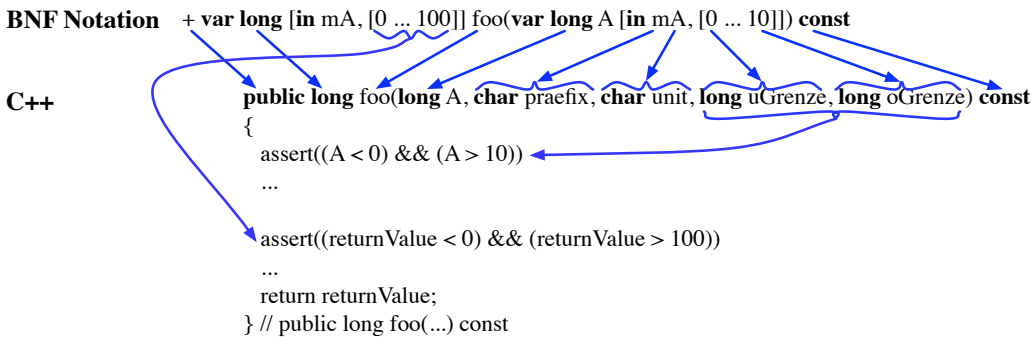


Abbildung 2.49.: Transformation einer in BNF notierten Methode eines Objekts/Klasse in C++.

Wie die Abbildung 2.49 zeigt, lassen sich nicht alle in der BNF spezifizierten Eigenschaften einer Methode einfach nach C++ transferieren. Dazu zählt insbesondere der Ergebnistyp. Hier kann zwar der Datentyp spezifiziert werden, jedoch weder das Präfix,

noch die Einheit oder das Gültigkeitsintervall lassen sich so transferieren. In C++ kann jedoch mit Hilfe von so genannten *templates* [Str90, 349ff] ein neuer Datentyp erzeugt werden, der für die Kompatibilitätsmodellierung geeignet ist. Die Einführung dieses erweiterten Datentyps würde jedoch den Rahmen dieser Arbeit sprengen.

Modellierung des Verhaltens eines Objekts in einem System mit Hilfe von Message Sequence Charts (MSCs)
Wie bereits im Kapitel „2.3.1.3.2 *Verhaltensbeschreibung eines Objekts mittels MSCs*“ ab Seite 55 gezeigt wurde, kann das Verhalten (genauer der Nachrichtenaustausch) zwischen den Objekten/Klassen eines Systems mit Hilfe der Message Sequence Charts modelliert und formal beschrieben werden. Auch für die Kompatibilitätsmodellierung und -bestimmung bieten sich die MSCs an. Um die MSCs jedoch für die Kompatibilitätsmodellierung und -bestimmung einsetzen zu können, müssen diese um einige grundlegende Konzepte der Kompatibilitätsmodellierung erweitert werden. Sämtliche notwendigen Erweiterungen und Ergänzungen gegenüber den allgemeinen MSCs sind in den beiden Diplomarbeiten von C. Eckl ([Eck07]) und J. Winkler [Win08] sowie der Dissertation von D. Koß [Koß09] beschrieben und werden hier nicht ausgeführt.

Das folgende Beispiel zeigt die Modellierung des Verhaltens mit Hilfe von MSCs anhand des Beispielsystems Kaffeemaschine.

Beispiel 28: Modellierung des Verhaltens des Beispielsystems Kaffeemaschine mit Hilfe von MSCs
Die Abbildung 2.50 zeigt einen Ausschnitt aus dem MSC des Beispielsystems Kaffeemaschine. Dabei wird die verhaltensbestimmende Methode `+ float [in A, [0 ... 20]] CalcStromverbrauch() const` des Objekts *Regelung* von einem nicht näher spezifizierten Objekt aufgerufen. Das Objekt *Regelung* genauer die Methode `... CalcStromverbrauch() ...` ruft ihrerseits zunächst die Methode `+ float [in A, [0 ... 10]] GetStromverbrauch() const` des Objekts *Heizplatte* auf und wartet, bis das Ergebnis zurückgegeben wird. Im Anschluss daran wird die Methode `+ float [in A, [0 ... 10]] GetStromverbrauch() const` des Objekts *RestlicheKomponenten* aufgerufen. Nachdem beide Aufrufe erfolgt sind, wird das Ergebnis an den Aufrufer des Objekts *Regelung* zurückgegeben.

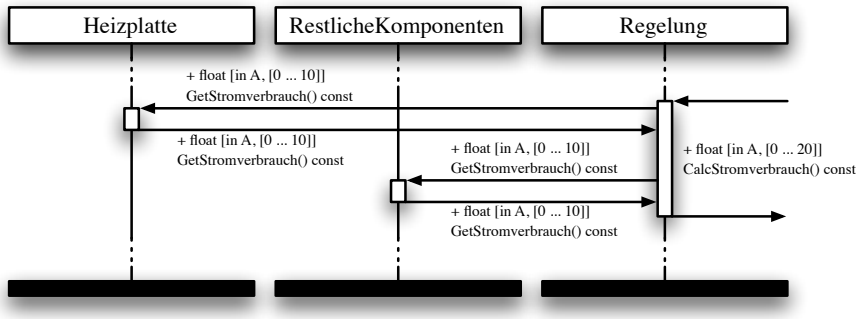


Abbildung 2.50.: Kompatibilitätsbestimmung mit Hilfe eines MSCs.

Ergebnis: Das MSC aus der Abbildung 2.50 enthält keinen Kompatibilitätsfehler. Alle Signale und sämtliche Signaturen der Methoden stimmen überein.

In der Abbildung 2.51 wurde das Objekt *Regelung* aus der obigen Abbildung 2.50 durch eine neuere Regelung *Regelung_neu* mit einem, gegenüber der ursprünglichen Regelung veränderten Verhalten eingesetzt. Die neue Regelung ruft zunächst die Methode `... CalcStromverbrauch() ...` des Objekts *RestlicheKomponenten* und im Anschluss daran die Methode `... CalcStromverbrauch() ...` des Objekts *Heizplatte* auf.

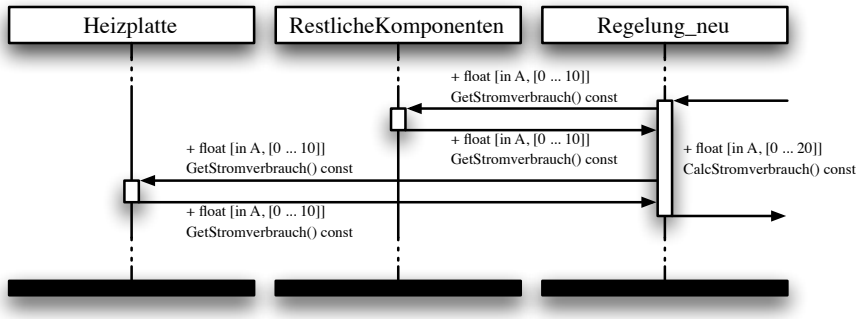


Abbildung 2.51.: Austausch des Objekts *Heizplatte* mit anschließender Kompatibilitätsbestimmung.

Ergebnis: Das neue Objekt *Regelung_neu* ist nicht kompatibel zum restlichen System, aufgrund der Tatsache, dass sich die Reihenfolge der Methodenaufrufe gegenüber dem Ausgangssystem geändert hat. Soll die neue Signalreihenfolge ebenfalls als kompatibel akzeptiert werden, so muss dies im Regelwerk explizit hinterlegt werden[98]. □

[98] Nähere Informationen zum Regelwerk finden Sie im Kapitel „2.4 *Kompatibilitätsregelwerk für die Modellierung und Bewertung von Kompatibilität*“ ab Seite 82.

2.3.2.4. Erweiterung der Objekte bzw. Klassen um explizit modellierte Schnittstellen für die Modellierung von Kompatibilität

Mit Hilfe des allgemeinen objektorientierten Modellbildungsparadigmas lassen sich, wie bereits im Kapitel „*Grundkonzepte der objektorientierten Modellbildung*“ gezeigt, nahezu beliebige technische Systeme, die aus Hard- und Software bestehen, vollständig modellieren und beschreiben. Dabei schreibt das generelle objektorientierte Modellbildungsparadigma nicht ausdrücklich vor, dass die Schnittstelle des Objekts/Klasse explizit modelliert und beschrieben werden muss. Für die Kompatibilitätsmodellierung genauso wie die anschließende Bewertung der Kompatibilität eines Objekts/Klasse ist jedoch die Existenz einer präzise modellierten, formal beschriebenen Schnittstelle eines Objekts/Klasse zwingend erforderlich, um sie – zum Beispiel bei der Verbindung zweier Objekte/Klassen in einem System – verifizieren zu können[99]. Aus diesem Grund muss die Schnittstelle eines Objekts/Klasse für die objektorientierte Kompatibilitätsmodellierung und -bestimmung erweitert werden. Zum einen müssen in der dezidierten Eingangsschnittstelle des Objekts/Klasse sämtliche Daten und Informationen hinterlegt werden die für die Ausführung der Arbeit des Objekt/Klasse benötigt werden. Zum anderen müssen in der dezidierten Ausgangsschnittstelle sämtliche von Objekt/Klasse angebotenen öffentlichen Eigenschaften und Methoden enthalten sein. Darüber hinaus muss insbesondere gelten: weder die Eingangs- noch die Ausgangsschnittstelle eines Objekts/Klasse darf leer sein.

Die folgende Definition fasst den oben beschriebenen Sachverhalt noch einmal knapp zusammen.

Definition 2.22 ***Erweiterung der Objekt- bzw. Klassenschnittstellen für die Kompatibilitätsmodellierung***
Jedes für die objektorientierte Kompatibilitätsmodellierung und -bewertung bestimmte Objekt/Klasse muss ***genau über eine nicht leere explizit modellierte Eingangs- und Ausgangsschnittstelle verfügen****. In der Eingangsschnittstelle müssen sämtliche von dem Objekt/Klasse benötigten Daten und Informationen eingetragen sein. Die Ausgangsschnittstelle enthält alle von dem Objekt/ Klasse angebotenen öffentlich zugänglichen Eigenschaften und Methoden des Objekts/Klasse.*

Dabei geht die hier eingeführte erweiterte Schnittstellendefinition über die zuvor, im Kapitel „2.3.1.4 *Modellierung von Klassenschnittstellen*“ ab Seite 56 vorgestellte allgemeine Lollipop-Schnittstellendefinition bei weitem hinaus. Zum einen müssen in der Eingangsschnittstelle eines Objekt/Klasse sämtliche benötigten Daten (Methodenaufrufe, Parameter etc.) im Gegensatz zur Lollipop-Notation explizit angegeben werden. Ebenso müssen sämtliche öffentlich zugänglichen Eigenschaften und Methoden des Objekts/Klasse in der Ausgangsschnittstelle vermerkt werden. Darüber hinaus müssen eventuell vorhandene Randbedingungen, wie beispielsweise Gültigkeitsbereiche oder verwendete Einheiten, ebenfalls ausdrücklich in der Schnittstelle des Objekts/Klasse angegeben werden. Dies gilt insbesondere für die Vor- und Nachbedingungen bzw. die Invarianten einer Methode eines Objekts/Klasse. Also beispielsweise die Invariante, dass eine bestimmte Eigenschaften eines Objekts/Klasse bei einem Methodenaufruf nicht verändert wird[100].

Die Abbildung 2.52 zeigt den grundsätzlichen Aufbau sowie die Struktur der erweiterten Objekt/Klassendefinition mit den beiden zwingenden Ein- und Ausgangsschnittstellen.

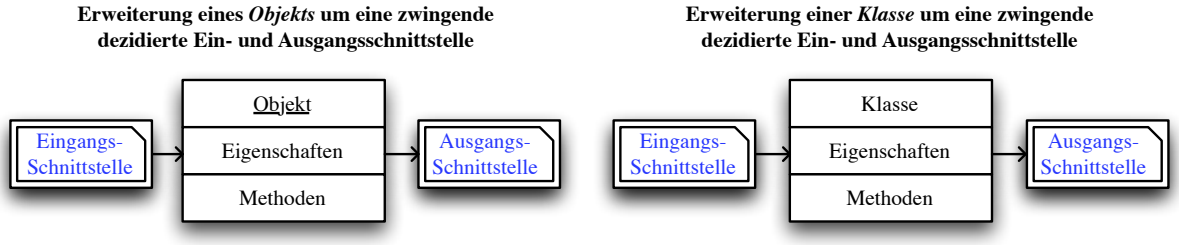


Abbildung 2.52.: Graphische Darstellung eines Objekts/Klasse für die Kompatibilitätsmodellierung und -bestimmung mit zwingender dezidierten Ein- und Ausgangsschnittstelle.

Sowohl hinter dem neu eingeführten Eingangs- als auch dem Ausgangsschnittstellenpfeil verbirgt sich eine Liste, in der sowohl die öffentlich zugreifbaren Eigenschaften, als auch die öffentlichen Methoden eines Objekts/Klasse in der zuvor eingeführten erweiterten Notation enthalten sind. Zusätzlich zur Eigenschafts- und Methodendeklaration kann die Schnittstellenbeschreibung Vor- und Nachbedingungen, Invarianten sowie für die Funktionalität der Klasse benötigte Eigenschaften und Methoden anderer Klassen enthalten. Die Abbildung 2.53 auf der nächsten Seite zeigt beispielhaft die Eingangsschnittstellenbeschreibung (links) der Klasse *Regelung_Komp* des Beispielsystems Kaffeemaschine, während auf der rechten Seite der Klasse die Ausgangsschnittstellenbeschreibung dargestellt ist.

In der Abbildung 2.53 ist beispielsweise die Zugriffsmethode `+ SetSpeichergroesse(long Speichergroesse)` der ursprünglichen Klasse *Regelung* (Abbildung 2.22) exemplarisch für alle anderen Methoden der Klasse herausgegriffen und entsprechend der oben eingeführten formalen Notation für die Kompatibilitätsmodellierung transformiert und sowohl in die Eingangsschnittstelle, als auch in die Klasse *Regelung_Komp* selbst eingetragen worden. Die erweiterte Methode `+ var void SetSpeichergroesse(long Speichergroesse [in KiBiByte, [0...100]]) var` kann nun für die Kompatibilitätsprüfung herangezogen werden. Zusätzlich enthält die Eingangsschnittstellenbeschreibung der Klasse *Regelung_Komp* zwei Hinweise, dass für die korrekte Funktion der Klasse zusätzlich die beiden Methoden `+ float [in A, [0 ... 10]] GetStromverbrauch() const` der Klassen *Heizplatte* und *RestlicheKomponenten* benötigt werden.

[99] Siehe hierzu insbesondere die beiden Kapitel „1.4.1 *Kompatibilitätsarten*“ ab Seite 22, sowie „1.4.2 *Kompatibilitätsszenarien*“ ab Seite 26.
[100] Nähere Informationen zu Vor- und Nachbedingungen sowie Invarianten finden Sie im Glossar ab Seite 354ff.

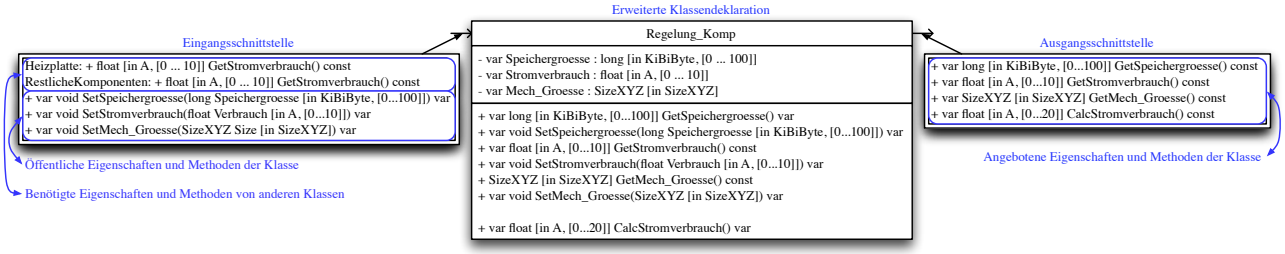


Abbildung 2.53.: Modellierung der Klasse *Regelung_Komp* mit explizit modellierter Ein- und Ausgangsschnittstelle.

Explizit modellierte Kommunikationsverbindungen zwischen Klassen/Objekten eines Systems
Zusätzlich zur expliziten Modellierung der Schnittstelle eines Objekts/Klasse ist die Modellierung sämtlicher Verbindungen, genauer der Kommunikationsverbindungen, zwischen den einzelnen Objekten/Klassen des Systems für die Kompatibilitätsbestimmung von entscheidender Bedeutung. Ohne die ausdrückliche Modellierung der Verbindungen ist eine Kompatibilitätsbestimmung nahezu unmöglich, da unbekannt ist, welche Objekte/Klassen des Systems Daten und Informationen austauschen[101]. Um sämtliche von einer Klasse benötigten Verbindungen mit anderen Klassen des Systems modellieren zu können, werden jeweils die Eingangsschnittstellenbeschreibungen der einzelnen Klassen herangezogen. Dabei werden für alle in der Eingangsschnittstellenbeschreibung aufgeführten Eigenschaften und Methoden der entsprechenden Klassen Verbindungen zu deren Schnittstelle hergestellt. Ob die Verbindung gültig (kompatibel) ist, wird dann während des Kompatibilitätstest[102] evaluiert. Die Abbildung 2.54 zeigt auszugsweise die beiden explizit modellierten Verbindung zwischen den Schnittstellen der Klassen *Regelung*, *Heizplatte* sowie der Klasse *RestlicheKomponenten*.

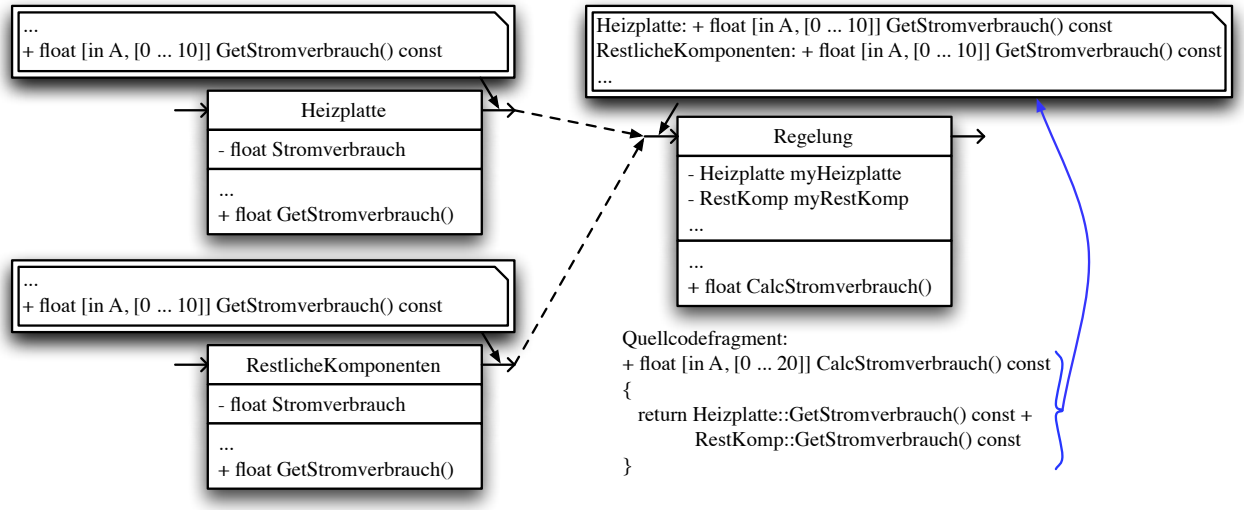


Abbildung 2.54.: Explizit modellierte Kommunikationsverbindung zwischen Schnittstellen unterschiedlicher Klassen eines Systems inklusive der für die Kompatibilitätsbestimmung notwendigen Schnittstellenbeschreibungen.

Das zur Klasse *Regelung* gehörige Quellcodefragment gibt an, dass sie für die korrekte Ausführung der Methode `+ var float [in A, [0 ... 20]] CalcStromverbrauch() const` Daten der beiden Klassen *Heizplatte* und *RestlicheKomponenten* benötigt. Diesen Sachverhalt zeigt die Klasse *Regelung* mit Hilfe der Eingangsschnittstellenbeschreibung an. Aufgrund dieses Eintrags im Beschreibungsfeld der Schnittstelle werden dann die Verbindungen zu den entsprechenden Klassen, genauer deren Schnittstellen der Klassen, hergestellt.

Anmerkung:
Es ist auch möglich, nur die explizit modellierte Schnittstelle in der erweiterten formalen Notation zu schreiben, und die ursprüngliche Klassendeklaration unverändert zu belassen. Dies wurde beispielsweise in der Abbildung 2.54 gemacht. Hier wurde lediglich die für die Kompatibilitätsmodellierung und -bestimmung notwendige Schnittstelle in der erweiterten formalen Notation angegeben, während die Eigenschaften und Methoden der Klassen unverändert gelassen wurden.

[101] Siehe hierzu insbesondere das Kapitel „2.3.2.1 *Identifikation der kompatibilitätsrelevanten Systemeigenschaften*“ ab Seite 59.
[102] Siehe hierzu das Kapitel „2.5 *Objektorientierte Kompatibilitätsprüfung*“ ab Seite 91.

2.4. Kompatibilitätsregelwerk für die Modellierung und Bewertung von Kompatibilität

Um die Kompatibilität beispielsweise eines Objekts mit einem System modellieren und bewerten zu können, wurde bereits im Kapitel „1.4.3 *Kompatibilitätsregelwerk*“ ab Seite 28 die Notwendigkeit eines Regelwerks erörtert. Aufbauend auf den dort beschriebenen Anforderungen an das Kompatibilitätsregelwerk werden in den nun folgenden beiden Unterkapiteln die Bestandteile des objektorientierten Regelwerks vorgestellt. Die Abbildung 2.55 zeigt den Aufbau und die Struktur des objektorientierten Kompatibilitätsregelwerks in Anlehnung an die Definition 1.21.

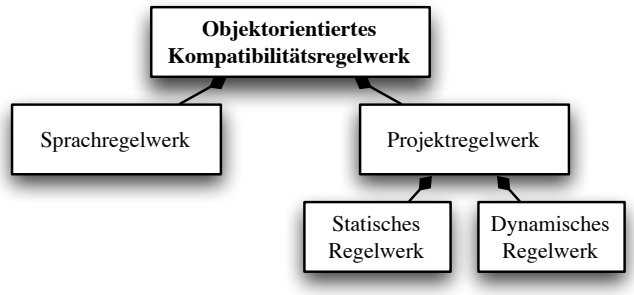


Abbildung 2.55.: Aufbau und Struktur des objektorientierten Kompatibilitätsregelwerks.

Das objektorientierte Kompatibilitätsregelwerk lässt sich in die beiden Bestandteile *Sprachregelwerk* und *Projektregelwerk* unterteilen. Dabei sind im *Sprachregelwerk* sämtliche strukturbildenden Sprachregeln, wie beispielsweise die formale BNF-artige Notation der Eigenschaften und Methoden eines Objekts/Klasse enthalten. Im Gegensatz dazu werden im *Projektregelwerk* logische Regeln hinterlegt, mit deren Hilfe beispielsweise die Kompatibilität von zwei Datentypen festgelegt werden kann. Um sämtliche Aspekte eines Systems auf Kompatibilität untersuchen zu können, muss das projektspezifische Regelwerk weiter unterteilt werden in ein *statisches Regelwerk*, in dem beispielsweise die Kompatibilität zweier Datentypen oder Einheiten hinterlegt ist, und ein *dynamisches Regelwerk*, mit dem die sich verändernden Aspekte des Systems überprüft werden können.

2.4.1. Sprachregelwerk

Im objektorientierten (projektunabhängigen) Sprachregelwerk ist die exakte formale Schreibweise sowohl der Eigenschaften als auch der verhaltensbeschreibenden Methoden der Klassen und Objekte eines Systems hinterlegt. Außerdem ist im Sprachregelwerk der graphische bzw. formale Aufbau und die Struktur der Schnittstellen der Objekte/Klassen und der Verbindungen zwischen den Klassen des Systems bindend festgelegt. Zusätzlich zur Beschreibung der formalen Syntax der Eigenschaften und Methoden der Klassen sind im Sprachregelwerk die grundsätzlichen Notationsregeln, sowohl der Klassen und Objekte, als auch der MSCs hinterlegt. Dabei gilt insbesondere: das projektunabhängige Sprachregelwerk darf nicht durch den Anwender der Sprache verändert werden. Sämtliche Regeln des Sprachregelwerks sind fest vorgegeben.

Definition 2.23 ***Eigenschaften des projektunabhängigen Sprachregelwerks***
Im ***projektunabhängigen Sprachregelwerk*** *sind sämtliche Regeln hinterlegt, die den formalen Aufbau und die Struktur der Eigenschaften, und der verhaltensbeschreibenden Methoden festlegen. Außerdem enthält das Sprachregelwerk Regeln für die Modellierung der Klassen, Objekte, Schnittstellen, Verbindungen sowie der MSCs.*

Grundregelwerk
Im Grundregelwerk sind sämtliche Regeln hinterlegt, die für die Kompatibilitätsmodellierung und -bestimmung notwendig sind. Beispielsweise sind folgende Regeln im Grundregelwerk enthalten (Auszug):

- Alle kompatibilitätsrelevanten Eigenschaften eines Objekts/Klasse müssen für die Kompatibilitätsmodellierung in der in der Definition 2.19 beschriebenen BNF-artigen Notation angegeben werden.
- Alle kompatibilitätsrelevanten verhaltensbeschreibenden Methoden eines Objekts/Klasse müssen für die Kompatibilitätsmodellierung in der, in der Definition 2.21 beschriebenen, BNF-artigen Notation angegeben werden.
- Jedes Objekt/Klasse muss mindestens einen dezidierten Ein- und mindestens einen dezidierten Ausgang besitzen.
- Die Schnittstelle einer Klasse/Objekts darf nicht leer sein.
- Sämtliche kompatibilitätsrelevanten Eigenschaften eines Objekts/Klasse müssen in der Schnittstelle des Objekts/Klasse hinterlegt sein.
- Sämtliche kompatibilitätsrelevanten verhaltensbeschreibenden Methoden eines Objekts/Klasse müssen in der Schnittstelle des Objekts/Klasse hinterlegt sein.
- Im Modell des Systems müssen sämtliche impliziten und expliziten Verbindungen zwischen den Klassen und Objekten des Systems eingetragen werden.

- Zu allen in der Schnittstellenbeschreibung einer Klasse eingetragenen benötigten Eigenschaften und Methoden muss es eine explizit modellierte Verbindung im Klassenmodell geben.
- Sämtliche über die Schnittstellenbeschreibung bzw. mittels explizit modellierter Verbindungen im Klassenmodell verbundenen Eigenschaften und Methoden müssen übereinstimmen.
- ...

Zusätzlich zum Grundregelwerk gibt es ein spezielles Regelwerk, in dem der Aufbau und die Struktur der zusammengesetzten Datentypen geregelt ist.

Regelwerk für die Modellierung von zusammengesetzten Datentypen
Bereits im Kapitel „2.3.2.2.2 *Verknüpfung von Datentyp, Präfix, Einheit und Gültigkeitsintervall*“ ab Seite 75 wurde die Notwendigkeit eines durchgängigen Regelwerks für die Modellierung von zusammengesetzten Datentypen erwähnt, um zu vermeiden, dass beispielsweise „sinnlose“ zusammengesetzte Datentypen in einem Modell verwendet werden. Die folgende Aufzählung illustriert beispielhaft einige Regeln, mit deren Hilfe valide zusammengesetzte Datentypen modelliert werden können.

- *Verbindung von Datentyp und Intervallgrenzen*
  Bei einem zusammengesetzten Datentyp muss der verwendete Datentyp mit den Intervallgrenzen übereinstimmen. Ist beispielsweise der Datentyp eines zusammengesetzten Datentyps vom Typ **`int`**, so müssen auch die beiden Intervallgrenzen des Gültigkeitsintervalls vom Datentyp **`int`** sein. So ist beispielsweise der folgende zusammengesetzte Datentyp

  ```
  + var Eigenschaft1 : int [in KW, [0...5]]
  ```

  wohlgeformt. Sowohl der verwendete Datentyp als auch der Datentyp der beiden Intervallgrenzen stimmt überein. Im Gegensatz dazu ist der folgende zusammengesetzte Datentyp falsch, aufgrund der Tatsache, dass hier die Untergrenze des Gültigkeitsintervalls nicht vom Typ **`int`**, sondern vom Typ **`float`** ist.

  `+ var` `Eigenschaft2 :` **`int`** `[in KW, [0,5...5]]` –> Fehler – Untere Intervallgrenze muss vom Typ **`int`** sein.

  Dabei gilt insbesondere, dass das Gültigkeitsintervall stets kleiner gleich der Mächtigkeit des verwendeten Datentyps sein muss. Beispielsweise reicht der gültige Zahlenbereich des vorzeichenlosen Datentyps **`unsigned int`**, auf einer 16-Bit Systemarchitektur von $0 \ldots 2^{16}$, also von 0 bis 65536. So darf das Gültigkeitsintervall ebenfalls nur Werte von 0 bis 65536 abdecken. Ansonsten kann es hier zu einer sehr schwer zu identifizierenden Inkompatibilität kommen.

- *Verbindung von Präfix und Einheit*
  In einem zusammengesetzten Datentyp dürfen lediglich verträgliche Präfixe und Einheiten verwendet werden. Beispielsweise sollte das Präfix K nur für physikalisch technische Einheiten verwendet werden, also beispielsweise für

  ```
  + var Eigenschaft3 : int [in KW, [0...5]].
  ```

  Im Gegensatz dazu dürfen sämtliche Präfixe, die auf dem Binärsystem beruhen, nicht für die Modellierung von physikalisch technische Einheiten verwendet werden.

  `+ var` `Eigenschaft4 :` **`int`** `[in KiBiByte W, [0...5]]` –> Fehler – Präfix und Einheit sind inkompatibel.

Beispiel 29: Erläuterung des objektorientierten Sprachregelwerk anhand des Beispielsystems Kaffeemaschine
Die Abbildung 2.56 auf der nächsten Seite zeigt einen kleinen Ausschnitt aus dem Klassenmodell des Beispielsystems Kaffeemaschine. In diesem Beispiel ist die Eigenschaft der Klasse *Heizplatte*, **`+`** `maxTemeratur :` **`float`**, mit Hilfe der oben eingeführten formalen Notation zur Beschreibung der kompatibilitätsrelevanten Eigenschaften (vgl. „Definition: 2.19“ auf Seite 68) modelliert und in die erweiterte Schnittstelle der Klasse eingetragen worden, also **`+ const`** `maxTemeratur :` **`float`** `[in K, [372.15 ... 374.15]]`. Des Weiteren enthält die Klasse *Heizplatte* die öffentliche Zugriffsmethode **`+ float`** `GetStromverbrauch()`. Auch diese Methode wurde mit Hilfe der formalen Notation an die Vorgaben für die Kompatibilitätsmodellierung angepasst und in die Schnittstelle der Klasse eingetragen (**`+ float`** `[in A, [0 ... 10]] GetStromverbrauch()` **`const`**).

Zusätzlich zu den Klassen und deren Schnittstellenbeschreibungen ist in der Abbildung 2.56 die Verbindung zwischen den Klassen, genauer der Schnittstellen der Klassen, des Systems graphisch modelliert und dargestellt. So kann beispielsweise überprüft werden, ob die beiden Klassen *Heizplatte* bzw. *RestlicheKomponenten*, genauer deren Ausgangsschnittstellenbeschreibungen, die von der Klasse *Regelung* geforderten Methoden **`+ float`** `[in A, [0 ... 10]] GetStromverbrauch()` **`const`** enthalten. Somit ist sichergestellt, dass der Aufruf der Methode `...` `CalcStromverbrauch()` `...` der Klasse *Regelung* nicht „ins leere läuft“, die entsprechenden Methoden in den beiden Klassen *Heizplatte* und *RestlicheKomponenten* vorhanden sind und deren Signatur mit der geforderten übereinstimmt.

Ergebnis: Das in der Abbildung 2.56 dargestellte Modell des Systems Kaffeemaschine enthält keine strukturellen bzw. syntaktischen Fehler – die Regeln des Sprachregelwerk sind alle erfüllt worden. □

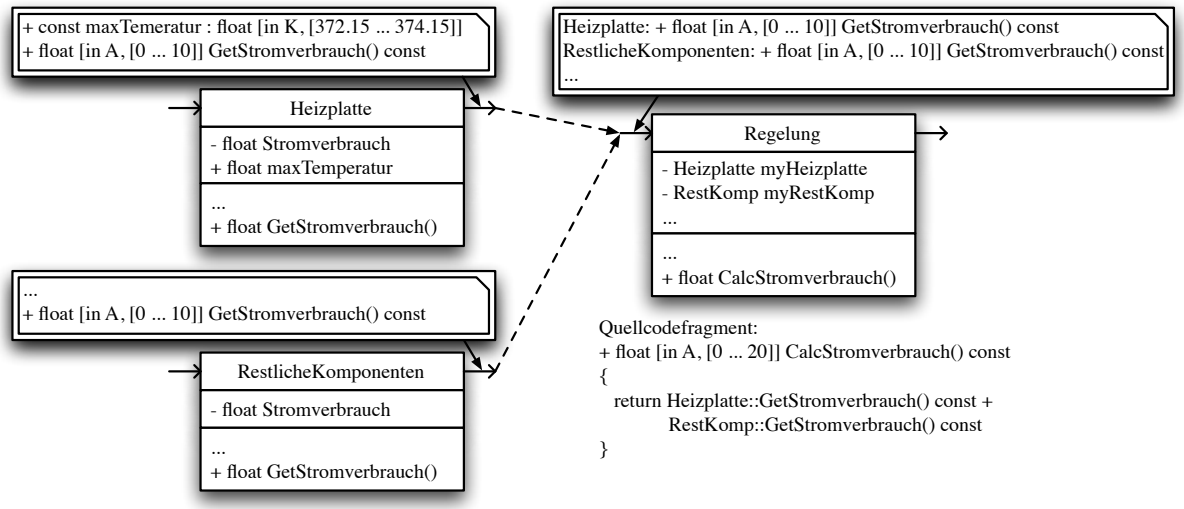


Abbildung 2.56.: Auszug aus dem Klassenmodell des Beispielsystems Kaffeemaschine.

2.4.2. Projektregelwerk

Nachdem im Kapitel „*Sprachregelwerk*“ das (objektorientierte) Sprachregelwerk vorgestellt wurde, folgt nun die Einführung des *projektspezifischen Regelwerks*. Im projektspezifischen Regelwerk sind sämtliche Regeln hinterlegt, mit deren Hilfe, die Kompatibilitätsbedingungen (beispielsweise von zwei Systemen) spezifiziert werden können. Dabei kann das projektspezifische Regelwerk im Gegensatz zum projektunabhängigen Sprachregelwerk an die speziellen Randbedingungen und Bedürfnisse eines Projekts individuell angepasst werden. So wird beispielsweise im projektspezifischen Regelwerk festgelegt, ob und wann zwei Datentypen kompatibel zueinander sind. Dabei ist das projektspezifische Regelwerk in genau die gleiche Anzahl an Unterregelwerken unterteilt, wie unterschiedliche Domänen an der Entwicklung eines Systems beteiligt sind. Wird beispielsweise eine Kaffeemaschine entwickelt, so sind im Allgemeinen daran mindestens die drei Domänen *Elektrotechnik*, *Mechanik* und *Software* beteiligt. Daraus folgt unmittelbar, dass das projektspezifische Regelwerk mindestens aus drei Teilregelwerken besteht. Die folgende Definition 2.24 beschreibt diesen Sachverhalt.

Definition 2.24 ***Eigenschaften des projektspezifischen objektorientierten Regelwerks***
*Im **projektspezifischen Regelwerk** sind sämtliche **statischen** und **dynamischen** Regeln hinterlegt, die die Kompatibilität zwischen einem Objekt/Klasse des Systems und einem (zweiten) System festlegen[103]. Dabei existieren genau so viele unterschiedliche Regelwerke wie Domänen an der Systementwicklung beteiligt sind. Dabei gilt: das projektspezifische Regelwerk kann an die individuellen Bedürfnisse eines Projekts angepasst werden.*

Obwohl das projektspezifische Regelwerk in so viele Regelwerke unterteilt ist, wie Domänen an der Systementwicklung beteiligt sind, sind vor allem diejenigen Regeln interessant, die über Domänengrenzen hinweggehen, da es vor allem an den Grenzen zwischen den Domänen zu Inkompatibilitäten kommt. Dies liegt vor allem daran, dass innerhalb einer Domäne meistens alle Randbedingungen bekannt sind, die zu einer Inkompatibilität führen können. Im Gegensatz dazu werden für gewöhnlich die Schnittstellen („Berührungspunkte“) mit anderen Domänen übersehen. So wird, wie in Abbildung 2.30 auf Seite 60 gezeigt wurde, der Zusammenhang zwischen CAD Daten und der Software eines Systems meistens übersehen, da er in keinem Modell explizit modelliert worden ist. Abhilfe schafft hier nur ein *integriertes domänenübergreifendes Systemmodell*, in dem sämtliche Daten über ein System hinterlegt und gespeichert sind. Im Kapitel „*Domänenübergreifendes integriertes Systemmodell*“ wird ein solches Systemmodell vorgestellt und seine Vorteile für die Kompatibilitätsmodellierung dargelegt.

Ein weiterer Lösungsansatz, um die bei der domänenübergreifenden Systemmodellierung auftretenden Probleme zu lösen, besteht in der durchgängigen Verwendung einer domänenübergreifenden Modellierungssprache, mit deren Hilfe ein zentrales Modell des gesamten Systems angefertigt werden kann, in dem sämtliche Systembestandteile modelliert sind. Im Kapitel „3 *Modellierungssprachen und -techniken für technische Systeme*“ ab Seite 109 werden mehrere Modellierungssprachen und Konzepte zur Modellierung von Systemen vorgestellt.

In den folgenden beiden Unterabschnitten wird zunächst das statische projektspezifische und im Anschluss daran das dynamische Regelwerk eingeführt und anhand von Beispielen erläutert.

2.4.2.1. Statisches projektspezifisches Regelwerk

Im statischen projektspezifischen Regelwerk sind diejenigen projektspezifischen Regeln hinterlegt, die für die statische Kompatibilitätsmodellierung und -bestimmung notwendig sind. Dabei unterteilt sich das statische projektspezifische Regelwerk, wie

[103] Dies gilt insbesondere für die beiden, im Kapitel „1.4 *Der Kompatibilitätsbegriff*“ ab Seite 18 vorgestellten Kompatibilitätsarten – Verträglichkeit (Def. 1.13) und Austausch- bzw. Ersetzungskompatibilität (Def. 1.14).

oben beschrieben, in so viele Teilregelwerke wie Domänen an der Entwicklung eines Systems beteiligt sind. Für eingebettete softwarelastige Systeme sind dies beispielsweise die drei Domänen Mechanik, Elektrotechnik und Software. Für jede der drei Domänen werden im nächsten Abschnitt einige Beispielregeln angegeben. Als Beispielsystem dient auch hier wieder das System Kaffeemaschine. Begonnen wird die Ausführung des Regelwerks mit der Beschreibung der statischen domänenübergreifenden Regeln für die Kompatibilitätsmodellierung und -bestimmung.

Domänenunabhängige projektspezifische statische Regeln zur Modellierung und Bewertung von Kompatibilität
In diesem Abschnitt sind diejenigen Regeln zusammengefasst die übergreifend für alle Domänen gleichermaßen gelten. Zu diesen Regeln gehört unter anderem die Festlegung der kompatiblen Präfixe, Einheiten und Teiler. Dabei gilt im Allgemeinen, dass nur identische Präfixe, Einheiten und Teiler als kompatibel angesehen werden. Eine Ausnahme bilden hier die Softwarepräfixe. Aufgrund der Tatsache, dass es zwei unterschiedliche Definitionen für Softwarepräfixe gibt[104], müssen diese prinzipiell als zueinander kompatibel angenommen werden. Werden beispielsweise bei einer Zuweisung die Präfixe Ki und KiBi verwendet, so kann grundsätzlich angenommen werden, dass diese identisch und daher kompatibel zueinander sind. Im Gegensatz dazu sind die beiden Präfixe K und Ki oder KiBi nicht kompatibel, da sie auf unterschiedlichen Zahlensystemen beruhen.

Soll in einem Modell eines Systems trotz der bekannten Inkompatibilität das Präfix K für die Modellierung von Softwaredatentypen erlaubt sein, kann dies durch das projektspezifische statische Kompatibilitätsregelwerk in drei Stufen festgelegt werden.

- **Fehler**
  Die Verletzung dieser Regel führt unmittelbar zu einem Kompatibilitätsfehler.
- **Warnung**
  Möglicherweise liegt eine Inkompatibilität vor. Diese Warnung sollte genauer untersucht werden.
- **OK**
  Es ist keine Inkompatibilität zu erwarten.

Diese Einteilung kann auf alle anderen projektspezifischen Regeln ebenfalls angewendet werden. Jedoch sobald eine Grundregel so modifiziert wird, kann dies zu schwerwiegenden Kompatibilitätsfehlern führen, die nur sehr schwer identifiziert werden können. Aus diesem Grund sollte das projektspezifische Regelwerk nur sehr behutsam an die unterschiedlichen Gegebenheiten eines Systems angepasst werden. Darüber hinaus sollte die Regelwerksanpassung nur von Spezialisten vorgenommen werden, die „wissen, was sie tun“.

Die folgende Aufzählung zeigt beispielhaft eine Auswahl an vordefinierten domänenübergreifenden Kompatibilitätsregeln.

- Nur identische Teiler sind kompatibel.
- Nur identische Präfixe sind kompatibel.
- Intervalle müssen – je nach Anwendung – ineinander passen oder deckungsgleich sein.
- ...

Für alle diese statischen projektspezifischen Kompatibilitätsregeln gilt: Sämtliche Regeln können durch den Anwender projektspezifisch angepasst und verändert werden.

Statische Kompatibilitätsregeln für die Modellierung und Kompatibilitätsbewertung von mechanischen Eigenschaften eines Objekts/Klasse
Im statischen projektspezifischen Regelwerk für die Modellierung und Bewertung von mechanischen Eigenschaften eines Systems, sind diejenigen Regeln hinterlegt, die sowohl für die Modellierung als auch für die Durchführung des Kompatibilitätstests von mechanischen Objekten/Klassen angepasst wurden. Dazu zählt insbesondere die Anpassung bzw. die unterschiedliche Interpretation der Gültigkeitsintervalle. Soll beispielsweise eine Passung aus zwei Bauteilen modelliert werden, so darf das Zielintervall nie kleiner als das Ausgangsintervall sein, da ansonsten beide Bauteile nicht ineinander passen. Auch der umgekehrte Fall ist nicht wünschenswert. Das folgende Beispiel illustriert den beschriebenen Zusammenhang.

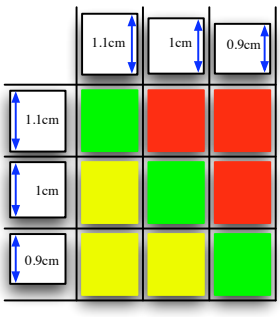


Abbildung 2.57.: Interpretation der Gültigkeitsintervalle anhand eines mechanischen Beispiels.

[104] Siehe hierzu insbesondere das Kapitel „2.3.2.2.1.2 *Modellierung von dekadischen und nicht dekadischen Präfixen mit Hilfe der erweiterten formalen Eigenschaftsdefinition*“ ab Seite 72

In der Abbildung 2.57 sind neun farbige Felder eingetragen, die anzeigen, ob zwei Boxen ineinander passen. So passt beispielsweise die Box in der ersten Zeile der Matrix lediglich in die Box in der ersten Spalte, ohne dass es dabei zu Spalten zwischen den beiden Boxen kommt. In die Boxen der restlichen beiden Spalten passt sie nicht hinein, da ihre Abmessung größer ist als die beiden anderen Boxen. Wiederum passt die Box in der untersten Zeile in sämtliche Boxen der Spalten. Dabei gilt jedoch: nur in die Box, die in der letzten Spalte aufgetragen ist, passt die Box ohne eine Lücke zu hinterlassen, bei allen anderen bleibt ein Spalt.

Mit Hilfe des statischen projektspezifischen Kompatibilitätsregelwerks kann nun definiert werden, ob auch andere Boxen im Modell ohne Fehler oder Warnung ineinander passen sollen. Dazu muss im obigen Beispiel lediglich die Farbe angepasst werden.

Die meisten mechanischen Regeln lassen sich auf das obige Beispiel zurückführen, also das ineinanderpassen von Intervallen[105].

Statische Kompatibilitätsregeln für die Modellierung und Kompatibilitätsbewertung von elektrotechnischen Eigenschaften eines Objekts/Klasse

Die meisten statischen Kompatibilitätsregeln für die Modellierung und Bewertung von Kompatibilität beruhen, wie bereits die mechanischen Kompatibilitätsregeln, auf der Schachtelung von Intervallen. Aus diesem Grund werden sie hier nicht noch einmal aufgeführt.

Statische Kompatibilitätsregeln für die Modellierung und Kompatibilitätsbewertung von Softwareeigenschaften eines Objekts/Klasse

Im Allgemeinen sind die beiden aus der Softwareentwicklung stammenden Datentypen **`unsigned int`** und **`unsigned long`** dann zuweisungskompatibel zueinander, wenn beispielsweise einer Variablen (Eigenschaft) `a` vom Datentyp **`unsigned long`** (**`unsigned long`** `a;`) die Variable `b` vom Datentyp **`unsigned int`** (**`unsigned int`** `b;`), also `a = b;` zugewiesen wird. Dies liegt daran, dass in den meisten Implementierungen der Programmiersprache C++ der Datentyp **`unsigned int`** 16-Bit ($2^{16} = 65536$) und der Datentyp **`unsigned long`** 32-Bit ($2^{32} = 4294967296$) groß ist und es somit zu keinem Datenverlust bei der Zuweisung kommen kann.

Wird in einem Modell jedoch der Variablen `b` die Variable `a` zugewiesen (`b = a;`) so kann es zu einem Datenverlust kommen, wenn Zahlen übergeben werden, die größer sind als sie von der Variablen `b` aufgenommen werden können, also beispielsweise `b` $= 2^{32} + 1$. Aus diesem Grund ist die Zuweisung dann inkompatibel. Ist jedoch bekannt, dass der verwendete Zahlenbereich der Variablen `a` nicht größer als der Zahlenbereich von `b` ist, so liegt keine Inkompatibilität (genauer: keine Zuweisungsinkompatibilität) vor. Aus diesem Grund ist der folgende zusammengesetzte Datentyp kompatibel.

```
1  + var a : unsigned long [in KW, [0...5]]
2  + var b : unsigned int [in KW, [0...5]]
3
4  a = b
5  b = a
```

Die beiden Zuweisungen 4 und 5 sind zuweisungskompatibel aufgrund der Tatsache, dass die beiden tatsächlich verwendeten Datenbereiche deckungsgleich sind und es somit trotz der unterschiedlichen Datentypen zu keinem Datenverlust kommt. Bei beiden wird jedoch eine Warnung ausgegeben, um anzuzeigen, dass es hier evtl. zu einem Kompatibilitätsproblem kommen kann.

Der gleiche Effekt kann erreicht werden, wenn die Gültigkeitsintervalle übereinstimmen, bzw. stets ein kleineres einem größeren zugewiesen wird. Dabei kommt es zu keinem Datenverlust und die Zuweisung kann als nicht strikt kompatibel angesehen werden. Wenn sowohl beide Gültigkeitsintervalle exakt deckungsgleich sind, als auch die beiden Datentypen übereinstimmen, liegt der Sonderfall der strikten Kompatibilität – die Identität – vor.

Die folgende Tabelle 2.6 zeigt einen Ausschnitt aus den elementaren Datentypen[106] der Programmiersprache C++ sowie deren Kompatibilität zueinander. Dabei gilt: Zwei elementare Datentypen sind genau dann zuweisungskompatibel, wenn es bei der Zuweisung zu keinem Datenverlust kommt.

Dabei wird stillschweigend angenommen, dass innerhalb eines Softwaresystems stets die gleiche Datentypausrichtung (d.h. little-endian oder big-endian[107]) verwendet wird. In verteilten Systemen und insbesondere bei eingebetteten verteilten Systemen kann diese Annahme jedoch zu fatalen Kompatibilitätsfehlern führen. Liefert beispielsweise der Sender ein Datum von Typ $\texttt{int}_{\text{little-endian}}$, während der Sender das Datum vom Typ $\texttt{int}_{\text{big-endian}}$ erwartet, stimmt zwar der Datentyp überein, jedoch repräsentieren die beiden Datentypen unterschiedliche Werte. Die Abbildung 2.58 zeigt diesen Sachverhalt.

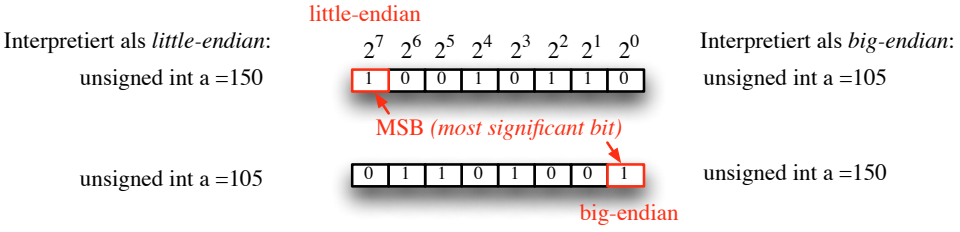


Abbildung 2.58.: Little- und big-endian Datentypausrichtung.

[105] Siehe hierzu auch das Kapitel „2.3.2.2.2 *Verknüpfung von Datentyp, Präfix, Einheit und Gültigkeitsintervall*“ ab Seite 75.

[106] Elementare Datentypen werden oft auch als Grunddatentypen bezeichnet. Im Gegensatz zu den Grunddatentypen stellt C/C++ dem Nutzer auch noch zusammengesetzte Datentypen, wie z.B. `struct`, zur Verfügung.

[107] Bei der big-endian (First-Byte-Order) wird das Byte mit höchstwertigen Bits (die signifikantesten Stellen) zuerst gespeichert, d.h. an der kleinsten Speicheradresse ([Wik06d][Wik08c][Wik08b]). Bei der Datentypausrichtung little-endian ist dies genau umgekehrt.

	char	short	int	long	float	double	long long	long double	unsigned char	unsigned short	unsigned int	unsigned long	bool
char	=	W	W	W	W	W	W	W	W	W	W	W	F
short	W	=	W	W	W	W	W	W	W	W	W	W	F
int	W	W	=	O	W	W	O	W	W	W	W	W	F
long	W	W	W	=	W	W	O	W	W	W	W	W	F
float	F	F	F	F	=	O	F	O	F	F	F	F	F
double	F	F	F	F	W	=	F	O	F	F	F	F	F
long long	W	W	W	W	W	W	=	W	W	W	W	W	F
long double	F	F	F	F	W	W	F	=	F	F	F	F	F
unsigned char	W	W	W	W	W	W	W	W	=	W	W	W	F
unsigned short	W	W	O	O	W	W	O	W	W	=	O	O	F
unsigned int	W	W	W	O	W	W	O	W	W	W	=	O	F
unsigned long	W	W	W	W	W	W	O	W	W	W	W	=	F
bool	F	F	F	F	F	F	F	F	F	F	F	F	=

Legende:

=	*Gleicher Datentyp*
O	*OK, keine Datentypkonvertierung, kein Informationsverlust*
W	*Warnung, es kann zu Datenverlust kommen*
F	*Fehler! Ungültige Konvertierung – Datenverlust*

Tabelle 2.6.: Datentypmatrix für C/C++ Datentypen.

In der Bildmitte ist jeweils ein 8-Bit Binärdatenwort mit eingezeichnetem MSB[108] eingezeichnet. Dieses Datenwort kann nun entweder als little-endian oder als big-endian interpretiert werden. Je nachdem, wie das Datenwort interpretiert wird ist der damit dargestellte Wert unterschiedlich – und damit nicht kompatibel zueinander, obwohl der Datentyp übereinstimmt.

Soll ein Systemmodell unabhängig von einer bestimmten Programmiersprache entwickelt werden, so wird eine generische Datentypmatrix mit eindeutig festgelegter Datentypausrichtung und fester Datentypbreite verwendet. Die folgende Tabelle 2.7 zeigt die programmiersprachenunabhängige generische Datentypmatrix, in der sämtliche Datentypen nach dem „little endian“ Paradigma ausgerichtet sind. Dabei sind die Datentypen **`bool`** 1-Bit, **`char`** und **`byte`** 8-Bit, **`short`** 16-Bit, **`int, long, float`** 32-Bit, sowie **`double, long long, long double`** 64-Bit breit.

2.4.2.2. Dynamisches projektspezifisches Regelwerk

Im projektspezifischen dynamischen Regelwerk sind sämtliche Regeln hinterlegt, mit deren Hilfe die dynamischen Aspekte der Kompatibilität eines Systems verifiziert werden können. Dazu zählen unter anderem die dynamische Über- oder Unterschreitung von Intervallgrenzen oder die Änderung des Sende-/Empfangsverhaltens von Nachrichten zwischen Objekten/Klassen des Systems. Dabei ist das dynamische Regelwerk wieder in so viele Teilregelwerke unterteilt, wie Domänen an der Systementwicklung beteiligt sind (domänenspezifisches Regelwerk). In diesem Regelwerk lassen sich die meisten dynamischen Regeln auf den Austausch von Nachrichten zwischen Klassen/Objekten bzw. das Über- bzw. Unterschreiten von Intervallgrenzen zurückführen.

Um die dynamische Kompatibilität eines Systems verifizieren zu können, muss das Modell des Systems bzw. das System ausgeführt (simuliert) werden[109]. Sämtliche dynamischen Kompatibilitätsprobleme können nur entdeckt werden, wenn das Modell/System ausgeführt wird, weil nur während der Ausführung des Modells/Systems dynamische Effekte wie beispielsweise eine Über- oder Unterschreitung eines statisch festgelegten Intervalls auftreten kann.

Dynamische Kompatibilitätsregeln für die Modellierung und Kompatibilitätsbewertung des mechanischen Verhaltens eines Objekts/Klasse
Um die dynamische Kompatibilität (Verhaltenskompatibilität) eines mechanischen Systems verifizieren und bewerten zu können, ist ein dynamisches Regelwerk notwendig, in dem beispielsweise Regeln für die mechanische Verhaltenskompatibilitätsbestimmung enthalten sind. Im folgenden Auszug aus dem Kompatibilitätsregelwerk für mechanische Klassen/Objekte sind einige Grundregeln exemplarisch aufgeführt. Für alle diese Regeln gilt: Die Anwendung des dynamischen projektspezifischen Regelwerks setzt die Ausführung/Simulation des entsprechenden Modells/Systems voraus.

- Sämtliche statisch festgelegten Intervallgrenzen dürfen während der Ausführung/Simulation des Modells/Systems nicht über- bzw. unterschritten werden. Dies gilt insbesondere für die

[108] Das Akronym *MSB* steht für **m**ost **s**ignificant **b**it – das höchstwertige Bit in einem Datenwort.
[109] Siehe hierzu die Definition 1.19 bzw. das Kapitel „2.2.2 *Simulation von Modellen*“ ab Seite 44.

	char	byte	short	int	long	float	double	long long	long double	unsigned char	unsigned short	unsigned int	unsigned long	boolean
char	=	W	W	W	W	W	W	W	W	W	W	W	W	F
byte	W	=	W	W	W	W	W	W	W	W	W	W	W	F
short	W	W	=	W	W	W	W	W	W	W	W	W	W	F
int	W	W	W	=	O	W	W	O	W	W	W	W	W	F
long	W	W	W	W	=	W	W	O	W	W	W	W	W	F
float	F	F	F	F	F	=	O	F	O	F	F	F	F	F
double	F	F	F	F	F	W	=	F	O	F	F	F	F	F
long long	W	W	W	W	W	W	W	=	W	W	W	W	W	F
long double	F	F	F	F	F	W	W	F	=	F	F	F	F	F
unsigned char	W	W	W	W	W	W	W	W	W	=	W	W	W	F
unsigned short	W	W	W	O	O	W	W	O	W	W	=	O	O	F
unsigned int	W	W	W	W	O	W	W	O	W	W	W	=	O	F
unsigned long	W	W	W	W	W	W	W	O	W	W	W	W	=	F
boolean	F	F	F	F	F	F	F	F	F	F	F	F	F	=

Legende:

=	*Gleicher Datentyp*
O	*OK, keine Datentypkonvertierung, kein Informationsverlust*
W	*Warnung, es kann zu Datenverlust kommen*
F	*Fehler! Ungültige Konvertierung – Datenverlust*

Tabelle 2.7.: Generische, programmiersprachenunabhängige Datentypmatrix.

  - Mechanische Abnutzung z.B. durch Verschleiß.
  - Wärmeentwicklung hervorgerufen beispielsweise durch Reibung.
  - Gewichtsveränderung beispielsweise hervorgerufen durch Abnutzung.
  - ...
- Mechanische Signale innerhalb des Systems müssen Reihenfolgetreu sein.
- ...

Das folgende Beispiel illustriert die Anwendung des projektspezifischen dynamischen Kompatibilitätsregelwerks für die Modellierung und Bewertung der Kompatibilität von mechanischen Klassen/Objekten anhand einer Passung zwischen zwei Bauteilen.

Beispiel 30: Anwendung des dynamischen projektspezifischen Regelwerks auf eine mechanische Passung
In diesem Beispiel soll demonstriert werden, dass obwohl die Intervallgrenzen im statischen Modell eingehalten worden sind, es dennoch zu einem Kompatibilitätsproblem kommt da ein Teil des Systems stärker erwärmt wird, als dies im statischen Modell vorgesehen war. Die tatsächliche Erwärmung und die damit verbundene Längenänderung kann jedoch nur durch die Ausführung/Simulation des Modells/Systems gefunden werden.

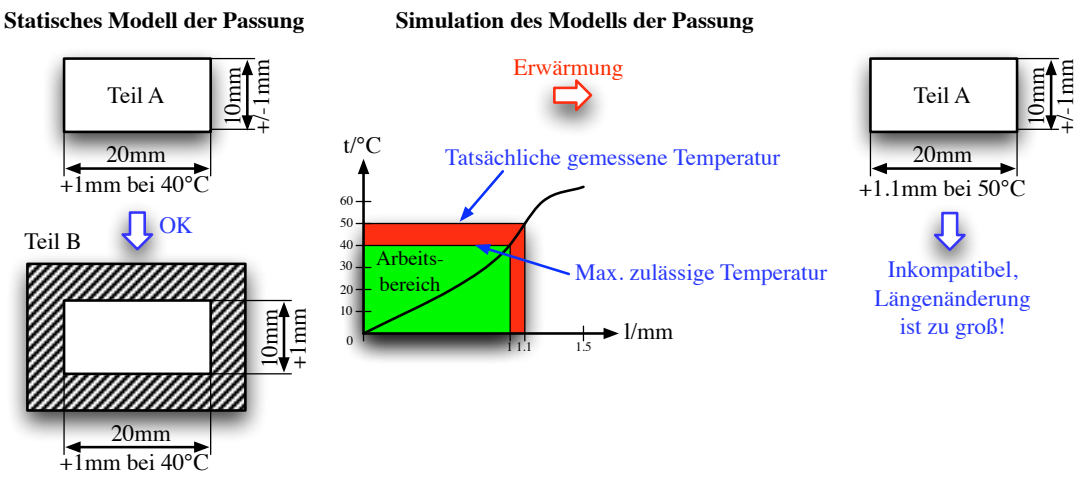


Abbildung 2.59.: Dynamisches Verhalten einer mechanischen Verbindung.

Die Abbildung 2.59 auf der vorherigen Seite zeigt auf der linken Seite oben das statische Modell eines Rechtecks (*Teil A*), das in die Nut des *Teil B* (linke Seite unten) eingepasst werden soll. Dabei ist die absolute Länge des *Teil A* 20mm mit einer Toleranzangabe von $+1\text{mm}$ bei einer Temperatur von $+40^\circ\text{C}$. Die Breite des *Teil A* beträgt 10mm – ebenfalls mit einer Toleranz von $+1\text{mm}$ bei einer Temperatur von $+40^\circ\text{C}$. Die Größenangaben der Nut des *Teil B* entsprechen den Angaben des *Teil A*.

Ergebnis: Bei der statischen Kompatibilitätsprüfung wird keine Kompatibilitätsverletzung festgestellt, weil sämtliche Intervalle ineinander passen. Das *Teil A* passt statisch in die Nut des *Teil B*.

Bei der Ausführung/Simulation des Modells/Systems, kann es jedoch vorkommen, dass beispielsweise durch die Erhöhung der Raumtemperatur die Ausdehnung des *Teil A* so groß wird, dass es nicht mehr in die Passung vom *Teil B* hineinpasst. Dieser Sachverhalt kann nun durch das dynamische Regelwerk gefunden werden, wenn das Systemverhalten simuliert wird. Im obigen Beispiel wird die maximale Längentoleranzangabe der Nut von *Teil B* um 0.1mm überschritten, weil sich das *Teil A* stärker erwärmt, als dies im statischen Modell vorgesehen wurde. Die tatsächliche Erwärmung des *Teil A* beträgt während der Simulation $+50^\circ\text{C}$ und nicht, wie im statischen Modell angegeben maximal $+40^\circ\text{C}$.

Ergebnis: Das *Teil A* passt nicht in die Nut des *Teil B*, weil sich das Modell/System stärker erwärmt, als dies im statischen Modell vorgesehen worden ist. Es liegt also eine dynamische Inkompatibilität vor. □

Dynamische Kompatibilitätsregeln für die Modellierung und Kompatibilitätsbewertung von elektrotechnischem Verhalten eines Objekts/Klasse
Um die dynamische Kompatibilität eines Systems zu verifizieren, muss zusätzlich die statische Kompatibilität überprüft werden. Also ob beispielsweise die elektrischen Signale innerhalb der statischen Intervallgrenzen liegen, bzw. ob die verwendeten Präfixe und Einheiten übereinstimmen. Ist die statische Kompatibilität gegeben, kann mit der dynamischen Kompatibilitätsprüfung begonnen werden. Um die dynamische Kompatibilität eines Systems überprüfen zu können, muss das Modell/System ausgeführt (simuliert) werden. In der folgenden Auszählung ist ein kleiner Auszug aus dem dynamischen Kompatibilitätsregelwerk für die Bestimmung der dynamischen projektspezifischen Kompatibilität aufgelistet. Mit Hilfe der dynamischen Kompatibilitätsregeln kann dann das dynamische Verhalten des Modells/Systems auf die Einhaltung der dynamischen Kompatibilitätsregeln während der Ausführung/Simulation hin überprüft werden.

- Um das projektspezifische dynamische Verhalten eines elektrischen Signals auf Kompatibilität überprüfen zu können, muss das Modell des Systems wohlgeformt und ausführbar (simulierbar) sein.
- Sämtliche statisch festgelegten Intervallgrenzen (ϵ-Umgebung) dürfen während der Ausführung/Simulation des Modells/Systems nicht über- bzw. unterschritten werden. Dazu zählen unter anderem die so genannten „Ausreißer“, die während der dynamischen Ausführung des Modells/Systems auftreten können.
- Elektrische/elektronische Signale innerhalb des Systems müssen Reihenfolgetreu sein.
- ...

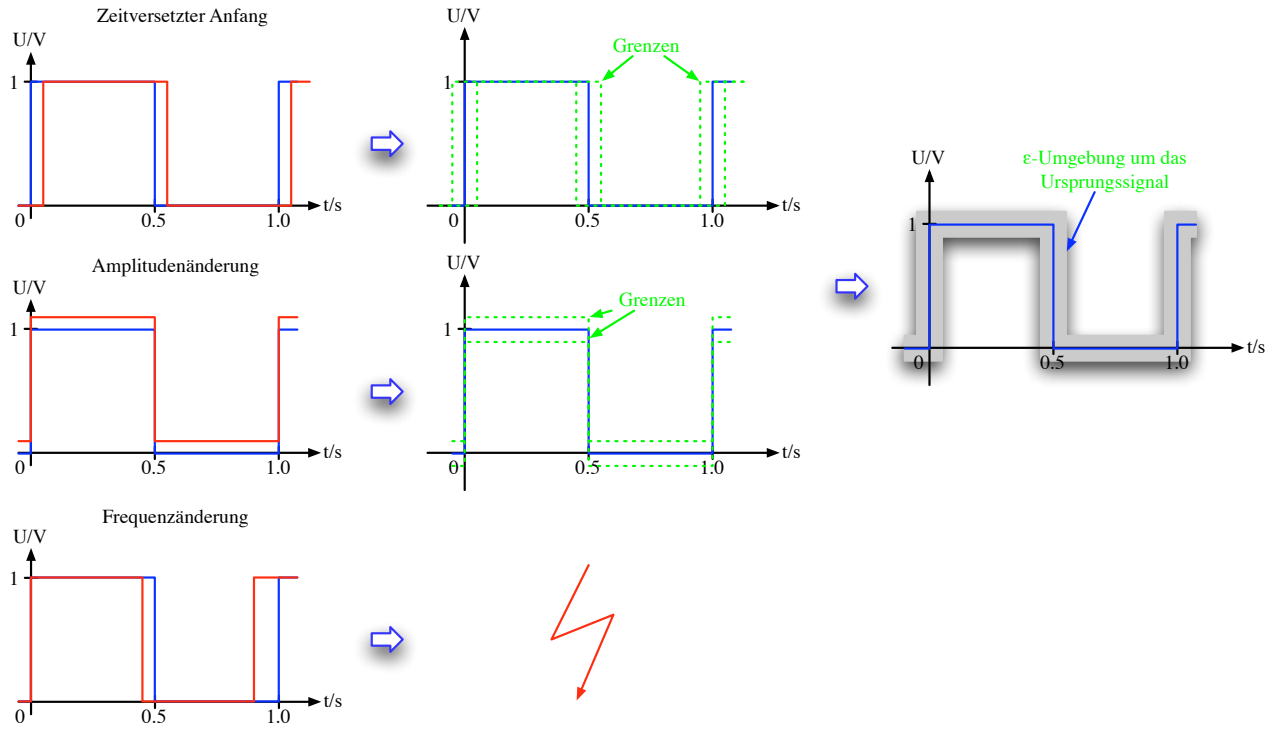


Abbildung 2.60.: Dynamisches Verhalten elektrischer Schwingungen.

Das folgende Beispiel illustriert die Anwendung des projektspezifischen dynamischen Kompatibilitätsregelwerks auf ein Rechtecksignal.

Beispiel 31: Anwendung des dynamischen projektspezifischen Regelwerks auf eine Rechteckschwingung
Die Abbildung 2.60 auf der vorherigen Seite zeigt links drei elektrische Signalverläufe. In blauer Farbe ist stets das identische Ausgangssignal, in roter Farbe das veränderte Signal dargestellt, das auf Kompatibilität zum blauen Ausgangssignal, mit Hilfe des dynamischen Regelwerks überprüft werden soll. Das hier in blauer Farbe dargestellte Ausgangssignal entspricht einem „idealen[110]“ Rechtecksignale mit einer Frequenz von 1Hz, einer Amplitude von 1V und beginnt stets bei $t = 0\text{s}$ mit der positiven Halbwelle. Dabei wird zwischen drei unterschiedlichen Änderungen gegenüber dem Ausgangssignal unterschieden. Im obersten Fall, ist das neue Signal um ca. 63ms zeitverschoben gegenüber dem Ausgangssignal. In der Bildmitte wiederum ist die Amplitude des neuen Signals um ca. 125mV größer als die des ursprünglichen Signals. Das Bild links unten zeigt in roter Farbe das veränderte Rechtecksignal mit einer Frequenz ca. 1.3Hz.

Alle drei links in Abbildung 2.60 dargestellten Signalverläufe (sowie sämtliche daraus ableitbaren Kombinationen) müssen durch das dynamische Kompatibilitätsregelwerk erfasst werden. In der Bildmitte sowie im Bild rechts sind Grenzen (gestrichelte grüne Linie) eingezeichnet, innerhalb derer sich das neue Signal bewegen muss (Kompatibilitätsregel) um als kompatibel zum Ausgangssignal angesehen zu werden. Aus der Kombination der beiden Grenzen (Bildmitte) entsteht die so genannte ϵ-Umgebung um das Ausgangssignal (graues Band). Liegt ein Signal innerhalb der ϵ-Umgebung, so ist das Signal kompatibel zum Ausgangssignal.

Dabei fällt auf, dass im Falle einer Frequenzänderung (Bild links unten) des neuen Signals gegenüber dem Ausgangssignal keine Kompatibilität vorliegen kann, da bereits die erste Schwingung die ϵ-Umgebung verlässt. Dies führt zu einer sofortigen Verletzung der Kompatibilitätsbedingung, dass das neue Signal nur dann kompatibel zum Ausgangssignal ist, wenn das neue Signal vollständig innerhalb der ϵ-Umgebung des Ausgangssignals liet (vgl. Abbildung: 2.60 rechts). □

Dynamische Kompatibilitätsregeln für die Modellierung und Kompatibilitätsbewertung von Softwareverhalten eines Objekts/Klasse
Für die dynamische projektspezifische Kompatibilitätsbestimmung des Softwareverhaltens eines Modells eines Systems sind ebenfalls spezielle dynamische Kompatibilitätsregeln notwendig. Ähnlich wie bei den dynamischen Kompatibilitätsregeln für mechanische und elektrische/elektronische Klassen/Objekte muss auch bei Softwaresystemen die Einhaltung der statisch definierten Grenzen während der Ausführung/Simulation des Modells/Systems untersucht werden. Die folgende Aufzählung zeigt eine kleine Auswahl an Regeln aus dem projektspezifischen dynamischen Regelwerk zur Kompatibilitätsbestimmung und -bewertung des Verhaltens von Software:

- Die vom Empfänger erwartete Signalreihenfolge muss mit der des Senders übereinstimmen.
- Das vom Empfänger erwartete Signal muss gleich dem des Senders sein.
- Sämtliche statischen Intervallgrenzen dürfen nicht über- oder unterschritten werden.
- Nachrichten/Signale dürfen nicht verloren gehen.
- ...

Die Abbildung 2.61 zeigt auf der linken Seite das MSC der Klasse/Objekt *C*. In diesem MSC ist die von der Klasse/Objekt *C* erwartete Signalreihenfolge eingetragen. Demnach erwartet die Klasse/Objekt *C* zunächst ein Signal von der Klasse/Objekt *A* und im Anschluss daran ein Signal von der Klasse/Objekt *B*. Auf der rechten Seite der Abbildung ist der tatsächliche dynamische Signalverlauf abgebildet. Dieser entspricht der von der Klasse/Objekt *C* erwarteten Signalreihenfolge.

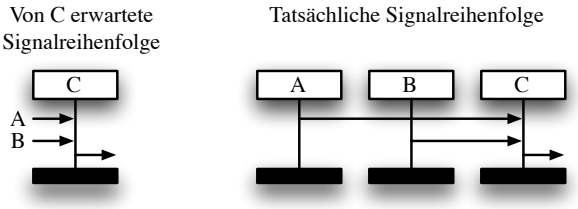


Abbildung 2.61.: Dynamisches Verhalten eines Softwaresystems.

Ergebnis: Bei der Ausführung/Simulation des Modells zeigt sich, dass die von der Klasse/Objekt *C* erwartete Signalreihenfolge der tatsächlichen Signalreihenfolge, die während der Simulation auftritt, entspricht. Somit ist die Klasse/Objekt *C* dynamisch kompatibel zum System.

Aufbauend auf dem hier auszugsweise vorgestellten projektspezifischen dynamischen Regelwerk ist es möglich, die dynamische Kompatibilität eines Objekts/Klasse zu seiner Umgebung verifizieren und bewerten zu können. Auf das hier vorgestellte Regelwerk wird insbesondere in den beiden Kapiteln „2.5 *Objektorientierte Kompatibilitätsprüfung*“ ab Seite 91 sowie „3.6.5 *(U)CML-Sprach- und Kompatibilitätsregelwerk*“ ab Seite 236 noch einmal eingegangen, nachdem die Kompatibilitätsmodellierungssprache (U)CML vorgestellt wurde.

[110] Das hier dargestellte Rechtecksignal wird als „ideal“ bezeichnet, da es weder Über- noch Unterschwingungen aufweist. Siehe hierzu die Abbildung F.2 auf Seite 342, in der der Aufbau einer Rechteckschwingung dargestellt ist.

2.5. Objektorientierte Kompatibilitätsprüfung

Aufbauend auf dem Kapitel „*Kompatibilitätsregelwerk für die Modellierung und Bewertung von Kompatibilität*“, in dem sowohl das Sprach- als auch das projektspezifische Regelwerk erläutert wurde, folgt in den nächsten beiden Unterabschnitten die ausführliche Beschreibung der Durchführung des eigentlichen Kompatibilitätstests. Dabei wird hier insbesondere auf das bereits im Kapitel „*Kompatibilitätsszenarien*“ vorgestellten Kompatibilitätsszenario – Kompatibilität im Sinne von austauschbar/ersetzbar (vgl. Definition: 1.14 auf Seite 20) – eingegangen da dieses Szenario am weitesten verbreiteten ist. Unabhängig von Kompatibilitätsszenario kann stets das gleiche generische Vorgehen bei der Durchführung des Kompatibilitätstest verwendet werden. In der Abbildung 2.62 ist das generische Vorgehen beim objektorientierten Kompatibilitätstest anhand eines Flussdiagramms erläutert.

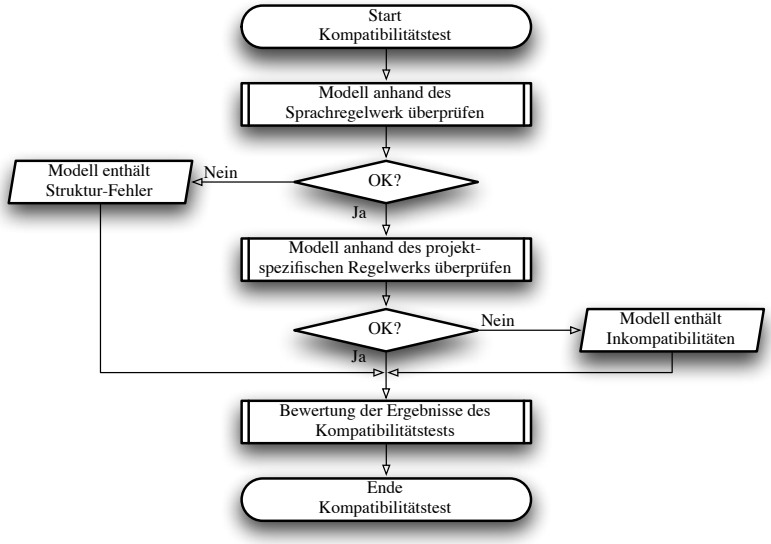


Abbildung 2.62.: Ablauf des objektorientierten Kompatibilitätstests.

Der Ablaufplan des Kompatibilitätstests beginnt zunächst mit der Überprüfung des Modells anhand des projektunabhängigen Sprachregelwerks[111]. Wurde der Sprachregeltest vom Modell erfolgreich absolviert, so folgt im zweiten Schritt die Prüfung der projektspezifischen Regeln. Wurde auch dieser Test erfolgreich durchgeführt, enthält das Modell des Systems im Allgemeinen keine Inkompatibilitäten. Liefert einer der beiden Tests einen Fehler, wird der Kompatibilitätstest abgebrochen. Dies gilt insbesondere für die Durchführung des Sprachregeltests. Tritt hier ein Fehler auf, so wird die weitere Überprüfung des Modells abgebrochen. Dies liegt daran, dass beispielsweise die Eigenschaft oder die verhaltensbeschreibende Methode, die einen Syntaxfehler enthält nicht weiter überprüft werden kann und somit die Durchführung des eigentlichen projektspezifischen Kompatibilitätstests somit keinen Sinn macht.

Ausgangsbasis für die Erläuterung der objektorientierten Kompatibilitätsprüfung ist wiederum das bereits mehrfach angesprochene Beispielsystem Kaffeemaschine.

2.5.1. Objektorientierte Kompatibilitätsprüfung und -bewertung des Systemmodells anhand des Sprachregelwerks

Der erste Schritt bei der objektorientierten Kompatibilitätsüberprüfung eines Modells eines Systems ist die Verifikation des Systemmodells anhand des zugrunde liegenden (projektunabhängigen) Sprachregelwerks (vgl. Abbildung: 2.62 auf Seite 91). Dabei wird das Modell des Systems schrittweise (bottom-up) untersucht. Tritt bei der Untersuchung des Modells eine Verletzung des Sprachregelwerks auf, so wird der Kompatibilitätstest an dieser Stelle abgebrochen, aufgrund der Tatsache, dass durch die Verletzung der Sprache keine gültige Aussage über die Kompatibilität des Modells des System getroffen werden kann. Denn hier gilt insbesondere der aus der Aussagenlogik stammende Grundsatz: „ex falso sequitur quodlibet“ (lat., aus Falschem folgt Beliebiges) [AHS08], [Wik08h], [Wik08a] und [DJ08, 23].

Vorgehen bei der objektorientierten Überprüfung des Systemmodells anhand des Sprachregelwerks
Um das Modell eines Systems auf die Einhaltung des Sprachregelwerks hin zu überprüfen, müssen zunächst sämtliche Objekte/Klassen des Systemmodells isoliert betrachtet werden. In diesem Schritt wird die Einhaltung der formalen Schreibweise (Syntax) der Eigenschaften, Methoden und der Schnittstellen des Objekts/Klasse betrachtet. Tritt dabei ein Fehler auf, wird die Verifikation an dieser Stelle abgebrochen. Ist die formale Schreibweise sämtlicher Objekte/Klassen fehlerfrei, wird im zweiten Schritt, mit der Verifikation der Verbindungen zwischen den Objekten/Klassen des Systemmodells begonnen. Die folgende Abbildung 2.63 auf der nächsten Seite sowie die unten stehende Aufzählung beschreiben das Vorgehen bei der Überprüfung des Systemmodells anhand des im Kapitel „*Sprachregelwerk*“ vorgestellten objektorientierten Sprachregelwerks.

[111] Nähere Informationen zu Programmablaufplänen finden Sie unter [DIN][Wik07q].

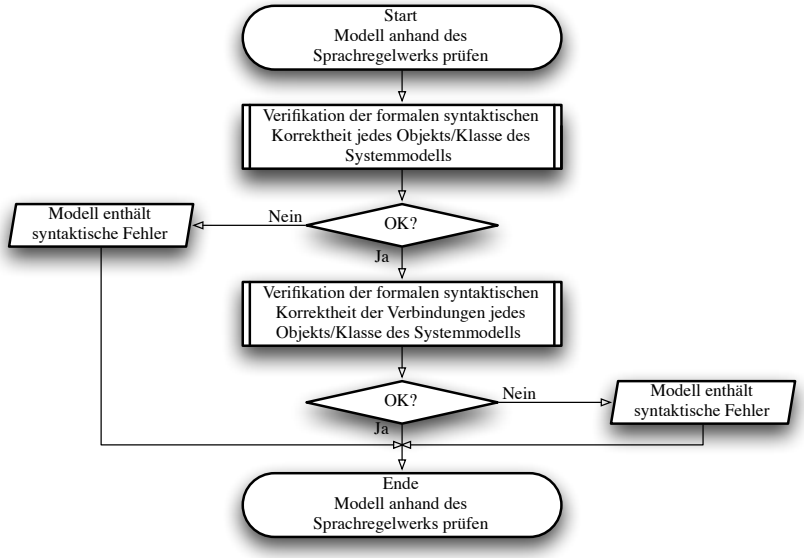


Abbildung 2.63.: Vorgehen bei der Überprüfung des Modells eines Systems anhand des Sprachregelwerks.

- **Verifikation der formalen syntaktischen Korrektheit jedes Objekts/Klasse des Systemmodells**
  Die Abbildung 2.64 zeigt das Modell der Klasse *Regelung_Komp* inklusive der beiden für die Kompatibilitätsmodellierung und -bestimmung notwendigen Schnittstellen.

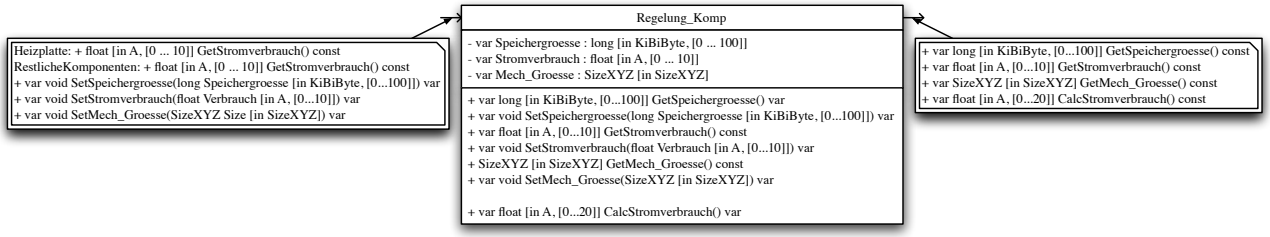


Abbildung 2.64.: Überprüfung der formalen syntaktischen Korrektheit der Klasse *Regelung_Komp* des Beispielsystems Kaffeemaschine anhand des Sprachregelwerks.

Zunächst wird jede Eigenschaft bzw. jede verhaltensbeschreibende Methode der Klasse auf die Einhaltung der formalen syntaktischen Schreibweise hin überprüft. Beispielsweise wird hier o.B.d.A. die Eigenschaft *Speichergroesse* der Klasse herausgegriffen und anhand des Sprachregelwerks auf die Einhaltung der korrekten Schreibweise hin untersucht.

```
- var Speichergroesse : long [in KiBiByte, [0 ... 100]]
```

Die Eigenschaft *Speichergröße* ist wohlgeformt. Sowohl die Sichtbarkeit, die Veränderbarkeit als auch der zusammengesetzte Datentyp ist nach der Sprachregelwerksdefinition wohlgeformt und enthält keinen syntaktischen Fehler. Aufgrund der eingeschränkten Sichtbarkeit – (privat), ist die Eigenschaft *Speichergröße* nur innerhalb der Klasse sichtbar und aus diesem Grund nicht in der Schnittstelle enthalten. Da die Speichergröße jedoch von außerhalb der Klasse manipuliert werden soll, existieren hierfür zwei Zugriffsmethoden, mit deren Hilfe die Eigenschaft manipuliert werden kann. Die beiden Zugriffsmethoden

```
+ var void SetSpeichergroesse(long Speichergroesse [in KiBiByte, [0 ... 100]]) var
+ var long [in KiBiByte, [0 ... 100]] GetSpeichergroesse() const
```

sind ebenfalls wohlgeformt und in den entsprechenden Schnittstellen der Klasse eingetragen.

Das Verhalten der Klasse *Regelung*, bzw. das Verhalten der verhaltensbeschreibenden Methode `+ float [in A, [0 ... 20]] CalcStromverbrauch() const` der Klasse *Regelung* kann nach Definition 2.15 auch mit Hilfe eines MSCs modelliert werden. Im Beispiel auf Seite 79 (Abbildungen 2.50 und 2.51) erfolgt die Verhaltensmodellierung der Methode `... GetStromverbrauch() ...` mit Hilfe eines MSCs. Auch das MSC muss syntaktisch wohlgeformt sein, um für die weitere Kompatibilitätsprüfung verwendet werden zu können. Die beiden dargestellten MSCs sind wohlgeformt und entsprechen somit allen vom Sprachregelwerk vorgegebenen Richtlinien.

Ergebnis: Sämtliche Eigenschaften und Methoden der Klasse *Regelung_Komp* sind, im Sinne des zuvor definierten Sprachregelwerks, wohlgeformt. Außerdem sind sämtliche öffentlich zugänglichen Eigenschaften und Methoden der Klasse in der Schnittstelle eingetragen. Auch das MSC der Methode `... GetStromverbrauch() ...` ist wohlgeformt.

- **Verifikation der formalen syntaktischen Korrektheit der Verbindungen jedes Objekts/Klasse des Systemmodells**
  Nachdem sämtliche Klassen des Systemmodells auf syntaktische Korrektheit hin überprüft worden sind, folgt nun die syntaktische Verifikation der Verbindungen zwischen den Klassen/Objekten des Systemmodells. Die Abbildung 2.65 illustriert die Überprüfung der Verbindungen zwischen Klassen des Systems Kaffeemaschine anhand der Schnittstellendeklaration, sowie der verhaltensbeschreibenden Methode `... CalcStromverbrauch() ...`.

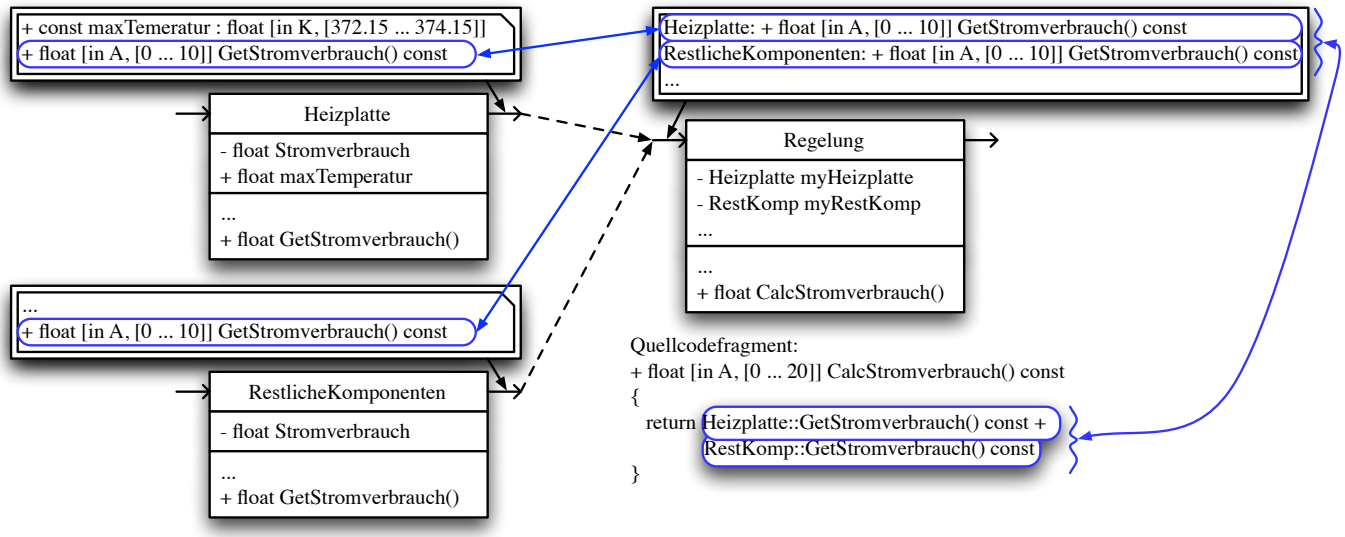


Abbildung 2.65.: Verifikation der Verbindung zwischen Klassen des Systems Kaffeemaschine.

Die verhaltensbeschreibende Methode `+ float [in A, [0 ... 20]] CalcStromverbrauch() const` (genauer: der Methodenrumpf) beschreibt, dass von den beiden Klassen *Heizplatte* und *RestlicheKomponenten* jeweils die Methode `... GetStromverbrauch() ...` aufgerufen wird. Die dafür benötigten explizit modellierten Verbindung sind sowohl im Klassenmodell als auch in den entsprechenden Schnittstellenbeschreibungen der Klassen hinterlegt. Anhand der explizit modellierten Verbindungen zwischen den Klassen des Systems bzw. den Einträgen in den Schnittstellenbeschreibungsfeldern kann die Kompatibilität der Signaturen der entsprechenden Methoden verifiziert werden[112]. So benötigt die Klasse *Regelung* laut Schnittstellenbeschreibung zur einwandfreien Funktion die beiden Methoden `+ float [in A, [0 ... 10]] GetStromverbrauch() const` der Klassen *Heizplatte* und *Regelung*. Im Modell sind sowohl die entsprechenden Verbindungen vorhanden als auch die Signaturen der Methoden fehlerfrei und dementsprechend kompatibel. Außerdem sind sämtliche Verbindungen innerhalb der beiden MSCs der beiden Abbildungen 2.50 und 2.51 im Klassenmodell des Systems (Abb.: 2.65) explizit hinterlegt und modelliert worden. Aus diesem Grund ist auch das MSC bzw. sind die Verbindungen im MSC wohlgeformt und verursachen keinen Kompatibilitätsfehler.

Ergebnis: Sämtliche Verbindungen im Modell, die für die Kompatibilitätsmodellierung notwendig sind, sind sowohl im Klassenmodell als auch im MSC hinterlegt. Des Weiteren stimmen die Signaturen der Methoden überein.

2.5.2. Objektorientierte Kompatibilitätsprüfung und -bewertung des Systemmodells anhand des projektabhängigen Kompatibilitätsregelwerks

Um die Kompatibilität eines Objekt oder eines ganzen Systems mit einem anderen Objekt/System (Verträglichkeit) bzw. den Austausch einer Komponenten (Objekt) eines Systems gegen eine andere (Austausch-/Ersetzungskompatibilität) bewerten zu können, wird zunächst die syntaktische Korrektheit des Modells verifiziert. Nachdem die syntaktische Verifikation des Systemmodells ohne Beanstandungen beendet ist, folgt die Überprüfung des Systemmodells anhand des vorgegebenen projektspezifischen Kompatibilitätsregelwerks. Dabei wird das projektspezifische Regelwerk, wie in den Definitionen 1.21 und 2.24 beschrieben, in zwei aufeinander aufbauende Teile unterteilt. Zum einen in das *statische Kompatibilitätsregelwerk*, in dem sämtliche statischen kompatibilitätsrelevanten Regeln hinterlegt sind. Und zum anderen in das *dynamische Kompatibilitätsregelwerk* in dem alle Regeln, die das dynamische Verhalten eines Systems einschränken abgelegt sind. Dabei kann das projektspezifische Kompatibilitätsregelwerk sowohl für die *Verträglichkeitsprüfung* nach Definition 1.13 als auch zur Verifikation eines Systemmodells auf *Austausch- und Ersetzungskompatibilität* nach Definition 1.14 verwendet werden. Insbesondere wird das projektspezifische Kompatibilitätsregelwerk für die Kompatibilitätsbestimmung in den nachfolgenden beiden Szenarien eingesetzt.

- **Szenario 1:** Verifikation eines neu entwickelten Systems anhand des projektspezifischen Kompatibilitätsregelwerks
  Nachdem das Modell eines Systems nach der objektorientierten Modellbildungsphase vollständig aufgebaut und anhand des Sprachregelwerks auf syntaktische Korrektheit überprüft worden ist, kann das Systemmodell mit Hilfe des vorgestellten

[112] Anmerkung:
Für die Kompatibilitätsprüfung und -bewertung zweier Signaturen wird zusätzlich das projektspezifische Regelwerk benötigt, da in diesem beispielsweise die Kompatibilität von Datentypen neu definiert werden kann.

projektspezifischen Kompatibilitätsregelwerks und der Spezifikation der Komponenten (Klassen/Objekte) auf Kompatibilitätsfehler hin untersucht und bewertet werden. Dabei wird beim Test mit dem projektspezifischen Regelwerk – angewendet auf ein noch nicht geprüftes Systemmodell – die Kompatibilität der Schnittstellen von miteinander verbundenen Bestandteile (Objekte/Klassen) des Systems untersucht und bewertet[113].

Die folgende Abbildung 2.66 zeigt das prinzipielle Vorgehen bei der Kompatibilitätsbestimmung eines neu entwickelten Systemmodells. Aus der objektorientierten Modellbildungsphase[114] stammen dazu die Klassen, die statischen Eigenschaften und Methoden der Klassen bzw. der Objekte und die dynamische Verhaltensbeschreibung der Klassen/Objekte (1a und 1b). Aus diesen Einzelbestandteilen wurde das Gesamtsystem aufgebaut (2). Dieses Gesamtmodell wird nun mit Hilfe des projektspezifischen Kompatibilitätsregelwerks evaluiert. Dabei wird stets auf die Spezifikation der Klassen/Objekte zurückgegriffen, um zu verifizieren, ob das Systemmodell in sich kompatibel ist. Also ob sämtliche definierten Bedingungen in den Schnittstellen der Klassen/Objekte (statisch) sowie das modellierte Verhalten der Klassen/Objekte des Systemmodells kompatibel zueinander sind.

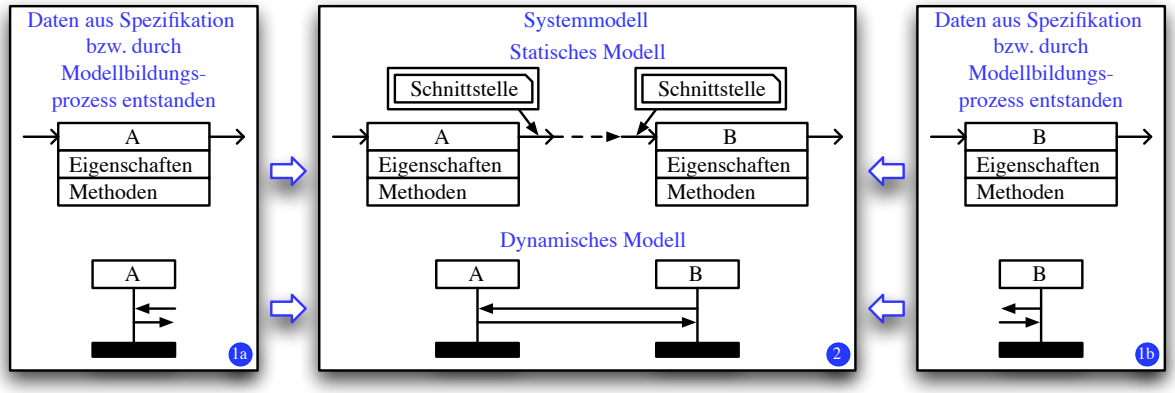


Abbildung 2.66.: Kompatibilitätsmodellierung und -bestimmung eines neu entwickelten Systemmodells.

- **Szenario 2:** Überprüfung eines Systems nach der Ersetzung/Austausch einer Komponente (Objekt) des Systems
  Das zweite Szenario beschreibt das Vorgehen beim Kompatibilitätstest, wenn ein Objekt des Systemmodells gegen ein anderes ausgetauscht/ersetzt wird. Dabei wird das ursprüngliche Systemmodell als Referenzsystem verwendet zu dem sowohl die neuen Komponenten als auch das gesamte neue Systemmodell kompatibel sein muss[115].

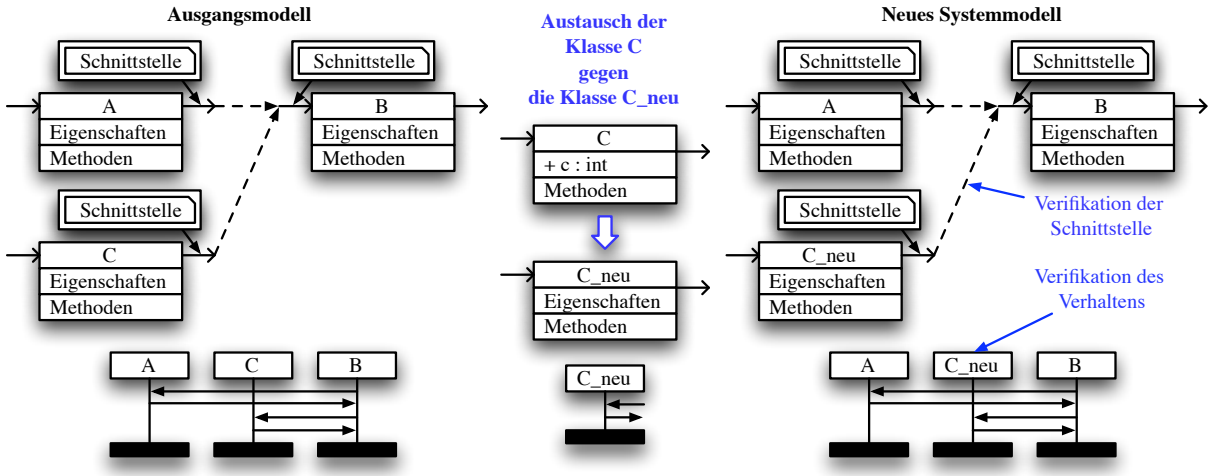


Abbildung 2.67.: Austausch eines Objekts aus dem Systemmodell mit anschließender Kompatibilitätsmodellierung und -bestimmung des neuen Systemmodells.

Die Abbildung 2.67 zeigt das prinzipielle Vorgehen beim Kompatibilitätstest nach dem Austausch einer Klasse/Objekts des Systemmodells. Dabei ist zunächst die statische Kompatibilität der neuen Klasse/Objekt mit dem Systemmodell sowie im Anschluss daran die dynamische Kompatibilität der neuen Klasse/Objekt mit dem Systemmodell sicherzustellen. Außerdem muss das neue Systemmodell kompatibel zum Referenzsystem sein.

[113] Anmerkung:
Das projektspezifische Kompatibilitätsregelwerk kann insbesondere bereits während der objektorientierten Modellbildung zur Verifikation des Systemmodells eingesetzt werden.
[114] Siehe hierzu: Objektorientierte Analyse (OOA – Object-Oriented Analysis) ab Seite 42ff.
[115] Siehe hierzu insbesondere auch das Szenario 2 im Kapitel „1.4.2 *Kompatibilitätsszenarien*“ ab Seite 26.

Beide hier beschriebenen Szenarien werden in den folgenden beiden Beispielen noch einmal aufgegriffen. Insbesondere das Szenario 2, in dem eine Komponente eines Systemmodells gegen eine neue ersetzt, und anschließend das neu entstandene System auf Kompatibilität untersucht und bewertet wird.

Nachdem die unterschiedlichen Einsatzmöglichkeiten des projektspezifischen Kompatibilitätsregelwerks vorgestellt wurden, folgt nun die Vorgehensbeschreibung bei der objektorientierten projektspezifischen Kompatibilitätsbestimmung. Die Abbildung 2.68 zeigt den prinzipiellen Ablauf des statischen und des dynamischen projektspezifischen Kompatibilitätstests des Modells eines Systems anhand des vorgegebenen projektspezifischen Regelwerks. Dabei wird die Kompatibilität der einzelnen Bestandteile des Systems, genauer die Schnittstellen zwischen den einzelnen Bestandteilen (Objekten) des Systems, auf Kompatibilitätsfehler untersucht und bewertet.

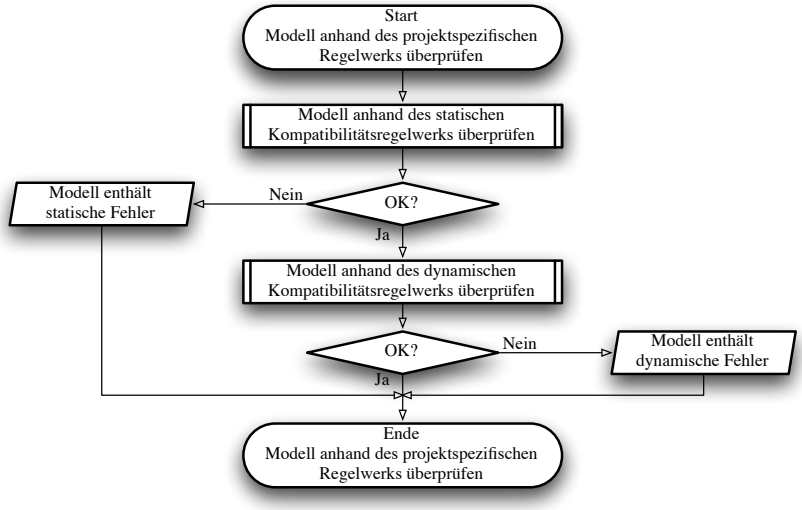


Abbildung 2.68.: Ablauf des statischen und des dynamischen projektspezifischen Kompatibilitätstests.

Wie der Ablaufpan zeigt, wird zunächst das Modell des Systems mittels des im Abschnitt 2.4.2.1 vorgestellten statischen projektspezifischen Regelwerks überprüft. Wurde die Prüfung erfolgreich absolviert, so wird im zweiten Schritt das Modell des Systems mit Hilfe des zuvor definierten dynamischen Regelwerks (siehe hierzu das Kapitel 2.4.2.2) evaluiert. Hat das Modell des zu verifizierenden Systems sowohl die statische als auch die dynamische Verifikation erfolgreich überstanden, so enthält das Modell des Systems im Allgemeinen keine Inkompatibilitäten.

Wenn der projektspezifische Kompatibilitätstest keine Fehler oder Inkompatibilitäten im Modell findet, bedeutet dies nicht zwangsläufig, dass das Modell des Systems keine Fehler oder Inkompatibilitäten enthält. Wurden beispielsweise kompatibilitätsrelevante Eigenschaften nicht modelliert oder vorhandene Verbindungen des Systems im Modell nicht hinterlegt, so findet der oben erläuterte Algorithmus keine Fehler oder Inkompatibilitäten im Modell – trotzdem ist System nicht fehlerfrei. Aus diesem Grund ist stets penibel darauf zu achten, dass bei der Identifikation der kompatibilitätsrelevanten Eigenschaften des Systems[116] größtmögliche Sorgfalt an den Tag gelegt wird, damit keine kompatibilitätsrelevanten Eigenschaften „übersehen“ werden.

Nachdem das Vorgehen bei der projektspezifischen Kompatibilitätsbestimmung sowie die damit verbundenen Probleme im letzten Abschnitt dargelegt wurden, folgt nun die Erläuterung der Durchführung des eigentlichen Kompatibilitätstests basierend auf den beiden im Kapitel „*Projektregelwerk*“ vorgestellten projektspezifischen Regelwerken. Begonnen wird dabei mit der statischen Kompatibilitätsprüfung.

Überprüfung der statischen Eigenschaften eines Objekts/Systems auf Kompatibilität
Mit Hilfe des statischen projektspezifischen Kompatibilitätsregelwerks wird das Modell des Systems auf die Existenz von statischen projektspezifischen Kompatibilitätsfehlern untersucht und bewertet. Das bedeutet, dass sämtliche in den Schnittstellen der Klassen/Objekten des Systems enthaltenen öffentlichen und geschützten Eigenschaften sowie die verhaltensbeschreibenden öffentlichen und geschützten Methoden, die über die Schnittstellen mit anderen Klassen (genauer den entsprechenden Methoden(-rümpfen) der Klassen) kommunizieren wollen, auf Kompatibilität untersucht werden müssen. Zusätzlich müssen die öffentlichen Zugriffsmethoden auf Kompatibilität zu den entsprechenden privaten Eigenschaften der Klasse überprüft werden. Die Abbildung 2.69 auf der nächsten Seite zeigt zwei Klassen eines Systems, von denen die Klasse *A* über die Schnittstelle mit der Klasse *B* verbunden ist. Im darauf folgenden Listing ist die Methode `+ void foo()` beschrieben. Innerhalb ihres Methodenrumpfes sind verschiedene Zuweisungen exemplarisch aufgeführt sowie ihr Ergebnis als Kommentar an die jeweilige Zuweisung angefügt.

```
1  + void foo()
2  {
3    ...
4    q1 = A::a1;       // OK
5    q1 = A::GetA2();  // Fehler: Intervallgrenzen stimmen nicht
6    q1 = A::a3;       // Fehler: Datentypen stimmen nicht (Datenverlust), Einheit stimmt nicht überein
7
8    q2 = A::GetA2();  // Warnung: Datentyp von "q2" größer als von "GetA2()"
```

[116] Siehe hierzu das Kapitel „2.3.2.1 *Identifikation der kompatibilitätsrelevanten Systemeigenschaften*“ ab Seite 59.

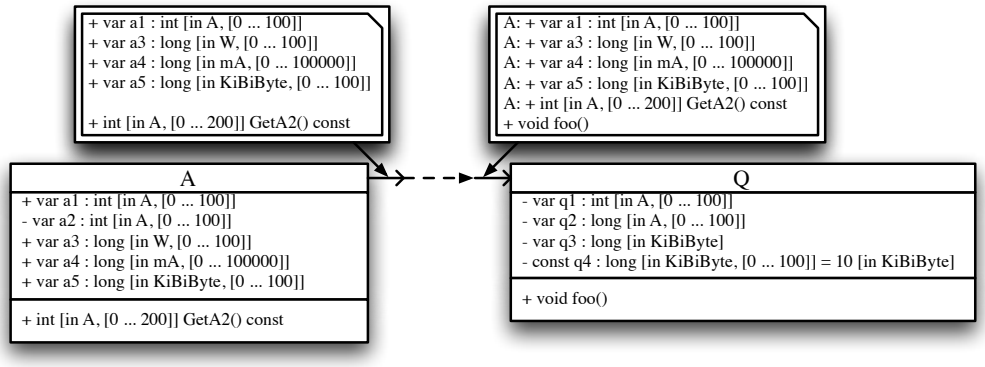


Abbildung 2.69.: Überprüfung der statischen Eigenschaften eins Objekts/Klasse auf Kompatibilitätsfehler.

```
 9     q2 = A::a3;       // Fehler: Einheit stimmt nicht überein
10     q2 = A::a4;       // Fehler: Präfix stimmt nicht überein, Intervallgrenzen stimmen nicht überein
11
12     q3 = A::a5;       // Warnung: Keine Intervallgrenze angegeben, Zuweisung möglicherweise fehlerhaft
13     q4 = A::a5;       // Fehler: Zuweisung an eine Konstante
14   }
```

Erläuterungen zum Listing:
Im oben aufgeführten Listing sind verschiedene Zuweisungen innerhalb des Methodenrumpfs der Methode `+ void foo()` der Klasse *Q* angegeben. Bis auf die erste Zuweisung (Zeile 4) sind sämtliche Zuweisungen entweder fehlerhaft oder produzieren eine Warnung. Insbesondere verursacht Zeile 12 eine Warnung, weil zwar der verwendete Datentyp und das Präfix modelliert wurde und übereinstimmen, jedoch kein Gültigkeitsintervall angegeben ist. Somit kann hier ein „unerwünschter" Effekt auftreten, vor dem gewarnt werden muss. Wie oben erwähnt, muss auch der Zugriff von öffentlichen Zugriffsmethoden auf interne Datenstrukturen überprüft werden. Im Beispiel verursacht die öffentliche Methode `+ int [in A, [0 ... 200]] GetA2() const` der Klasse *A* eine Warnung beim Zugriff auf die interne (private) Eigenschaft `- var a2 : int [in A, [0 ... 100]]`, da die beiden definierten Gültigkeitsintervalle nicht übereinstimmen.

Wie bereits mehrfach erwähnt, kann das projektspezifische Regelwerk an die individuellen Bedürfnisse eines Systems angepasst werden. So kann es vorkommen, dass in einem Modell eines Systems beispielsweise der Datentyp **`long`** sehr wohl dem Datentyp **`int`** zugewiesen werden kann, ohne dass es dabei zu einem Informationsverlust kommen kann, weil beispielsweise beide Datentypen die gleiche Repräsentation im Speicher haben, also gleich groß sind (**`int`** $= 2^{32}$ und **`long`** $= 2^{32}$) und die selbe Ausrichtung haben. Oder weil beispielsweise der Gültigkeitsbereich des Datentyps **`long`** mittels eines Intervalls so eingeschränkt wurde, dass er vollständig in den Datentyp **`int`** passt. Dann wird kein Fehler ausgegeben, sondern lediglich eine Warnung, die signalisiert, dass möglicherweise eine Unstimmigkeit vorliegt. Das folgende Listing beschreibt den zweiten Fall.

```
1   + var a : int [in A, [0 ... 32000]]
2   + var b : long [in A, [0 ... 32000]]
3   + var c : long [in A, [0 ... 32000]]
4
5   a = b; // Warnung, jedoch kein Kompatibilitätsfehler, da beide Gültigkeitsintervalle gleich groß sind!
6   c = b; // OK: c ist strikt kompatibel zu b
```

Die Zuweisung in Zeile 5 ist somit *nicht strikt kompatibel* (vgl. Definition: 1.17) weil die beiden Datentypen nicht übereinstimmen. Es kommt jedoch durch die Einschränkung des Gültigkeitsbereichs zu keinem Informationsverlust. Die Zuweisung in der Zeile 6 hingegen ist *strikt kompatibel*. Sowohl die Datentypen als auch die Einheit und das Gültigkeitsintervall stimmen vollständig überein (Sonderfall der strikten Kompatibilität – Identität (vgl. Satz: 1.1)). Dadurch werden sämtliche Forderungen der strikten Kompatibilität erfüllt (vgl. Definition: 1.16).

Das folgende Beispiel illustriert die Durchführung der statischen projektspezifischen Regelwerksprüfung anhand eines kleinen Auszugs aus dem Beispielsystem Kaffeemaschine.

Beispiel 32: Überprüfung des Beispielsystems Kaffeemaschine anhand des statischen projektspezifischen Regelwerks
Das folgende Beispiel unterscheidet zwei Szenarien. Im ersten Szenario wird das ursprüngliche Modell des Systems Kaffeemaschine mit Hilfe des statischen projektspezifischen Kompatibilitätsregelwerks auf Kompatibilität untersucht. Im zweiten Szenario wird dann eine Komponente (Klasse/Objekt) des Systems gegen eine andere ausgetauscht und das neue Systemmodell auf Kompatibilität untersucht.

- **Szenario 1:** Verifikation eines neu entwickelten (oder ungeprüften) Systems anhand des projektspezifischen Kompatibilitätsregelwerks
  Die folgende Abbildung (2.70) illustriert den Test mit dem projektspezifischen statischen Kompatibilitätsregelwerk anhand eines kleinen Auszugs aus dem Klassenmodell des Beispielsystems Kaffeemaschine. Das Klassenmodell zeigt dabei die drei Klassen *Heizplatte, RestlicheKomponenten* und *Regelung*, die dazugehörigen Schnittstellen, die Verbindungen zwischen den Klassen sowie die Implementierung der entsprechenden Methoden. Als Grundlage (Referenz) dienen hier die in vorangegangen Kapiteln erarbeiteten Klassen/Objekte der Kaffeemaschine.

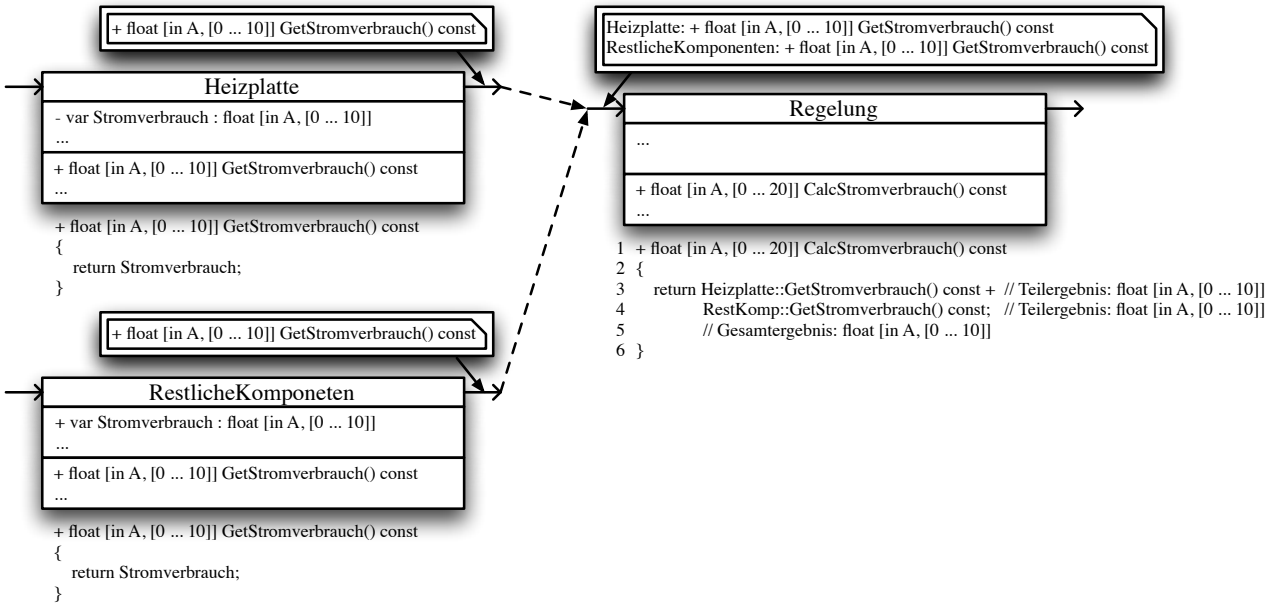


Abbildung 2.70.: Anwendung des statischen projektspezifischen Kompatibilitätstests auf das Beispielsystem Kaffeemaschine.

Im folgenden wird insbesondere die statische Kompatibilität der Methode **+ float** [**in** A, [0 ... 20]] CalcStromverbrauch() **const** der Klasse *Regelung* untersucht. Die Methode berechnet den aktuellen Stromverbrauch der beiden an die Regelung angeschlossenen Klassen/Objekte. Dazu ruft sie in ihrem Methodenrumpf (Zeile 3) zunächst die Methode **+ float** [**in** A, [0 ... 10]] GetStromverbrauch() **const** der Klasse *Heizplatte* auf und addiert das Ergebnis dieses Methodenaufrufs mit dem Methodenaufruf **+ float** [**in** A, [0 ... 10]] GetStromverbrauch() **const** der Klasse *RestlicheKomponenten* und gibt das Ergebnis der Berechnung an den Aufrufer der Methode ... CalcStromverbrauch() ... zurück.

Um die Kompatibilität des Methodenaufrufs ... CalcStromverbrauch() ... verifizieren zu können, werden die beiden Methodenaufrufe in Zeile 3 in mehrere Teilaufrufe zerlegt und einzeln bewertet. Der Aufruf der Methode ... GetStromverbrauch() ... der Klasse *Heizplatte* liefert als Ergebnis eine Zahl vom Datentyp **float**, der Einheit A und einem Gültigkeitsintervall von [0...10]. Somit ist sowohl der Datentyp (**float**) als auch die Einheit (A) identisch mit dem Ergebnis der Methode ... CalcStromverbrauch() .... Das Gültigkeitsintervall des Ergebnisses des Methodenaufrufs ... GetStromverbrauch() ... ist jedoch kleiner als das Gültigkeitsintervall der Methode ... CalcStromverbrauch() .... Somit liefert dieses erste Teilergebnis eine Warnung – die beiden Intervalle nicht übereinstimmen. Genauso verhält es sich beim zweiten Methodenaufruf der Klasse *RestlicheKomponenten*. Auch hier stimmt wieder der Datentyp und die Einheit überein, während das Gültigkeitsintervall nicht übereinstimmt.

Beim Ergebnis der Addition der beiden Teilergebnisse stimmt wiederum sowohl der Datentyp und die Einheit, während das Gültigkeitsintervall nicht stimmt. Das Ergebnis des Methodenaufrufs ... CalcStromverbrauch() ... produziert demnach die Warnung – das angegebene Gültigkeitsintervall nicht übereinstimmt.

Ergebnis: Der statische projektspezifische Kompatibilitätstest der Klasse *Regelung* hat eine Gültigkeitsverletzung der Intervalle ergeben. Aufgrund der Tatsache, dass dabei kein Datenverlust auftritt, wird lediglich eine Warnung und kein Fehler ausgegeben.

- **Szenario 2:** Überprüfung eines Systems nach der Ersetzung/Austausch einer Komponente (Objekt) des Systems
  Der zweite Anwendungsfall für das objektorientierte statische projektspezifische Kompatibilitätsregelwerk erfolgt nach dem Austausch einer Klasse/Objekts des Systems gegen eine andere[117]. Die Abbildung 2.71 zeigt den Austausch der Klasse *Heizplatte* gegen die neue Klasse *Heizplatte_neu*, die eine leicht veränderte Schnittstelle hat (Bild oben). Im unteren Teil der Abbildung ist das neue Klassenmodell des Systems Kaffeemaschine abgebildet, das nach dem Austausch der Klasse *Heizplatte* entstanden ist. Dieses neu entstandene Klassenmodell soll nun ebenfalls mit Hilfe des zuvor vorgestellten statischen projektspezifischen Kompatibilitätsregelwerks auf Inkompatibilitäten untersucht und bewertet werden. Dabei liegt der Schwerpunkt der Kompatibilitätsuntersuchung auf der Identifikation von Kompatibilitätsfehlern der neuen Klasse mit dem ursprünglichen Modell des Systems.

  Im neuen Klassenmodell des Systems Kaffeemaschine kommuniziert – wie im ursprünglichen Klassenmodell – lediglich die Klasse *Regelung* bzw. die verhaltensbeschreibende Methode ... CalcStromverbrauch() ... mit der neuen Klasse *Heizplatte_neu* bzw. der Zugriffsmethode ... GetStromverbrauch() .... Aus diesem Grund muss hier nur noch dieser Methodenaufruf auf Kompatibilität mit dem restlichen System untersucht werden, weil das übrige Systemmodell bereits untersucht und bewertet wurde.

[117] Siehe hierzu insbesondere das Kapitel „1.4.2 *Kompatibilitätsszenarien*“ ab Seite 26 – Szenario 2: Kompatibilität im Sinne von austauschbar/ersetzbar.

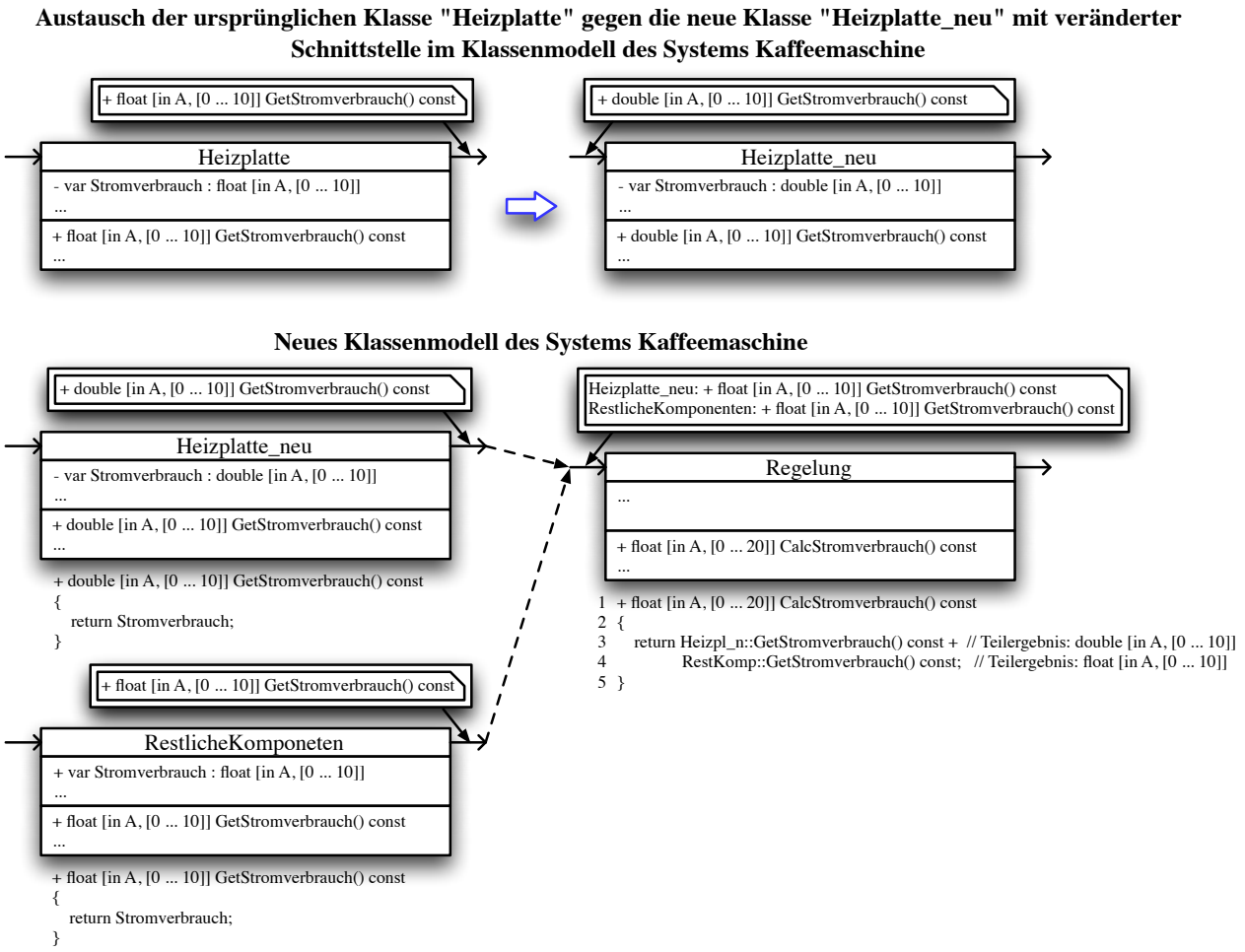


Abbildung 2.71.: Austausch der Klasse *Heizplatte* gegen eine neue mit veränderter Schnittstelle.

Wird das neu entstandene Systemmodell auf *Identität* – Sonderfall der strikten Kompatibilität[118] – hin untersucht, so führt der Methodenaufruf in Zeile 3 der Klasse *Regelung* zu einem Kompatibilitätsfehler, weil die von der Klasse *Regelung* erwartete Signatur der Methode `+ float [in A, [0 ... 10]] GetStromverbrauch() const` mit der Signatur der angebotenen Methode `+ double [in A, [0 ... 10]] GetStromverbrauch() const` der Klasse *Heizplatte_neu* nicht mehr übereinstimmt. Trotzdem kann die neue Komponente *Heizplatte_neu strikt kompatibel* zum ursprünglichen System sein. Genau dann, wenn, wie von der strikten Kompatibilität gefordert, die neue Komponente *Heizplatte_neu* dem System Kaffeemaschine das gleiche Verhalten garantiert, wie dies bereits die ursprüngliche Komponente *Heizplatte* getan hat. Von der neuen Komponente *Heizplatte_neu* wird lediglich das Intervall [0...10] verwendet und nicht der gesamte Wertebereich, den der Datentyp **double** zur Verfügung stellt. Deshalb ist die neue Komponente *Heizplatte_neu* strikt kompatibel zum System, da sie dem System das gleiche Verhalten garantiert, wie dies bereits die ursprüngliche Komponente getan hat. Im Beispiel sind die beiden genutzten Intervalle identisch. Aus diesem Grund ist der Austausch der Komponente *Heizplatte* gegen die neue Komponente *Heizplatte_neu* strikt kompatibel zum System Kaffeemaschine. Wird beispielsweise der gesamte Wertebereich des Datentyps **double** ausgenutzt, so ist die neue Komponente *Heizplatte_neu nicht strikt kompatibel* zum System Kaffeemaschine, weil die neue Komponente ein erweitertes Verhalten besitzt.

Ergebnis: Die neue Klasse *Heizplatte_neu* ist strikt statisch kompatibel zum ursprünglichen System der Kaffeemaschine. Die neue Klasse *Heizplatte_neu* kann im System verwendet werden.

Anmerkung:
Sollte das Ausgangsmodell bereits Kompatibilitätsfehler enthalten, führt die Ersetzung einer Klasse/Objekts im Allgemeinen ebenfalls wieder zu einem Systemmodell mit Kompatibilitätsfehlern. Aus diesem Grund sollte eine Klasse/Objekt nur in einem fehlerfreien Systemmodell ersetzt werden.

□

Überprüfung der dynamischen verhaltensbeschreibenden Methoden eines Objekts/Systems auf Kompatibilität
Nachdem im letzten Schritt die statische Kompatibilität überprüft wurde, folgt nun die Verifikation eines Modells des Systems mit Hilfe des, im Kapitel *„Dynamisches projektspezifisches Regelwerk“* vorgestellten dynamischen Kompatibilitätsregelwerks. Voraussetzung für die dynamische Kompatibilitätsüberprüfung des Modells eines Systems ist dabei, dass sowohl das statische Systemmodell fehlerfrei ist, als auch dass das Modell des Systems ausgeführt (simuliert) werden kann. Sollte das statische Modell Fehler enthalten

[118] Siehe hierzu insbesondere die Definition 1.16 und den Satz 1.1.

oder nicht ausgeführt/simuliert werden können, so kann die dynamische Kompatibilitätsprüfung nicht ausgeführt werden (siehe hierzu die Abbildung 2.68 auf Seite 95). Eine weitere Voraussetzung, um die dynamische Kompatibilität eines Modells verifizieren zu können, ist die Existenz eines dynamischen Modells, in dem das Verhalten der Klassen/Objekte bzw. die Verhaltensbeschreibung der Klassen/Objekte des Systems hinterlegt ist. Sind beide Anforderungen erfüllt, kann mit der Durchführung des dynamischen Kompatibilitätstests begonnen werden.

Anmerkung:
Die theoretischen Grundlagen für die dynamische Kompatibilitätsmodellierung und -bewertung wurden von D. Koß in ihrer Dissertation [Koß09] bzw. den beiden Diplomarbeiten von C. Eckl [Eck07] und J. Winkler [Win08] beschrieben und werden aus diesem Grund hier nicht aufgeführt. Um die dynamische Kompatibilitätsprüfung dennoch zu demonstrieren, wird wieder das Beispielsystem Kaffeemaschine herangezogen.

Beispiel 33: Überprüfung des Beispielsystems Kaffeemaschine anhand des dynamischen projektspezifischen Regelwerks
Wie bereits bei der statischen Kompatibilitätsprüfung des Systemmodells der Kaffeemaschine mit dem statischen projektspezifischen Kompatibilitätsregelwerk, muss auch bei der Untersuchung des Systemmodells auf dynamische Kompatibilität wieder zwischen zwei unterschiedlichen Szenarien unterschieden werden.

- **Szenario 1:** Verifikation eines neu entwickelten Systems anhand des projektspezifischen Kompatibilitätsregelwerks
  Um die dynamische Kompatibilität des Systems Kaffeemaschine anhand des dynamischen projektspezifischen Regelwerks überprüfen zu können, muss wie oben erwähnt, das dynamische Verhalten im Modell hinterlegt sein und das Modell ausgeführt/simuliert werden können. Diese beiden Voraussetzungen sind für das Modell der Kaffeemaschine erfüllt. Das Verhalten der Klassen wurde zum Teil mittels der verhaltensbeschreibenden Methoden und teilweise mit Hilfe der MSCs beschrieben. Aufbauend auf dem dynamischen Systemmodell wird nun die dynamische Kompatibilitätsprüfung durchgeführt. Grundlage für die dynamische Kompatibilitätsmodellierung ist auch hier wieder das modellierte dynamische Verhalten der einzelnen Klassen/Objekte des Systems.

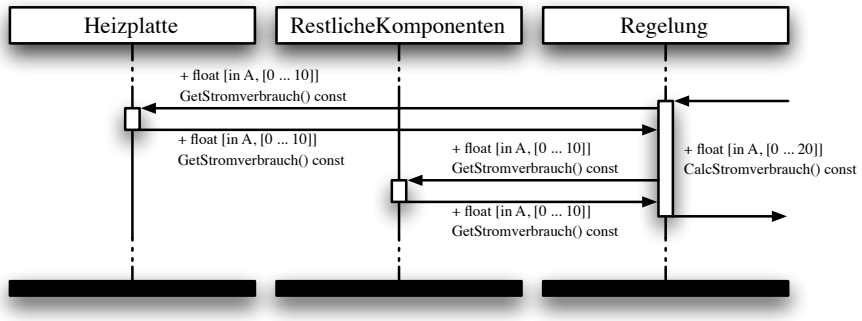


Abbildung 2.72.: Anwendung des dynamischen projektspezifischen Kompatibilitätstests auf das Beispielsystem Kaffeemaschine.

Die Abbildung 2.72 zeigt das MSC mit dessen Hilfe die Kommunikation (d.h. das dynamische Verhalten) der Objekte *Regelung*, *Heizplatte* und *RestlicheKomponenten* des Systems Kaffeemaschine modelliert ist. Dabei stimmt das modellierte Verhalten mit dem statischen Verhalten in der Abbildung 2.70 in Zeile 3 und dem modellierten Einzelverhalten der Klassen/Objekte überein. Alle Nachrichten des Beispielmodells sind in der richtigen Reihenfolge angeordnet. Ob das modellierte Verhalten dem tatsächlichen Verhalten des Systems entspricht wird mittels der Ausführung/Simulation des Modells überprüft.

Ergebnis: Das modellierte Teilsystem ist kompatibel im Sinne der dynamischen Kompatibilitätsprüfung.

- **Szenario 2:** Überprüfung eines Systems nach der Ersetzung/Austausch einer Komponente (Objekt) des Systems
  Wie bereits im statischen Systemmodell (Abbildung 2.71) wird nun die Klasse *Regelung* gegen die neue Klasse *Regelung_neu2* mit modifiziertem Verhalten ersetzt. Die Abbildung 2.73 auf der nächsten Seite zeigt das entsprechende MSC.

  Die neue Klasse *Regelung_neu2* hat gegenüber der ursprünglichen Klasse/Objekt *Regelung* ein modifiziertes erweitertes Verhalten. Die neue Regelung kann die Signale in der ursprünglichen Reihenfolge empfangen oder in der umgekehrten Reihenfolge. Dies wird innerhalb des MSCs durch den Block *alt* symbolisiert. Für die Kompatibilitätsbestimmung bedeutet dies, dass die neue Komponente nicht *strikt kompatibel* zum ursprünglichen Systemmodell ist, da sich das Verhalten der neuen Komponente von dem der alten unterscheidet. Jedoch ist die neue Komponente *Regelung_neu2 nicht strikt kompatibel* zum ursprünglichen Modell, weil eine der beiden Alternativen der neuen Komponente *Regelung_neu2* dem ursprünglichen Verhalten der Komponente *Regelung* entspricht.

  Ergebnis: Der Austausch der Komponente *Regelung* gegen die neue Komponente *Regelung_neu2* führt zu einer Verletzung der *strikten Kompatibilität* zwischen den beiden Regelungen bzw. der Regelung mit der Umwelt. Aufgrund der Tatsache, dass das Verhalten der neuen Regelung *Regelung_neu2* durch die erste Alternative der ursprünglichen Regelung entspricht, ist das Systemmodell dennoch kompatibel im Sinne der *nicht strikten Kompatibilitätsdefinition*.

□

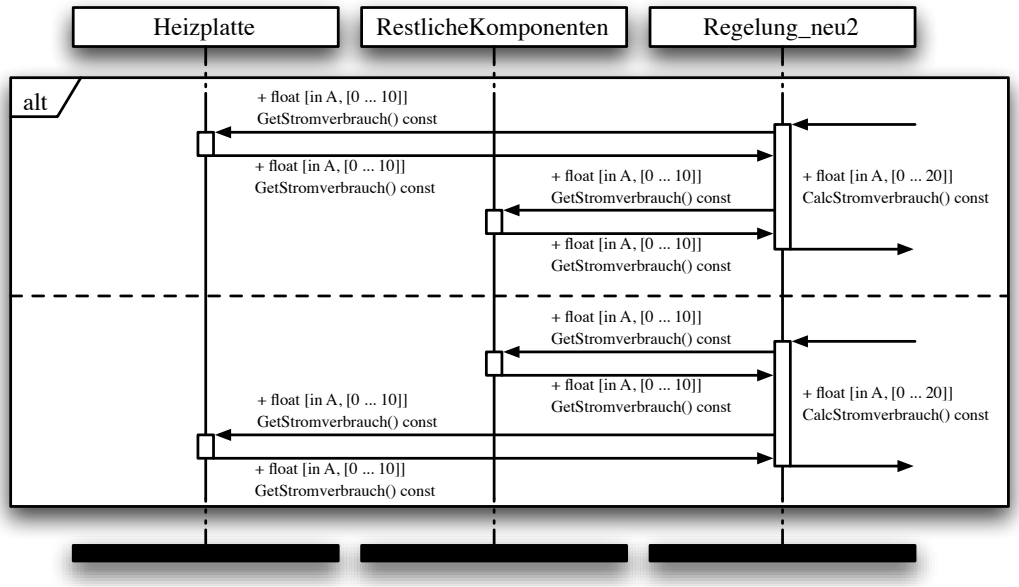


Abbildung 2.73.: Anwendung des dynamischen projektspezifischen Kompatibilitätstests auf das Beispielsystem Kaffeemaschine nach dem Austausch der Klasse/Objekt *Regelung*.

2.5.3. Bewertung der Ergebnisse der objektorientierten Kompatibilitätsverifikation

Nachdem die unterschiedlichen Phasen der objektorientierten Kompatibilitätsmodellierung durchlaufen und die daran anschließende Kompatibilitätsverifikation abgeschlossen wurde, müssen die Ergebnisse des Kompatibilitätstest ausgewertet und gegebenenfalls geeignete Gegenmaßnahmen ergriffen werden, sofern eine Inkompatibilität im Modell festgestellt worden ist. Bereits im Kapitel „*Was hat Systems Engineering mit modellbasierter Kompatibilitätsbewertung zu tun?*“ wurde der DIBMUK-Kompatibilitätssicherungsprozess vorgestellt[119], mit dessen Hilfe die Kompatibilität eines Systems bewertet sowie geeignete Maßnahmen zur Kompatibilitätssicherung ergriffen werden können. Der DIBMUK-Kompatibilitätssicherungsprozess zielt dabei insbesondere auf die Einhaltung der Kompatibilität beim Austausch einer Komponente[120] in einem System gegen eine neue. Er kann jedoch, mit geringen Anpassungen, auch während der im Kapitel „*Modellbildung*“ beschriebenen objektorientierten Systementwicklung eingesetzt werden um die Kompatibilität während der Entwicklung des Systemmodells sicherzustellen.

In diesem Kapitel wird nun aufbauend auf dem DIBMUK-Prozess sowie den Ergebnissen der objektorientierten Kompatibilitätsverifikation die Auswertung der Ergebnisse der Kompatibilitätsverifikation des Systemmodells beschrieben. Jedoch bevor mit der eigentlichen Auswertung der Kompatibilitätsergebnisse begonnen werden kann, wird zunächst der bereits vorgestellte DIBMUK-Kompatibilitätssicherungsprozess an die Bedürfnisse der Kompatibilitätsbewertung angepasst.

Anpassung des allgemeinen Kompatibilitätssicherungsprozesses DIBMUK an die Bedürfnisse der objektorientierten Kompatibilitätsbestimmung
Der allgemeine Kompatibilitätssicherungsprozess DIBMUK wurde von F. Bornemann speziell für die Sicherung und Wahrung der Kompatibilität eines Systems entwickelt, das bereits vollständig modelliert vorliegt. Dabei wird ebenfalls vorausgesetzt, dass das Modell des Systems keine syntaktischen Sprachfehler enthält, also wohlgeformt ist. Um den DIBMUK-Kompatibilitätssicherungsprozess auch während der objektorientierten Modellbildungsphase einsetzen zu können, muss dieser leicht angepasst und erweitert werden. Die Abbildung 2.74 auf der nächsten Seite zeigt die dazu notwendigen Anpassungen in tabellarischer Form.

In der ersten Phase des erweiterten DIBMUK-Kompatibilitätssicherungsprozesses – der Definitionsphase – werden die Grundregeln definiert, nach denen das zu verifizierende Systemmodell auf Kompatibilität bzw. auf syntaktische Sprachfehler untersucht werden soll. Aufbauend auf den dort hinterlegten Sprach- bzw. Kompatibilitätsregeln wird das Modell des Systems in der Identifikationsphase auf Kompatibilitäts- bzw. Syntaxfehler untersucht. Treten während der Identifikationsphase Fehler auf, so wird – in Abhängigkeit von der Fehlerklasse (Syntaxfehler oder Kompatibilitätsfehler) – der aufgetretene Fehler nach seiner Wichtigkeit bewertet. In der darauf folgenden Maßnahmenphase werden geeignete Maßnahmen identifiziert, mit deren Hilfe das aufgetretene Kompatibilitätsproblem bzw. die Syntaxverletzung beseitigt werden kann. In der Umsetzungsphase werden dann die beschlossenen Maßnahmen umgesetzt und anschließend in der Kontrollphase das Ergebnis der Umsetzung kontrolliert. Ist das Systemmodell nun fehlerfrei, so wird der erweiterte DIBMUK-Prozess abgeschlossen. Ansonsten wird zur Phase Maßnahmen zurückgesprungen, um die noch vorhandene Kompatibilitätsverletzung zu beseitigen.

Der hier vorgestellte erweiterte DIBMUK-Prozess bildet die Grundlage für die nun folgende Beschreibung der objektorientierten Kompatibilitätsbewertung.

[119] Siehe hierzu die Beschreibung des DIBMUK-Kompatibilitätssicherungsprozessbeschreibung „1.2 *Was hat Systems Engineering mit modellbasierter Kompatibilitätsbewertung zu tun?*“ ab Seite 7.
[120] Siehe hierzu das Kapitel „1.4.2 *Kompatibilitätsszenarien*“ ab Seite 26 und insbesondere das dort beschriebene Szenario 2.

DIBMUK-Phasen	Objektorientierte Modellbildung	Auswertung der Ergebnisse des Kompatibilitätstests
Definition	Definition der projektunabhängigen Sprachregeln	Definition und Festlegung der projektspezifischen Kompatibilitätsregeln
Identifikation	Identifikation der Sprachregelwerksverletzungen	Identifikation der Kompatibilitätsfehler und Warnungen
Bewertung	Bewertung der Sprachregelwerksverletzungen *Kritisch:* Fehler: Maßnahmen zur sofortigen Beseitigung einleiten Warnung: Kann ignoriert werden (Entscheidung protokollieren) *Nicht kritisch:* Fehler: Kann ignoriert werden (Entscheidung protokollieren) Warnung: Kann ignoriert werden (Entscheidung protokollieren)	Bewertung der Kompatibilitätsverletzung *Kritisch:* Fehler: Maßnahmen zur sofortigen Beseitigung einleiten Warnung: Kann ignoriert werden (Entscheidung protokollieren) *Nicht kritisch:* Fehler: Maßnahmen zur späteren Beseitigung einleiten (Entscheidung protokollieren) Warnung: Kann ignoriert werden (Entscheidung protokollieren)
Maßnahmen	Kritische Sprachregelwerksverletzungen beseitigen	Kompatibilitätsfehler oder Warnung: *Kritisch:* Maßnahmen zur sofortigen Beseitigung ergreifen *Nicht Kritisch:* Kann ignoriert werden (Entscheidung protokollieren)
Umsetzung	Umsetzung der zuvor festgelegten Maßnahmen	
Kontrolle	Erfolgskontrolle der zuvor festgelegten Maßnahmen. Sollte der Fehler/Warnung nicht beseitigt worden sein, Rücksprung zur Phase Maßnahmen	

Abbildung 2.74.: Anpassung des DIBMUK-Kompatibilitätssicherungsprozesses an die speziellen Anforderungen der objektorientierten Kompatibilitätsmodellierung.

Ablauf der objektorientierten Kompatibilitätsbewertung eines Systemmodells

Nachdem im vorangegangenen Abschnitt dieses Kapitels die Erweiterung des allgemeinen DIBMUK-Prozesses vorgestellt wurde, folgt nun die Beschreibung der Ablaufreihenfolge bei der Bewertung der Kompatibilitätsergebnisse. Die Abbildung 2.75 illustriert beispielhaft die Ablaufreihenfolge der Bewertung der Kompatibilität eines Systemmodells.

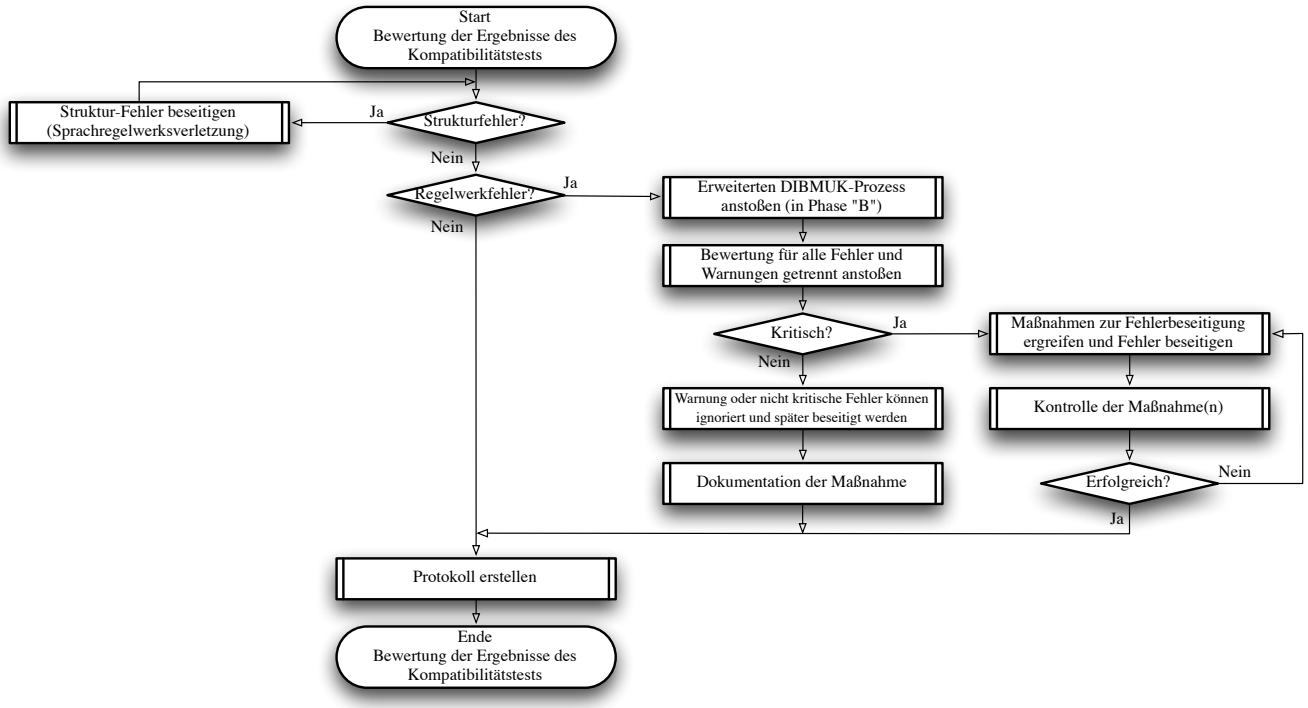


Abbildung 2.75.: Ablauf der Kompatibilitätsbewertung eines Systemmodells.

Dabei wird wieder zwischen einer Sprachregelwerksverletzung und der Verletzung des projektspezifischen Regelwerks unterschieden. Sollte das Modell des Systems eine Verletzung des Sprachregelwerks enthalten, so muss dieser Fehler beseitigt werden bevor mit der eigentlichen projektspezifischen Kompatibilitätsbewertung begonnen werden kann. Trat während der Überprüfung des Systemmodells anhand des projektspezifischen Regelwerks eine Warnung oder ein Fehler auf, so wird der erweiterte DIBMUK-Prozess gestartet, ansonsten wird das Prüfungsergebnis protokolliert und der Bewertungsprozess abgeschlossen. Im Fehler- und Warnungsfall wird der erweiterte DIBMUK-Prozess mit der separaten Bewertung aller aufgetretenen Warnungen und Fehler angestoßen. Dabei gilt: Für alle kritischen Kompatibilitätsfehler müssen geeignete Maßnahmen ergriffen werden um diese zu beseitigen. Daran anschließend werden die Warnungen und Fehler beseitigt und der Erfolg der Beseitigung kontrolliert. War die Beseitigung erfolgreich, so wird dies im Auswertungsprotokoll dokumentiert. Sollte die Warnung- oder Fehlerbeseitigung erfolglos sein, müssen erneut geeignete Maßnahmen zur Fehlerbeseitigung ergriffen werden. Diese Schleife wird so lange wiederholt, bis kein Kompatibilitätsfehler mehr auftritt.

Ergibt die Bewertung der Warnungen und Fehler, dass diese nicht kritisch für die Kompatibilität des Systems sind, so können sie ignoriert und zu einem späteren Zeitpunkt behandelt werden. Die Entscheidung, eine Warnung oder eine nicht kritische

Kompatibilitätsverletzung zu ignorieren wird in einem speziellen Protokoll dokumentiert, so dass diese Entscheidung jederzeit nachgeschlagen werden kann, falls sich die Entscheidung den Fehler bzw. die Kompatibilitätsverletzung als nicht kritisch einzustufen als Fehler herausstellt und diese nun berichtigt werden muss.

Behandlung von Warnungen und Fehlern, die während der Überprüfung des Systemmodells anhand des projektunabhängigen Sprachregelwerks auftreten
Sämtliche Warnungen und Fehler dieser Klasse können lediglich während der *objektorientierten Modellbildungsphasen*[121] bzw. während des Austauschs einer Komponente des Systems gegen eine neue (Szenario 2) auftreten, weil nach Abschluss der Modellbildungsphase das fertige, wohlgeformte Systemmodell vorliegt, das keine Sprachregelwerksverletzungen mehr enthalten darf. Alle Fehler und Warnungen, die während der objektorientierten Modellbildungsphase auftreten, werden sofort beseitigt.

Wird während der objektorientierten Modellbildungsphase bzw. nach dem Austausch einer Komponente des Systems gegen eine neue die Kompatibilitätsüberprüfung anhand des objektorientierten projektunabhängigen Sprachregelwerks angestoßen, so kann es vorkommen, dass das Modell Sprachregelwerksverletzungen enthält, da das Modell noch unvollständig bzw. Teile des Modells in unterschiedlichen Versionen vorliegen. In diesem Fall kann das projektspezifische Regelwerk dazu benutzt werden, das Modell des Systems fehlerfrei (im Sinne von Syntaxfehlerfreiheit) aufzustellen. Die syntaktische Fehlerfreiheit ist die Voraussetzung für die anschließende Durchführung des projektspezifischen Kompatibilitätstests.

Behandlung von Warnungen und Fehlern, die während der Überprüfung des Systemmodells anhand des projektspezifischen Regelwerks auftreten
Nachdem das Systemmodell vollständig und wohlgeformt vorliegt, wird es mit Hilfe des projektspezifischen Kompatibilitätsregelwerks auf mögliche Kompatibilitätsfehler untersucht. Bei der Verifikation des Systemmodells anhand des projektspezifischen Kompatibilitätsregelwerks können dabei sowohl Kompatibilitätsfehler als auch Warnungen auftreten. Dabei gilt: Im Allgemeinen führen alle Fehler im Modell unmittelbar zu einer Inkompatibilität des Systems und müssen beseitigt werden. Um die Beseitigung der Kompatibilitätsfehler zu initiieren, wird der erweiterte DIBMUK-Kompatibilitätssicherungsprozess in der Bewertungsphase mit sämtlichen zuvor identifizierten Kompatibilitätsfehlern angestoßen[122]. Die aufgetretenen Kompatibilitätsverletzungen werden einzeln, zum einen nach ihren möglichen Auswirkungen auf das System, zum anderen nach ihrer Wichtigkeit bewertet, geeignete Maßnahmen ergriffen um die Kompatibilitätsfehler zu beseitigen und im Anschluss daran die Umsetzung der Fehlerbeseitigung eingeleitet. Abgeschlossen wird der DIBMUK-Prozess mit der Kontrolle und Dokumentation der zuvor eingeleiteten Umsetzungsmaßnahmen zur Beseitigung der Inkompatibilität sowie der Erfolgskontrolle, ob die Beseitigungsmaßnahmen erfolgreich waren und somit der/die zuvor identifizierten Kompatibilitätsfehler im Systemmodell nicht mehr vorhanden ist/sind.

Ebenso wie bei der Behandlung von Kompatibilitätsfehlern wird mit den aufgetretenen Warnungen vorgegangen. Im Unterschied zu Kompatibilitätsfehlern, die beseitigt werden müssen, kann in der Bewertungsphase einer Warnung entschieden werden, dass die aufgetretene Warnung ignoriert wird, da daraus keine Kompatibilitätsprobleme resultieren. In diesem Fall wird die Entscheidung dokumentiert und der DIBMUK-Prozess an dieser Stelle abgebrochen.

Bewertung der Kompatibilitätsergebnisse des Beispielsystems Kaffeemaschine
Bei der objektorientierten Modellierung der Klasse *Regelung* wurde gegen die formale Schreibweise einer Methodendeklaration verstoßen. Diese Verletzung der formalen Syntax wurde bei der Durchführung des Kompatibilitätstests identifiziert und soll nun mit Hilfe des in Abbildung 2.75 vorgestellten Algorithmus bewertet werden. Die Abbildung 2.76 zeigt den hierfür relevanten Auszug aus dem Modell der Klasse *Regelung*.

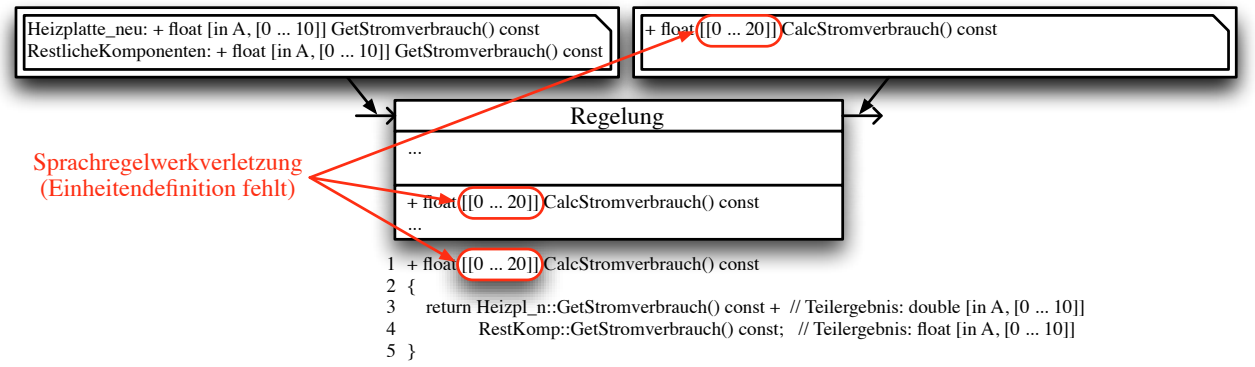


Abbildung 2.76.: Sprachregelwerksverletzung bei der Modellierung der Klasse Regelung.

Dabei fällt auf, dass bei der Methodendeklaration der Methode `+ float [[0 ... 20]] CalcStromverbrauch() const` eine Verletzung der formalen Syntax auftritt. Der Ergebniswert besitzt keine Angabe über eine Einheit. Dies ist jedoch laut der formalen Syntax für die Methodendeklaration nicht erlaubt[123].

[121] Siehe hierzu insbesondere das Grundlagenkapitel „2.2.1 *Unterschiedliche Modellbildungsparadigmen*“ ab Seite 38, sowie das Kapitel „2.3.1 *Grundkonzepte der objektorientierten Modellbildung*“ ab Seite 46.
[122] Anmerkung: Für alle identifizierten Kompatibilitätsfehler, die durch die Verifikation des Systemmodells mit Hilfe des projektspezifischen Regelwerks identifiziert wurden, wird ein eigener DIBMUK-Prozess gestartet und vollständig abgearbeitet.
[123] Siehe hierzu die Definition 2.21 auf Seite 77.

Der oben vorgestellte Algorithmus zur Bewertung der Kompatibilitätsergebnisse liefert in diesem Fall das Ergebnis: „Die Methode `+ float [[0 ... 20]] CalcStromverbrauch() const` enthält eine Verletzung des formalen Sprachregelwerks. Der Rückgabewert der Methode enthält keine Angabe über die verwendete Einheit.“. Aufgrund der Tatsache, dass es sich um eine Verletzung des Sprachregelwerks handelt, muss diese umgehend beseitigt werden, bevor mit der eigentlichen Kompatibilitätsbewertung begonnen werden kann. Nachdem der Syntaxfehler beseitigt worden ist, kann mit der Durchführung des eigentlichen Kompatibilitätstests begonnen werden.

2.6. Domänenübergreifendes integriertes Systemmodell

Das Systems Engineering propagiert, wie im Kapitel *„Was hat Systems Engineering mit modellbasierter Kompatibilitätsbewertung zu tun?“* gezeigt, die „ganzheitliche modellbasierte Systementwicklung“ angefangen bei der Planung (Phase 0[124]) bis hin zur Entsorgung (Phase F) des Systems. Dabei spielt das ganzheitliche modellbasierte Systementwicklungsparadigma besonders während der Planungs- und Entwicklungsphasen (0-D) des Systems eine besondere Rolle. In dieser bedeutsamen Phase wird das Modell des Systems entworfen, geplant und schließlich das abstrakte Systemmodell in ein konkretes System überführt und das System gefertigt. Um diese vier Phasen der Systementwicklung möglichst effizient unterstützen zu können, untermauert T. Weilkiens in seinem Buch „Systems Engineering mit SysML/UML“ [Wei06] den zentralen domänenübergreifenden Denkansatz des Systems Engineerings und insbesondere die immer wichtiger werdende domänenübergreifende Systementwicklung. In diesem Abschnitt werden nun die grundsätzlichen Thesen des ganzheitlichen modellbasierten Systems Engineerings noch einmal aufgegriffen und ihre Vorteile für die objektorientierte Modellbildung und die Kompatibilitätsbestimmung und -bewertung von eingebetteten Systemen aufgezeigt und erläutert.

Wie bereits mehrfach erwähnt, sind an der Entwicklung eines eingebetteten softwarelastigen Systems mindestens Ingenieure der drei Domänen Elektrotechnik, Mechanik und Software beteiligt. Um beispielsweise ein eingebettetes System wie die Kaffeemaschine zu entwickeln, sind Ingenieure mindestens aus den drei zuvor erwähnten Fachbereichen beteiligt. Um das Produkt (z.B. eine Kaffeemaschine) effizient und vor allem die Einzelteile des Systems kompatibel zueinander zu entwickeln, kann auf zwei unterschiedliche Art und Weisen vorgegangen werden.

Klassisches Vorgehen bei der Systementwicklung
Bei der klassischen Systementwicklung werden im Allgemeinen die Bestandteile des Systems getrennt voneinander gefertigt und anschließend zu einem System integriert. Dabei kommt es jedoch immer wieder zu erheblichen Kompatibilitätsproblemen, weil beispielsweise zuvor festgelegte Schnittstellen nicht eingehalten oder einseitig, ohne die Zustimmung der anderen an der Systementwicklung beteiligten Bereiche, geändert worden sind. Die Abbildung 2.77 zeigt das Vorgehen bei der klassischen Systementwicklung.

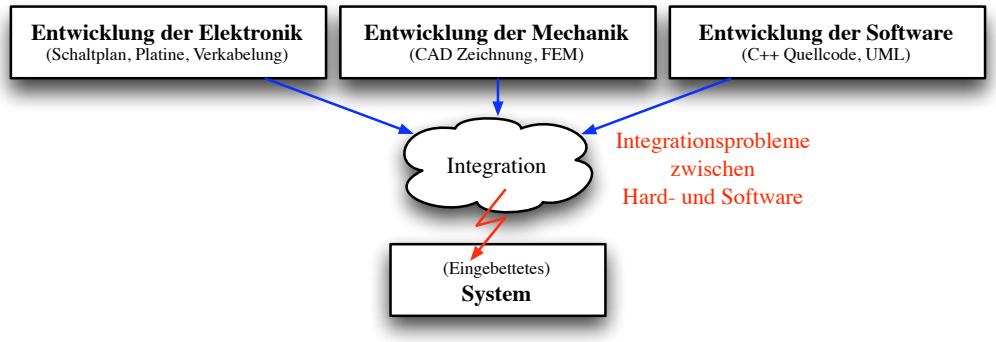


Abbildung 2.77.: Klassisches Vorgehen bei der Systementwicklung.

Dabei entwickeln die drei an der Systementwicklung beteiligten Fachbereiche jeweils einzeln und getrennt voneinander die später zu integrierenden Systembestandteile. Bei der späteren Integration der einzelnen Bestandteile des Systems kommt es dann in den meisten Fällen zu erheblichen Kompatibilitätsproblemen. Dies liegt zum einen daran, dass innerhalb einer Domäne unterschiedliche Modelle verwendet werden, die von den anderen an der Systementwicklung beteiligten Gruppen nicht gelesen und verstanden werden können. So kann es sehr schnell zu Dissonanzen während der Systementwicklung kommen. Ein weiteres Problem bei der klassischen Systementwicklung liegt in der verteilten Entwicklung. Aufgrund der Tatsache, dass in den meisten Fällen jede Systemkomponente von einem anderen Zulieferer stammt und diese „nur“ zum eigentlichen System zusammengebaut werden müssen. Die einzelnen Zulieferer wissen meistens nichts voneinander, so dass sich mögliche Änderungen in der Spezifikation einer Systemkomponente erst bei der Integration des Systems bemerkbar machen, wodurch erhebliche Folgekosten entstehen, um diese Inkompatibilitäten zu beseitigen.

Im Gegensatz dazu verfolgt das Systems Engineering den Ansatz, alle an der Entwicklung eines Systems beteiligten Ingenieursgruppen dazu zu ermutigen, ein gemeinsames integriertes Modell des zu entwickelnden Systems zu verwenden, um das Kompatibilitätsproblem bei der Integration der unterschiedlichen Systembestandteile zu verringern bzw. ganz zu vermeiden.

[124] Phasenbezeichnungen nach dem ECSS Vorgehensmodell.

Domänenübergreifendes integriertes Vorgehen bei der Systementwicklung
Beim domänenübergreifenden integrierten Vorgehen bei der Systementwicklung wird im Gegensatz zum klassischen Ansatz mit nur einem, für alle beteiligten Ingenieursgruppen verständlichen, Systemmodell gearbeitet. Dieses Systemmodell enthält sämtliche für das Verständnis bzw. die Kompatibilität des Systems notwendigen Daten und Informationen. Dazu zählt unter anderem eine eineindeutige Schnittstellendefinition, die für alle an der Entwicklung des Systems beteiligten Ingenieursgruppen bindend ist. Des Weiteren sind im gemeinsamen Systemmodell sämtliche Daten und Informationen so hinterlegt, dass sie von allen an der Entwicklung beteiligten Gruppen gleichermaßen verstanden werden. Dazu ist eine gemeinsame formale Sprache notwendig, in der sämtliche für die Systementwicklung relevanten Daten hinterlegt werden können. Dies gilt insbesondere für Daten, die für die Kompatibilität des zu entwickelnden Systems relevant sind. Die Abbildung 2.78 zeigt beispielhaft das Vorgehen bei der Verwendung eines domänenübergreifenden Systemmodells.

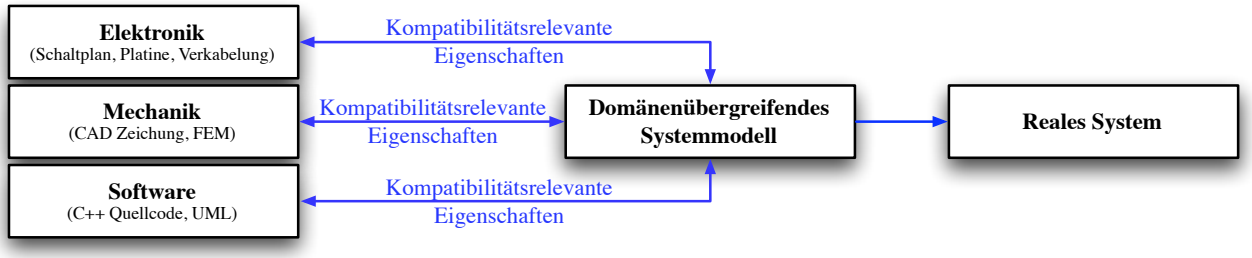


Abbildung 2.78.: Integriertes domänenübergreifendes Systemmodell.

Den zentralen Kern der domänenübergreifenden Systementwicklung bildet das domänenübergreifende Systemmodell (Bildmitte). In diesem Modell befinden sich sämtliche kompatibilitätsrelevanten Eigenschaften aus den an der Systementwicklung beteiligten Fachbereichen. Dabei ist das Modell in einer für alle verständlichen, eineindeutigen und formalen Notation dargestellt. Zusätzlich zu den kompatibilitätsrelevanten Eigenschaften des Systems sind im gemeinsamen Datenmodell die Schnittstellen zwischen den einzelnen Komponenten des Systems exakt beschrieben. Sollte beispielsweise ein Ingenieur-Team eine Schnittstelle ändern, so würden alle an der Systementwicklung beteiligten Teams dies unverzüglich mitbekommen, weil sie ebenfalls das gemeinsame Modell für ihre Entwicklung verwenden. Das gleiche gilt für sämtliche im Datenmodell enthaltenen Daten. Auch diese sind allen an der Systementwicklung beteiligten Ingenieur-Teams gleichermaßen zugänglich. Aus diesem Grund sollten nur solche Daten und Informationen im zentralen domänenübergreifenden Systemmodell abgelegt werden, die für alle Ingenieur-Teams relevant sind, um das Modell nicht zu überfrachten. Um das domänenübergreifende Systemmodell zu realisieren, können die im Kapitel „*Modellbildung*“ vorgestellte objektorientierte Modellbildungsparadigma bzw. die im Kapitel „*Modellierungssprachen und -techniken für technische Systeme*“ vorgestellten Modellierungssprachen verwendet werden. Insbesondere verfolgen die beiden Modellierungssprachen SysML bzw. die (U)CML das domänenübergreifende Modellbildungsparadigma zur Erstellung eines Systems, das das Systems Engineering vorschlägt.

Problematisch bei der Nutzung eines gemeinsamen domänenübergreifenden Systemmodells ist jedoch der Mehraufwand, der durch die Transformation und Synchronisation der ursprünglichen Entwicklungsdaten mit dem gemeinsamen Modell entsteht. Dieser Mehraufwand wird jedoch durch den wesentlich größeren Nutzen gerechtfertigt, da auf diese Weise „fast keine Kompatibilitätsprobleme“ mehr auftreten, und sich somit der Mehraufwand gegenüber dem klassischen Vorgehen rechnet.

Anmerkung:
Nähere Informationen zum domänenübergreifenden integrierten Systemmodellansatz finden Sie unter [BE08] und [Eck12].

2.7. Objektorientierte Modellierung kompatibilitätsrelevanter Eigenschaften von eingebetteten Systemen

Nachdem in den vorangegangenen Kapiteln die Grundlagen für die objektorientierte (Kompatibilitäts-) Modellbildung sowie die Bewertung der Kompatibilität von technischen Systeme vorgestellt wurde, folgt nun eine knappe Einführung in die grundlegende Modellierungstechnik von kompatibilitätsrelevanten Eigenschaften technischer Systembestandteile, wie beispielsweise von elektrischen Signalen, mechanischen Größen oder der Modellierung von Softwaredaten. Dabei werden in diesem Kapitel aus jedem der drei Fachbereiche – Elektrotechnik, Mechanik und Software – jeweils ein Beispiel exemplarisch für diese Klasse herausgegriffen und sowohl mit Hilfe der im Kapitel „*Erweiterung des objektorientierten Modellbildungsparadigmas auf die objektorientierte Kompatibilitätsmodellierung und -bestimmung*“ eingeführten formalen Notation als auch der dort beschriebenen graphischen Modellierungssprache modelliert. Begonnen wird mit der (Kompatibilitäts-) Modellierung eines elektrischen Signals[125].

Kompatibilitätsmodellierung von elektrischen Signalen
Um die während der objektorientierten Modellbildung, genauer während der Identifikationsphase der kompatibilitätsrelevanten Eigenschaften eines Systems, identifizierten elektrischen Systemeigenschaften zu modellieren, wird das grundsätzliche Vorgehen exemplarisch anhand einer Rechteckschwingung erläutert. Ausgangsbasis ist die technische Beschreibung des Signals,

[125] Anmerkung:
Die in diesem Kapitel vorgestellte grundlegende Modellierungstechnik für kompatibilitätsrelevante Systemeigenschaften wird insbesondere in den beiden Kapiteln „*3 Modellierungssprachen und -techniken für technische Systeme*“ ab Seite 109 sowie „*4 Modellierung der Fallstudie – Gleitschutzsystem – in (U)CML*“ ab Seite 263 angewendet.

so wie sie in einer Anleitung oder technischen Spezifikation enthalten ist. Die Abbildung 2.79 zeigt auf der linken Seite oben das idealisierte Modell[126] einer Rechteckschwingung mit einer Frequenz von 1Hz und einer Amplitude von $+1.0$V mit einer maximalen Toleranz von ± 125mV, die zum Zeitpunkt $t = 0$s mit der positiven Halbwelle beginnt. Dieses ideale Modell der Rechteckschwingung ist links unten noch einmal mit den in der Legende der (technischen Spezifikation) definierten Grenzen graphisch dargestellt. Dabei wurde die zulässige Abweichung des Signals mit Hilfe der bereits in der Abbildung 2.60 auf Seite 89 eingeführten ϵ-Umgebung modelliert.

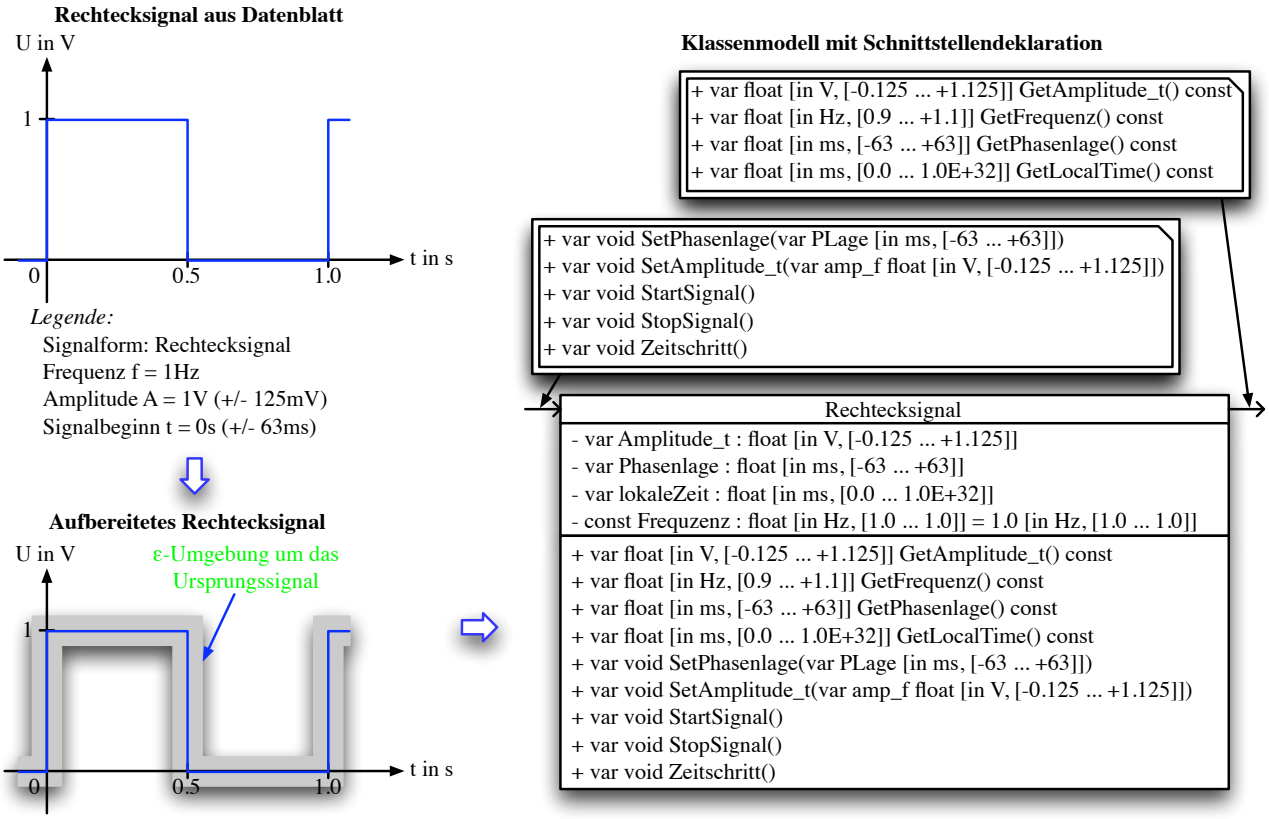


Abbildung 2.79.: Modellierung der kompatibilitätsrelevanten Eigenschaften einer elektrischen Schwingung für die Kompatibilitätsbestimmung.

Auf der rechten Seite der Abbildung ist das für die Kompatibilitätsmodellierung und -bestimmung erweiterte Klassenmodell der Rechteckschwingung inklusive der Schnittstellenbeschreibung dargestellt. Im Klassenmodell sind die drei, das Rechtecksignal charakterisierenden Eigenschaften – Amplitude, Phasenlage und Frequenz – , dargestellt und mit den entsprechenden Grenzen modelliert. Um das Verhalten des Rechtecksignals über die Zeit beschreiben zu können, ist zusätzlich die Definition einer lokalen Zeit notwendig. Die lokale Zeit wird mit Hilfe der privaten Eigenschaft `- var lokaleZeit : float [in ms, [0.0 ... 1.0E+32]]` beschrieben. Mit dieser Definition wird vereinbart, dass die kontinuierliche Zeit mit Hilfe von Zeitscheiben mit einer maximalen Genauigkeit von 1ms angegeben werden kann.

Zusätzlich zur Modellierung der Eigenschaften des Rechtecksignals sind in der Klassenbeschreibung sechs Methoden angegeben, mit deren Hilfe auf die internen (privaten) Eigenschaften der Klasse *Rechtecksignal* zugegriffen bzw. diese manipuliert werden können. Des Weiteren enthält die Klasse *Rechtecksignal* zwei Methoden, um das zeitabhängige Verhalten des Rechtecksignals beschreiben zu können. Mit Hilfe der öffentlichen Methode `+ var void StartSignal()` wird das Signal gestartet, während mit Hilfe der Methode `+ var void StopSignal()` die Ausführung beendet werden kann. Dabei wird nach dem Aufruf der Methode `... StartSignal() ...` bei jedem weiteren Aufruf der Methode `... Zeitschritt() ...` die interne lokale Zeit (`- var lokaleZeit : float [in ms, [0.0 ... 1.0E+32]]`) der Klasse um genau einen Zeitschritt von 1ms erhöht. Dies geschieht solange der maximale Gültigkeitsbereich der lokalen Zeit der Klasse nicht überschritten wird. Dabei kann der aktuelle Amplitudenwert des Rechtecksignals mit Hilfe der Methode `... GetAmplitude_t() ...` zu jedem Zeitpunkt (genauer: nach jedem Zeitschritt) abgefragt werden.

Kompatibilitätsmodellierung von mechanischen Größen

Ebenso wie bei der Modellierung von elektrischen Signalen kann bei der Modellierung von mechanischen Größen vorgegangen werden. Auch hier werden die während der objektorientierten Identifikationsphase der kompatibilitätsrelevanten Eigenschaften des Systems identifizierten mechanischen Eigenschaften mit Hilfe der zuvor eingeführten Modellierungsmethode in einem Modell abgelegt. Die Abbildung 2.80 auf der nächsten Seite zeigt links die CAD Zeichnung des mechanischen Bauteils Durchbruch. In der Zeichnung sind sämtliche Werte inklusive der zulässigen Toleranzen eingetragen. Diese in der CAD Zeichnung enthaltenen Daten werden nun in das Modell überführt und dabei so aufbereitet, dass sie für die Kompatibilitätsbestimmung verwendet werden können.

[126] Siehe hierzu das Kapitel „F.1 *Modellierung von elektrotechnischen Signalen*“ ab Seite 341 und insbesondere die Abbildung F.2.

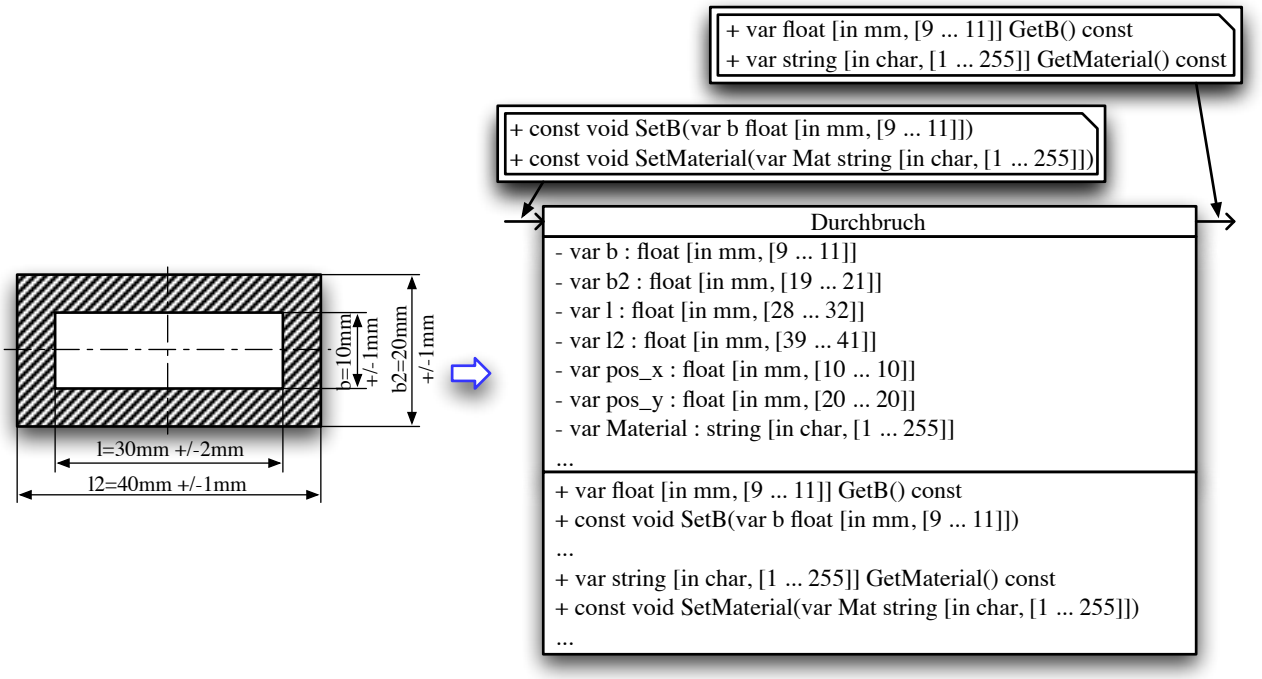


Abbildung 2.80.: Modellierung der kompatibilitätsrelevanten Eigenschaften einer mechanischen Verbindung für die Kompatibilitätsbestimmung.

Auf der rechten Seite der Abbildung 2.80 ist das vereinfachte Klassenmodell des mechanischen Durchbruchs mit sämtlichen charakteristischen Werten aus der CAD Zeichnung dargestellt. Die Werte aus der CAD Zeichnung wurden dabei mit Hilfe von Eigenschaften modelliert und die Einheiten und Toleranzen mittels der zuvor eingeführten formalen Notation für die Kompatibilitätsmodellierung und -bewertung den Eigenschaften hinzugefügt. Zusätzlich enthält die Klasse *Durchbruch* die Eigenschaft `- var Material : string [in char, [1 .. 255]]` mit deren Hilfe das Material des Durchbruchs angegeben werden kann. Die Eigenschaft `... Material ...` ist in der ursprünglichen CAD Zeichnung nicht enthalten, wurde jedoch durch den Identifikationsprozess für kompatibilitätsrelevante Eigenschaften ebenfalls als kompatibilitätsrelevant klassifiziert und aus diesem Grund in das Modell übernommen.

Aufgrund der Tatsache, dass sämtliche Eigenschaften der Klasse *Durchbruch* privat spezifiziert wurden, müssen für jede Eigenschaft der Klasse spezielle Zugriffsmethoden (Accessoren) definiert werden, um die internen (privaten) Eigenschaften auslesen bzw. verändern zu können.

Kompatibilitätsmodellierung von Softwaredaten

Aufgrund der Tatsache, dass bei der modellbasierten Softwareentwicklung das objektorientierte Modellierungsparadigma schon seit geraumer Zeit Einzug gehalten hat und angewendet wird, ist die Transformation der im Allgemeinen in der objektorientierten Modellierungssprache UML/UML2 vorliegenden Klassendiagramme sehr einfach. Die Abbildung 2.81 zeigt auf der linken Seite die Klasse *A* in UML/UML2 Notation. Auf der rechten Seite der Abbildung ist die UML/UML2 Klasse *A* in der formalen Notation zur Modellierung der kompatibilitätsrelevanten Eigenschaften dargestellt.

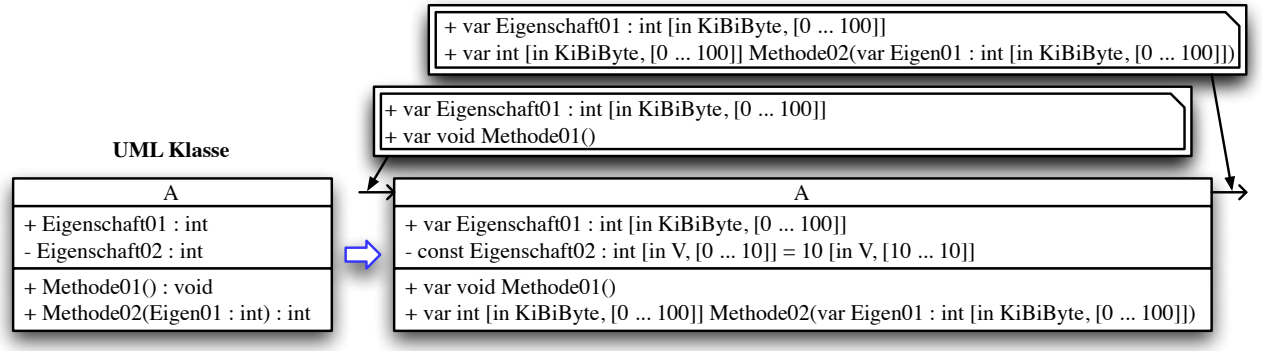


Abbildung 2.81.: Modellierung der kompatibilitätsrelevanten Eigenschaften einer UML Klasse für die Kompatibilitätsbestimmung.

Auf der rechten Seite der Abbildung 2.81 ist das für die objektorientierte Kompatibilitätsbestimmung erweiterte Klassenmodell inklusive der beiden explizit modellierten Schnittstellen dargestellt. Dabei können prinzipiell sämtliche Einträge aus dem UML/UML2 Klassendiagramm in das erweiterte formale Schema zur Kompatibilitätsmodellierung transferiert werden. Im Allgemeinen sind für die Transformation jedoch Zusatzinformationen notwendig, wie beispielsweise die physikalische Größe, die ein UML/UML2 Attribut repräsentiert. In der Abbildung besitzt beispielsweise die UML/UML2 Klasse *A* die Eigenschaft *Eigenschaft01* die nur, wie in der Softwareentwicklung üblich, einen explizit modellierten Datentyp (hier **`int`**) besitzt. Im UML/UML2 Klassendiagramm werden dabei weder die Gültigkeitsintervalle der Eigenschaft, noch deren Einheit angegeben. Diese Angaben sind jedoch beispielsweise für die Programmiersprache C/C++ implizit in so genannten maschinenabhängigen „Header-Files" enthalten. Dort ist exakt hinterlegt, wie groß ein bestimmter Datentyp sein kann. Außerdem wird implizit angenommen, dass jeder Datentyp stets auf dem Binärsystem – also 2^x – basiert.

Für die Kompatibilitätsmodellierung und -bestimmung bergen jedoch implizite Angaben großes Fehlerpotential. Beispielsweise dann wenn eine Softwareeigenschaft eine physikalische Eigenschaft repräsentiert. Die Eigenschaft *Eigenschaft02* der UML/UML2 Klasse *A* repräsentiert eine physikalische Eigenschaft (im Beispiel Volt). Um in diesem Fall Typfehler, wie beispielsweise beim NASA Sonden Absturz, vermeiden zu können, müssen solche Eigenschaften mit physikalischen Einheiten und Teilern versehen werden. So kann ein Typfehler effektiv vermieden werden. Aus diesem Grund ist das Attribut *Eigenschaft02* mit der explizit angegebenen Einheit Volt versehen worden. Die Anreicherung von Eigenschaften um Einheiten und Teiler kann jedoch nur durchgeführt werden, wenn Zusatzwissen über die Eigenschaften des Systemmodells vorhanden sind.

Anmerkung:
In allen drei Beispielen hat es sich gezeigt, dass jeweils nur „eine spezielle" Klasse modelliert wurde, ohne dabei die speziellen Eigenschaften der Objektorientierung wie beispielsweise die Vererbung, auszunutzen.

Anwendung des Konzepts der Vererbung bei der Modellierung von technischen Systemen
Auch bei der objektorientierten Modellierung von technischen Systemen kann das allgemein gültige Vererbungskonzept der objektorientierten Modellbildung eingesetzt werden, um die Vorteile der objektorientierten Systementwicklung, wie beispielsweise die Wiederverwendbarkeit von Klassen, ausnutzten zu können. In der Abbildung 2.82 ist ein stak vereinfachtes Klassenmodell mit Vererbungsbeziehungen zwischen den Klassen des Systems dargestellt. Dabei wird deutlich, dass die Nutzung des Vererbungskonzepts nicht auf die Modelle einer Domäne beschränkt ist, sondern über Domänengrenzen hinweg eingesetzt werden kann.

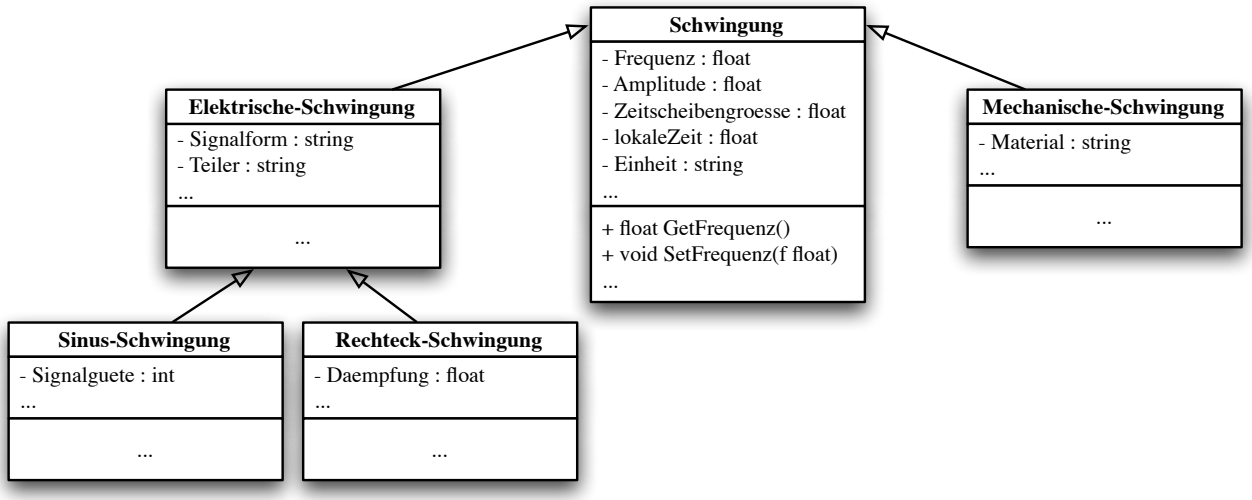


Abbildung 2.82.: Klassenmodell mit Vererbung.

Im oben angeführten Beispiel wurde die Klasse *Schwingung* als Basisklasse sowohl für die Klasse *Elektrische-Schwingung*, als auch für die Klasse der mechanischen Schwingungen eingeführt. Des Weiteren wurde die Klasse *Elektrische-Schwingung* weiter in die beiden Unterklassen *Sinus-Schwingung* und *Rechteck-Schwingung* verfeinert. Dabei werden sämtliche öffentlichen und geschützten Eigenschaften (Attribute) und Methoden der jeweiligen Basisklasse an die Kindklasse vererbt. Durch die konsequente Anwendung der Vererbung bei der Systementwicklung kann sowohl die Güte des Modells verbessert als auch die Entwicklungszeit erheblich verringert werden, da nicht jedes Mal das komplette Systemmodell neu aufgebaut werden muss, sondern Grundbausteine wiederverwendet werden können. So entsteht mit der Zeit eine „große" Klassendatenbank, in der sich viele unterschiedliche Klassen befinden, die später wiederverwendet werden können.

Auch das obige Klassenmodell kann wieder in eine für die Kompatibilitätsmodellierung und -bestimmung einsetzbare formale Form transformiert werden. Dazu müssen lediglich die Eigenschaften und Methoden in die im Kapitel *„Erweiterung des objektorientierten Modellbildungsparadigmas auf die objektorientierte Kompatibilitätsmodellierung und -bestimmung"* vorgeschlagene Form gebracht werden. Sobald dieser Schritt vollzogen ist, kann das UML/UML2-Klassendiagramm für die Kompatibilitätsmodellierung und -bewertung eingesetzt werden. Durch die Wiederverwendung von bereits entwickelten und auf Kompatibilität hin untersuchten Klassen kann außerdem die Güte des Modells erheblich gesteigert werden, da diese als „sichere Klassen" gelten, in denen im Allgemeinen sehr wenige Kompatibilitätsfehler enthalten sind.

2.8. Alternative Identifikationsmethode zur Identifikation der kompatibilitätsrelevanten Eigenschaften eines Systems

Eine alternative Methode, um die Kompatibilität zwischen Steuergeräten eines Fahrzeugs zu modellieren, beschreibt Dr. M. Glockner in seiner Dissertation [Glo07] bzw. dem dazugehörigen Fachartikel [MPDHDFDM07]. Dabei versucht er die folgende Frage zu klären: „Kann ein neu entwickeltes Steuergerät (SG_2) ein Vorgänger-Steuergerät (SG_1) im selben Fahrzeug oder in anderen Fahrzeugmodellen ersetzt d.h. ist SG_2 rückwärtskompatibel zu SG_1 [MPDHDFDM07].

Um diese Frage beantworten zu können, definiert er zunächst den Begriff der Rückwärtskompatibilität und die damit verbundenen Anforderungen an die Entwicklung eines Steuergeräts anhand eines Lastenhefts. In dem Lastenheft des Steuergeräts sind sämtliche statischen und dynamischen Anforderungen an das Steuergerät hinterlegt. Diese textuelle Notation des Lastenhefts wird anschließend in die **E**xtensible **M**arkup **L**anguage (XML) transferiert. Anschließend wird aus dem XML-Modell des Steuergeräts eine Instanz gebildet, die im nächsten Schritt auf Kompatibilität hin untersucht werden soll. Der eigentliche Kompatibilitätstest wird dann mit Hilfe eines X-Diff Algorithmus [Glo07, 42ff], der Änderungen innerhalb einer XML-Datei erkennt durchgeführt.

Durch diese Art des Kompatibilitätstests ergeben sich unterschiedliche Probleme, für die jedoch weder in der Dissertation noch in dem Fachartikel eine Lösung angegeben wird:

- Identifikation der kompatibilitätsrelevanten Eigenschaften des Objekts/Systems
  Die Identifikation der kompatibilitätsrelevanten Eigenschaften des Systems basiert ausschließlich auf der Transformation der Daten aus den Lastenheften. Kompatibilitätsrelevante Eigenschaften des Systems, die in keinem Lastenheft erwähnt wurden, werden nicht erfasst, wodurch es wieder zu Inkompatibilitäten kommen kann.
- Betrachtung der Umgebung um das zu ersetzende Objekt des Systems
  Bei der Betrachtung der Rückwärtskompatibilität eines Steuergeräts gegenüber einem Vorgängermodell wird die Systemumgebung nicht betrachtet. Dies hat zur Folge, dass es zu Kompatibilitätsproblemen kommen kann, die sich erst dann ergeben, wenn das Steuergerät im Kontext eines Systems betrachtet wird.

Ergebnis: Die von Dr. M. Glockner vorgeschlagene Identifikationsmethode für kompatibilitätsrelevante Eigenschaften eines Steuergeräts ist bei weitem nicht ausreichend um Kompatibilität zu beschreiben, da die Umgebung des zu untersuchenden Steuergeräts nicht in die Kompatibilitätsbewertung einfließt. Dies ist jedoch zwingend notwendig, wie die letzten Beispiele gezeigt haben.

Kapitel 3.

Modellierungssprachen und -techniken für technische Systeme

> Alles, von dem sich der Mensch eine Vorstellung machen kann, ist machbar.
>
> *(Wernher von Braun)*

In diesem Kapitel wird eine kleine Auswahl an unterschiedlichen Modellierungssprachen und -techniken vorgestellt, mit deren Hilfe technische Systeme, bestehend aus Hard- und Software, modelliert und graphisch dargestellt werden können. Insbesondere wird hier auf die Modellierung der im Kapitel *„Objektorientierte Modellierung kompatibilitätsrelevanter Eigenschaften von eingebetteten Systemen“* vorgestellten kompatibilitätsrelevanten Eigenschaften eines technischen Systems eingegangen.

3.1. Aufbau und Struktur des Kapitels

Um die verschiedenen existierenden Modellierungssprachen und -techniken für technische Systeme im Hinblick auf die Modellierung und Bewertung von Kompatibilität besser vergleichen und einschätzen zu können, werden zunächst im Abschnitt *„Anforderungen an eine Kompatibilitätsmodellierungssprache“* allgemeine Anforderungen an eine domänenübergreifende Kompatibilitätsmodellierungssprache für technische Systeme formuliert. Anhand der identifizierten Anforderungen an eine domänenunabhängige Modellierungssprache wird im darauf folgenden Abschnitt ein typisches Beispielsystem konstruiert und erläutert, so wie es im realen technischen Umfeld anzutreffen ist. Das Beispielsystem dient später als gemeinsame Modellierungs- und Bewertungsgrundlage für alle in diesem Kapitel vorgestellten Modellierungssprachen und -techniken.

Der zweite Teil dieses Kapitels beginnt mit der Einführung einer neuen, speziell für die Anforderungen der Kompatibilitätsmodellierung und -bewertung von technischen Systemen, bestehend aus Hard- und Software, entwickelten Modellierungssprache. Abgeschlossen wird das Kapitel *„Modellierungssprachen und -techniken für technische Systeme“* mit dem Vergleich und der Bewertung aller vorgestellten Modellierungssprachen und -techniken im Hinblick auf die Modellierung und Bewertung von Kompatibilität.

3.2. Anforderungen an eine Kompatibilitätsmodellierungssprache

Um unterschiedliche Modellierungssprachen und -techniken für komplexe technische Systeme miteinander vergleichen zu können, ist es zwingend notwendig, zunächst die wesentlichen Bewertungskriterien, nach denen die Modellierungssprachen und -techniken verglichen werden sollen, zu identifizieren. Während des vom BMBF geförderten Forschungsvorhabens MOKOMA wurden unterschiedliche Anforderungen an eine domänenunabhängige Kompatibilitätsmodellierungssprache für eingebettete softwarelastige Systeme gesammelt und anhand ihrer Wichtigkeit, in Haupt- und Nebenanforderungen eingeteilt und gewichtet. Die nachfolgende Aufzählung enthält stichpunktartig die im MOKOMA-Projekt identifizierten Anforderungen an eine Kompatibilitätsmodellierungssprache – nach *Haupt-* und *Nebenanforderungen*[127]:

- **Hauptanforderungen**
  - Unterstützung von Austausch- und Ersetzungskompatibilität.
  - Aufbauend auf bereits existierenden Modellierungssprachen und -techniken.
  - Domänenunabhängigkeit (gleichermaßen einsetzbar in der Informatik, dem Maschinenbau und der Elektrotechnik).
  - Modellierung von Hard- und Software in einem gemeinsamen Modell.
  - Sowohl graphische als auch textuelle Notation aller Sprachelemente.
  - Definition von Kompatibilitätsregeln für Hard- und Software.
  - Modellierung von neuen bzw. erweiterten Datentypen (z.B. Ki, Mi).
  - Beschreibung von Intervallen.
  - Verbindung von Datentyp und Einheiten.
  - Graphische und textuelle Notation der Kompatibilitätsregeln.
  - Graphische und textuelle Bewertung, Auswertung und Darstellung von Kompatibilität bzw. Inkompatibilität.
- **Nebenanforderungen**
  - Einfache Erweiterung auf andere Fachgebiete (z.B. Automobilindustrie, Raumfahrt, etc.).
  - Sowohl von Informatikern als auch von Ingenieuren einfach zu erlernen.

[127] Die Haupt- und Nebenanforderungen an eine generische domänenübergreifende Kompatibilitätsmodellierungssprache wurden aufgrund der im Kapitel „2.2 *Modellbildung*“ ab Seite 31 vorgestellten Grundanforderungen aufgestellt.

- Einbettung der Modellierungssprache in den generischen Kompatibilitätsprozess.
- Entwicklung bzw. Anpassung eines Werkzeugs.
- Beachtung des SIL-Level[128].

Anhand der in der Aufzählung identifizierten Haupt- und Nebenanforderungen an eine domänenunabhängige Kompatibilitätsmodellierungssprache für eingebettete softwarelastige Systeme werden sämtliche, in den beiden Kapiteln „3.4 *Existierende Modellierungssprachen*“ ab Seite 113 und „3.6 *Einführung in die Kompatibilitätsmodellierungssprache (U)CML*“ ab Seite 153 aufgeführten Modellierungssprachen und -konzepte verglichen und bewertet. Eine kompakte Zusammenfassung der Bewertung finden Sie im Kapitel „3.7 *Vergleich und Bewertung der vorgestellten Modellierungssprachen*“ ab Seite 253.

3.3. Technische Spezifikation des zu modellierenden Beispielsystems „KOMPTEST“

In diesem Abschnitt wird das technische Beispielsystem „KOMPTEST“, das aus Hard- und Software besteht, eingeführt. Dieses dient als gemeinsame Grundlage für die Modellierung und Kompatibilitätsbewertung aller in diesem Kapitel vorgestellten Modellierungssprachen und -konzepte. Das Beispielsystem ist so gewählt, dass es sämtliche zuvor definierten Anforderungen[129] an eine universell einsetzbare Kompatibilitätsmodellierungssprache für eingebettete softwarelastige Systeme abdeckt.

Die Untersuchung der unterschiedlichen Modellierungssprachen und -methoden erfolgt dabei in zwei aufeinander aufbauenden Schritten. Im ersten Szenario wird zunächst das Strukturmodell des Beispielsystems in der jeweiligen Modellierungssprache modelliert und dargestellt. Anschließend folgt die Verhaltensbeschreibung des Systems oder eines Teilsystems. Nachdem sowohl die Struktur als auch das Verhalten des Grundsystems vollständig modelliert worden sind folgt die Bewertung der Modellierung im Hinblick auf die Modellierbarkeit der unterschiedlichen technischen Aspekte des Systems. Das zweite Szenario beschreibt den Austausch einer alten Komponente des Grundsystems gegen eine neue Komponente mit leicht veränderter Schnittstelle. Auch das neu entstandene Systemmodell wird wieder auf Kompatibilität untersucht und das Ergebnis bewertet.

In den folgenden beiden Unterabschnitten werden sämtliche, für die Modellierung des Beispielsystems notwendigen, technischen Eigenschaften detailliert spezifiziert und beschrieben.

3.3.1. Szenario 1: Modellierung des Beispielsystems „KOMPTEST“

Das Szenario 1 unterteilt sich in zwei aufeinander aufbauende Unterabschnitte. Zum einen in die *Strukturmodellierung* des Beispielsystems „KOMPTEST“, in der sämtliche, für die Strukturbeschreibung notwendigen Eigenschaften des Systems, wie zum Beispiel die Komponenten des Systems, die Ein- und Ausgänge der Komponenten (Komponentenschnittstellen) sowie die Verbindungen zwischen den Ein- und Ausgängen der Komponenten festgelegt werden. Und zum anderen in die *Verhaltensbeschreibung* der einzelnen Komponenten (genauer: der Schnittstellen der Komponenten) sowie deren Interaktion (Datenaustausch) untereinander.

Szenario 1: Strukturmodellierung
Das im Szenario 1 zu modellierende technische Grundsystem besteht aus fünf unterschiedlichen Komponenten *Element1, Element2, Element3, SubElement1* und *SubElement2*. Die beiden Komponenten *SubElement1* und *SubElement2* sind in einem Subsystem zusammengefasst. Von sämtlichen Komponenten des Systems sind sowohl die externen Schnittstellen, also die Ein- und Ausgänge, sowie das Schnittstellenverhalten bekannt. Der innere Aufbau genauso wie das innere Verhalten der einzelnen Komponenten des Systems sind nicht bekannt. Für die Kompatibilitätsmodellierung bzw. -bestimmung ist weder das innere Verhalten noch der innere Aufbau einer Komponente von Bedeutung, weil sämtliche kompatibilitätsrelevanten Eigenschaften einer Komponente stets Schnittstelleneigenschaften sind[130].

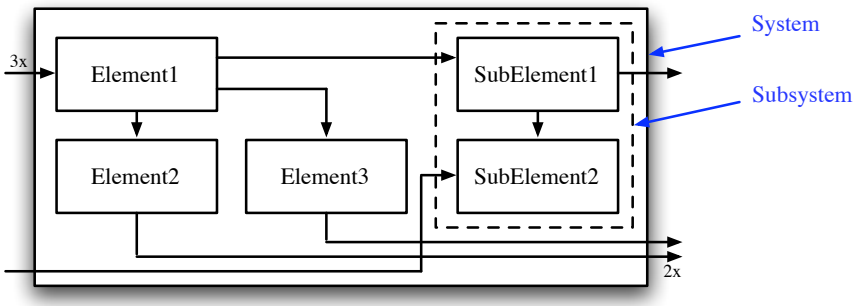


Abbildung 3.1.: Strukturmodell des Beispielsystems „KOMPTEST“ inklusive Verbindungen zwischen den Komponenten des Systems.

[128] Das Akronym *SIL* steht für **S**afety-**I**ntegrity-**L**evel (Sicherheits-Integritätslevel). Nähere Informationen zu SIL finden Sie unter: [Kem], [RAM07], [Wik07f] sowie [IEC07].
[129] Vgl. Abschnitt „3.2 *Anforderungen an eine Kompatibilitätsmodellierungssprache*“ ab Seite 109.
[130] Vgl. „Definition: 1.15“ auf Seite 22.

In der Tabelle 3.1 sind die fünf Komponenten des Systems, die Anzahlen der Ein- und Ausgänge und das aus der Schachtelung resultierende Subsystem eingetragen. Die Tabelle 3.2 zeigt beispielhaft für das gesamte System die technischen Spezifikationen des Ausgangs *O*1 der Komponente *Element1* sowie des Eingangs *I*1 der Komponente *Element2*. Anhand dieser beiden Schnittstellen wird beispielhaft für alle Anschlüsse der Komponenten des Systems sowohl die Strukturmodellierung als auch das Schnittstellenverhalten untersucht. Die Abbildung 3.1 illustriert die Struktur des zusammengebauten Systems inklusive der Verbindungen zwischen den einzelnen Bestandteilen des Systems.

Komponentenname	**Eingänge**		**Ausgänge**	
	Anzahl	Bezeichnung	Anzahl	Bezeichnung
System	4	I1, I2, I3, I4	4	O1, O2, O3, O4
Element1	3	I1, I2, I3	3	O1, O2, O3
Element2	1	I1	2	O1, O2
Element3	1	I1	1	O1
SubSystem	2	I1, I2	1	O1
SubElement1	2	I1, I2	1	O1
SubElement2	1	I1	1	O1

Tabelle 3.1.: Komponentenbeschreibung des Beispielsystems „KOMPTEST“.

Komponentenname	**Technische Spezifikation**		
	Anschlussbezeichnung	*Beschreibung*	
Element1	O1	Richtung	Ausgang
		Anschlussart	Elektrotechnik
		Spannung	9
		Einheit	Volt
		Definitionsbereich	[0..10[
		Strom	1
		Einheit	Ampere
		Definitionsbereich	[0..1]
Element2	I1	Richtung	Eingang
		Anschlussart	Elektrotechnik
		Spannung	9
		Einheit	Volt
		Definitionsbereich	[0..10[
		Strom	1
		Einheit	Ampere
		Definitionsbereich	[0..1]

Tabelle 3.2.: Spezifikation der wesentlichen Ein- und Ausgänge der Komponenten *Element1* und *Element2* des Beispielsystems „KOMPTEST“.

Szenario 1: Verhaltensbeschreibung
Nachdem die Strukturbeschreibung des Beispielsystems „KOMPTEST“ abgeschlossen ist, folgt nun die Beschreibung des Verhaltens der einzelnen Komponenten des Systems bzw. die Modellierung der Interaktionen der einzelnen Komponenten des Systems untereinander. In der Abbildung 3.2 auf der nächsten Seite ist sowohl das Schnittstellenverhalten, als auch die Interaktion (Datenaustausch) der beiden Komponenten *Element1* (O1) und *Element2* (I1), beispielhaft für alle anderen Schnittstellen des Systems herausgegriffen, dargestellt.

Die Komponente *Element1* (1) bekommt von der Systemumwelt drei Signale in der Reihenfolge *I2, I1* und *I3*, übermittelt. Nachdem die drei Signale an der Komponente *Element1* angekommen und verarbeitet worden sind, sendet die Komponente *Element1* die Nachricht *O1* an die Komponente *Element2* (2) sowie eine weitere Nachricht an die Komponente *Element3*. Nachdem die Komponente *Element2* das empfangene Datum verarbeitet hat, sendet sie ihrerseits zwei Signale an die Systemumwelt (*O2* und *O1*).

3.3.2. Szenario 2: Austausch einer Komponente des Beispielsystems „KOMPTEST“

Im zweiten Szenario wird ausgehend vom zuvor modellierten Grundsystem eine Komponente gegen eine neue mit leicht modifizierter Schnittstelle ersetzt. Bei der neuen Komponente *ElementX* handelt es sich um eine Weiterentwicklung der ursprünglich im System verbauten Komponente *Element2*. Bei der neuen Komponente ist lediglich ein weiterer optionaler Eingang *I2* hinzugekommen. Aufgrund der Tatsache, dass der neu hinzugekommene Anschluss optional ist – also nicht angeschlossen werden muss – kann die Komponente *Element2* ohne Probleme gegen die Komponente *ElementX* ausgetauscht werden, falls das Verhalten (Schnittstellenverhalten) der neuen Komponente (ohne den angeschlossenen Anschluss *I2*) gleich dem Verhalten der alten ist. Die Abbildung 3.3 auf der nächsten Seite zeigt links das Strukturmodell der zu ersetzende Komponente *Element2* des Grundsystems. Auf der rechten Seite der Abbildung ist die neue Komponente *ElementX* dargestellt.

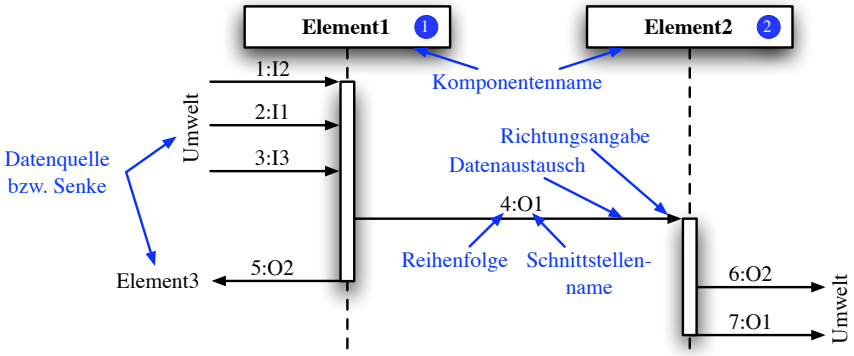


Abbildung 3.2.: Verhaltensmodell der Verbindung zwischen der Komponente *Element1* und *Element2*.

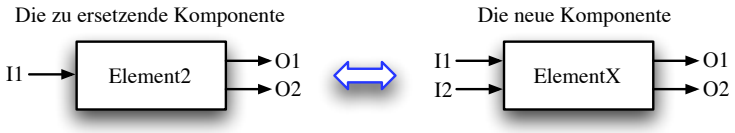


Abbildung 3.3.: Austausch der Komponente *Element2* durch die Komponente *ElementX* (Szenario 2).

Das Strukturmodell der neuen Komponente *ElementX* hat gegenüber dem alten einen optionalen Eingang (*I2*) hinzugewonnen. Die Tabelle 3.3 beschreibt die technische Spezifikation des neu hinzugekommenen Eingangs *I2* der Komponente *ElementX*.

Komponentenname	**Technische Spezifikation**		
	Anschlussbezeichnung	*Beschreibung*	
ElementX	I2	Richtung	Eingang
		Anschlussart	Elektrotechnik
		Spannung	5
		Einheit	Volt
		Definitionsbereich	[0..5]
		Strom	0.5
		Einheit	Ampere
		Definitionsbereich	[0..0.5]

Tabelle 3.3.: Spezifikation des hinzugekommenen Eingangs *I2* der Komponente *ElementX*.

Wie bereits erwähnt, ist das Schnittstellenverhalten der Komponente *ElementX* gleich dem Schnittstellenverhalten der Komponente *Element2*, wenn der optionale Anschluss *I2* der neuen Komponente nicht angeschlossen ist. Sollte der optionale Anschluss beschalten sein, so verändert sich das Schnittstellenverhalten der Komponente *ElementX*. Das Schnittstellenverhalten der neuen Komponente *ElementX* ist in der nachfolgenden Abbildung 3.4 dargestellt. Ohne den optionalen Eingang *I2* entspricht das Schnittstellenverhalten dem der Komponente *Element2* (vgl. Abbildung: 3.2 auf Seite 112 (rechts)).

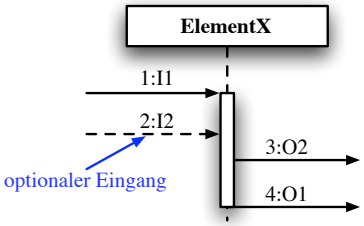


Abbildung 3.4.: Schnittstellenverhaltensmodell der Komponente *ElementX*.

Wie sich das innere Verhalten der Komponente *ElementX* gegenüber dem inneren Verhalten der ursprünglichen Komponente *Element2* verändert ist unbekannt. Für die Verifikation der statischen Struktur der Schnittstelle und des Schnittstellenverhaltens ist das innere Verhalten der Komponente unwichtig und wird für die Kompatibilitätsbestimmung nicht herangezogen.

3.3.3. Kompatibilitätsregeln für das Beispielsystem „KOMPTEST“

Um sowohl das Strukturmodell als auch die Verhaltensbeschreibung auf Kompatibilität hin untersuchen zu können, werden in diesem Abschnitt Kompatibilitätsregeln für das Beispielsystem „KOMPTEST“ definiert, anhand derer das Modell des Beispielsystems verifiziert werden kann. Die Kompatibilitätsregeln werden zu diesem Zweck in zwei aufeinander aufbauende Gruppen unterteilt: in die *statischen Schnittstellenregeln* und die *Schnittstellenverhaltensregeln*.

Statische Schnittstellenregeln
Um die statische Struktur des Beispielsystems „KOMPTEST“ auf Kompatibilität hin untersuchen zu können, werden folgende Regeln verwendet:

- Alle zwingenden Anschlüsse einer Komponente müssen angeschlossen sein.
- Die Einheiten und Teiler der Quelle müssen gleich der Senke sein.
- Das Zielintervall/Definitionsbereich muss das Quellintervall/Definitionsbereich umfassen.
- Ein Datentypen muss sich ohne Informationsverlust in einen anderen überführen lassen.

Kommt es bei keiner dieser Regeln zu einer Regelverletzung im Modell, so sind sämtliche statischen Schnittstellen des Modells kompatibel.

Schnittstellenverhaltensregeln
Nachdem ein Modell auf statische Schnittstellenkompatibilität erfolgreich untersucht worden ist, folgt im zweiten Schritt der Kompatibilitätsüberprüfung die Untersuchung des Schnittstellenverhaltens (Kommunikation) der Komponenten. Dazu müssen im einfachsten Fall lediglich folgende Fälle überprüft werden:

- Die Reihenfolge der gesendeten und empfangenen Nachrichten muss übereinstimmen.
- Es dürfen keine Nachrichten innerhalb des Systems verloren gehen.
- Es darf weder zu einer *Verklemmung*[131] noch zum *Aushungern*[132] einer Komponente kommen.

Ist der Nachrichtenfluss innerhalb eines Systems in Ordnung, so ist das (Beispiel-) System kompatibel. Nachdem nun sowohl die statische Struktur als auch das Verhalten des Beispielsystems in diesem Kapitel festgelegt wurden, folgt nun die Vorstellung verschiedener Ansätze zur Modellierung von Systemen.

3.4. Existierende Modellierungssprachen

In diesem Abschnitt wird eine kleine Auswahl von am Markt befindlichen Modellierungssprachen und -konzepten vorgestellt, mit deren Hilfe es möglich ist, komplexe technische Systeme, bestehend aus Hard- und Software, zu modellieren. Besonderes Augenmerk wird bei der Einführung und Evaluierung der unterschiedlichen Modellierungssprachen auf die Anwendbarkeit für die Modellierung und Bewertung der Kompatibilität von eingebetteten softwarelastigen Systemen gelegt.

Begonnen wird mit der Einführung der „einfachsten“ Modellierungsmethode: dem so genannten *Input-Process-Output Modell* (IPO), in dem lediglich die Elemente des Systems zusammen mit den Flüssen zwischen den einzelnen Systembestandteilen modelliert und dargestellt werden. Daran anschließend folgt die Einführung des am Lehrstuhl für Raumfahrttechnik der Technischen Universität München mitentwickelten *Systems Engineering Element-Konzepts*. Mit Hilfe des Element-Konzepts lassen sich komplexe technische Systeme, bestehend aus Hard- und Software einfach graphisch modellieren.

In den beiden darauf folgenden Kapiteln wird zunächst die aus der Softwareentwicklung stammende objektorientierte Modellierungssprache *UML* – Unified Modelling Language – und daran anschließend die Systemmodellierungssprache *SysML* einführend beschrieben. Abgeschlossen wird das Kapitel mit der Bewertung der existierenden Modellierungssprachen für die Modellierung und Bewertung der Kompatibilität von eingebetteten softwarelastigen Systemen.

3.4.1. Input-Process-Output Modell

Das ursprünglich aus der Informationsverarbeitung stammende IPO – Input → Process → Output – Systemmodell repräsentiert einen einfachen, allgemein gültigen Modellierungsansatz für Systeme. Aufgrund des übersichtlichen Aufbaus sowie der leicht zu erlernenden Struktur der IPO-Modelle ist es besonders gut für die interdisziplinäre Zusammenarbeit bei der Modellierung von Systemen geeignet. Mit Hilfe des IPO-Modells kann sowohl das grundsätzliche Verhalten, als auch der Aufbau und die Struktur des Systems modelliert und beschrieben werden.

Nähere Informationen zum IPO-Modellierungsansatz finden Sie unter [Ver06b], [Wik07h] sowie [Mol94].

[131] Eine *Verklemmung* liegt vor, wenn z.B. ein Prozess *A* auf das Ergebnis des Prozesses *B* wartet und der Prozess *B* auf ein Ergebnis des Prozesses *A* wartet. Von einer Verklemmung wird ebenfalls gesprochen, wenn ein oder mehrere Prozesse auf ein exklusives Betriebsmittel warten, sich dieses durch die Nutzung selbst blockieren oder dieses vom eigenen oder einem anderen Prozess (dauerhaft) belegt ist. Siehe auch [PDB94, 47].

[132] *Aushungern* liegt vor, wenn ein Prozess unendlich lange warten muss, obwohl immer wieder die Möglichkeit besteht weiter zu arbeiten. Siehe auch [PDB94, 62].

Aufbau und Struktur des IPO-Modells
Das IPO-Systemmodell besteht im Wesentlichen aus drei hintereinander angeordneten Bausteinen:

- Einem oder mehreren *Eingängen (input)*:
  Im Eingang werden sämtliche für die Verarbeitung notwendigen Daten und Informationen gesammelt und an die Verarbeitungseinheit, den Prozess, übergeben. Ohne die vollständigen Eingaben kann der nachfolgende Prozess nicht gestartet werden.
- Genau einem *Prozess (process)*:
  Als Prozess versteht man die „Verarbeitungseinheit“, mit deren Hilfe die vom Eingang gelieferte Eingabe bearbeitet und anschließend das Ergebnis der Bearbeitung an die Ausgabe weitergeleitet wird.

  Anmerkung:
  Der Prozess lässt sich um einen lokalen Speicher erweitern. In diesem Speicher kann der Prozess Daten ablegen, diese intern nutzen oder sie zu einem späteren Zeitpunkt an den Ausgang weiterreichen.
- Einem oder mehreren *Ausgängen (output)*:
  Im Ausgang werden sämtliche Systemausgaben gesammelt und an die Systemumwelt bzw. an ein anderes System/Teilsystem weitergereicht.

Die Abbildung 3.5 zeigt links die klassische Darstellung des IPO-Prozessmodells (a). Rechts ist das gleiche System – allerdings in einer leicht modifizierten Darstellung – abgebildet (b). In dieser modernen Darstellung wurde die Pfeilform des Prozesses durch ein Rechteck ersetzt. Dadurch ist es einfacher, mehrere Ein- oder Ausgänge eines Prozesses darzustellen, als dies in (a) möglich ist.

a) "Klassische" IPO-Prozessdarstellung b) "Moderne" IPO-Prozessdarstellung

Abbildung 3.5.: Das IPO-Modell

Die linke Darstellungsform des IPO-Modells (a) ist besonders gut geeignet, wenn der Prozess lediglich über einen Ein- und einen Ausgang verfügt. Benötigt ein Prozess mehr als eine Ein- oder Ausgabe, so muss dieser entweder zusammengefasst, oder für jede Eingabe/Ausgabe ein eigener Pfeil gezeichnet werden. In diesem Fall ist die rechte Darstellungsform (b) besser geeignet. Für die Modellierung und Bestimmung von Kompatibilität ist es im Allgemeinen besser, jeden Ein- und Ausgang einzeln zu modellieren anstatt einen zusammengefassten Ein- oder Ausgang zu verwenden. Aus diesem Grund wird hier nur noch die Darstellungsart auf der rechten Seite verwendet.

Um ein komplexes IPO-Modell übersichtlicher gestalten bzw. Teilsysteme bilden zu können, ist es in einem IPO-Modell erlaubt, Modelle hierarchisch ineinander zu verschachteln. Als weitere Maßnahme, um die Übersichtlichkeit eines IPO-Modells zu verbessern, können Teilsysteme graphisch durch einen „umgebenden Rahmen“ zu einer Einheit zusammengefasst werden. Diese Zusammenfassung hat jedoch keinerlei Auswirkungen auf die Struktur oder das Verhalten des Systems.

Struktur- und Verhaltensmodellierung
Für die Modellierung der *Systemstruktur* und des *Systemverhaltens* werden im IPO-Modellierungsansatz die selben Symbole eingesetzt und verwendet. So beschreibt die Kombination aus Eingabe, Prozess und Ausgabe sowohl die Struktur als auch das Verhalten des Systems.

Die Struktur des Systems – also die vorhandenen Ein- und Ausgänge – werden im IPO-Modell mit den Eingangs- bzw. Ausgangssymbolen links und rechts neben dem Prozesskasten modelliert. Die Eingänge werden dabei stets links und die Ausgänge stets rechts am Prozesskasten angebracht. Die Verarbeitungseinheit (Prozess) wird als rechteckiger Kasten zwischen Ein- und Ausgängen dargestellt, in dem der Name des Prozesses eingetragen ist. Das Verhalten des Systems kann auf unterschiedliche Art und Weisen modelliert werden. Die gebräuchlichsten Beschreibungsvarianten sind Text, als Formel, Graph oder Diagramm. Das Verhalten kann entweder in den Prozesskastens eingetragen werden oder auf einem weiteren Blatt notiert werden.

Beispiel 34: Dualismus zwischen IPO-Struktur- und -Verhaltensmodell
Soll in IPO ein logisches *NAND*-Gatter[133], das mit zwei Ein- und einem Ausgang modelliert wird dargestellt werden, so kann dies mittels des in der Abbildung 3.6 dargestellten IPO-Modells erfolgen. Diese Abbildung beschreibt sowohl die Struktur als auch das Verhalten des Systems *NAND-Gatter*.

- *Strukturbeschreibung:*
  Ein logischer *NAND*-Baustein besitzt zwei Eingänge (I1 und I2), eine Verarbeitungseinheit (NAND) und einen Ausgang (O). Im obigen Modell sind die beiden Eingänge, die Verarbeitungseinheit sowie der Ausgang explizit modelliert und dargestellt.
- *Verhaltensbeschreibung:*
  Wenn an beiden Eingängen (I1 und I2) eine positive Spannung von $+5V$ anliegt, so liegt am Ausgang (O) eine Spannung von $0V$ an. Das Verhalten des Systems *NAND-Gatter* kann nun verbal in textueller Form, als Formel $O = \neg(I1 \wedge I2)$, Graph, Schaltzeichen (vgl. Abbildung: 2.5 auf Seite 34 rechts unten) etc. im Prozesskasten hinterlegt werden.

□

[133] Vgl. Abbildung: 2.5 auf Seite 34.

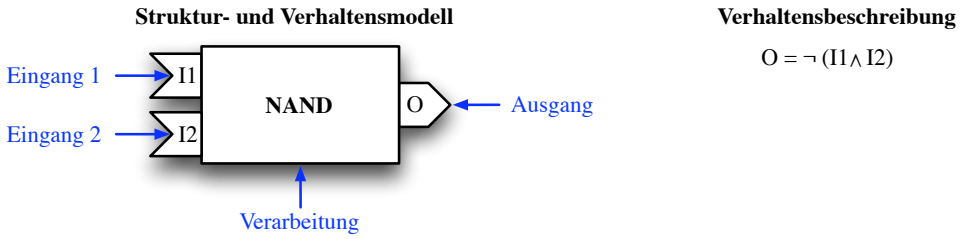


Abbildung 3.6.: Dualismus zwischen Struktur- und Verhaltensmodell

3.4.1.1. Modellierung des Beispielsystems „KompTest“ in IPO

Nachdem im vorangegangen Abschnitt die Grundlagen des IPO-Modellierungskonzepts eingeführt wurden, folgt nun, wie im Kapitel 3.3 erwähnt, die Anwendung des IPO-Modellierungskonzepts auf das dort vorgestellte Beispielsystem „KompTest“ (vgl. Abbildung: 3.1 auf Seite 110).

Im Szenario 1 wird zunächst die Struktur und das Verhalten des Beispielsystems in IPO-Darstellung modelliert und das Ergebnis der Modellierung bewertet. Daran anschließend folgt im Szenario 2 der Austausch einer Komponente des im Szenario 1 modellierten Systems gegen eine neue mit leicht modifizierter Schnittstelle. Nachdem der Austausch der Komponente durchgeführt worden ist, wird das neue System ebenfalls auf Kompatibilität hin untersucht und das Ergebnis bewertet.

Szenario 1: Modellierung von Struktur und Verhalten des Beispielsystems „KompTest“
Zunächst wird aus den im Kapitel 3.3 angegebenen Informationen und Daten des Beispielsystems „KompTest“ das statische Systemmodell erzeugt. Die Abbildung 3.7 zeigt das Ergebnis der Strukturmodellierung des Beispielsystems in IPO-Notation. Um die für die Kompatibilitätsmodellierung und -bestimmung notwendigen Informationen über die technischen Aspekte der Schnittstellen der Komponenten im IPO-Strukturmodell hinterlegen zu können, müssen diese in textueller Form in den entsprechenden Schnittstellen der Komponenten eingetragen werden. Zum Beispiel wird die technischen Spezifikation des Ausgangs $O1$ der Komponente *Element1*: $9V$ Definitionsbereich $[0...10[$, $1A$ Definitionsbereich $[0..1]$ direkt im Ausgang $O1$ hinterlegt (Abb.: 3.7 rechts).

Nachdem die Struktur des Systems modelliert worden ist, folgt im nächsten Schritt die Verhaltensbeschreibung des Systems. Aufgrund der Tatsache, dass im IPO-Modell die Struktur sowie das Verhalten des Systems mit den gleichen Bausteinen modelliert und dargestellt wird, ist das statische Modell identisch mit dem Verhaltensmodell.

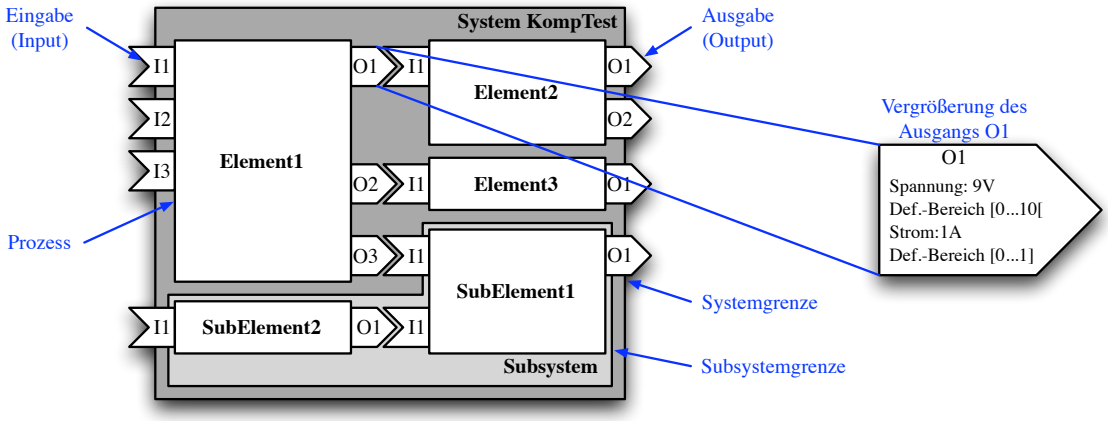


Abbildung 3.7.: Modellierung des Beispielsystems „KompTest“ in IPO-Notation.

Zusätzlich muss nun das Verhalten der einzelnen Elemente des Modells beschrieben werden. Hierzu wird beispielhaft das Verhalten der Kommunikation zwischen den beiden Systembausteinen *Element1* und *Element2* herausgegriffen und in textueller Form beschrieben. Das Element *Element1* bekommt auf seinen drei Eingängen von der Umwelt je ein Datum in der Reihenfolge I2, I1 und I3 übergeben. Diese drei Eingaben werden vom Baustein *Element1* verarbeitet und anschließend an den Ausgängen O1, O2 und O3 ausgegeben.

Der Baustein *Element2* bekommt das Datum aus dem Ausgang (O1) des Bausteins *Element1* übertragen, verarbeitet es und gibt es anschließend auf seinen beiden Ausgängen O2 und O1 (in dieser Reihenfolge) an die Systemumwelt zurück.

Ergebnis: Mit Hilfe des IPO-Modells kann sowohl die Struktur als auch das Verhalten eines einfachen technischen Systems modelliert werden. Dabei hinkt die Modellierung der Struktur eines Systems der Beschreibung des Verhaltens bei weitem hinterher, da das IPO-Konzept hauptsächlich für Modellierung des Eingangs-/Ausgangsverhaltens eines Systems konzipiert wurde. Für die Kompatibilitätsbewertung reichen sowohl die statische Strukturbeschreibung als auch die Verhaltensmodellierung nicht aus. Da beispielsweise weder zusammengesetzte Einheiten noch Kompatibilitätsregeln im Modell hinterlegt werden können. Erschwerend kommt hinzu, dass es keine Werkzeugunterstützung für die Modellierung in IPO gibt.

Szenario 2: Austausch der Komponente *Element2* gegen die neue Komponente *ElementX*
In diesem Szenario wird aus dem zuvor modellierten System eine Komponente durch eine andere, mit leicht veränderter Schnittstelle, ausgetauscht (Ersetzungskompatibilität). Laut Vorgabe soll die Komponente *Element2* gegen die neue Komponente *ElementX* ausgetauscht werden. Die neue Komponente weist gegenüber der ursprünglichen eine modifizierte Schnittstelle sowie ein verändertes Verhalten auf. Das beschriebene Austauschszenario ist in Abbildung 3.8 dargestellt.

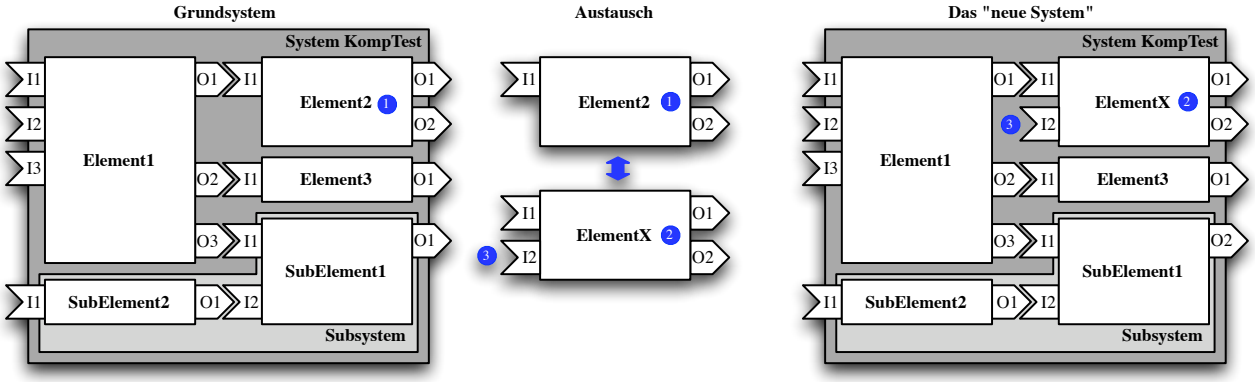


Abbildung 3.8.: Austausch des Elements *Element2* gegen das Element *ElementX*.

Auf der linken Seite der Abbildung ist das Grundsystem aus dem Szenario 1 vollständig abgebildet. In der Bildmitte ist oben die ursprüngliche Komponente *Element2* dargestellt (1), die nun gegen die neue Komponente *ElementX* (2) ersetzt werden soll. Rechts ist das neu entstandene System mit der neuen Komponente *ElementX* abgebildet. Dabei wurden sämtliche zuvor im Ausgangsmodell vorhandenen Schnittstellen wieder miteinander verbunden. Die neu hinzugekommene Schnittstelle *I2* kann nicht verbunden werden. Wenn diese Schnittstelle nicht zwingend verbunden sein muss, kann es sein, dass das System wieder so funktioniert, wie das Grundsystem. Dies ist zum Beispiel der Fall, wenn es sich bei den Eingang *I2* um einen optionalen Eingang handelt und somit das ursprüngliche Verhalten des Systems nicht negativ beeinflusst. Ist für das Verhalten der Komponente *ElementX* jedoch die Schnittstelle *I2* von entscheidender Bedeutung, so kann das System nicht einwandfrei funktionieren und es kommt zu einem Kompatibilitätsfehler. Aufgrund des Fehlens von Kompatibilitätsregeln kann es jedoch vorkommen, dass der Kompatibilitätsfehler im Modell unentdeckt bleibt.

Ergebnis: Das IPO-Modellierungskonzept unterstützt die geforderte Austauschkompatibilität nur bedingt. Es können zwar Komponenten des Systeme einfach gegen andere ersetzt werden, jedoch kann ein dabei auftretender Fehler im Modell nicht erkannt werden, weil im IPO-Modell kein Kompatibilitätsregelwerk enthalten ist.

3.4.1.2. Einsetzbarkeit und Bewertung von IPO für die Bestimmung der Kompatibilität von eingebetteten Systemen

Mit Hilfe der IPO-Modellierungssprache lässt sich vor allem das Eingabe-/Ausgabeverhalten einfacher Systeme auf einem sehr abstrakten Level beschreiben. Dabei tritt die Modellierung der Systemstruktur in den Hintergrund. Aus diesem Grund lassen sich mit Hilfe der IPO-Notation reale technische System bestehend aus Hard- und Software nur schwerlich modellieren. Vor allem dann, wenn das Modell auf Kompatibilitätsfehler untersucht werden soll ist die Erstellung des Modells beschwerlich, da dafür sehr viele technische Systemdetails im Modell abgebildet werden müssen.

Ein weiteres Problem des IPO-Ansatzes liegt zum einen an der fehlenden Unterstützung einer standardisierten formalen Beschreibungsmethode für das (System-) Verhalten wie z.B. durch Sequenzdiagramme (MSC) und zum anderen daran, dass im IPO-Modell keine Kompatibilitätsregeln definiert werden können.

Dennoch sollte während der frühen Entwicklungsphasen eines System die leicht zu erlernende, weitgehend domänenunabhängige Modellierungssprache IPO eingesetzt werden, um das gewünschte grundlegende Verhalten eines Systems zu beschreiben. Die Stärke des IPO-Ansatzes liegt dabei vor allem bei der barrierefreien Kommunikation zwischen den unterschiedlichen Entwicklungspartnern und mit Kunden. Für die konkrete Modellierung des technischen Systems und vor allem für die Kompatibilitätsbewertung, kann IPO allerdings nicht eingesetzt werden.

3.4.2. Das Systems Engineering Element-Konzept – „Die Münchner Schule“

Eine weitere Modellierungssprache für komplexe technische und nichttechnische Systeme stellt das so genannte Systems Engineering „Element-Konzept“ dar. Das klassische Element-Konzept wurde bereits im Jahre 1982 von Prof. Dr. G. Patzak [PDDP06] in seinem Buch „Systemtechnik – Planung komplexer innovativer Systeme“ [Pat82, 19ff] in seinen Grundzügen vorgestellt. Dort formuliert er verbal ein System als eine Menge von Elementen die miteinander in Relation stehen.

Definition 3.1 ***SE Element***
Ein System besteht aus einer Menge von ***Elementen****, welche* ***Eigenschaften*** *besitzen und welche durch* ***Relationen*** *miteinander verknüpft sind [Pat82, 19].*

Aufbauend auf der allgemeinen gehaltenen Systemdefinition von Prof. Dr. G. Patzak hat Prof. Dr.-Ing. E. Igenbergs im Jahre 1992/93 am Lehrstuhl für Raumfahrttechnik – Systems Engineering Group – der Technischen Universität München das klassische Element-Konzept wieder aufgegriffen und um weitere Aspekte, wie beispielsweise eine graphische Repräsentation des Systems oder die Einführung einer hierarchischen Systemstruktur erweitert. In der nachfolgenden Abbildung 3.9 ist ein einfaches System, das aus den drei Elementen *Element 1*, *Element 2* und *Element 3* besteht, in Element-Notation dargestellt. Jedes Element hat Eigenschaften (Attribute) und Funktionen. Außerdem sind die drei Elemente des Systems untereinander durch Relationen verbunden. Die Relationen werden ebenfalls verwendet, um die Elemente des Systems mit anderen Systemen oder der Umwelt zu verbinden.

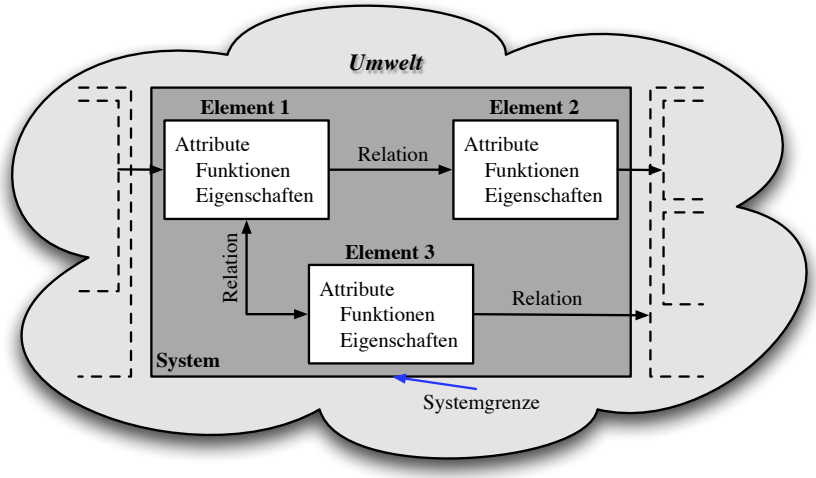


Abbildung 3.9.: Das ursprüngliche Systems Engineering Element-Konzept von 1992/93 (nach [DIW93, 7ff]).

Das von Prof. Dr.-Ing. E. Igenbergs überarbeitete klassische Element-Konzept wurde und wird auch heute noch in zahlreichen Industrieprojekten, Veröffentlichungen und Dissertationen verwendet. Haupteinsatzgebiet des Element-Konzepts sind zahlreiche Raumfahrtprojekte des Lehrstuhls für Raumfahrttechnik der Technischen Universität München. Beispielhaft möchte ich zwei Raumfahrtprojekte herausgreifen: Das Projekt S_2C_2[134] sowie das Cube-Sat[135] Projekt MOVE. In beiden Projekten wird das Element-Konzept zur Auslegung und Planung (Phase 0 und A) von Missionen und zur Simulation des Satelliten verwendet. Genauere Informationen zu beiden Projekten finden Sie unter http://www.lrt.mw.tu-muenchen.de/de/wissenschaft/.

Aufgrund des allgemein gehaltenen generischen Modellierungsansatzes des SE Element-Konzepts lässt es sich problemlos auch in anderen, nichttechnischen Domänen anwenden. Beispielsweise wurde das Element-Konzept für die Modellierung von Geschäftsprozessen und in der Produktentwicklung eingesetzt [DIF98][DIV01].

Im nächsten Kapitel wird zunächst das klassische Systems Engineering Element-Konzept nach Prof. Dr.-Ing. E. Igenbergs eingeführt und anhand von einfachen Beispielen ausführlich erläutert. Daran anschließend, im Kapitel „3.4.2.2 *Das erweiterte Systems Engineering Element-Konzept*“ ab Seite 126, werden die wohl wichtigste Erweiterungen des klassischen Element-Konzepts erläutert, die speziell für die Modellierung von komplexen softwarelastigen eingebetteten Systemen verwendet werden. Nachdem die Einführung des SE Elemente-Konzepts abgeschlossen ist, folgt die Modellierung des im Abschnitt „3.3 *Technische Spezifikation des zu modellierenden Beispielsystems „KOMPTEST““* ab Seite 110 eingeführten Beispielsystems. Anhand dieses Beispielsystems sowie der zuvor formulierten Anforderungen an eine Kompatibilitätsmodellierungssprache wird die Verwendbarkeit des Element-Konzepts zur Modellierung und Bewertung von Kompatibilität eingeschätzt (Siehe hierzu: Kapitel „3.4.2.4 *Einsetzbarkeit und Bewertung des SE Element-Konzepts für die Bestimmung der Kompatibilität von eingebetteten Systemen*“ ab Seite 131.).

Anmerkung:
Im Anhang „A.3 *Mathematische Definition des Systems Engineering Element-Konzepts*“ ab Seite 319 finden Sie eine „formal-mathematische“ Beschreibung des Systems Engineering Element-Konzepts und der verschiedenen Erweiterungen.

3.4.2.1. Das ursprüngliche Systems Engineering Element-Konzept aus dem Jahre 1993

Wie bereits in der Einleitung zu diesem Kapitel beschrieben, stammt das ursprüngliche Element-Konzept von Prof. Dr. G. Patzak und wurde im Jahre 1992/93 von Prof. Dr.-Ing. E. Igenbergs am Lehrstuhl für Raumfahrttechnik weiterentwickelt. Aufbauend auf der einfachen System- bzw. Elementdefinition, hat Prof. Dr.-Ing. E. Igenbergs weitere Eigenschaften für Systeme/Elemente formuliert bzw. die bereits in der Definition 3.1 enthaltenen Begriffe *System*, *Element*, *Eigenschaften* und deren *Verbindung (Relationen)* erweitert.

In der nachfolgenden Aufzählung wird das Systems Engineering Element eingeführt und seine Bestandteile und Eigenschaften erläutert. Nachdem der Aufbau und die Struktur des Elements eingeführt sind, folgt ein ausführliches Beispiel, in dem ein System, das aus mehreren Elementen besteht, mit Hilfe des SE Elementkonzepts modelliert und beschrieben wird.

[134] S_2C_2 steht für *Space System Concept Center*.
[135] Als *Cube-Sat* werden Satelliten bezeichnet, deren maximale Abmessung $10 * 10 * 10\text{cm}$ beträgt und die ein maximales Gewicht von 1kg haben. Genauere Informationen finden Sie unter http://de.wikipedia.org/wiki/CubeSat.

Aufbau und Struktur eines SE Elements
Ein Element stellt einen nach außen abgegrenzten Bereich innerhalb eines Systems dar (vgl. Systemdefinition 1.2 auf Seite 12 sowie die Abbildung 3.9 auf der vorherigen Seite). Es enthält *Attribute* und *Funktionen*, mit deren Hilfe das Element dezidierte Aufgaben erledigen kann. Dabei kann ein Element selbst wieder zu einem System werden. Die folgende Definition 3.2 beschreibt formal ein Element bzw. ein System bestehend aus Elementen.

Definition 3.2 ***System und Element***
1. *Ein System besteht aus Elementen.*
2. *Elemente haben Attribute (Funktionen, Variablen und Konstanten).*
3. *Elemente stehen über Relationen. miteinander in Verbindung*
4. *Ein Element kann selbst wieder ein System sein.*

Die vier Teildefinitionen aus Definition 3.2 werden oft auch als „Axiome [Wie06] des Systems Engineering“ bezeichnet, da sie für alle Systeme gelten, jedoch nicht bewiesen werden können.

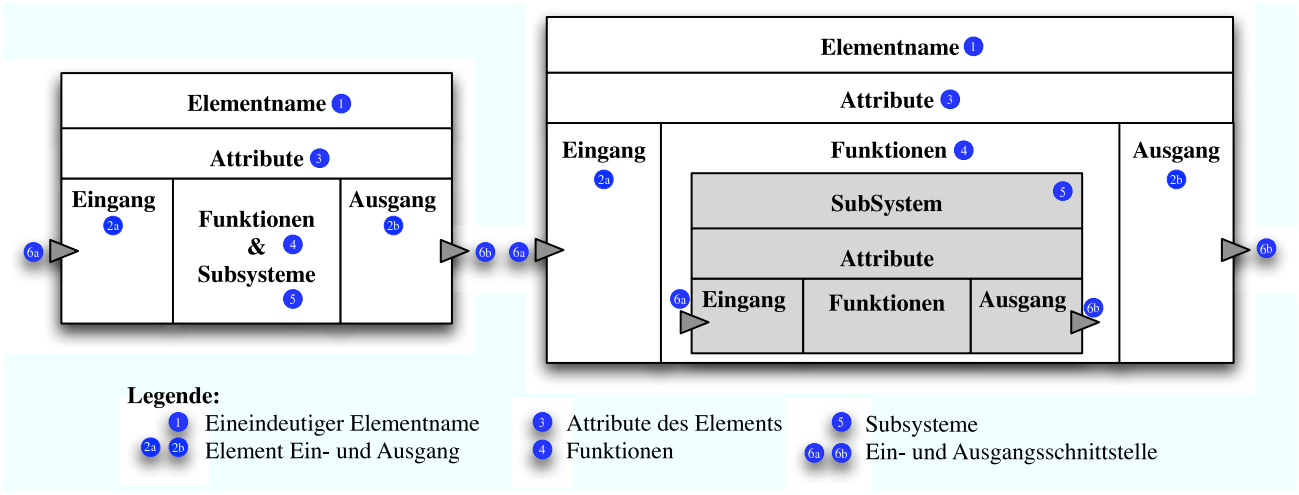


Abbildung 3.10.: Links: Darstellung des generischen SE Elements
Rechts: Geschachteltes System; Besteht aus einem Element sowie einem Subelement

In der Abbildung 3.10 (links) ist der Aufbau und die innere Struktur eines Elements schematisch dargestellt. Das abgebildete Element hat einen eineindeutigen Elementnamen *Elementname* (1), der im gesamten System nur einmal vergeben werden darf und über den es jederzeit eineindeutig identifiziert werden kann. Der Name des Elements sollte so gewählt werden, dass er die Funktionalität bzw. die Aufgabe(n) des Elements möglichst genau beschreibt. Durch eine aufgabenspezifische bzw. funktionsbestimmende Benennung des Elements ist es möglich, die Funktion eines Elements besser verstehen zu können, ohne die einzelne(n) Funktion(en) des Elements im Funktionsblock (4) betrachten zu müssen. Des Weiteren haben Elemente dezidierte Ein- und Ausgänge[136] (2a und 2b) sowie Schnittstellen[137] (6a und 6b), über die sie mit anderen Systemen/Elementen oder der Umwelt kommunizieren können. Eine Kommunikation mit einem Element ist ausschließlich über seine ausgezeichnete Schnittstelle (6a und 6b) möglich. Ein direkter Zugriff (lesen oder schreiben) auf die interne Struktur (Attribute, Funktionen und Subsysteme) des Elements ist nicht möglich[138]. Ein weiterer wichtiger Bestandteil eines Elements sind die Attribute. Sie lassen sich in *Eigenschaften* und *Funktionen* unterteilen. Im Element werden die Eigenschaften in (3) und die Funktionen des Elements in (4) eingetragen. Im Abschnitt (6) des SE Elements werden die Subsysteme/Subelemente eingetragen sofern das System/Element ein Subsystem/Subelement besitzt (vgl. 3.10 rechts).

Ein Element kann beliebig viele weitere Elemente enthalten und somit selbst zu einem System werden[139] (vgl. Abbildung 3.10 rechts). Hat ein Element ein Subsystem/Subelement, so wird dieses im Abschnitt (4) des Elements eingetragen. Durch die „Ineinanderschachtelung“ von Elementen bzw. Systemen entsteht ein hierarchisch gegliedertes System. Das Element auf der obersten Hierarchieebene, im Allgemeinen das System-Element, wird als das „Top-Level-Element“, die untergeordneten Systeme/Elemente als „Subsystem/Subelement“ bezeichnet. Allgemein: Ein Element ist *Vater* eines anderen, wenn es diesem übergeordnet ist. Im umgekehrten Fall wird das untergeordnete Element als *Kind* bezeichnet (vgl. Abbildung 3.11).

Beispiele für einfache und geschachtelte Systeme/Elemente: Komponenten (z.B. Otto-Motor, Turbine, Festplatte etc.), Bauteile (z.B. Stecker, Diode, Widerstand etc.) oder Prozesse.

[136] Im Englischen werden die Ein- und Ausgänge eines Elements oft auch als „Input“ bzw. „Output“ bezeichnet.
[137] Eine Schnittstelle wird im Englischen als „Interface“ bezeichnet.
[138] Das Konzept des Schutzes interner (Daten-) Strukturen eines Elements/Klasse wird auch als „Datenkapselung“ bzw. einfach als „Kapselung“ oder als „Information Hiding“ bezeichnet. Nähere Informationen zum Konzept der Kapselung bzw. des Information Hidings finden Sie unter [Wik06c], [Kon06], [Str90, 29, 811ff] sowie den Definitionen 2.8 und C.9.
[139] Anmerkungen:
- Wenn z.B. Element A Element B enthält ($B \subset A$), so wird das Element B oft auch als *Subsystem* von A bezeichnet
- Nach dem Satz A.3 auf Seite 319 besteht ein System mindestens aus zwei Elementen

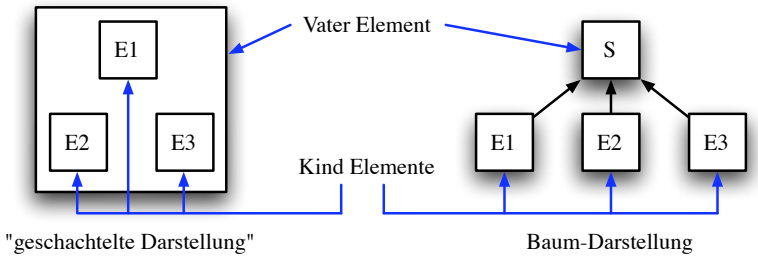


Abbildung 3.11.: Vater – Kind Beziehung zwischen Elementen

Elemente haben Attribute
Die *Eigenschaften* und *Funktionen* bilden zusammen die *Attribute*[140] [Wik06a] des Elements. Im Element-Konzept Kontext werden die Eigenschaften und Funktionen eines Elements oft auch als *Formalisierungskomponenten*[141] des Elements bezeichnet. In der Abbildung 3.12 ist ein Element mit Attributen und Funktionen dargestellt. Die Attribute werden in einem Element in (1), die Funktionen in (2) eingetragen.

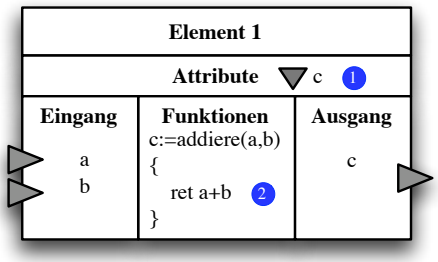


Abbildung 3.12.: Formalisierungskomponenten eines SE Elements.

- **Eigenschaften:**
  Die Elemente können mittels Eigenschaften (Elementparameter, Eingangs-/Ausgangsparameter) beschrieben werden. Die Eigenschaften – auch Parameter genannt – können dabei sowohl Platzhalter sein (qualitative Beschreibung) als auch Werte annehmen (quantitative Beschreibung). Außerdem können die Parameter eines Elements sowohl konstante als auch variable Eigenschaften aufweisen. Im Fall eines konstanten Parameters kann dieser weder durch Eingangs- noch durch interne Funktionen des Elements verändert werden. Im Gegensatz dazu können alle variablen Parameter sowohl von den Eingängen als auch von allen internen Funktionen uneingeschränkt manipuliert und verändert werden. Wird bei der Deklaration eines Parameters keine Angabe über seine Verwendung gemacht, so wird stets angenommen, dass der Parameter *variablen* Charakter hat – also beliebig verändert werden kann. Dies gilt insbesondere, wenn ein interner Parameter auch als Ausgabeparameter verwendet werden soll.

  Das Element *Element 1* (Abbildung 3.12) hat einen Parameter c. Dieser Parameter wird sowohl als interner Speicher für das Funktionsergebnis der Funktion *addiere(a,b)* als auch als Ausgangsparameter verwendet.

- **Funktionen:**
  Die Funktionen eines Elements beschreiben die Funktionalität bzw. das Verhalten des Elements. Die ausgezeichneten Ein- und Ausgabefunktionen übernehmen dabei die Kommunikation des Elements mit anderen Elementen bzw. mit der Umwelt sofern die entsprechenden Relationen (Flussrelationen[142]) existieren. Im klassischen SE Element-Konzept gibt es keine speziellen Ein- und Ausgangsfunktionen.

  Außer für die Kommunikation wird mit Hilfe der Funktionen die Funktionalität bzw. das Verhalten des Elements festgelegt. Soll ein Element z.B. zwei Zahlen a, b miteinander addieren, so muss es in diesem Element eine Funktion c := *addiere(a, b){ret a + b}* geben, die zwei Zahlen vom Elementeingang oder von den internen Variablen (a und b) einliest, das Ergebnis berechnet ($a + b$) und anschließend das errechnete Ergebnis an den Ausgang c oder in der internen Variable c ablegt.

Für die Formalisierungskomponenten eines Elements (die Eigenschaften und Funktionen des Elements) gilt: Ein Lese- oder Schreibzugriff auf die elementinternen (Daten-) Strukturen ist von außerhalb des Elements nicht möglich (vgl. Abbildung 3.13). Auf die internen Eigenschaft(en), Funktion(en) und Subsysteme des Elements kann nur über die ausgezeichnete Schnittstelle (Ein- und Ausgänge) des Elements zugegriffen werden. Eine unbefugte Manipulation von außerhalb des Elements wird somit wirkungsvoll unterbunden. Das Konzept des Schutzes interner (Daten-) Strukturen bzw. das „verbergen“ interner Strukturen eines Elements, wird in Anlehnung an die objektorientierte Programmierung, als „Datenkapselung“ oder „Kapselung“ bzw. als „Information Hiding“ bezeichnet.

[140] Vgl. Definition A.4, Satz A.3 und Lemma A.3 auf Seite 319ff. des Anhangs.
[141] Im SE Element-Konzept werden die Attribute eines Elements manchmal auch als „Formalisierungskomponenten“ bezeichnet (vgl. [PDII06, 8]).
[142] Dr. Negele führt in seiner Dissertation [DIN98, 75ff] die Begriffe *Flussrelation* und *Ordnungsrelation* ein, um zwischen *Datenfluss* und *Struktur* eines Elements zu unterscheiden. Die dort eingeführte Klassifikation der Relationen wird hier aufgegriffen und verfeinert.

Elemente haben Ein- und Ausgänge
Jedes Systems Engineering Element hat mindestens einen Eingang und mindestens einen Ausgang. Die Ein- und Ausgänge des Elements werden an der linken bzw. rechten Seite des Elements eingetragen. Zusammen bilden sie die ausgezeichnete Schnittstelle des Elements. Über diese Schnittstelle kommuniziert das Element (über die Flussrelationen s.u.) mit anderen Elementen oder der Umwelt.

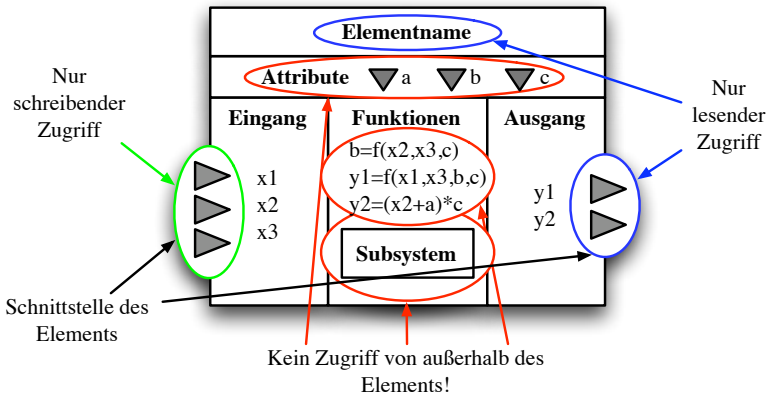


Abbildung 3.13.: Zugriffsmöglichkeiten auf ein SE Element.

In der Abbildung 3.13 ist ein Element mit Ein- und Ausgängen dargestellt. Auf die Eingänge (links) kann nur schreibend zugegriffen werden, während auf die Ausgänge (rechts) lesend zugegriffen werden kann. Genauer: Der Ausgang des Quell-Elements, hier *Element 1*:y_1, schreibt das intern errechnete Datum über die Flussrelation R_{F_1} in den Eingang x_1 des Ziel-Elements *Element 2*. Dieser Sachverhalt ist in der Abbildung 3.14 schematisch für zwei Elemente dargestellt.

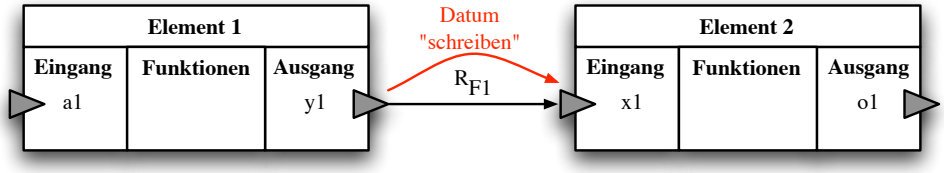


Abbildung 3.14.: Schreib- und Lesezugriff auf die Schnittstelle eines Elements.

Anmerkung:
Der Elementname kann von allen Elementen des Systems, die mittels einer Ordnungsrelation (s.u.) mit diesem verbunden sind, gelesen, aber nicht verändert werden. Diese Erweiterung ist für die Implementierung des SE Element-Konzepts, z.B. im Werkzeug (v)Sys-ed[143], notwendig.

Elemente stehen über Relationen in Verbindung/Wechselwirkung
Elemente stehen über gerichtete Relationen[144] R miteinander in Verbindung/Wechselwirkung. Die Relationen, auch Verbindungen zwischen Elementen genannt, lassen sich in zwei Gruppen unterteilen. Zum einen in die strukturbildenden *Ordnungsrelationen* R_O und zum anderen in die informationstransportierenden *Flussrelationen* R_F.

- **Ordnungsrelationen** R_O:
  Ordnungsrelationen verbinden einzelne Elemente eines Systems miteinander und bilden so die Struktur des Systems/Elements.

  Im Systems Engineering Element-Konzept sind zwei unterschiedliche Arten der Darstellung der Ordnungsrelationen zwischen einzelnen Elementen eines Systems möglich. In der Abbildung 3.15 ist auf der linken Seite die klassische geschachtelte Darstellung der Elemente abgebildet. Auf der rechten Seite ist das gleiche System in Baum-Darstellung gezeichnet.

  - *Klassische geschachtelte Darstellung der Ordnungsrelationen* R_O*:*
    Bei der klassischen geschachtelten Darstellung ist die hierarchische Struktur des Systems/Elements durch die Schachtelung der einzelnen Elemente repräsentiert. Die Schachtelung kann als *enthalten sein* Relation bzw. mengentheoretisch als Teilmenge (Kind-Element (K) ist ein Teil des Vater-Elements (V) also $K \subset V$) interpretiert werden. Durch die geschachtelte Modellierung der Elemente werden keine expliziten Pfeile zwischen den einzelnen Elementen benötigt um die Struktur des Systems/Elements darzustellen. Die Struktur des Systems ist sozusagen *implizit* in der Schachtelung der Elemente enthalten. In der Abbildung 3.15 (links) ist ein geschachteltes System, bestehend aus insgesamt sieben Elementen: dem Systemelement S sowie sechs Elementen E_1 bis E_6 abgebildet. Das Element E_4 kann selbst wieder als System interpretiert werden, da es aus zwei Elementen/Subelementen E_5 und E_6 besteht.

[143] Genauere Informationen zu (v)Sys-ed finden Sie unter [DIS07].
[144] Vgl. verbale Definition der SE-Relation A.2 bzw. mathematische Definition A.5 und Lemma A.4 auf Seite 320ff.

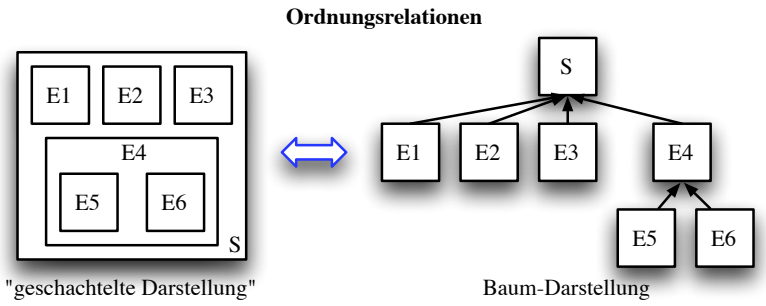


Abbildung 3.15.: Darstellung der Ordnungsrelationen.

Anmerkung:
Es wird keine Aussage darüber getroffen, ob ein Subelement auch ohne das übergeordnete Vater-Element „überleben“ kann oder nicht, falls es aus dem System/Element entfernt wird (vgl. Unterscheidung von Aggregation und Komposition in UML/UML2 im Kapitel *„Die Unified Modelling Language – UML/UML2“*, Abbildung 3.30 auf Seite 135).

– *Baumdarstellung der Ordnungsrelationen R_O:*
Im Gegensatz zur geschachtelten Darstellung der Struktur eines Systems/Elements wird in der Baumdarstellung die Struktur des Systems durch Pfeile zwischen den Elementen repräsentiert. Dabei gilt, dass die Pfeilspitze stets zum übergeordneten Element (dem Vater-Element) zeigt. Bei dieser Art der Darstellung ist die Struktur des Systems *explizit* dargestellt. In der Abbildung 3.15 (rechts) sind die Ordnungsrelationen zwischen den Elementen eines System in der Baumdarstellung dargestellt.

Je nach Anwendung wird die eine oder die andere Darstellungsweise bevorzugt. Soll ein System vollständig dargestellt werden – also mit Ordnungs- und Flussrelationen – so muss die geschachtelte Darstellung benutzt werden, da in der Baumdarstellung die Flussrelationen nicht eingetragen werden können. Ein Beispiel für ein geschachteltes System mit Flussrelationen ist in der Abbildung 3.19 auf Seite 125 exemplarisch dargestellt.

- **Flussrelationen** R_F:
Flussrelationen verbinden genau einen Ausgang eines Elements mit genau einem Eingang eines anderen Elements, wobei das Quell-Element ungleich dem Ziel-Element sein muss. Jede Flussrelation ist eine gerichtete Kommunikation, d.h. das Datum „fließt“ nur in einer Richtung: von der Quelle (Pfeilfuß) hin zur Senke (Pfeilspitze). Für jede Kommunikationsverbindung zwischen zwei Elementen sollte eine eigene Flussrelation zwischen dem Ausgang des Quell-Elements und dem Ziel-Element gezeichnet werden. Es ist aber auch möglich mehrere Flussrelationen zu einer Flussrelation zusammenzufassen.

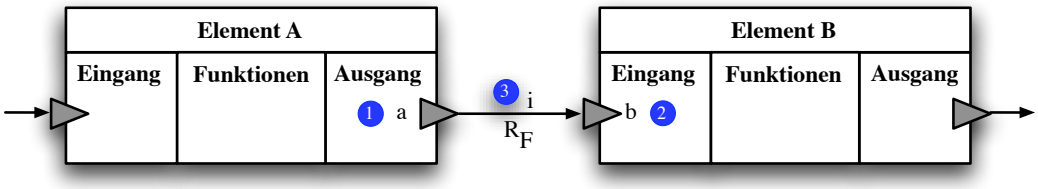


Abbildung 3.16.: Darstellung der Flussrelation.

In der obigen Abbildung 3.16 sind die beiden Elemente (*Element A* und *Element B*) dargestellt, die durch die Flussrelation R_F miteinander verbunden sind. Dabei wird das Datum i von der Quelle (*Element A*) zur Senke (*Element B*) transportiert. Formal lässt sich die Flussrelation R_F zwischen den beiden Elementen *Element A* und *Element B* schreiben als:

$$R_{F:ElementA \xrightarrow{i} ElementB}$$

Nach Definition von Prof. Dr.-Ing. E. Igenbergs haben alle Relationen (Fluss- und Ordnungsrelationen) im SE Element-Konzept stets die Länge „Null“, d.h. es ist unerheblich, ob zwischen zwei Elementen eine „kurze“ oder eine „lange“ Verbindungslinie (Relation) gezeichnet wird. Des Weiteren gilt: Flussrelationen zwischen einzelnen Elementen eines Systems haben „keine eigene Funktionalität“, sie verändern das zu übertragene Datum nicht. Außerdem übertragen sie das Datum i.A. sofort und ohne Zeitverlust ($t = 0$) vom Ausgang eines Elements zum Eingang des anderen.

Ist es für die Modellierung eines Systems wichtig, die Länge der Verbindungen/Relationen zu modellieren, so muss die gewünschte Eigenschaft der Relation (z.B. die Länge einer Leitung zwischen zwei Elementen) explizit in einem eigenen Element modelliert werden. Die Relationseigenschaften des Element-Konzepts werden mit dem nachfolgenden Beispiel verdeutlicht.

Beispiel 35: Anwendung der Flussrelation
Die Festlegung von Prof. Dr.-Ing. E. Igenbergs, dass eine Relation „keine Länge“ und „keine Funktion“ hat, ist in Abbildung 3.17 exemplarisch dargestellt.

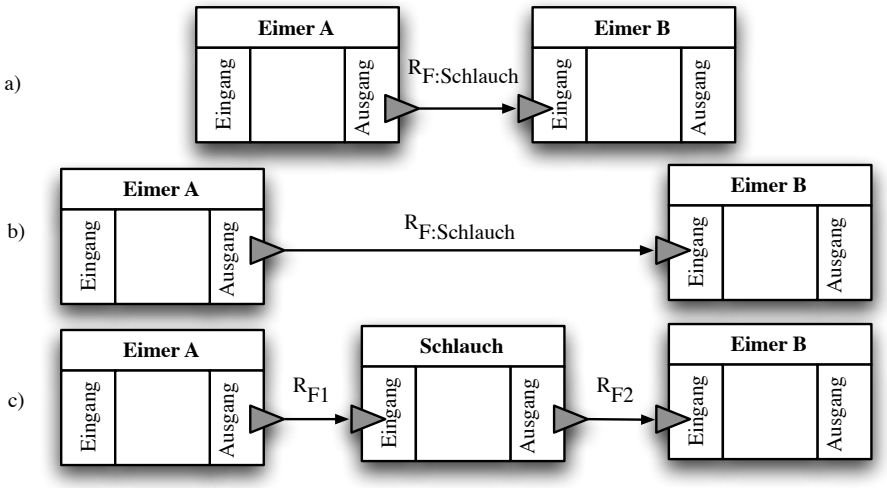


Abbildung 3.17.: Modellierung eines Schlauchs zwischen zwei Eimern.

In der Abbildung ist ein System, das aus zwei Elementen (*Eimer A* und *Eimer B*) besteht dargestellt, die durch eine Flussrelation $R_{F:Schlauch}$ miteinander verbunden sind. Der Inhalt von *Eimer A* soll mittels der Flussrelation $R_{F:Schlauch}$ in den *Eimer B* transportiert werden ($R_{F:EimerA \rightarrow EimerB}$). In a) und b) ist dargestellt, dass es unerheblich ist, wie lang die Relationspfeile gezeichnet sind. Das Datum – in diesem Fall Wasser – braucht stets gleich lang (i.A. $t = 0$) um „durch" oder „über" die Relation $R_{F:Schlauch}$ übertragen zu werden. Für die Modellierung ist es lediglich von Interesse, dass die gerichtete Flussrelation von *Eimer A* zu *Eimer B* führt und nur in diese Richtung etwas (in diesem Fall Wasser) übertragen werden kann.

Im Abschnitt c) der Abbildung wurde die Flussrelation $R_{F:Schlauch}$ explizit als eigenständiges Element *Schlauch* mit Attributen und Funktionalität modelliert. Die ursprüngliche Flussrelation $R_{F:Schlauch}$ aus a) und b), die die beiden Elemente *Eimer A* und *Eimer B* miteinander verbindet, wurde in zwei Teilrelationen unterteilt. Der Ausgang des Elements *Eimer A* wurde durch die neue Flussrelation R_{F_1} mit dem Eingang des Elements *Schlauch* verbunden. Außerdem wurde der Ausgang des Elements *Schlauch* über die Flussrelation R_{F_2} mit dem Eingang des Elements *Eimer B* verbunden ($R_{F_1:EimerA \rightarrow Schlauch}$ und $R_{F_2:Schlauch \rightarrow EimerB}$). In dieser Modellierung können nun dem Element *Schlauch* explizite Eigenschaften und Funktionen, wie z.B. die Form des Schlauches oder die maximale Durchflussmenge, zugewiesen werden. Die beiden Flussrelationen R_{F_1} und R_{F_2} haben weiterhin *keine Länge* und *keine Funktionalität*[145]. □

Die folgende erweiterte System- und Elementdefinition 3.3 enthält sämtliche notwendigen Erweiterungen des ursprünglichen Relationsbegriffs (3) aus der Systems Engineering Elementdefinition 3.2, die sich durch den oben beschriebenen Sachverhalt ergeben. Neu hinzugekommen sind die Unterscheidung der Relationen in Fluss- und Ordnungsrelationen, wie sie bereits Dr. Negele in seiner Dissertation vorgeschlagen hat. Darüber hinaus wurde der Relationsbegriff um die speziellen Eigenschaften der SE-Relationen erweitert.

Definition 3.3 ***System und Element (erweitert)***

1. *Ein System besteht aus Elementen*
2. *Elemente haben Attribute (Funktionen, Variablen und Konstanten)*
3. *Elemente stehen über (Fluss- und Ordnungs-) Relationen miteinander in Verbindung*
   - *i Fluss- und Ordnungsrelationen haben die Länge „null"*
   - *ii Flussrelationen haben „keine eigene Funktionalität"*
   - *iii Flussrelationen übertragen ein Datum im Allgemeinen ohne Zeitverlust ($t = 0$)*
4. *Ein Element kann selbst wieder ein System sein*

Mit Hilfe der vier erweiterten SE-Axiome lassen sich nun nahezu beliebig strukturierte technische und nichttechnische Systeme modellieren. Ein Beispiel für eine System, das aus sieben Elementen besteht, die mittels Ordnungs- und Flussrelationen untereinander und mit der Umwelt verbunden sind, ist in der Abbildung 3.19 auf Seite 125 dargestellt. Wie das in der Abbildung dargestellte System nahe legt, tendiert das SE Element-Konzept dazu, bereits nach wenigen modellierten Elementen sehr groß und unübersichtlich zu werden. Um diesem Umstand entgegenzuwirken, kann die Struktur eines Systems/Elements auch mit Hilfe der „Element-Element-Matrix" dargestellt werden.

SE Element-Element-Matrix

Im Systems Engineering kommt es häufig vor, dass sehr große Systeme mit mehreren hundert Elementen und Relationen zwischen den Elementen betrachtet werden. Um diese enorme Komplexität beherrschbar zu machen, macht sich das Element-Konzept eine kompakte Notation für Systeme – die so genannte „Element-Element-Matrix[146]" – zu Nutze.

[145] Anmerkung: Eine ausführliche formale Beschreibung der Systems Engineering Ordnungs- und Flussrelationen finden Sie im Anhang unter A.2 auf Seite 318.
[146] Im Englischen Sprachraum wird die *Element-Element-Matrix* auch als ***D**esign **S**tructure **M**atrix* kurz *DSM* bezeichnet

In der Element-Element-Matrix werden alle Elemente des Systems der Reihe nach in die Spalten und Zeilen einer quadratischen Adjazenz-Matrix [Wik06j][DN06] eingetragen. Anschließend wird an jedem Kontenpunkt, an dem zwei Elemente durch eine Relation in Verbindung miteinander stehen, eine „1“ eingetragen. Alle anderen Knotenpunkte werden mit einer „0“ gefüllt. Ist es für die Modellierung des Systems von Interesse, zwischen den *Ordnungs-* und *Flussrelationen* des Systems zu unterscheiden, kann für jede der beiden Relationsklassen eine eigene Element-Element-Matrix oder ein erweitertes Werteschema für die Element-Element-Matrix benutzt werden, wie beispielsweise ein „1“ für *Ordungsrelationen* und eine „2“ für *Flussrelationen*.

In der nachfolgenden Matrix M ist ein beliebiges System dargestellt, das aus i Elementen besteht. An jeder Stelle, an der ein Element mit einem anderen Element in Relation (Verbindung) steht, wird eine „1“, in die Matrix eingetragen, ansonsten eine „0“.

$$M_E(i \times i) = \begin{array}{c|cccc} & E_1 & E_2 & \cdots & E_i \\ E_1 & \mathbf{0} & e_{1,2} & \cdots & e_{1,i} \\ E_2 & e_{2,1} & \mathbf{0} & \cdots & e_{2,i} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ E_i & e_{i,1} & e_{i,2} & \cdots & \mathbf{0} \end{array}$$

Bei einer Element-Element-Matrix gibt es eine Besonderheit zu beachten: Per Definition hat ein Element keine Verbindung/Relation zu sich selbst, d.h. alle Relationen (Ordnungs- und Flussrelationen) in einem System sind stets nicht reflexiv. Durch diese Einschränkung ist die Hauptdiagonale der Element-Element-Matrix stets mit „0“ besetzt. Eine ausführliche formale Beschreibung des SE-Matrixkonzept finden Sie im Anhang „A.3 *Mathematische Definition des Systems Engineering Element-Konzepts*“ ab Seite 319 unter Satz A.4 sowie in den beiden Lemmata A.5 und A.6.

Nachdem das SE-Element-Konzept nun theoretisch eingeführt und erläutert wurde, folgt nun ein ausführliches Beispiel, in dem alle oben eingeführten Begriffe und Definitionen noch einmal erläutert werden.

Beispiel 36: Anwendung des Systems Engineering Element-Konzept
Das einfachste SE-Element hat einen Namen und mindestens einen Ein- und Ausgang. Solch ein Element ist in der Abbildung 3.18 (links) schematisch dargestellt. Auf der rechten Seite der Abbildung ist ein konkretes Element mit Attributen und Funktionen modelliert, wie es in einem System vorkommen könnte:

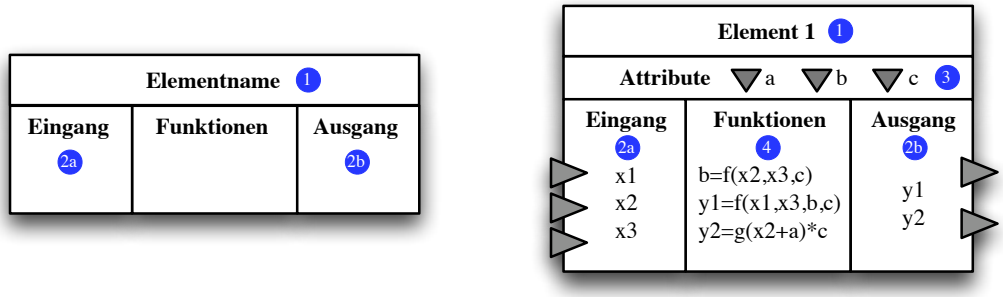


Abbildung 3.18.: Links: Grundstruktur eines SE-Elements.
Rechts: Element mit Eingängen, Ausgängen, Attributen sowie Funktionen.

- **Elementname**
  Im Elementkopf (1) ist der Name des Elements „*Element 1*“ eingetragen. Der Name muss eineindeutig und nicht leer sein, sonst kann das Element in einem System nicht eindeutig identifiziert werden. Des Weiteren sollte der Name möglichst genau die Funktionalität des Elements beschreiben. Je genauer die Bezeichnung eines Elements ist, desto schneller kann dessen Funktionalität bzw. Aufgabe erkannt werden.
- **Eingang und Ausgang**
  Jedes Element hat Ein- und Ausgänge (Abbildung 3.18, 2a und 2b), auch gemeinsame Schnittstelle des Elements genannt, mit deren Hilfe es mit anderen Elementen oder der Umwelt kommunizieren kann (vgl. Abbildung 3.9, 3.18 und 3.19). Dabei ist der Eingang (2a) eines Elements stets auf der linken, der Ausgang (2b) stets auf der rechten Seite des Elements eingezeichnet. Im Beispielelement *Element 1* repräsentieren x_1, x_2, x_2 die Eingangsparameter und y_1, y_2 die Ausgangsparameter des Elements.

  Eine Kommunikation mit dem Element ist ausschließlich über seine ausgezeichnete Schnittstelle möglich. Ein direkter Zugriff (lesen oder schreiben) auf die interne Struktur (Attribute, Funktionen und Subsysteme) des Elements ist nicht möglich. Beispielsweise kann auf die Ein- und Ausgangsparameter (2a: x_1, x_2, x_3 und 2b: y_1, y_2) von außerhalb des Elements ungehindert zugegriffen werden. Die internen Attribute (3: a, b und c) und Funktionen (4: f und g) sind von außen nicht sichtbar und somit nicht veränderbar.

  Das einfache Element *Element 1* enthält kein Subsystem bzw. Subelement. In der Abbildung 3.19 auf Seite 125 ist das System *System KompTest* dargestellt, das mehrere Subsysteme *Subsystem, SubElement 1* und *SubElement 2* enthält. Das Subsystem *Subsystem* enthält zwei eigenständige Subsysteme *SubElement 1* und *SubElement 2*. Die Subsysteme bilden die Hierarchie des Systems (vgl. Unterpunkt *Subsystem* in dieser Aufzählung).

  Eine ausführliche Beschreibung des Schnittstellenkonzepts des ursprünglichen SE Element-Konzepts finden Sie unter [DIS99, 10f und 27-37], [DIN98, 66-103] bzw. [DIW93, 24-45].

- **Attribut(e)**
  Im SE Element-Konzept werden die Attribute eines Elements in *Eigenschaften* und *Funktionen* unterteilt. Die Attribute eines Elements werden manchmal auch als Formalisierungskomponenten bezeichnet.

  – *Eigenschaften:*
  Im SE Element-Konzept gibt es zwei Arten von Eigenschaften:

    * variable
    * konstante

  Als konstante Eigenschaften werden Eigenschaften bezeichnet, die sich während der gesamten Lebensdauer des Elements nicht verändern. Im Gegensatz dazu können die als variabel deklarierten Eigenschaften jederzeit verändert werden. Zusätzlich zu den beiden Eigenschaften konstant und variabel gibt es noch zwei weitere Verfeinerungen der Eigenschaften eines Elements:

    * qualitative
    * quantitative

  Als *qualitative Eigenschaften* werden Eigenschaften bezeichnet, bei denen der genaue Wert einer Eigenschaft unwichtig für die Modellierung und Simulation ist; alle anderen Eigenschaften werden als *quantitativ* bezeichnet.

  Im Beispiel sind die Eigenschaften des Elements *Element 1*, (3) a, b, c; dabei wurde keine genauere Aussage darüber getroffen, ob diese Parameter konstant oder variabel sind. Implizit werden alle Parameter, die nicht explizit als konstant deklariert sind, als variabel angenommen. Außerdem sind die Parameter nur quantitativ angegeben, es fehlen die konkreten Werte.

  – *Funktion(en):*
  Im Block Funktionen (Abbildung 3.18, 4: f und g) sind alle Funktionen des Elements zusammengefasst, die das Verhalten bzw. die Funktionalität des Elements festlegen. Alle Funktionen sind stets nur innerhalb des Elements sichtbar, d.h. auf die Funktionen kann von außerhalb nicht zugegriffen werden (vgl. Kapselung). Die Funktionen werden von den Eingangsvariablen (2a: x_1, x_2, x_3) mit Daten versorgt und speichern nach Abschluss der Berechnung das Ergebnis in den Ausgangsparametern (2b: y_1, y_2) ab. Ein anderes Element oder die Umwelt kann nach Abschluss der Berechnung das Ergebnis am Ausgang abholen. Des Weiteren wird der Parameter b von der Funktion $b = f(x_2, x_3, c)$ berechnet.

  Beispielsweise wird die interne Funktion $y_2 = g(x_2 + a) * c$ vom Eingang mit dem Eingangsparameter x_2 und den internen Parameter a und c mit Daten versorgt, berechnet das Ergebnis und liefert es an den Ausgang y_2 wo es gespeichert wird. Dort kann es von einem anderen Element abgeholt und weiterverarbeitet werden. Außerdem sind die beiden Funktionen f und g nicht weiter spezifiziert. Sollte das Modell „sinnvoll“ eingesetzt werden, so müssten die beiden Funktionen f und g ebenfalls im Block (4) deklariert werden.

- **Subsystem(e)**
  In Abbildung 3.19 auf der nächsten Seite ist das System *System KompTest*, bestehend aus drei Elementen und einem Subsystem, das aus zwei Elementen aufgebaut ist, dargestellt. Aufgrund des vierten SE-Axioms „ein Element kann selber wieder ein System sein“ lassen sich hierarchische Systeme aufbauen. Das System in der Abbildung 3.19 besteht aus zwei Hierarchieebenen [Wik06e]. Auf der ersten Ebene befinden sich die Elemente *Element 1, Element 2*, und *Element 3* sowie das Subsystem *Subsystem*. Die Elemente *SubElement 1* und *SubElement 2* bilden die zweite Hierarchieebene in diesem System. Mit Hilfe des vierten SE-Axioms lassen sich beliebig tiefe hierarchisch geschachtelte Systeme zusammensetzen.

- **Relationsart(en)**
  In einem System gibt es zwei Arten von binären Relationen. Zum einen die so genannten *Ordnungsrelationen*, mit deren Hilfe die Struktur (Hierarchie) eines Systems beschrieben wird, und zum anderen die *Flussrelationen*, die einen Eingang eines Element/Systems mit einem Ausgang eines anderen verbindet [DIN98, 75ff].

  – *Flussrelation*
  Über die Flussrelation werden sämtliche Daten[147] zwischen zwei Elementen/Systemen übertragen bzw. transportiert. Während des Transports der Daten „über die Relation“ werden diese durch die Relation nicht beeinflusst oder verändert. Die Flussrelation ist somit datenerhaltend.

  – *Ordnungsrelationen*
  Mit Hilfe der Ordnungsrelationen wird die Struktur des Systems beschrieben. Der „physikalische Abstand“ bzw. die „reale Lage“ zwischen zwei Elementen in einem System wird so jedoch nicht festgelegt. Die Ordnungsrelation trifft nur eine Aussage darüber, ob zwei Elemente miteinander in Verbindung stehen bzw. eine Struktur bilden.

  Für beide Relationstypen gilt: Die Relationen haben „keine Länge“ und „keine Funktionalität“. (Eine ausführliche mathematische bzw. Systems Engineering Definition der verschiedenen Relationstypen finden Sie im Anhang A.2.)

Element-Element-Matrixschreibweise für Flussrelationen
Die Flussrelationen des Beispielsystems „KOMPTEST“ in der Abbildung 3.19 auf der nächsten Seite lassen sich in eine 7×7-SE Element-Element-Matrix M_{E_F} eintragen (siehe Tabelle 3.4 auf der nächsten Seite). In der Tabelle ist an jeder Stelle eine „1“ eingetragen, an der im Beispielsystem eine Relation (Verbindung) zwischen zwei Elementen besteht, ansonsten eine „0“.

[147] Unter dem Begriff *Daten* werden sowohl Informationen, Energie als auch Materie etc. zusammengefasst.

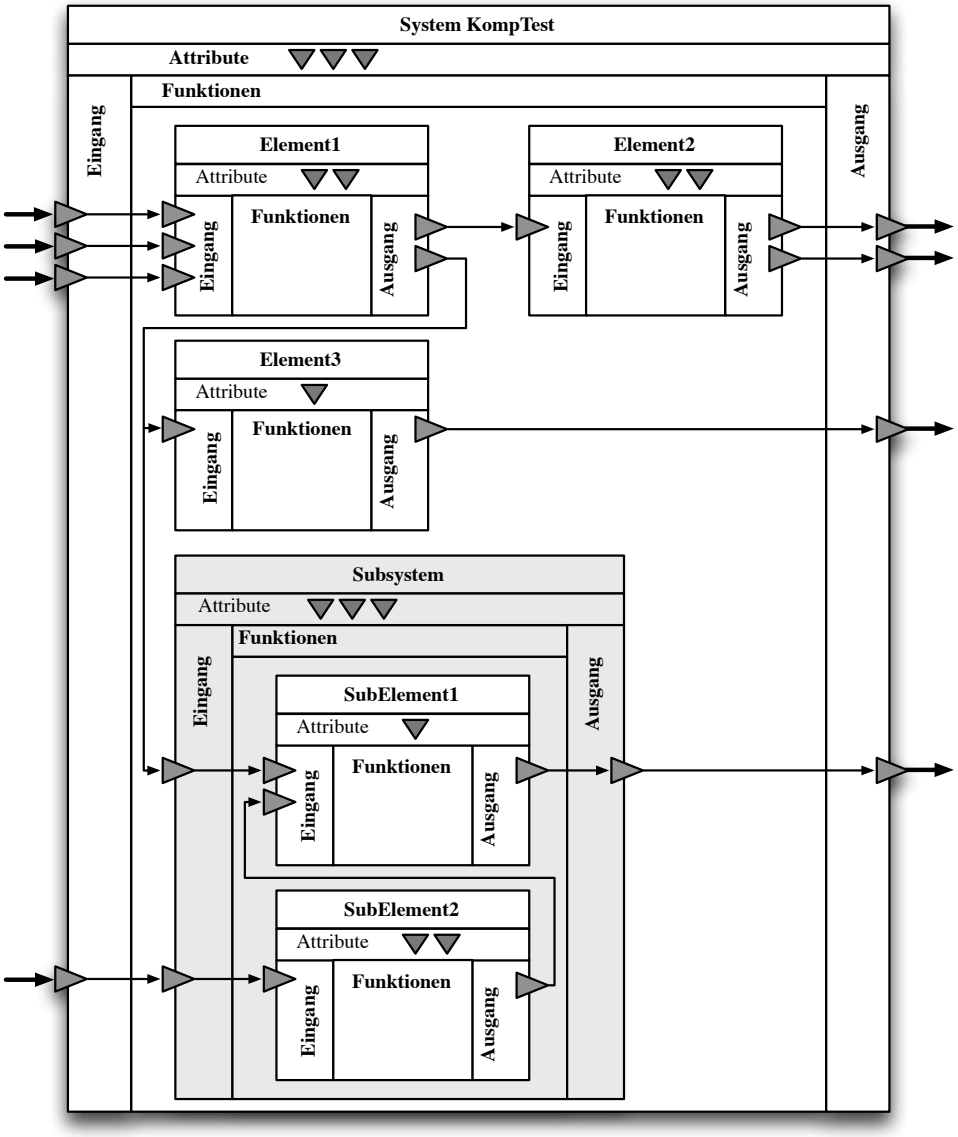


Abbildung 3.19.: Beispielsystem „KOMPTEST“ – bestehend aus drei Elementen und einem Subsystem, das wiederum aus zwei Elementen besteht.

Element/Element-Matrix	SYSTEM KOMPTEST	ELEMENT 1	ELEMENT 2	ELEMENT 3	SUBSYSTEM	SUBELEMENT 1	SUBELEMENT 2
SYSTEM KOMPTEST	**0**	1	0	0	1	0	0
ELEMENT 1	0	**0**	1	1	1	0	0
ELEMENT 2	1	0	**0**	0	0	0	0
ELEMENT 3	1	0	0	**0**	0	0	0
SUBSYSTEM	1	0	0	0	**0**	1	1
SUBELEMENT 1	0	0	0	0	1	**0**	0
SUBELEMENT 2	0	0	0	0	0	1	**0**

Tabelle 3.4.: Element-Element-Matrix des Beispielsystems „KOMPTEST“ (Abbildung 3.19).

Wenn für die Modellierung eines Systems zusätzlich von Interesse ist, welche Elemente mit der Umwelt kommunizieren, kann statt der einfachen Element-Element-Matrix auch die erweiterte Element-Element-Matrix verwendet werden (vgl. Satz A.4 und Lemmata A.5 und A.6 auf Seite 321 ff). □

3.4.2.2. Das erweiterte Systems Engineering Element-Konzept

Wie bereits in der Kapiteleinleitung erwähnt, wurde und wird das Systems Engineering Element-Konzept in zahlreichen Veröffentlichungen und Dissertationen[148] des Lehrstuhls für Raumfahrttechnik der Technischen Universität München verwendet. Aus diesem Grund, wurde und wird das klassische generische Element-Konzept ständig erweitert bzw. an spezielle Projektanforderungen angepasst. Eine für die Implementierung des Element-Konzepts sehr wichtige Erweiterung wurde von Dr.-Ing. L. Schrepfer [DIS99] in seiner Dissertation vorgenommen. Er erkannte, dass das SE Element-Konzept der objektorientierten Programmierung in C++[149] bzw. der Modellierung in UML/UML2[150] sehr ähnlich ist. Um die notwendige Anpassung bzw. Transformation des klassischen SE Element-Konzepts durchführen zu können, unterteilt er den dafür notwendigen Überführungsprozess, der das klassische SE Element-Konzept in eine objektorientierte Umgebung bzw. Programmiersprache überführt, in drei generische Schritte.

Im ersten Schritt wird die statische Struktur der SE-Elemente neben den Ordnungsrelationen zwischen den Elementen des Systems in ein UML/UML2-Klassendiagramm überführt (siehe Kapitel 3.4.3 auf Seite 131). Dabei werden sämtliche SE Elemente in UML-Klassen und alle vorhandenen Ordnungsrelationen des Systems in UML-Relationen[151] transformiert. Im nächsten Schritt werden die SE-Formalisierungskomponenten in die entsprechenden UML/UML2-Konstrukte überführt (SE-Funktionen ↔ UML-Methoden, SE-Attribute ↔ UML-Attribute). Nachdem alle Informationen aus dem SE Element-Konzept in UML/UML2 überführt worden sind, folgt im letzten Schritt des Transformationsprozesses die automatische Überführung der UML/UML2-Klassendiagramme in C++ Code[152]. Nach Abschluss aller Prozessschritte kann das so entstandene Modell/Programm direkt auf einem Computer ausgeführt (simuliert) werden.

Durch den Transformationsprozess von Dr.-Ing. L. Schrepfer wurde die „semantische Lücke[153]“ zwischen dem „SE-Papiermodell“ und einer „Ingenieursanwendung“ geschlossen. Eine ausführliche Beschreibung des Transformationsprozesses bzw. der Simulation eines komplexen technischen Systems finden Sie in [DIS99].

Objektorientierte Erweiterung des Systems Engineering Element-Konzepts
Dr.-Ing. L. Schrepfer hat in seiner Dissertation [DIS99] gezeigt, dass das Systems Engineering Element-Konzept in drei Schritten in ein UML/UML2-Modell bzw. direkt in C++ überführt werden kann. Die wichtigsten Erkenntnisse für die Transformation des SE Element-Konzepts nach UML/UML2 bzw. C++ sind in der nachfolgenden Aufzählung kurz aufgelistet. Die meisten Erweiterungen, die Dr.-Ing. L. Schrepfer in seiner Dissertation eingeführt hat, sind auch für die Erweiterung des SE Element-Konzepts zur Modellierung kompatibilitätsrelevanter Systeme notwendig (siehe Kapitel D auf Seite 331).

Umbenennung der Attribute eines Elements
Im klassischen Element-Konzept nach Prof. Dr.-Ing. E. Igenbergs werden unter dem Begriff *Attribut* sowohl die *Variablen* und *Konstanten* als auch die *Funktionen* eines Elements zusammengefasst. In UML/UML2 bzw. C++ wird jedoch zwischen Variablen und Konstanten auf der einen Seite und Funktionen auf der anderen Seite strikt unterschieden. Aus diesem Grund unterteilte Dr.-Ing. L. Schrepfer in seiner Dissertation die Attribute des klassischen Element-Konzepts in *Elementparameter* (Variable und Konstante) und *Funktionen*. Durch diese Unterteilung lassen sich die Attribute des klassischen Element-Konzepts leicht nach UML/UML2 bzw. C++ überführen.

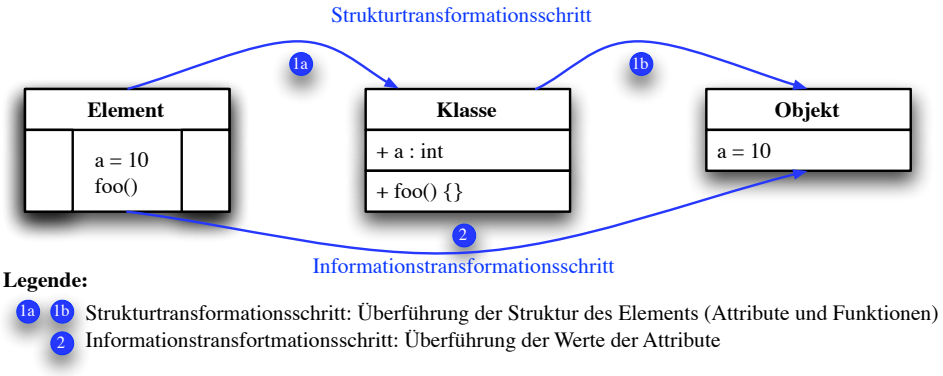


Abbildung 3.20.: Überführung eines SE Elements in eine UML-Klasse bzw. ein UML-Objekt.

Dualismus „Instanz einer Klasse“ – „SE Element“
Die wichtigste Gemeinsamkeit zwischen dem klassischen SE Element-Konzept und der objektorientierten Modellierung in UML/UML2 bzw. der Programmierung in C++ ist die Erkenntnis, dass das klassische SE Element in ein Objekt [Wik06h] bzw. in eine Instanz einer Klasse überführt werden kann. Diese Transformation ist jedoch nicht vollkommen homogen, so dass bei der Überführung des Elements in ein Objekt[154] ein kleiner Zwischenschritt gemacht werden muss.

[148] Eine Auswahl an Büchern und Veröffentlichungen zum Thema Systems Engineering Element-Konzept finden Sie unter [Ver06a] und [TM06].
[149] C++ wurde von Bjarne Stroustrup ab 1979 bei AT&T entwickelt.
[150] Siehe hierzu das Kapitel 3.4.3 auf Seite 131ff.
[151] Im Allgemeinen können alle SE-Ordnungsrelationen in UML-Aggregations-Beziehungen überführt werden.
[152] Die Erzeugung von Code aus einem Modell wird auch als automatische Codegenerierung oder Modell zu Text Transformation bezeichnet.
[153] Siehe hierzu auch das Kapitel „3.6.6 *Der (U)CML-Modellbildungsprozess*“ ab Seite 243.
[154] Der Prozess der Überführung einer Klasse in ein Objekt wird als *Instantiierung* bezeichnet. Nähere Informationen zum *Objekt* finden Sie auch im Kapitel „2.3.1.1 *Klassen und Objekte eines Systems*“ ab Seite 46.

Im ersten Schritt – dem Strukturtransformationsschritt – (Abbildung 3.20) wird aus dem klassischen SE Element eine generische UML-Klasse mit allen Methoden und Variablen erzeugt (1a und 1b). Für alle Variablen der Klasse gilt: sie enthalten keine konkreten Werte und sind also lediglich qualitative Platzhalter. Daran anschließend wird im zweiten Schritt – dem Informationstransformationsschritt – aus der Klasse ein Objekt (Instanz einer Klasse) erzeugt (2), in das alle Werte des SE Elements eingetragen werden (quantitativ). Diese Zweiteilung ist zwingend notwendig da in den meisten objektorientierten Modellierungssprachen und -konzepten zunächst eine Klasse existieren muss, bevor aus der Klasse ein Objekt erzeugt werden kann. Aus dieser Klasse wird im nächsten Schritt – bei der Instantiierung der Klasse – eine Instanz (Objekt) erzeugt, die die konkreten Werte enthält.

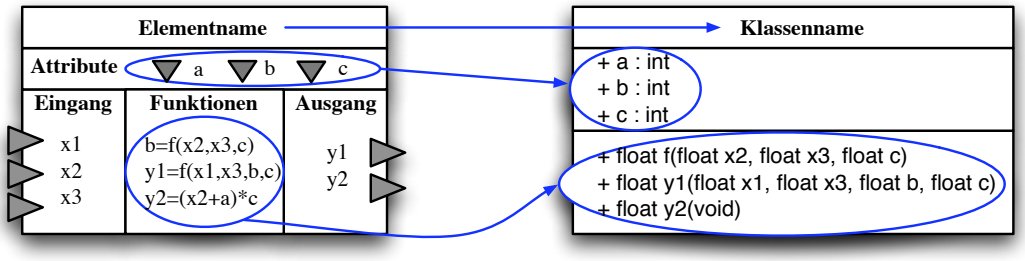


Abbildung 3.21.: Überführung eines klassischen SE Elements in eine UML-Klasse.

In der Abbildung 3.21 ist der Transformationsprozess dargestellt, der die Strukturelemente (Elementparameter und Funktionen) des klassischen Elements in eine UML-Klasse überführt. Für die Variablen der UML-Klasse gilt: sie enthalten keine konkreten Werte. Sollen in die Klasse konkrete Werte eingetragen werden, so muss aus der Klasse eine Instanz, also ein konkretes Objekt erzeugt werden (siehe Abbildung 3.20).

Subsysteme werden als „UML-Aggregation oder Komposition" modelliert
Ein weiterer Unterschied zwischen einem SE Element und einer Klasse bzw. einem Objekt besteht im strukturellem Aufbau. Ein SE Element kann Subsysteme enthalten, während eine Klasse im Klassendiagramm keine weiteren Klassen enthalten kann[155]. Aus diesem Grund müssen alle Subsysteme/Subelemente, die ein SE Element enthält, außerhalb des Elements als eigenständige Klassen modelliert werden. Diese Klassen werden dann mittels einer Aggregation oder Komposition mit der Oberklasse verbunden. Die Begriffe *Aggregation* und *Komposition* werden im Kapitel „3.4.3 *Die Unified Modelling Language – UML/UML2*" ab Seite 131 genauer erläutert.

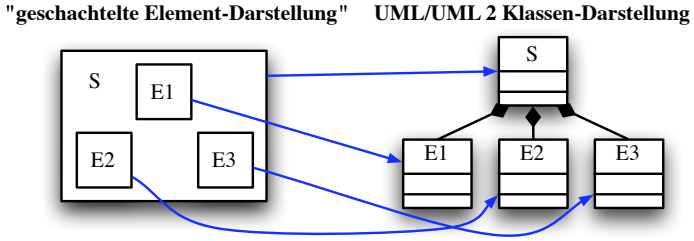


Abbildung 3.22.: Überführung der geschachtelten Elementstruktur in eine UML-Klassenstruktur

In der Abbildung 3.22 ist die Überführung eines geschachtelten Elements (links) in ein baumartiges UML-Klassendiagramm (rechts) exemplarisch dargestellt. Die Überführung der geschachtelten Element-Struktur in eine baumartige Element- bzw. Klassenstruktur ist gleich der in der Abbildung 3.15 auf Seite 121 dargestellten Methode.

Unterscheidung in innere und äußere Element-Schnittstellen
Um das klassische Systems Engineering Element-Konzept, und vor allem die „geschachtelte Struktur" der Elemente besser nach UML/UML2 bzw. C++ überführen zu können, hat Dr.-Ing. L. Schrepfer die Ein- und Ausgänge eines Elements, also die Schnittstelle eines klassischen SE Elements, modifiziert. Die Modifikation der Schnittstelle eines Elements ist der wichtigste Schritt bei der Transformation des Element-Konzepts nach UML/UML2 bzw. nach C++, da weder in UML/UML2 noch in C++ eine „Ineinanderschachtelung" von Klassen möglich ist. In UML/UML2 sowie in C++ werden für die Strukturierung von Klassen Pakete (UML) bzw. „Namensräume" und Module (C++) verwendet, die im Gegensatz zu Elementen keine eigene Funktionalität haben; sie dienen lediglich zur Strukturierung der Klassen bzw. des Quellcodes eines Programms[156].

In der Abbildung 3.23 ist ein Element mit allen Modifikationen, die von Dr.-Ing. L. Schrepfer eingeführt worden sind dargestellt. Die wichtigste Neuerung gegenüber dem klassischen Element-Konzept von Prof. Dr.-Ing. E. Igenbergs ist die Unterteilung der Ein- und Ausgänge des Elements in innere und äußere Ein- und Ausgänge. Dabei entsprechen die äußeren Ein- und Ausgänge des Elements (2a, 2b) den Ein- und Ausgängen des klassischen Element-Konzepts. Neu hinzugekommen sind die inneren Ein-

[155] Anmerkung:
Im Kompositionsstrukturdiagramm ist die Schachtelung von Klassen erlaubt. Allerdings gibt es das Kompositionsstrukturdiagramm erst seit der Version 2.0 der UML, so dass Dr.-Ing. L. Schrepfer diese Diagrammart noch nicht verwenden konnte, um die Schachtelung von Elementen in eine Schachtelung von Klassen zu überführen.

[156] Eine ausführliche Beschreibung der Strukturierungsmöglichkeiten von C++ Programmen finden Sie unter [Str90].

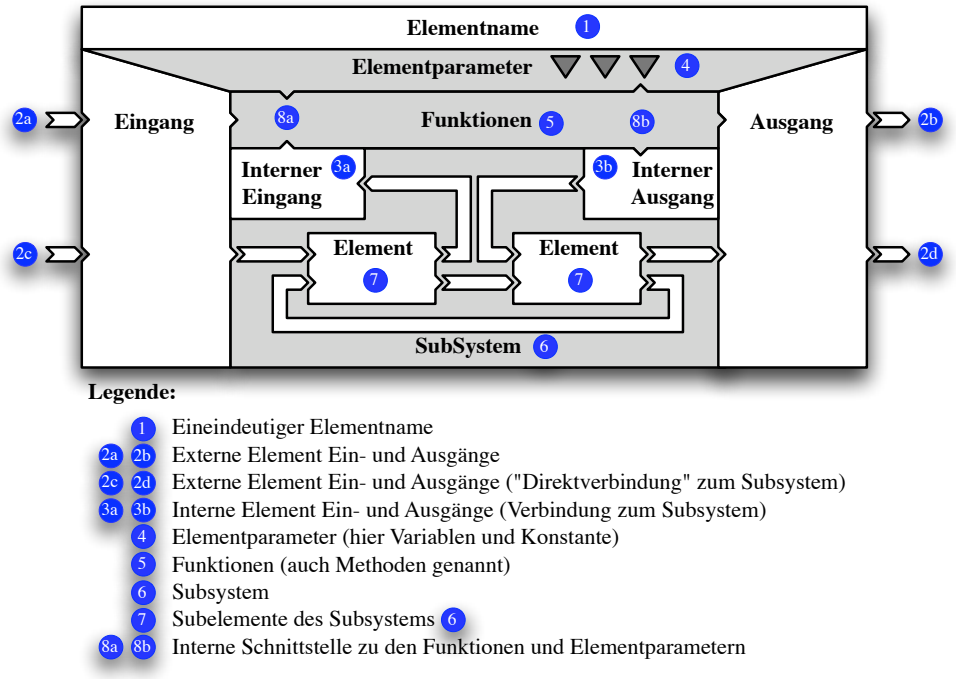


Abbildung 3.23.: Modifikation der Ein- und Ausgänge des erweiterten Element-Konzepts von Dr.-Ing. L. Schrepfer gegenüber dem klassischen SE Element-Konzepts.

und Ausgänge eines Elements (3a, 3b), mit deren Hilfe das Element mit seiner „inneren Struktur“, also den Subsystemen (6) bzw. Subelementen (7) kommunizieren kann. Ebenfalls neu ist die Möglichkeit, dass ein Ein- bzw. Ausgang eines Elements (2c, 2d) *direkt* mit dem Subsystem bzw. den Elementen des Subsystems kommunizieren kann. Von außerhalb des Elements ist es nicht ersichtlich, ob ein Ein- bzw. Ausgang mit dem Element selbst oder dessen Subsysteme verbunden ist. Des Weiteren definiert Dr.-Ing. L. Schrepfer eine interne Schnittstelle zu den Funktionen und Elementparametern des Elements.

3.4.2.3. Modellierung des Beispielsystems „KompTest“ mit Hilfe des Systems Engineering Element-Konzepts

Abgeschlossen wird die Einführung in das Systems Engineering Element-Konzept mit der Modellierung des im Kapitel 3.3 beschriebenen Beispielsystems „KompTest“. Das dort vorgestellte System soll nun mit Hilfe des Element-Konzepts modelliert und auf Kompatibilität hin untersucht werden.

Begonnen wird mit der Modellierung der Struktur des Beispielsystems. Daran anschließend wird das Verhalten einer Komponente des System beispielhaft für alle anderen herausgegriffen und erläutert. Sowohl bei der Modellierung der Struktur als auch des Verhaltens des Systems wird besonders auf die Modellierung und Bewertung von Kompatibilität im Element-Konzept eingegangen. Nachdem die Modellierung des Grundsystems abgeschlossen worden ist, folgt der Austausch einer Komponente des Systems gegen eine andere mit leicht veränderter Schnittstelle.

Anmerkung:
Für die Modellierung des Beispielsystems wird eine „Mischung“ aus dem klassischen sowie dem objektorientierten Element-Konzept verwendet.

Szenario 1: Modellierung der Struktur des Beispielsystems „KompTest“
Die Abbildung 3.19 auf Seite 125 zeigt die Struktur des in diesem Schritt zu modellierenden Beispielsystems in Element-Notation. In dieser Abbildung sind sämtliche Elemente und Verbindungen (Relationen) zwischen den Ein- und Ausgängen der Elemente modelliert und dargestellt. Auf die exakte Beschreibung der Attribute und Funktionen der einzelnen Elemente wurde aus Gründen der Übersichtlichkeit in dieser Abbildung nicht explizit eingegangen. Dies wird hier exemplarisch für die beiden Elemente *Element1* und *Element2* nachgeholt. Die Abbildung 3.24 auf der nächsten Seite zeigt die beiden Elemente *Element1* und *Element2* sowie die Verbindung der beiden mittels einer Flussrelation.

Das Elemente *Element1* hat zwei Attribute *O1Spannung* und *O1Strom* sowie mehrere Funktionen, die nicht explizit ausgeführt sind, da sie sich ausschließlich auf das interne Verhalten des Elements und nicht auf dessen Schnittstellenverhalten beziehen und somit für die Kompatibilitätsbestimmung irrelevant sind. Der konkrete Wert des Attributs *O1Spannung* = *9* bzw. der Wert des Attributs *O1Strom* = *1* sind aus der Tabelle 3.2 auf Seite 111 übernommen worden. Alle weiteren Angaben, wie beispielsweise der Definitionsbereich oder die Einheit, können nicht direkt in das Modell des Elements übernommen werden. Das Element *Element2* hat ebenfalls zwei Attribute *I1Spannung* = *9* und *I1Strom* = *1*. Auch hier stammen die konkreten Werte der Attribute aus der Tabelle 3.2.

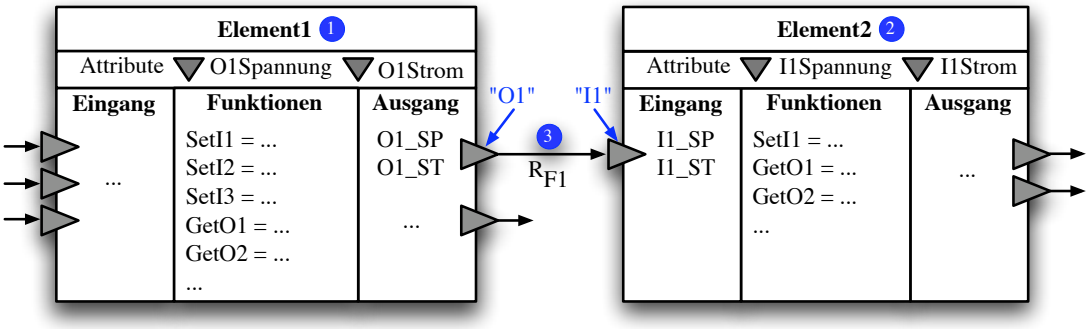


Abbildung 3.24.: Darstellung der beiden Elemente *Element1* und *Element2* inklusive Flussrelation (3).

Anmerkung:
Aufgrund des flexiblen Aufbaus der Attribute und Funktionen eines Elements ist es möglich, zum Beispiel die Attributdeklaration so zu erweitern, dass (wie in den Anforderungen für die Kompatibilitätsbestimmung gefordert) Einheiten und Definitionsbereiche direkt modelliert werden können. So kann z.B. das Attribut *O1Spannung* auch wie folgt deklariert werden: `float O1Spannung = 9V, [0..10]`. Das so veränderte Attribut kann nun direkt für die Kompatibilitätsbestimmung verwendet werden, da sämtliche für die Kompatibilitätsbestimmung notwendigen Bestandteile (Datentyp, Wert, Einheit/Teiler und Definitionsbereich) in der Attributbeschreibung enthalten sind.

Die beiden Elemente sind mittels der Flussrelation R_{F1} (3) miteinander verbunden. Genauer: Der Ausgang „O1“ des Elements *Element1* ist durch die Flussrelation R_{F1} mit dem Eingang „I1“ des Elements *Element2* verbunden. Über die Flussrelation R_{F1} transferiert das Element *Element1* die beiden Daten *O1_SP, O1_ST* vom Ausgang des Elements *Element1* an den Eingang des Elements *Element2*, also

$$R_{F1:Element1 \xrightarrow{O1\_SP,O1\_ST} Element2}$$

und legt die beiden Daten in den entsprechenden Attributen *I1_SP* bzw. *I1_ST* im Element *Element2* ab. In diesem Beispiel wird über die Relation R_{F1} sowohl das Datum *O1_SP*, als auch das Datum *O1_ST* übertragen. Für die Kompatibilitätsbestimmung ist es jedoch besser, wenn über eine Relation nur eine Art von Datum übertragen wird, da es so zu keinem Konflikt bei der Reihenfolge der zu übertragenden Daten kommen kann. Die Abbildung 3.25 zeigt noch einmal die beiden Elemente *Element1* und *Element2* – jedoch mit aufgeteilten Flussrelationen für die beiden zu übertragenden Daten.

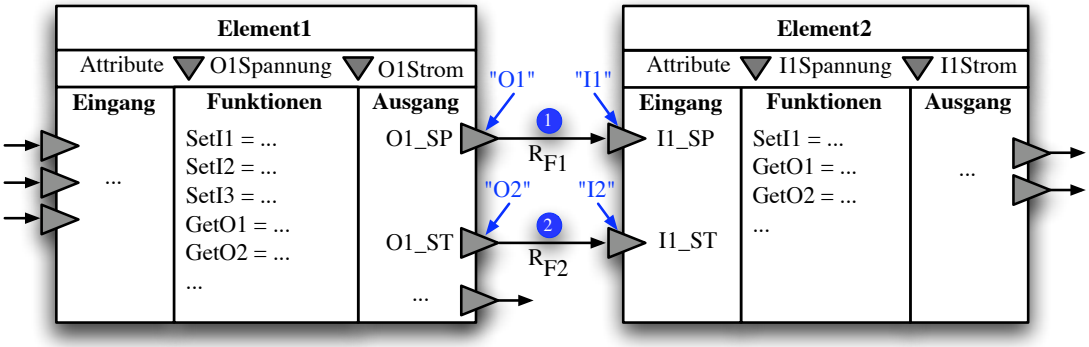


Abbildung 3.25.: Darstellung der beiden Elemente *Element1* und *Element2* mit aufgetrennten Flussrelationen (1) und (2).

In der Abbildung 3.25 wird über die Flussrelation R_{F1} das Datum *O1_SP* vom Ausgang des Elements *Element1* (O1) zum Eingang des Elements *Element2* (I1) übertragen und dort im Attribut *I1_SP* abgelegt.

$$R_{F1:Element1 \xrightarrow{O1_{SP}} Element2}$$

Ebenso wird über die Relation R_{F2} das Datum *O1_ST* vom Element *Element1* zum Element *Element2* übertragen.

$$R_{F2:Element1 \xrightarrow{O1_{ST}} Element2}$$

Die Tabelle 3.5 zeigt die vollständige Flussrelationsmatrix des Systems (vgl. Abbildung: 3.19 auf Seite 125) ohne die oben eingeführte Aufteilung der Flussrelationen. Dabei illustrieren die Zahlen in den Zeilen und Spalten der Matrix die Anzahl der tatsächlichen Flussrelationen zwischen den einzelnen Elementen des Systems. Zum Beispiel besitzt das Element *Element2* zwei Flussrelationen, die vom Elementausgang zum System gehen ($R_{F1:Element2 \xrightarrow{O1,O2} System}$) und eine, die vom Ausgang des Elements *Element1* zum Eingang des Elements *Element2* geht ($R_{F2:Element1 \xrightarrow{O1\_SP,O1\_ST} Element2}$). Falls die genaue Anzahl der Flussrelationen für

die Modellierung des Systems nicht von Bedeutung ist, so kann, wie in der Element-Element-Matrix, auch eine „1“ eingetragen werden. Für die Betrachtung der Kompatibilität zwischen einzelnen Elementen eines Systems ist es jedoch zwingend, alle Flussrelationen in die Flussrelationsmatrix einzutragen, da sonst nicht überprüft werden kann, ob einzelne zwingende Eingangs- oder Ausgangsschnittstellen nicht belegt worden sind.

Flussrelationsmatrix	UMWELT	SYSTEM KOMPTEST	ELEMENT 1	ELEMENT 2	ELEMENT 3	SUBSYSTEM	SUBELEMENT 1	SUBELEMENT 2
UMWELT	**0**	4	0	0	0	0	0	0
SYSTEM KOMPTEST	4	**0**	3	0	0	1	0	0
ELEMENT 1	0	0	**0**	1	1	1	0	0
ELEMENT 2	0	2	0	**0**	0	0	0	0
ELEMENT 3	0	1	0	0	**0**	0	0	0
SUBSYSTEM	0	0	0	0	0	**0**	1	1
SUBELEMENT 1	0	0	0	0	0	1	**0**	0
SUBELEMENT 2	0	0	0	0	0	1	0	**0**

Tabelle 3.5.: Flussrelationsmatrix des Beispielsystems „KOMPTEST“ aus der Abbildung 3.19 auf Seite 125.

Ergebnis: Mit Hilfe des Systems Engineering Element-Konzepts kann die Struktur eines (eingebetteten) Systems vollständig modelliert und dargestellt werden. Die für den Kompatibilitätstest zwingend notwendige Verbindung von Datentyp und Einheit kann im Element-Konzept nicht direkt modelliert werden. Des Weiteren lassen sich die Schnittstellen sowie die Flussrelationen eines Elements nicht ausreichend formal beschreiben, da es keine allgemein gültige Notation z.B. für die Attribute und Schnittstellen eines Elements gibt (vgl. die Axiome des Element-Konzepts).

Durch die Erweiterung zum Beispiel der Attributdeklaration um einen eineindeutigen Datentyp, eine Einheit/Teiler sowie einen Definitionsbereich bzw. eine formale Notation der Flussrelationen zwischen Ein- und Ausgängen von Elementen, kann die Verwendbarkeit des Element-Konzepts für die Modellierung und Bewertung von statischer Kompatibilität erheblich verbessert werden.

Szenario 1: Modellierung des Verhaltens des Beispielsystems „KOMPTEST“
Nachdem im letzten Abschnitt die Struktur des Systems „KOMPTEST“ modelliert wurde, folgt nun die Modellierung des Schnittstellenverhaltens des Systems. Auch hier wird wieder das Schnittstellenverhalten der beiden Elemente *Element1* und *Element2* beispielhaft für das Schnittstellenverhalten des Gesamtsystems herausgegriffen.

In der Abbildung 3.2 auf Seite 112 wurde das geforderte Verhalten, genauer das Schnittstellenverhalten, der beiden Komponenten *Element1* und *Element2* genau spezifiziert und festgelegt. Die Reihenfolge der hier spezifizierten Transaktionen kann nicht ohne weiteres im Element-Konzept abgebildet werden, da im Element-Konzept weder eine Beschreibungsmethode für das Schnittstellenverhalten, wie beispielsweise Sequenzdiagramme, noch die Zeit definiert ist. Beides ist jedoch für die Modellierung des Schnittstellenverhalten zwingend notwendig, da ohne diese Angaben das Schnittstellenverhalten eines Elements nicht beschrieben werden kann. Beispielsweise kann die geforderte Nachrichtenreihenfolge (*I2, I1, I3, O1* und *O2*) des Elements *Element1* nicht modelliert werden.

Ergebnis: Mit Hilfe der Funktionen eines Elements kann das innere Verhalten jedes Elements eingeschränkt festgelegt werden. Über die Reihenfolge der einzelnen Funktionsaufrufe (Schnittstellenverhalten) sowie deren Verknüpfungen untereinander wird allerdings keine Aussage getroffen. Für die Kompatibilitätsmodellierung und -bestimmung ist aber vor allem die exakte Beschreibung der Reihenfolge der Funktionsaufrufe sowie die Kommunikationsreihenfolge der Elemente von entscheidender Bedeutung.

Anmerkung:
Die Funktionen des Element beschreiben das innere Verhalten eines Elements und nicht das Schnittstellenverhalten. Aus der Beschreibung des inneren Verhaltens eines Elements lässt sich jedoch das Schnittstellenverhalten des Elements teilweise ableiten, insofern bekannt ist, in welcher Reihenfolge die einzelnen Funktionen des Elements abgearbeitet werden. Die so gewonnenen Informationen über das Schnittstellenverhalten reichen jedoch für den Kompatibilitätstest bei weitem nicht aus.

Szenario 2: Austausch der Komponente *Element2* gegen die neue Komponente *ElementX*
In diesem Abschnitt wird eine Komponente des zuvor modellierten Beispielsystems „KOMPTEST“ gegen eine andere mit leicht modifizierter Schnittstelle ausgetauscht. Die Abbildung 3.26 auf der nächsten Seite zeigt exemplarisch den Austausch des Elements *Element2* (1) gegen das neue Element *ElementX* (2) mit leicht modifizierter Schnittstelle.

Das Element *ElementX* hat gegenüber dem Element *Element2* einen optionalen Eingang (3) mehr. Dieser Anschluss muss laut Beschreibung nicht zwingend angeschlossen sein. Ist der Anschluss nicht angeschlossen, so hat laut Dokumentation das neue Element *ElementX* genau das gleiche Schnittstellenverhalten wie das ursprüngliche Element *Element2*. Deshalb kann das Element *ElementX* problemlos gegen das Element *Element2* ausgetauscht werden, ohne dass es zu einer Inkompatibilität kommt.

Die optionale Eigenschaft des neu hinzugekommenen Anschlusses (3) kann im Element-Konzept nicht direkt modelliert werden. Aus diesem Grund ist im Modell kein Unterschied zwischen einem zwingenden und einem optionalen Eingang zu erkennen (gilt ebenfalls für Ausgänge). Dies führt unwillkürlich zu versteckten Schnittstelleninkompatibilitäten im Modell, die sich nur sehr

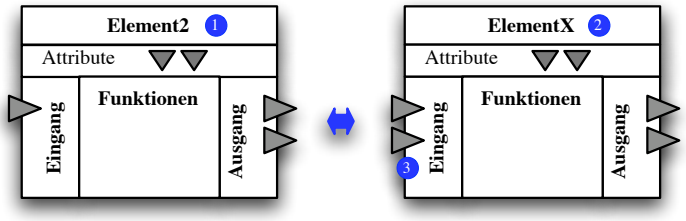


Abbildung 3.26.: Darstellung des Austausches des Elements *Element2* gegen das Element *ElementX*.

schwer eingrenzen und beseitigen lassen. Eine mögliche Lösung dieses Problems besteht in der Einführung spezieller optionaler Schnittstellen. Dann kann zwischen optionalen und zwingenden Schnittstellen zweifelsfrei unterschieden werden. Zusätzlich zur Einführung der optionalen Schnittstellen muss noch eine Regel eingeführt werden, die festlegt, dass im Gegensatz zu zwingenden Schnittstellen optionale Schnittstellen nicht angeschlossen sein müssen und so keine Fehler im Modell verursachen.

Ergebnis: Mit Hilfe des Element-Konzepts kann der statische Austausch eines Elements gegen ein anderes einfach modelliert und dargestellt werden. Sollte es bei dem Austausch zu einer statischen Inkompatibilität der Schnittstelle kommen, so kann diese nicht detektiert werden, weil dazu ein entsprechendes Regelwerk fehlt, in dem festgelegt ist, unter welchen Bedingungen zwei Schnittstellen kompatibel sind. Die vier Axiome des Element-Konzepts reichen hier bei weitem nicht aus.

Aufgrund des Fehlens einer eindeutigen Schnittstellenverhaltensbeschreibung kann eine mögliche Inkompatibilität, die beim Austausch eines Elements gegen ein anderes entsteht, nicht entdeckt werden. Dies liegt zum einen daran, dass es im Element-Konzept keine Möglichkeit gibt, das Schnittstellenverhalten eines Elements zu spezifizieren und zum anderen daran, dass es nicht möglich ist, das Transaktionsverhalten (Austauschverhalten) zwischen zwei Elementen zu spezifizieren. Beide Einschränkungen lassen sich jedoch umgehen, wenn zusätzlich das Schnittstellenverhalten, zum Beispiel mit einem Flussdiagramm oder MSC, explizit modelliert wird.

3.4.2.4. Einsetzbarkeit und Bewertung des SE Element-Konzepts für die Bestimmung der Kompatibilität von eingebetteten Systemen

Das Systems Engineering Element-Konzept kann besonders gut in der frühen Modellentwicklungsphase von technischen Systemen eingesetzt werden, in der unterschiedliche Domänenverantwortliche gemeinsam an einem Modell die Eigenschaften und Funktionen des Systems entwickeln und das System auslegen. Für die Modellierung und Bewertung von Kompatibilität ist das Element-Konzept jedoch nicht geeignet, da aufgrund seiner flexiblen Struktur nur wenige Eigenschaften des Element-Konzepts eineindeutig spezifiziert und festgelegt sind. Für die Kompatibilitätsmodellierung und -bewertung ist die Flexibilität des Element-Konzepts kontraproduktiv, da jeder Nutzer sein eigenes „Element-Konzept“ definieren kann. Somit ist es nahezu unmöglich die Elemente eines Systems auf Kompatibilität zu untersuchen. Wenn sich jedoch sämtliche Projektbeteiligte an eine gemeinsame formale Notation halten, kann das Element-Konzept für die Modellierung von statischer Kompatibilität eingesetzt werden. Für die Modellierung und Bewertung von Schnittstellenverhaltenskompatibilität reicht der vorhandene Sprachschatz des Element-Konzepts nicht aus. Zum einen fehlt das Konzept Zeit in der Element-Definition und zum anderen gibt es zum gegenwärtigen Zeitpunkt keine Beschreibungsmöglichkeit für die zeitliche Interaktion zwischen zwei Elementen. Beides sind jedoch zwingende Voraussetzungen für die Kompatibilitätsbestimmung eines eingebetteten Systems.

3.4.3. Die Unified Modelling Language – UML/UML2

Die UML/UML2 ist eine in der Informatik weit verbreitete graphische Modellierungssprache für Softwaresysteme. Der besondere Fokus der UML/UML2 liegt dabei auf der vereinheitlichten graphischen Modellierung und Beschreibung von Softwarestrukturen sowie der Modellierung des Verhaltens der Software. Die erste Version der UML entstand in den 1990er Jahren bei dem Versuch, eine vereinheitlichte graphische Modellierungssprache für die aufkommende objektorientierte Softwareentwicklung zu etablieren. Im Jahre 2005 wurde die UML 1.x durch die neue, grundlegend überarbeitete Version UML2 abgelöst. Für die Weiterentwicklung, Wartung und Pflege der UML/UML2 ist heute maßgeblich die Dachorganisation OMG[157] verantwortlich.

Aufbau und Struktur der UML/UML2

Für die Modellierung von Softwaresystemen stellt die UML/UML2 dem Benutzer zahlreiche Diagrammarten zur Verfügung, mit deren Hilfe die unterschiedlichen Aspekte des zu modellierenden Softwaresystems beschrieben und dargestellt werden können. Ein weiterer Schwerpunkt der UML/UML2 liegt in der automatischen, werkzeugunterstützten Generierung von Quellcode aus den unterschiedlichen Diagrammarten des Systems bzw. dem umgekehrten Fall, der automatischen Generierung von UML/UML2-Diagrammen aus einer im Quellcode vorliegenden Software.

Die Abbildung 3.27 auf der nächsten Seite zeigt die unterschiedlichen in UML/UML2 vorhandenen Diagrammarten. Mit Hilfe dieser Diagrammarten ist es möglich Softwaresysteme vollständig zu modellieren. Dabei unterteilen sich die Diagrammarten der UML/UML2 in zwei Bereiche: die *Strukturdiagramme*, mit deren Hilfe der Aufbau und die Struktur eines Systems beschrieben werden kann und den *Verhaltensdiagrammen*, durch die das Verhalten des Systems modelliert wird.

[157] Das Akronym OMG steht für **O**bject **M**anagement **G**roup, ein internationales Konsortium, das Standards für die objektorientierte Programmierung entwickelt und spezifiziert [OMG07a].

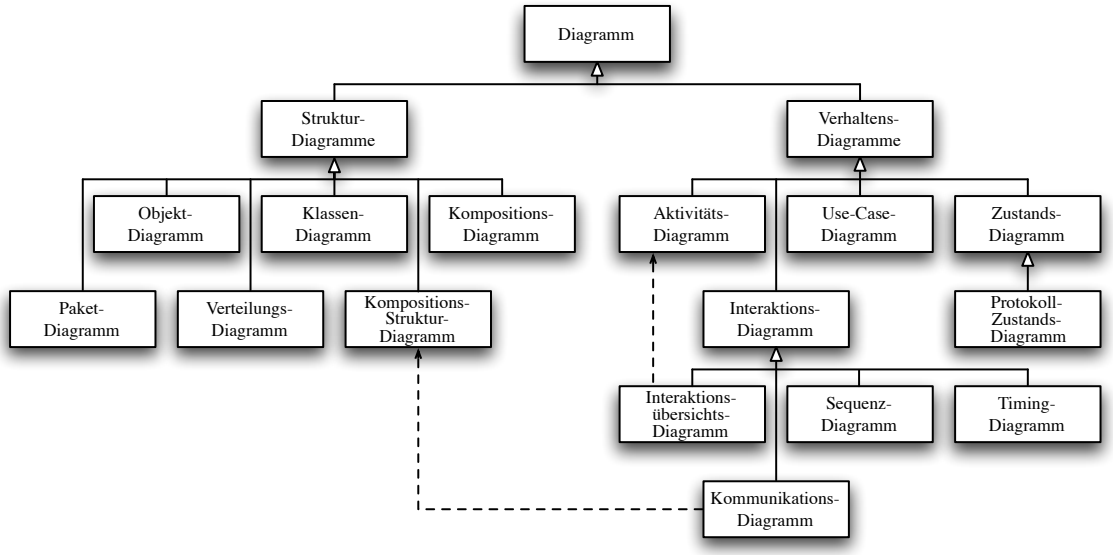


Abbildung 3.27.: Aufbau und Struktur der UML/UML2 nach [CS04, 30].

Für die Modellierung und Bestimmung von Kompatibilität – insbesondere von eingebetteten Systemen, die aus Hard- und Software bestehen, sind vor allem das *Klassendiagramm*, das *Objektdiagramm* sowie das *Kompositionsstrukturdiagramm* für die Modellierung der Struktur des Systems entscheidend. Für die Modellierung des Verhaltens des Systems wird vor allem das *Sequenzdiagramm* verwendet. Die hier aufgeführten UML/UML2-Diagramme werden in den nachfolgenden Abschnitten dieses Kapitels erläutert, mit dem Fokus auf der Modellierung von Hard- und Software sowie der Bestimmung von Kompatibilität.

Anmerkung:
Eine ausführliche Beschreibung der Softwaremodellierungssprache UML/UML2 finden Sie unter [OMG07c], [CS04], [PDB05], [Gro06b], [Gro06a], [Gro06c] sowie [BRJ98].

3.4.3.1. Die wichtigsten Diagrammarten der UML/UML2 für die Modellierung von Kompatibilität

In diesem Abschnitt werden die wesentlichen Diagrammarten der UML/UML2 eingeführt und anhand von einfachen Beispielen erläutert, die für die Modellierung von technischen Systemen, bestehend aus Hard- und Software, benötigt werden. Dabei wird besonders auf die Sprachbestandteile Wert gelegt, die für die Modellierung und die Bewertung von Kompatibilität benötigt werden.

Begonnen wird die Einführung der wichtigsten Diagrammarten der UML/UML2 mit der Beschreibung der *Strukturdiagramme*. Im darauf folgenden Abschnitt werden die *Verhaltensdiagramme* vorgestellt, die für die Modellierung des Verhaltens eines Systems benötigt werden.

3.4.3.1.1. UML/UML2-Strukturdiagramme

Für die Modellierung der Struktur von Softwaresystemen bietet die UML/UML2 zahlreiche *Strukturdiagramme* wie z.B. das *Objekt-*, *Klassen-*, *Kompositions-*, *Paket-*, *Verteilungs-* oder das *Kompositionsstrukturdiagramm*[158] an. Für die Modellierung von Softwaresystemen reichen jedoch im Allgemeinen das *Klassen-* bzw. das *Komponentendiagramm* vollkommen aus. Die restlichen Diagrammarten dienen der Konkretisierung bzw. Ergänzung der anderen Diagramme.

Nachfolgend wird zuerst das Klassen- bzw. Objektdiagramm erläutert. Daran anschließend folgt die Beschreibung des Paket- und Komponentendiagramms.

Klassen- und Objektdiagramm

Die wohl wichtigste Diagrammart der UML/UML2 ist das *Klassendiagramm*. Mit Hilfe des Klassendiagramms werden die strukturbildenden Bestandteile – die Klassen – eines Systems spezifiziert. Dabei bilden die Klassen „Schablonen" aus denen beliebig viele Objekte mit gleichen Eigenschaften gebildet werden können. Die Abbildung 3.28 auf der nächsten Seite (links) zeigt den allgemeinen Aufbau einer UML/UML2-Klasse (a). Im Klassenkopf wird der *Name* der Klasse eingetragen. Im darunter liegenden Abschnitt folgt die Definition der *Attribute* (Eigenschaften) der Klasse. Abgeschlossen wird die Klassendefinition mit der Sektion Operationen. Hier werden sämtliche *Operationen* (Methoden), die die Klasse zur Verfügung stellt, eingetragen.

Auf der rechten Seite der Abbildung 3.28 (b) ist ein *Objekt* einer Klasse dargestellt. Ein *Objekt* wird in UML/UML2 ähnlich wie eine Klasse spezifiziert. Im Objektkopf wird der *Name* des Objekts gefolgt von einem Doppelpunkt sowie dem Klassennamen eingetragen, von dem das Objekt *instantiiert* wurde. Zusätzlich ist der gesamte Name unterstrichen dargestellt, um anzuzeigen, dass es sich um ein Objekt handelt[159].

[158] Vgl. Abbildung: 3.27 auf Seite 132. (links)

[159] In UML/UML2 ist es auch zulässig, ein Objekt nur durch einen Doppelpunkt gefolgt vom unterstrichenen Klassennamen zu beschreiben (:Klassenname), wenn es unerheblich ist, wie das Objekt heißt („Anonymes Objekt")[CS04, 71].

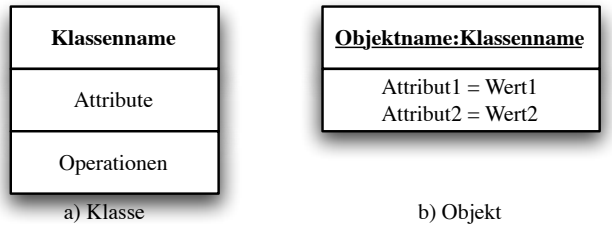


Abbildung 3.28.: UML/UML2-Klasse und Objekt (nach [CS04, 73] und [PDB05, 12]).

Ein Objekt entsteht aus einer Klasse, wenn diese *instantiiert* wird[160]. Dabei entsteht aus der allgemeinen Klassendefinition ein konkretes Objekt mit einem eineindeutigen Namen. Des Weiteren wird bei der Instantiierung der Klasse jedem Attribut der Klasse (genauer des Objekts) ein konkreter Wert zugewiesen. Hat beispielsweise die Klasse *A* das Attribut *Größe*, dann wird bei der Instantiierung der Klasse *A* dem Attribut *Größe* des Objekts *myObjekt : A* beispielsweise der Wert *Größe = 10* zugewiesen. Außerdem besitzen alle Objekte, die aus einer Klasse gebildet (instantiiert) werden die selben Operationen.

Sichtbarkeit von Attributen und Operationen
Für alle *Attribute* und *Operationen* einer Klasse kann die *Sichtbarkeit* definiert werden, anhand der festgelegt wird, ob und wie die Attribute oder Operationen der Klasse von außerhalb angesprochen werden können.

- **Öffentlich (public)**:
  Ist ein Attribut oder eine Operation einer Klasse als *öffentlich* deklariert, so können alle anderen Klassen, bzw. ihre Operationen ohne Einschränkung darauf zugreifen. Öffentliche Attribute und Operationen werden in UML/UML2 mit dem Sonderzeichen `+` eingeleitet. `+ Attribut1` bedeutet demnach, dass alle Klassen/Objekte beliebig auf das Attribut `Attribut1` zugreifen können.
- **Privat (private)**:
  Ist ein Attribut oder eine Operation einer Klasse als *privat* deklariert, so kann keine andere Klasse auf dieses Attribut/Operation zugreifen. Die Deklaration von privaten Attributen/Operationen werden in UML/UML2 mit dem Sonderzeichen `-` eingeleitet.
- **Geschützt (protected)**:
  Eine Sonderform der Sichtbarkeit wird durch das Sonderzeichen `#` (*geschützt*) eingeleitet. Solche Attribute/Operationen sind nur für von der Klasse *abgeleitete* oder die Klasse *besitzende* Klassen möglich (vgl. nächsten Abschnitt).

Mit den unterschiedlichen Sichtbarkeiten von Attributen und Operationen einer Klasse wird die so genannte *Kapselung* realisiert[161]. Mit Hilfe der Kapselung ist es möglich, die öffentliche Schnittstelle einer Klasse zu beschreiben[162]. Nachdem Klassen, Objekte sowie die Sichtbarkeit von Attributen und Operationen eingeführt wurden, folgt nun die Verbindung einzelner Klassen zu größeren Strukturen.

Verbindungsarten zwischen Klassen
In UML/UML2 können einzelne Klassen zu Verbünden zusammengefasst werden. Dazu bietet die UML/UML2 verschiedene Verbindungsarten mit unterschiedlichen Eigenschaften an. In der Abbildung 3.29 sind die vier gebräuchlichsten Verbindungsarten der UML/UML2 dargestellt.

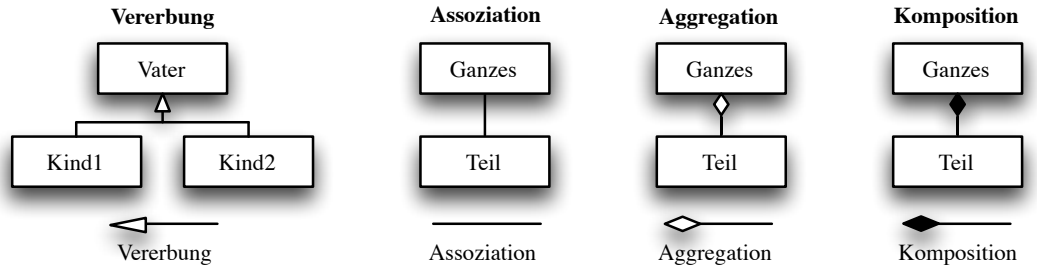


Abbildung 3.29.: Verbindungsarten der UML/UML2.

Erläuterung der unterschiedlichen Verbindungsarten zwischen Klassen aus der Abbildung 3.29:

- **Vererbung**
  Mit Hilfe der *Vererbung* werden in UML/UML2 Vater – Kind Beziehungen modelliert. Ein Vater gibt dabei seine, ihn bestimmenden öffentlichen sowie seine als geschützt spezifizierten Attribute an seine Kinder weiter. Jedes Kind kann zusätzliche Attribute und Operationen besitzen, die der Vater nicht besitzt. Aus diesem Grund wird die Vererbung oft

[160] Vgl. Abschnitt „2.3.1.1 *Klassen und Objekte eines Systems*“ ab Seite 46.
[161] Vgl. „Definition: C.9“ auf Seite 329..
[162] Siehe hierzu den Abschnitt „Schnittstellen von Klassen“ auf Seite 135.

auch als *Verfeinerung* bezeichnet. Des Weiteren ist es in UML/UML2 möglich, öffentliche und geschützte Attribute und Operationen der Vaterklasse in einer Kindklasse zu überschreiben. Dadurch bleibt die statische Schnittstelle der Klasse unverändert erhalten, lediglich das Verhalten der Klasse kann sich dadurch ändern.

Die Vererbung ist die wohl wichtigste Eigenschaft aller objektorientierten (Programmier-) Sprachen. Mit ihrer Hilfe ist es möglich, bereits existierende Klassen weiter zu verwenden und um fehlende/neue Attribute und Operationen zu erweitern, um das darin enthaltene Wissen weiter nutzen zu können. Die Vererbung von Attributen und Operationen ist auch für die Kompatibilitätsmodellierung von entscheidender Bedeutung. Da heutzutage kaum noch Neuentwicklungen durchgeführt werden, sondern im Allgemeinen lediglich Änderungen an bereits existierenden Klassen vorgenommen werden, kann durch die konsequente Nutzung der Vererbung sowohl das in den Klassen enthaltene Wissen als auch die statische (Grund-) Schnittstelle der Klassen von Generation zu Generation unverändert übernommen werden[163]. Alle neu hinzugekommenen Attribute und Operationen können somit als „optional“ und somit als nicht zwingend notwendig für die Benutzung der Klasse angesehen werden.

- **Assoziation**
  Soll in UML/UML2 eine einfache Verbindung zwischen zwei Klassen dargestellt werden, so kann dies mittels einer *Assoziation* modelliert werden. Assoziationen sind in UML/UML2 die einfachste Verbindungsart. Sie werden in UML/UML2 hauptsächlich zur Darstellung einer *losen/lockeren* Beziehung zwischen zwei Klassen verwendet.
- **Aggregation**
  Mit Hilfe der Aggregation wird eine schwache Teil – Ganzes – Beziehung modelliert. Das heißt, dass die beteiligten Partner auch alleine, ohne den jeweiligen anderen existieren können.
- **Komposition**
  Die Komposition stellt die „starke“ Form der Aggregation dar. Das Ganze kann hier nicht ohne seine Teile und umgekehrt existieren.

Zusätzlich zur rein graphischen Darstellung der Verbindung kann jede Verbindung zwischen zwei Klassen mit einer erläuternden textuellen Beschreibung versehen werden, die die Art der Verbindung genauer spezifiziert. Außerdem kann in UML/UML2 jedes Attribut bzw. jede Verbindung mit einer *Kardinalität* versehen werden, die zum Beispiel angibt, wie viele Objekte aus dieser Klasse gebildet werden dürfen. Die Tabelle 3.6 beschreibt, wie mit Hilfe der unterschiedlichen Kardinalitäten die Häufigkeit einer Klasse oder eines Attributs der Klasse quantisiert und festgelegt werden kann.

Anzahl in UML/UML2-Notation	Bedeutung
1	Exakt 1 (die Vorgabe)
2	Exakt 2
1..3	Zwischen 1 und 3 (inklusive 1 und 3)
3, 5	Entweder 3 oder 5
1..*	Mindestens 1 und ansonsten unbeschränkt
*	Unbeschränkt, einschließlich der 0 für kein Element
0..1	Entweder 0 oder 1, also auch kein Element erlaubt

Tabelle 3.6.: UML/UML2-Notation verschiedener Zählweisen – Kardinalitäten [CS04, 66].

Mit Hilfe der in der Tabelle 3.6 eingeführten Notation kann z.B. für das Attribut `age: Integer [0..1]` festgelegt werden, dass das Attribut optional ist, also keinmal bzw. maximal einmal vorhanden sein darf.

Beispiel 37: Verbindungsarten und Kardinalitäten der UML/UML2
Ein Haus besteht im Allgemeinen aus vielen Zimmern. Ein Zimmer kann ohne ein Haus nicht existieren. Aus diesem Grund wird das Verhältnis zwischen einem Haus und einem Zimmer mit Hilfe der *Komposition* – der starken Bindung – modelliert[164]. Außerdem besteht ein Haus aus mindestens einem und maximal unendlich vielen Zimmern[165]. Die Kompositionsbeziehung zwischen dem Haus und seinen Zimmern wird durch die Kardinalität $1...*$ festgelegt.

Im Gegensatz zur starken Bindung zwischen dem Haus und seinen Zimmern kann ein Stuhl, der sich in einem Zimmer befindet, auch ohne das Zimmer existieren. Aus diesem Grund wird hier die *Aggregation* – die schwache Bindung – zur Modellierung dieses Zusammenhangs verwendet. Im Beispiel kann jedes Zimmer des Hauses kein bis maximal vier Stühle enthalten $(0...4)$.

Die Abbildung 3.30 auf der nächsten Seite zeigt auf der linken Seite eine stark vereinfachte technische Zeichnung eines Hauses, das im Wesentlichen aus drei Zimmern besteht. In der Bildmitte ist das Modell des Hauses inklusive der maximal darin enthaltenen Stühle dargestellt. Auf der rechten Seite der Abbildung ist das entsprechende Klassenmodell abgebildet, das die Beziehungen zwischen dem Haus und den Zimmern sowie den Zimmern und den darin befindlichen Stühlen beschreibt. □

Kommentarfelder
In UML/UML2 ist es zur Verbesserung der Lesbarkeit bzw. Verständlichkeit von Diagrammen möglich, textuelle Kommentarfelder einzufügen und beispielsweise an eine Klasse oder eine Verbindung zu heften. Die Abbildung 3.31 auf der nächsten Seite zeigt den Aufbau eines einfachen UML/UML2-Kommentarfelds.

[163] Für die Modellierung von Kompatibilität ist bei einer Änderung eines Attributs/Operation einer Klasse darauf zu achten, dass dadurch das äußere Verhalten der Klasse nicht verändert wird, da es ansonsten zu einer Verhaltensinkompatibilität kommen kann. (vgl. Abschnitt „3.4.3.1.2 *UML/UML2-Verhaltensdiagramme*“ ab Seite 137 sowie [Koß09] und [Eck07]).
[164] Erweiterung des Beispiels aus [KE01].
[165] Dies ist nur eine theoretische Annahme. In der realen Welt kann es kein Haus geben, das unendlich viele Zimmer hat.

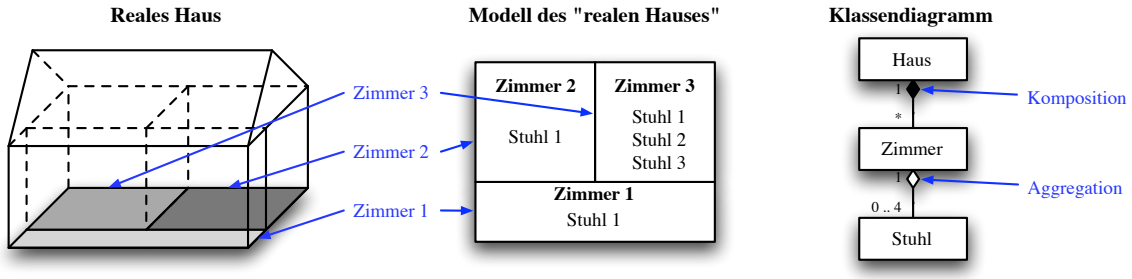


Abbildung 3.30.: Unterschied Aggregation – Komposition sowie Kardinalität der Verbindung.

Abbildung 3.31.: UML/UML2-Kommentarfeld.

Innerhalb des Kommentarfelds kann sowohl beliebiger Text als auch Formeln, Quellcodefragmente etc. hinterlegt werden. Kommentarfelder sind vor allem zur Beschreibung von „Randbedingungen“ wie z.B. zur Annotation von für die Modellierung/Anwendung der Klasse notwendigen Zusatzinformationen. Insbesondere können Kommentarfelder zur Beschreibung von kompatibilitätsrelevanten Eigenschaften verwendet werden.

Schnittstellen von Klassen
In UML/UML2 ist es möglich die öffentliche Schnittstelle einer Klasse explizit zu modellieren. Dabei unterteilt sich die öffentliche Schnittstelle der Klasse in zwei Bereiche. Zum einen in die von der Klasse *angebotene Schnittstelle*, in der sämtliche öffentlich zugänglichen Attribute und Operationen der Klasse enthalten sind, und zum anderen in die *benötigte Schnittstelle*. Darunter werden alle Eingaben, die die Klasse für ihr einwandfreies Verhalten (Funktionalität) benötigt, zusammengefasst.

Die Abbildung 3.32 zeigt auf der linken Seite die bereits in der Abbildung 3.28 (links) eingeführte Standardnotation einer Klasse in UML/UML2. Zusätzlich ist an der hier dargestellten Klasse die öffentliche *Schnittstelle* der Klasse in der so genannten „Lollipop-Notation“ angebracht. Das heißt: Die Klasse bietet eine dezidierte Schnittstelle und benötigt für ihr Verhalten eine Schnittstelle. In UML/UML2 ist es auch möglich, dass eine Klasse lediglich eine Schnittstelle anbietet oder benötigt.

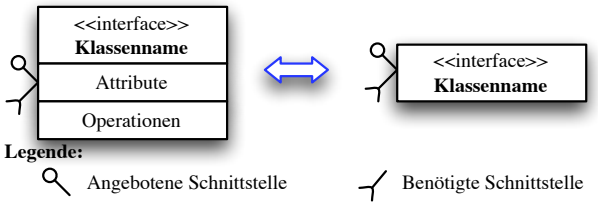


Abbildung 3.32.: UML/UML2-Klasse mit Schnittstellen.

Auf der rechten Seite der Abbildung 3.32 ist die selbe Klasse wie links in der vereinfachten Darstellung – also ohne Attribute und Operationen dargestellt. Auch hier ist wieder die öffentliche Schnittstelle der Klasse in der Lollipop-Notation angebracht.

Für die Modellierung und Bewertung von Kompatibilität reicht die Lollipop-Schnittstellennotation nicht aus, da lediglich die Art der Schnittstelle, also z.B. ob eine Schnittstelle angeboten oder benötigt wird spezifiziert werden kann. Die genaue Ausprägung der Schnittstelle kann jedoch nicht direkt hinterlegt werden, sondern muss über den Umweg eines zusätzlichen Kommentarfelds angegeben werden.

Beispiel 38: Lollipop-Schnittstellennotation einer Klasse mit zusätzlichem Kommentarfeld
In UML/UML2 kann jede Klasse mit einer Lollipop-Schnittstelle versehen werden. Für die Kompatibilitätsmodellierung und -bewertung reicht die so spezifizierte Schnittstelle nicht aus. Um dennoch die Lollipop-Schnittstellennotation für die Kompatibilitätsbestimmung nutzen zu können, muss diese mit einem zusätzlichen Kommentarfeld, in dem die dafür notwendigen Zusatzinformationen enthalten sind, versehen werden. In der Abbildung 3.33 auf der nächsten Seite ist links die Klasse *Rechteck* mit der dazugehörigen Lollipop-Schnittstellennotation dargestellt. Rechts daneben sind sowohl für die angebote, als auch für die benötigte Schnittstelle Kommentarfelder angegeben, in denen die notwendigen Zusatzinformationen für die Kompatibilitätsbestimmung enthalten sind. Die in den Kommentarfeldern hinterlegten Informationen gehen über die in der Klasse spezifizierten Attribute und Operationen hinaus. So ist z.B. das Attribut `laenge` der Klasse *Rechteck* mit dem Datentyp **`int`** versehen, jedoch ist weder der Gültigkeitsbereich[166] noch die Einheit angegeben.

[166] Hier ist nicht der Gültigkeitsbereich des Datentyps **`int`**, sondern der Gültigkeitsbereich des Attributs `laenge` gemeint. Beide unterscheiden sich im Allgemeinen stark voneinander.

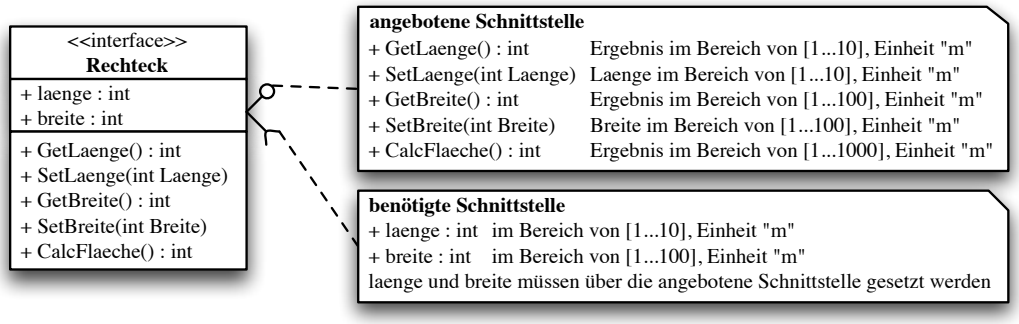


Abbildung 3.33.: UML/UML2-Klasse mit Lollipop-Schnittstelle und zusätzlichem Kommentarfeld.

Durch den „Umweg“ über die Kommentarfelder ist es möglich, sämtliche für die Kompatibilitätsbestimmung notwendigen Zusatzinformationen zu hinterlegen. Aufgrund der Tatsache, dass Kommentarfelder in UML/UML2 nur „Prosa“ sind, können sie nicht für die automatische Kompatibilitätsbestimmung verwendet werden. □

Paket- und Komponentendiagramm
UML/UML2-Klassendiagramme von realen Softwaresystemen können sehr groß und unübersichtlich werden. Um dieser Tatsache Rechnung zu tragen, können Modellelemente (Klassen, Objekt und Komponenten) in so genannte *Pakete* zusammengefasst werden. Durch die Zusammenfassung von Modellelementen in einem Paket entstehen abstrakte Teilsysteme, durch die die Übersichtlichkeit des Softwaresystems erheblich verbessern wird. Dabei kann auf zwei unterschiedliche Weisen vorgegangen werden. Zum einen können die Systembestandteile nach ihrer logischen Zusammengehörigkeit in Pakete zusammengefasst werden oder zum anderen nach ihrer „räumlichen“ Anordnung in einem System. Des Weiteren dürfen in UML/UML2 Pakete auch Pakete enthalten. So entsteht eine hierarchische Struktur eines Systems. Um ein Paket innerhalb eines Systems eineindeutig identifizieren zu können, hat jedes Paket innerhalb eines Modells einen eineindeutigen Namen.

Komponenten werden wie Pakete in UML/UML2 benutzt, um ein komplexes Softwaresystem in handhabbare Teilsysteme zu unterteilen. Im Gegensatz zu einem Paket kann mit Hilfe einer Komponente eine Softwareeinheit (Subsystem) gebildet werden, die über *Ports* und eine eigene dezidierte *Schnittstelle* verfügt. Komponenten eines Softwaresystems sollen im Allgemeinen so strukturiert werden, dass sie einfach, ohne Änderung der Ports bzw. Schnittstellen gegen eine andere Komponente ersetzt werden können. Aus diesem Grund schotten Komponenten ihren Inhalt gegenüber der Umwelt vollständig ab (Kapselung). Ein Zugriff auf die innere Struktur der Komponente ist nur über die Ports bzw. die Schnittstelle der Komponente möglich. Ein *Port* einer Komponente dient dabei als „Übergabepunkt“ zwischen der Komponente und der Umwelt der Komponente. An einem Port wird im Allgemeinen eine Schnittstelle in Lollipop-Notation angeschlossen. Über diese dezidierte Schnittstelle tauscht die Komponente Daten mit ihrer Umwelt aus. Komponenten können jedoch auch ohne Ports und Schnittstellen dargestellt werden, wenn die Schnittstelle in textueller Form innerhalb der Komponente spezifiziert ist.

Die Abbildung 3.34 zeigt links (a) ein *Paket*. Dieses wird in UML/UML2 als Rechteck mit einem „Reiter“ in der linken oberen Ecke dargestellt. In der Bildmitte (b) ist eine *Komponente* in Black-Box Darstellung mit einem *Port* und einer daran angeschlossenen *Lollipop-Schnittstelle* dargestellt. Auf der rechten Seite (c) ist die Komponente *Komponenten A* mit Inhalt, also in Glass-Box Darstellung, abgebildet. Hier ist zu erkennen, dass die Komponente *Komponente A* ihrerseits aus drei Subkomponenten besteht.

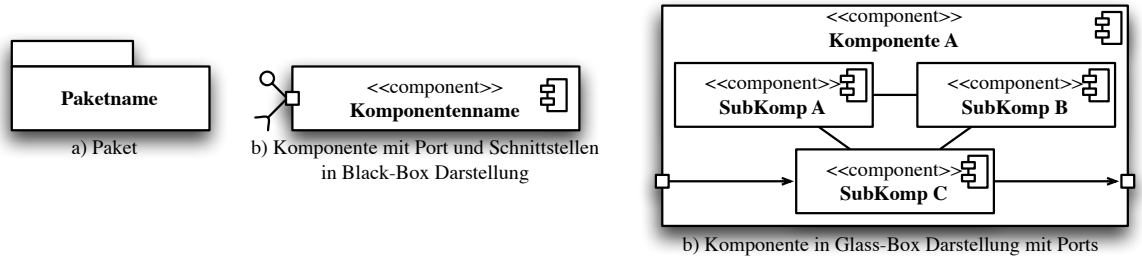


Abbildung 3.34.: UML/UML2-Paket und Komponente (nach [CS04, 147, 348]).

Zusätzlich zur graphischen Lollipop-Schnittstellennotation ist es in UML/UML2 möglich, die Schnittstelle einer Komponente auch in textueller Form innerhalb der Komponente zu beschreiben. Beide Varianten der Beschreibung von Schnittstellen sind gleichwertig, wie die Abbildung 3.35 auf der nächsten Seite illustriert.

Die Abbildung 3.35 zeigt auf der linken Seite eine Komponente mit zwei Schnittstellen *SS1* und *SS2*, wobei die Schnittstelle *SS1* von der Komponente *angeboten*, und die Schnittstelle *SS2 benötigt* wird. Auf der rechten Seite der Abbildung ist die selbe Komponente noch einmal – allerdings mit explizit modellierter Schnittstelle dargestellt. Diese Darstellungsart ist für die Modellierung bzw. die anschließende Bestimmung der Kompatibilität von Vorteil, da im Allgemeinen mehr Informationen wie z.B. die Signatur der Operationen, enthalten sind und diese für die Bestimmung der Kompatibilität herangezogen werden können.

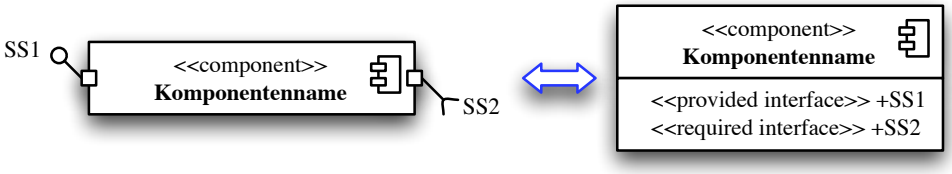


Abbildung 3.35.: UML/UML2-Komponente mit erweiterter Schnittstellendefinition (nach [CS04, 348]).

Wird bei der Entwicklung eines (Software-) Systems die Komponentenbildung konsequent eingesetzt, um einzelne Systembestandteile später einfach gegen andere austauschen/ersetzen zu können, so entspricht dies im Wesentlichen dem Entwicklungsparadigma, das hinter der Austausch- bzw. Ersetzungskompatibilität (vgl. „Definition: 1.14“ auf Seite 20) steht und bildet so die Grundlage für die Modellierung von Austausch- bzw. Ersetzungskompatibilität in UML/UML2.

In UML/UML2 gibt es weitere Möglichkeiten Schnittstellen zu definieren. Diese unterscheiden sich jedoch nicht wesentlich von den beiden hier vorgestellten Varianten und werden aus diesem Grund nicht weiter ausgeführt. Weitere Informationen zu Schnittstellen und Ports in UML/UML2 finden Sie unter [CS04, 449ff][OMG07c, 86ff, 447ff].

Kompositionsstrukturdiagramm
In UML/UML2 ist es erlaubt, Klassen in Klassen zu verpacken, wenn diese eine enge Koppelung wie z.B. bei einer Aggregation oder Komposition aufweisen. Solche Klassen in Klassen werden in einem Kompositionsstrukturdiagramm dargestellt. Das Kompositionsstrukturdiagramm wird benutzt, um zu zeigen, wie ein konkretes System/Klasse im Inneren strukturiert und aufgebaut ist und wie die Systembestandteile mit einander interagieren (verbunden sind). Die Abbildung 3.36 zeigt links ein einfaches Klassendiagramm, bestehend aus drei Klassen, die mittels strukturbildender Kompositionen verbunden sind. Rechts daneben ist das Kompositionsstrukturdiagramm des Systems dargestellt. In diesem Diagramm wird explizit der Zusammenhang zwischen den Klassen (genauer: den Objekten) dargestellt. Dabei können die explizit dargestellten Ports auch weggelassen werden.

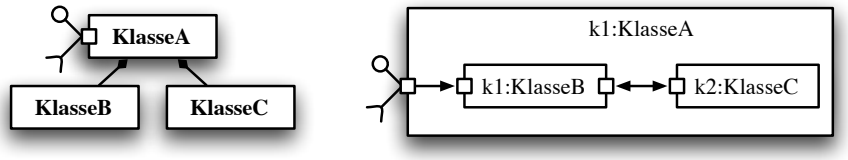


Abbildung 3.36.: Aufbau und Struktur des UML/UML2-Kompositionsstrukturdiagramms.

Mit Hilfe des Kompositionsstrukturdiagramms wird der konkrete innere Aufbau sowie die Struktur eines Systems exakt modelliert. Des Weiteren werden, wie auch bei der Komponentenbildung, die Ports und Schnittstellen des Systems dezidiert modelliert. Sowohl die Schnittstellen als auch die Ports können beide für die Kompatibilitätsmodellierung und -bestimmung herangezogen werden. Zusätzlich sollten die Schnittstellen jedoch mit Beschreibungsfeldern versehen werden, in denen weitere Informationen über die Attribute und Operationen des Systems spezifiziert sind. Vergleichen sie hierzu die Abbildung 3.33 auf der vorherigen Seite.

Anmerkung:
Durch die Einführung von Klassen, Objekten, Sichtbarkeiten, Hierarchie (Pakete) sowie der Vererbung von Eigenschaften von Klassen wurden die vier Grundeigenschaften der objektorientierten (Programmier-) Sprachen erfüllt (vgl. „Definition: C.9“ auf Seite 329). Somit handelt es sich bei UML/UML2 um eine objektorientierte (Programmier-) Sprache.

3.4.3.1.2. UML/UML2-Verhaltensdiagramme

Um das Verhalten eines Softwaresystems zu modellieren, bietet die UML/UML2 eine Vielzahl von unterschiedlichen Diagrammtypen an. In der Abbildung 3.27 auf Seite 132 (rechts) sind die unterschiedlichen Diagrammarten aufgelistet, die zur Modellierung des Verhaltens eines Systems verwendet werden können. Für die Modellierung von Kompatibilität ist vor allem der zeitliche Verlauf einer Interaktion interessant – also ob z.B. der Sender einer Nachricht diese rechtzeitig an den Empfänger versendet oder nicht. Um den zeitlichen Verlauf einer Aktion zu modellieren, bietet sich das *Sequenzdiagramm* bzw. *Timing Diagramm* an.

Mit Hilfe der unterschiedlichen Verhaltensdiagramme der UML/UML2 kann jedoch nicht nur das Verhalten eines Softwaresystems modelliert werden, sondern auch technische Systeme bestehend aus Hard- und Software. Für die Modellierung von technischen Systemen ist es allerdings erforderlich, die einzelnen Verhaltensdiagramme behutsam zu erweitern, z.B. um die Modellierung von kontinuierlichen Signalen (vgl. [Wei06]).

Sequenzdiagramm
In einem Sequenzdiagramm lässt sich die zeitliche Reihenfolge von Nachrichten modellieren jedoch nicht der exakte Zeitpunkt, wann eine Nachricht versandt oder empfangen wird. Die Abbildung 3.37 auf der nächsten Seite zeigt den grundsätzlichen Aufbau und die Struktur des UML/UML2-Sequenzdiagramms. Im Kopf des Sequenzdiagramms steht der Bezeichner *sd* gefolgt vom Diagrammnamen.

Innerhalb des Diagramms sind die am Nachrichtenaustausch beteiligten *Objekte* als Rechtecke dargestellt[167]. Zu jedem Objekt gehört genau eine *Lebenslinie*, die anzeigt, wie lange ein Objekt innerhalb des Diagramms aktiv („lebendig“) ist. Der zeitliche Verlauf innerhalb eines Sequenzdiagramms verläuft für ein Objekt stets von oben nach unten bzw. für eine Nachricht vom Pfeilanfang zur Pfeilspitze. Am linken bzw. rechten Rand des Diagramms werden die Nachrichten eingezeichnet, die in das Diagramm hinein bzw. aus dem Diagramm herausgeführt werden. Dies ist notwendig, damit nicht alle Objekte eines Systems in einem Sequenzdiagramm modelliert werden müssen, sondern damit Sequenzdiagramme ebenfalls (logisch) geschachtelt werden können.

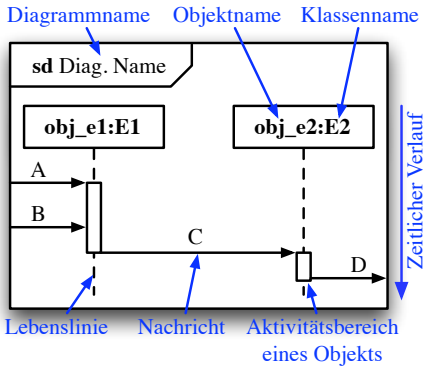


Abbildung 3.37.: Aufbau und Struktur des UML/UML2-Sequenzdiagramms.

Der wichtigste Bestandteil eines Sequenzdiagramms sind die *Nachrichten*[168], die zwischen den Lebenslinien der beteiligten Objekte fließen. Sie werden mittels gerichteter Pfeile vom Sender der Nachricht zum Empfänger gezeichnet. Jede Nachricht innerhalb eines Sequenzdiagramms muss mit einem Namen versehen werden, so dass die einzelnen Nachrichten innerhalb eines Diagramms eindeutig identifiziert werden können.

In einem Sequenzdiagramm ist es außerdem möglich, Nachrichten zwischen Objekten genauer zu klassifizieren. Die Klassifikation von Nachrichten ist vor allem für die Kompatibilitätsmodellierung und -bestimmung besonders wichtig. Zum Beispiel kann mit Hilfe eines alternativen Abschnitts innerhalb eines Sequenzdiagramms der Wegfall bzw. das hinzukommen von Nachrichten modelliert werden, ohne dass sich das Verhalten des ursprünglichen Grundsystems dadurch verändert. Dieser Sachverhalt ist in der Abbildung 3.38 genauer dargestellt.

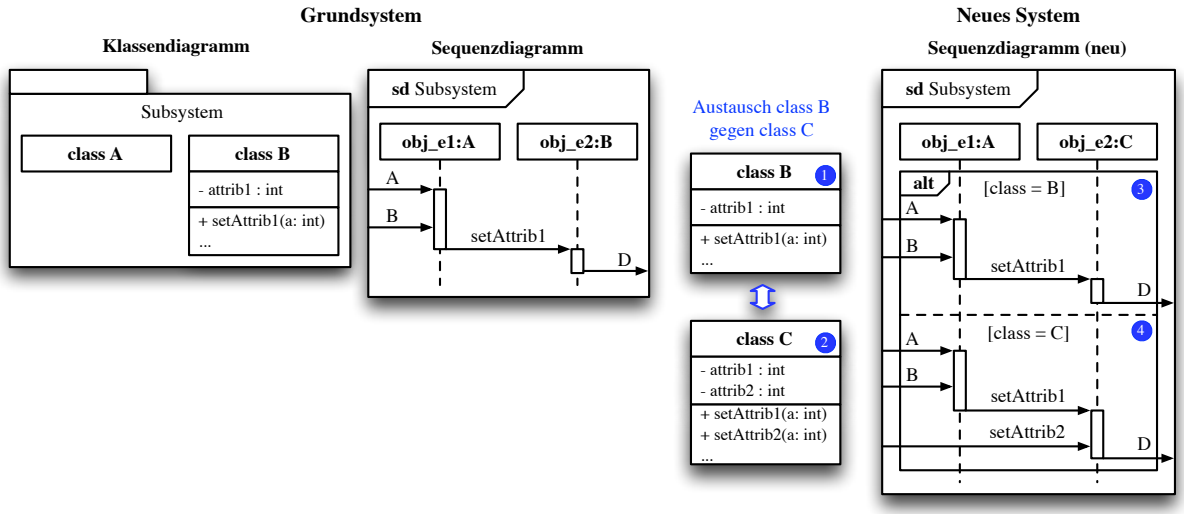


Abbildung 3.38.: Alternatives Verhalten in einem UML/UML2-Sequenzdiagramm.

Die Abbildung 3.38 zeigt auf der linken Seite das Grundsystem. Im Klassendiagramm sind die beiden essentiellen Klassen und im Sequenzdiagramm die Interaktion der beiden aus den Klassen gebildeten Objekte dargestellt. In der Abbildungsmitte ist der Austausch der Klasse *class B* (1) gegen die neue Klasse *class C* (2) mit einer gegenüber der *class B* veränderten Schnittstelle dargestellt[169]. Das Sequenzdiagramm auf der rechten Seite stellt die Interaktion der Klasse *class A* mit der Klasse *class C* dar. Im oberen Teil des Sequenzdiagramms (3) ist das selbe Verhalten wie im Sequenzdiagramm des Grundsystems auf der linken Seite dargestellt (vgl. Bedingung *class = B*). Im unteren Abschnitt (4) ist das Verhalten des Subsystems abgebildet, wenn zusätzlich der neu hinzugekommene Eingang der Klasse *class C* genutzt wird.

[167] Anmerkung: In einem Sequenzdiagramm muss mindestens eine Interaktion (Nachrichtenaustausch) zwischen zwei Objekten modelliert sein.
[168] Nachrichten werden oft als *Interaktionen* bezeichnet.
[169] Anmerkung: In der üblichen Lollipop-Notation für Schnittstellen von Klassen oder Komponenten ist nicht ersichtlich, dass sich die Schnittstelle der Klasse *class B* (1) gegenüber der Schnittstelle der Klasse *class C* (2) geändert hat.

Durch die Nutzung der Alternativen (*alt*) in Sequenzdiagrammen ist es möglich, das veränderte Verhalten eines Systems so zu beschreiben, dass in einem System unterschiedliche Klassen (Bausteine) verwendet werden können (Alternative (3) – Grundsystem, Alternative (4) – erweitertes System). Dies ist eine wichtige Voraussetzung für die direkte Modellierung von Austauschkompatibilität.

Anmerkung:
Eine Kurzbeschreibung der UML/UML2-Sequenzdiagramme finden Sie unter [Dum06]. Ausführliche Beschreibungen unter [OMG07c, 455ff], [CS04, 211ff] bzw. [Wei06, 288ff].

Timing Diagramm
Wie bereits erwähnt, kann in einem Sequenzdiagramm der zeitliche Ablauf von Nachrichten nicht exakt modelliert und beschrieben werden. Aus diesem Grund wurde das Standardsequenzdiagramm um eine weitere Unterart – das *Timing Diagramm* – erweitert [OMG07c, 515ff]. Mit Hilfe des *Timing Diagramms* kann die zeitliche Abfolge von Nachrichten mit dem Fokus auf das Eintreffen von Ereignissen exakt modelliert werden.

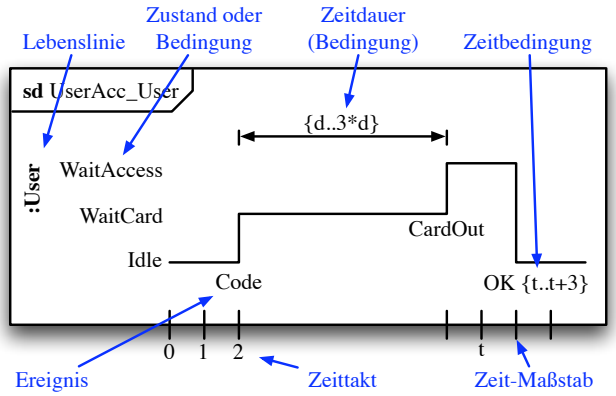


Abbildung 3.39.: Timing Diagramm nach [OMG07c, 517].

Die Abbildung 3.39 zeigt ein einfaches Timing Diagramm, in dem die wesentlichen Bestandteile des UML/UML2-Timing Diagrams eingetragen sind. Mit Hilfe des Timing Diagramms ist es möglich, ansatzweise kontinuierliche Signale zu modellieren.

3.4.3.2. Modellierung von Hardware in UML/UML2

Die UML/UML2 ist, wie in der Einleitung zu diesem Kapitel beschrieben, eine Modellierungssprache deren Fokus auf der Modellierung und Beschreibung von Softwaresystemen liegt. In UML/UML2 ist es jedoch auch möglich Hardware (also Elektrotechnik und Mechanik) zu modellieren. Um die Modellierung von Hardware in UML/UML2 zu realisieren, ist es notwendig, einige Grundkonzepte der UML/UML2 auf die beiden Domänen Elektrotechnik und Mechanik anzupassen. In diesem Abschnitt wird anhand eines einfachen Beispiels gezeigt, dass sich sowohl Elektrotechnik als auch Mechanik mit Hilfe von UML/UML2 beschreiben lässt.

Modellierung von Elektrotechnik in UML/UML2
In UML/UML2 kann mit Hilfe einer *Klasse* die Struktur und mit einem *Sequenz-* bzw. *Timing Diagramm* das Verhalten eines elektrischen/elektronischen Bauteils modelliert werden. Die Abbildung 3.40 zeigt auf der linken Seite das Blockschaltbild eines *Schmitt-Triggers* (vgl. hierzu auch die Abbildung F.4 auf Seite 343). Auf der rechten Seite der Abbildung ist der Schmitt-Trigger als UML/UML2-Klasse dargestellt, in der die wichtigsten Attribute sowie die Signatur der Operationen des Schmitt-Triggers eingetragen sind. Zusätzlich zum Klassendiagramm ist ein Kommentarfeld mit angefügt, in dem festgelegt wird, dass alle in der Klasse `Schmitt_Trigger` verwendeten Spannungen stets in Volt angegeben werden müssen.

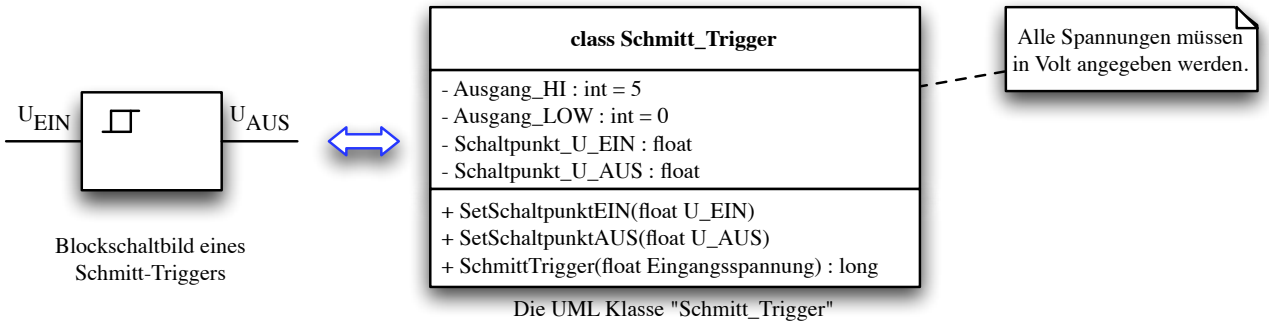


Abbildung 3.40.: Klassendiagramm des elektrischen Bauteils Schmitt-Trigger.

Um das Verhalten des Schmitt-Triggers zu beschreiben, bietet sich das UML/UML2-Sequenzdiagramm an. Das Sequenzdiagramm aus der Abbildung 3.41 zeigt die zeitliche Abhängigkeit des Ausgangssignals *U_AUS* vom Eingangssignal *U_EIN*. Für die Beschreibung des Verhaltens des Schmitt-Triggers reichen diese Informationen jedoch nicht aus, weil keine Informationen über die eigentliche Signalumwandlung hinterlegt sind. Um diese zu beschreiben, ist es notwendig, die Methode `SchmittTrigger(float Eingangsspannung):long` z.B. in einer Programmiersprache auszuformulieren. Das nachfolgende Listing zeigt eine mögliche, stark vereinfachte Implementierung der Methode `SchmittTrigger()` in einer stark vereinfachten C++ Notation.

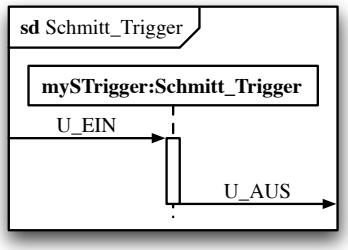


Abbildung 3.41.: Sequenzdiagramm des elektrischen Bauteils Schmitt-Trigger.

```
1 public long Schmitt_Trigger::SchmittTrigger(float Eingangsspannung)
2 {
3   if (Eingangsspannung > Schaltpunkt_U_EIN)
4     return Ausgang_HI;
5   else
6     return Ausgang_LOW;
7 }
```

Ergebnis: In UML/UML2 lassen sich einfache elektrotechnische Bauteile modellieren. In einem Klassendiagramm können die statischen Eigenschaften des Bauteils modelliert werden, während mit Sequenzdiagrammen das Verhalten spezifiziert werden kann. Zur Verhaltensbeschreibung und insbesondere zur Simulation des Bauteils ist zusätzlich die Ausformulierung der Operationen (Methoden) in einer Programmiersprache notwendig.

In UML/UML2 können jedoch nicht sämtliche Fälle modelliert werden. Wie bereits bei der Einführung der Sequenzdiagramme erwähnt, können kontinuierliche Signale in UML/UML2 nicht direkt dargestellt werden. Um dieses Problem in der Praxis zu umgehen, bietet sich die Einführung eines zentralen Zeitpunkts (Zeitschritts) an, zu dem alle Zustände des Systems neu berechnet werden.

Modellierung von Mechanik in UML/UML2

Um ein mechanisches Bauteil in UML/UML2 zu beschreiben, bietet sich das gleiche Vorgehen wie bei der Modellierung von elektrischen Bauteilen an. Zunächst wird die statische Struktur des Bauteils in eine Klasse mit Attributen und Operationen überführt und im Anschluss daran das Verhalten des Bauteils mittels eines Sequenzdiagramms bzw. von Quellcode beschrieben. In der Abbildung 3.42 (links) ist das Gehäuse eines Schmitt-Triggers dargestellt und rechts ist ein Ausschnitt aus dem dazugehörigen Klassendiagramm abgebildet.

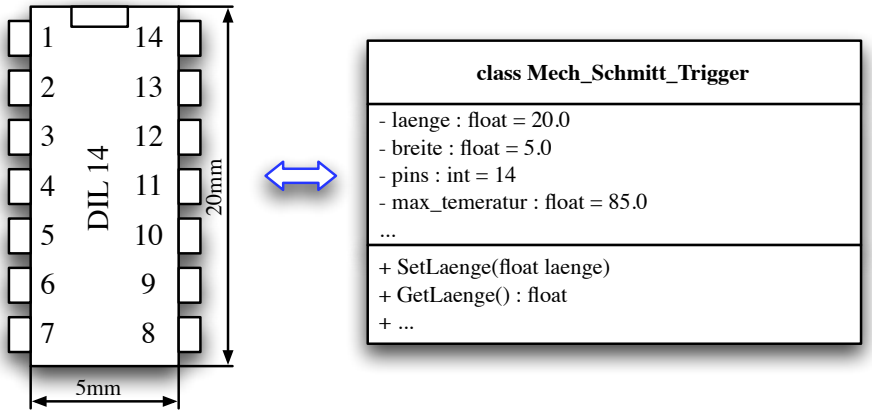


Abbildung 3.42.: Klassendiagramm des mechanischen Bauteils Schmitt-Trigger.

Ergebnis: In UML/UML2 können mechanische Baugruppen genauso modelliert und beschrieben werden wie elektrische Bauteile. Auch für die Modellierung der mechanischen Bauteile gelten die gleichen Einschränkungen wie für die elektrischen Bauteile.

3.4.3.3. Modellierung des Beispielsystems „KOMPTEST“ in UML/UML2

Nachdem in den vorangegangen Abschnitten dieses Kapitels die grundlegenden Diagrammarten der UML/UML2, sowie die Modellierung von elektrischen und mechanischen Bauteile in UML/UML2 vorgestellt wurden, wird nun das in der Einleitung vorgestellte Beispielsystem „KOMPTEST“ (vgl. Kapitel 3.3) wieder aufgegriffen und in UML/UML2 modelliert.

Im Szenario 1 wird zunächst die Struktur und das Verhalten des Grundsystems in UML/UML2 modelliert. Daran anschließend folgt im Szenario 2 die Modellierung des Austausches einer Klasse des Grundsystems gegen eine neue mit veränderter Schnittstelle.

Szenario 1: Modellierung der Struktur des Beispielsystems „KOMPTEST“
Im ersten Schritt der Modellierung des Beispielsystems „KOMPTEST“ wird zunächst das Klassendiagramm des Systems aufgestellt. In der Abbildung 3.43 ist das vollständige Klassendiagramm des Systems *System* inklusive einer beschränkten Auswahl an öffentlichen Attribute und Operationen (Methoden) der einzelnen Klassen dargestellt. Das System besteht aus fünf Klassen, wovon die beiden Klassen *SubElement1* und *SubElement2* in einem eigenen Paket *Subsystem* zusammengefasst sind. Durch die Zusammenfassung von einzelnen Klassen in einem Paket wird die hierarchische Struktur des Systems bzw. die logischen Zusammenhänge zwischen den einzelnen Systembestandteilen verdeutlicht. Im vorliegenden Fall bedeutet dies, dass das System auf der obersten Systemebene aus drei Klassen (*Element1, Element2* und *Element3*) und einem Paket (*SubSystem*) besteht. Ein weiterer Vorteil, der sich aus der hierarchischen Aufteilung des Systems ergibt, ist die leichtere Identifikation der Komponenten des Systems.

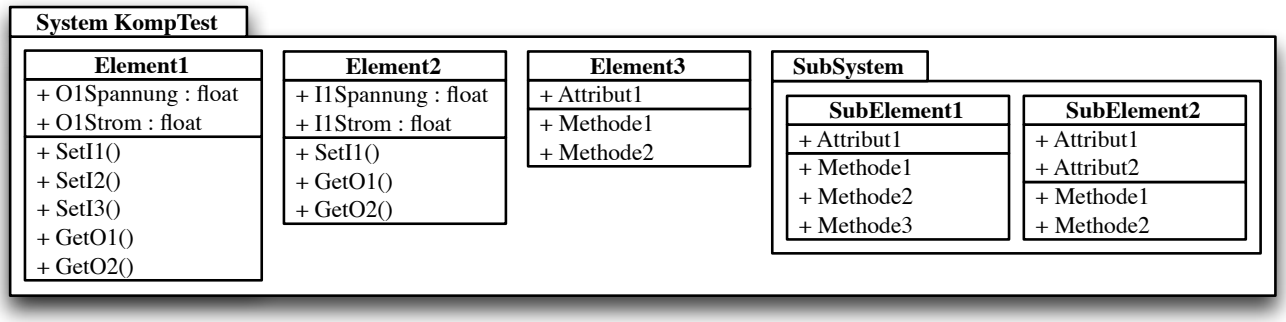


Abbildung 3.43.: UML/UML2-Klassendiagramm des Beispielsystems „KOMPTEST“.

Die Abbildung 3.44 zeigt die beiden Klassen *Element1* und *Element2* mit einem Auszug aus der Schnittstellendefinition in der Lollipop-Notation mit den für die Kompatibilitätsbestimmung notwendigen Kommentarfeldern, in denen die Signatur der Operationen, die verwendete Einheit sowie der Definitionsbereich der Attribute der Operationen textuell hinterlegt ist.

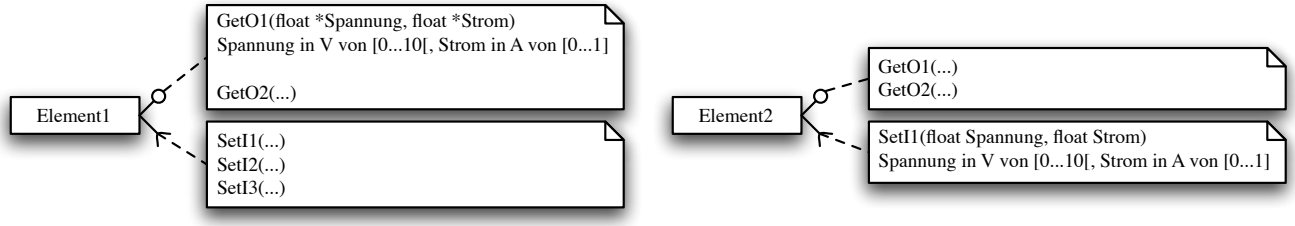


Abbildung 3.44.: UML/UML2-Klassendiagramm mit Schnittstellendefinition und Kommentarfeld der beiden Klassen *Element1* und *Element2*.

Um die Struktur des Beispielsystems „KOMPTEST“ vollständig zu modellieren, reicht die Modellierung der Klassen alleine nicht aus. Im Komponentendiagramm 3.45 auf der nächsten Seite (links) sind die beiden zentralen Komponenten des Beispielsystems, die Komponente *System* und die Komponente *SubSystem*, mit allen dazugehörigen Schnittstellen abgebildet. Des Weiteren sind der innere Aufbau und die Struktur der beiden Komponenten sowie die internen Zusammenhänge (Verbindungen) explizit dargestellt. Die explizit modellierten Verbindungen zwischen den Schnittstellen der Klassen sind vor allem für die Kompatibilitätsüberprüfung des Systems von entscheidender Bedeutung, da nur so verifiziert werden kann, ob die Schnittstellen zweier Klassen bzw. die Instantiierung einer Klasse oder ein Methodenaufruf (Signatur) übereinstimmen. Auf der rechten Seite der Abbildung sind zusätzlich die äußeren Schnittstellen der beiden Komponenten des Systems in Black-Box Darstellung dargestellt. In dieser Darstellungsform der Komponenten wird die Ausprägung der Schnittstelle der Komponente besonders hervorgehoben. Im Vergleich zur linken Darstellung kann in der Abbildung rechts die genaue Anzahl und die jeweilige Signatur der Schnittstellen der Komponenten entnommen werden. Dies ist links nicht der Fall. Dort ist lediglich die Existenz einer Schnittstelle dargestellt. Für die Modellierung von Kompatibilität ist sowohl die linke Abbildung als auch die rechte essentiell für die Beschreibung der Schnittstellen und der Verbindungen zwischen den Schnittstellen der beteiligten Klassen und Komponenten des Systems.

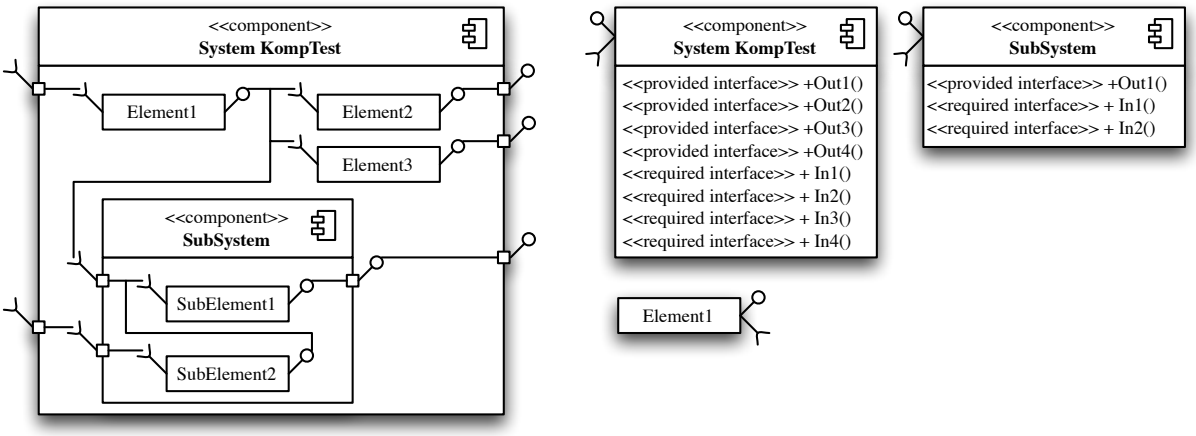


Abbildung 3.45.: UML/UML2-Komponentendiagramm mit Schnittstellen des Beispielsystems „KOMPTEST“.

Für die Kompatibilitätsbestimmung reicht die Beschreibung der Struktur der Schnittstellen des Systems jedoch noch nicht aus. Zusätzlich zur Definition der Schnittstellen (wie in Abbildung 3.45 gezeigt) müssen sowohl die Verbindungen (Operationsaufrufe) der Klassen als auch die konkreten technischen Werte im Modell hinterlegt werden. Aufgrund der Tatsache, dass weder in einem Klassen- noch in einem Komponentendiagramm die konkreten Werte hinterlegt werden können, muss nun das gesamte System instantiiert werden, um die konkreten technischen Werte in die entstandenen Objekte eintragen zu können. Die Abbildung 3.46 illustriert beispielhaft die Instantiierung der beiden Klassen *Element1* und *Element2*. Die Attribute der beiden Objekte enthalten nun die konkreten technischen Spezifikationen. Mittels konkreter Spezifikationen kann die statische Verbindung der beiden Objekte auf Kompatibilität hin untersucht werden.

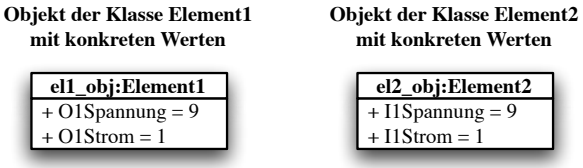


Abbildung 3.46.: Objekt der Klasse *Element1* und *Element2* mit konkreten Werten befüllt.

Zusätzlich zur graphischen Strukturmodellierung in einem Klassen- oder Komponentendiagramm ist für die Kompatibilitätsüberprüfung die Modellierung der Methoden- und Konstruktoraufrufe[170] der Klassen notwendig, um den korrekten Aufruf der jeweils angesprochenen Klassenschnittstelle zu überprüfen. In dem nachfolgenden Listing ist ein Auszug aus der Klasse *Element2* in C++ Notation dargestellt. Dabei wird die gerichtete Verbindung zwischen den beiden Klassen *Element1* und *Element2* auf Kompatibilität untersucht. Zu diesem Zweck holt sich der Konstruktor `Element2::Element2()` der Klasse *Element2* die Ausgabewerte der Klasse *Element1* über die öffentliche Methode `GetO1()` der Klasse *Element1* (Zeile 7) und vergleicht sie mit den maximal zulässigen internen Werte (Zeile 11). Stimmen die gelieferten Werte der Klasse *Element1* mit den internen maximal zulässigen Werten der Klasse *Element2* überein, so ist der Methodenaufruf (hier: Klasseninstantiierung) kompatibel.

```
1  public Element2::Element2()
2  {
3    float E1_Spannung, E1_Strom;
4    ...
5    // Aufruf der Methode GetO1 der Klasse Element1
6    el1_obj.GetO1(&E1_Spannung, &E1_Strom);
7    ...
8    // Vergleich der zulaessigen Werte mit den gelieferten Werten
9    if ((E1_Spannung <= I1Spannung) && (E1_Strom <= I1Strom))
10   {
11     // Eingabe OK
12     ...
13   }
14   else
15   {
16     // Kompatibilitätsfehler - Fehlerbehandlung anstossen
17   }
18 }
```

[170] Mit Hilfe eines Konstruktors wird in C++ eine Instanz der Klasse erzeugt. Nähere Informationen finden Sie unter [Str90, 35, 140, 324].

Um die Kompatibilität des Strukturmodells vollständig verifizieren zu können, ist ein Vergleich der Schnittstellendefinition aus Abbildung 3.44 mit den konkreten Werten, die in den Objekten gespeichert sind, notwendig (vgl. Abb.: 3.46). Außerdem müssen sämtliche Konstruktor- und Methodenaufrufe auf ihre Korrektheit hin überprüft werden (Listing). Erst wenn sämtliche Verifikationen des Strukturmodells erfolgreich abgeschlossen worden sind, ist das Strukturmodell kompatibel. Die Überprüfung der Werte und Methodenaufrufe im vorliegenden Fall ergibt, dass sowohl die konkreten Werte der Objekte innerhalb der Definitionsbereiche liegen als auch die Signaturen der Klassen übereinstimmen und sämtliche Methodenaufrufe kompatibel sind; die Schnittstellen des Strukturmodells sind somit kompatibel.

Ergebnis: Die Struktur des Beispielsystems lässt sich in UML/UML2 vollständig modellieren. Die für die Kompatibilitätsbestimmung wichtigen Schnittstellen der Klassen und Komponenten lassen sich nicht ausreichend genau spezifizieren, so dass es zu Inkompatibilitäten kommen kann, weil z.B. keine Einheiten oder Intervalle direkt spezifiziert werden können. Diese Einschränkung gilt auch für die Modellierung der Attribute und Operationen (Methoden) der Klassen. In UML/UML2 ist es jedoch möglich, Hilfsklassen zu modellieren, die den im Kapitel „2.3.2.2.2 *Verknüpfung von Datentyp, Präfix, Einheit und Gültigkeitsintervall*“ ab Seite 75 beschriebenen zusammengesetzten Datentypen entsprechen. Somit lassen sich zumindest mögliche Inkompatibilitäten zwischen unterschiedlichen Datentypen und Einheiten vermeiden. Der selbe Modellierungsansatz bietet sich auch für die Beschreibung der Intervalle an. Für Neuentwicklungen ist die Verwendung von Hilfsklassen mit relativ geringem Aufwand durchzuführen. Dies gilt jedoch nicht für bestehende Systeme, da z.B. sämtliche Bibliotheksschnittstellen neu entwickelt bzw. angepasst werden müssten.

Zusätzlich zur Schnittstellenüberprüfung ist eine Verifikation der Methodenaufrufe (Konstruktoren) des Systems notwendig um sicherzustellen, dass sämtliche Methodenaufrufe kompatibel zueinander sind. Auch hier gilt wieder die Einschränkung, dass ebenfalls Einheiten und Intervalle nicht direkt modelliert und auf Kompatibilität hin untersucht werden können. In diesem Fall können ebenfalls Hilfsklassen zur erweiterten Signaturbeschreibung der Operationen verwendet werden.

Szenario 1: Modellierung des Verhaltens des Beispielsystems „KOMPTEST“
Nachdem im ersten Modellierungsschritt das Klassendiagramm des Beispielsystems „KOMPTEST“ modelliert wurde, folgt nun die Beschreibung des Verhaltens des Systems. Um den Rahmen dieser Arbeit nicht zu sprengen, ist in der Abbildung 3.47 lediglich das Verhalten (Interaktion) der beiden Klassen (genauer: der Objekte *el1_obj:Element1* und *el2_obj:Element2*) beschrieben. Im Komponentendiagramm 3.45 ist die (statische) Verbindung der beiden Klassen *Element1* und *Element2* in Lollipop-Notation dargestellt. Das Sequenzdiagramm aus Abbildung 3.47 beschreibt die Reihenfolge, in der die einzelnen Nachrichten zwischen den beiden Objekten ausgetauscht werden.

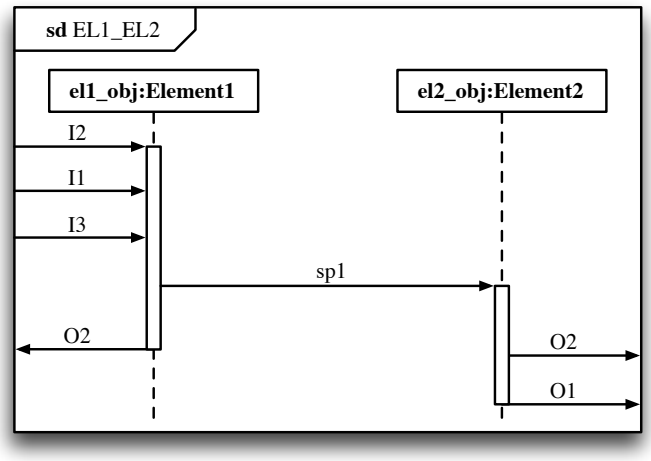


Abbildung 3.47.: UML/UML2-Sequenzdiagramm des Teilsystems Element1, Element2 des Beispielsystems „KOMPTEST“.

Im obigen Fall sendet das Objekt *el1_obj:Element1* die Nachricht *sp1* an das Objekt *el2_obj:Element2* nachdem es selbst die drei Nachrichten *I2, I1* und *I3* in dieser Reihenfolge erhalten hat. Das Objekt *el2_obj:Element2* wiederum sendet nach dem Eintreffen der Nachricht *sp1* vom Objekt *el1_obj:Element1* zwei Nachrichten *O2* und *O1* weiter.

Ergebnis: Mit Hilfe des Sequenzdiagramms kann das Verhalten des Beispielsystems und insbesondere die Reihenfolge der Nachrichten zwischen den einzelnen Systembestandteilen spezifiziert werden.

Szenario 2: Austausch der Komponente *Element2* gegen die neue Komponente *ElementX*
Im Szenario 1 wurde das UML/UML2-Klassen- bzw. Komponentendiagramm des Beispielsystems „KOMPTEST“ aus der Einleitung modelliert. Ausgehend von diesem Modell wird nun im Szenario 2 eine Komponente (Klasse) des Systems gegen eine andere mit einer modifizierter Schnittstelle ausgetauscht. Die Abbildung 3.48 auf der nächsten Seite illustriert den Austausch der Klasse *Element2* gegen die neue Klasse *ElementX*.

Im Klassendiagramm kann die veraltete Klasse *Element2* (1) leicht gegen die neue Klasse *ElementX* (2) ersetzt werden, ohne dass die anderen Klassen des Systempakets tangiert werden, weil im Klassendiagramm im Allgemeinen Verbindungen zwischen den Schnittstellen der einzelnen Klassen des Systems nicht explizit modelliert und dargestellt werden. Genauso verhält es sich im Komponentendiagramm. Hier sind zwar die Schnittstellen der Klassen eingetragen, jedoch in einer zu allgemeinen Form (Lollipop), so dass die veränderte Schnittstelle der neuen Klasse *ElementX* nicht weiter auffällt und es zu einer „versteckten“ Inkompatibilität

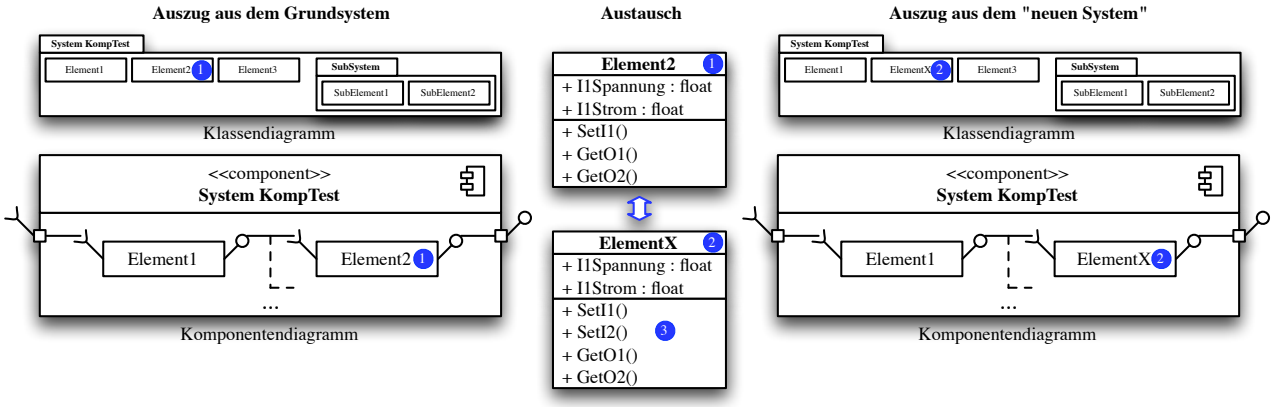


Abbildung 3.48.: Austausch der Klasse *Element2* gegen die Klasse *ElementX* des Beispielsystems „KOMPTEST“.

kommen kann. Diese Inkompatibilität tritt genau dann auf, wenn die Methode `SetI2()` der Klasse *ElementX* angesprochen wird (z.B. bei einem Konstruktor- oder Methodenaufruf). Handelt es sich bei der Methode `SetI2()` um eine optionale, also nicht zwingend für die Struktur bzw. das Verhalten der Klasse benötigte Methode, so hat der Austausch der Klasse *Element2* gegen die Klasse *ElementX* keine negative Auswirkung. Wird die neu hinzugekommene Methode `SetI2()` jedoch von der Klasse *ElementX* zwingend benötigt, kommt es zu einem Fehler. Dieser Fehler kann in UML/UML2 jedoch nicht (automatisch) identifiziert werden, da die Schnittstellendefinition nicht genau genug ist, um diese Art von Kompatibilitätsverletzung aufzuzeigen.

Ergebnis: Der Austausch einer Klasse in UML/UML2 kann zwar direkt modelliert werden jedoch kann es dabei zu erheblichen, im Modell unentdeckten Inkompatibilitäten kommen, da eine evtl. auftretende Schnittstelleninkompatibilität, die durch den Austausch einer Klasse entsteht, nicht entdeckt werden kann, ohne dass sämtliche Methoden jeder beteiligten Klasse einzeln betrachtet werden.

3.4.3.4. Einsetzbarkeit und Bewertung von UML/UML2 für die Bestimmung der Kompatibilität von eingebetteten Systemen

Mit Hilfe der Softwaremodellierungssprache UML/UML2 ist es möglich, eingebettete Systeme zu modellieren. Im Klassen- bzw. Komponentendiagramm kann die Struktur des Systems inklusive der für die Kompatibilitätsbestimmung wichtigen Schnittstellen der Klassen bzw. Komponenten des Systems beschrieben werden. Mit den unterschiedlichen Verhaltensdiagrammen und insbesondere dem Sequenzdiagramms kann das Verhalten eines Systems modelliert und beschrieben werden.

Für die Kompatibilitätsbestimmung ist die UML/UML2 dennoch ungeeignet, da wesentliche Anforderungen an eine domänenübergreifende Kompatibilitätsmodellierungssprache nicht erfüllt sind[171] wie z.B. die direkte Unterstützung von SI-Einheiten sowie die Modellierung von Intervallen. Außerdem ist es in UML/UML2 nicht möglich, die Schnittstellen zwischen Systembausteinen so zu modellieren, dass sie für die Kompatibilitätsbestimmung direkt benutzt werden können. Diese Problematik lässt sich zwar zum Teil durch die Benutzung von Kommentarfeldern, Hilfsfunktionen und zusätzlichen Datentypen abmildern, jedoch das grundsätzliche Problem bleibt dennoch bestehen. Ein weiterer Nachteil der UML/UML2 besteht in der mangelnden Unterstützung von kontinuierlichen Signalen. So ist es in UML/UML2 nicht möglich, z.B. elektrische Signale wie beispielsweise Strom direkt zu modellieren. Dieses Problem kann jedoch dadurch umgangen werden, dass z.B. Zeitschritte in Sequenzdiagrammen eingeführt werden. Dadurch lässt sich jedes kontinuierliche Signal als „kontinuierlicher Nachrichtenfluss“ modellieren.

Der größte Nachteil der UML/UML2 liegt jedoch in der mangelnden Unterstützung eines „Kompatibilitätsregelwerks“, in dem Regeln für die Kompatibilitätsbestimmung hinterlegt werden können sowie einer graphischen Repräsentation der Kompatibilitätsergebnisse. Beides sind jedoch wesentliche Bestandteile einer domänenübergreifenden graphischen Kompatibilitätsmodellierungssprache.

3.4.4. Die System Modelling Language – SysML

Die SysML[172] ist eine noch sehr junge Systemmodellierungssprache, deren Fokus auf der ganzheitlichen Modellierung komplexer technischer Systeme liegt. Insbesondere soll mit Hilfe der SysML das Systems Engineering Entwicklungsparadigma – die ganzheitlichen domänenübergreifende Betrachtung eines Systems von der (Vor-) Entwurfsphase bis hin zur Herstellung des Systems und darüber hinaus – erfasst und beschrieben werden. Zusätzlich zur Unterstützung bei der Produktentwicklung ist die SysML auch in der Lage, Anforderungen und Prozesse, die beide für die erfolgreiche Produktentwicklung essenziell sind, zu erfassen und graphisch zu modellieren. Durch die Integration sämtlicher Systems Engineering Grunddisziplinen in der SysML erlangt sie bereits jetzt eine große Bedeutung bei der Entwicklung von komplexen technischen Systemen.

Treibende Kräfte hinter der Entwicklung der SysML sind INCOSE, OMG sowie einige große Konzerne aus der Luft- und Raumfahrt, die sich hauptsächlich mit der Entwicklung komplexer technischer Systeme, die aus Hard- und Software bestehen, beschäftigen. Die beiden Organisationen INCOSE und OMG hingegen bemühen sich intensiv um die Unterstützung und Ausbildung von Ingenieuren bei der Systemmodellierung und der Standardisierung von (objektorientierten) Modellierungssprachen wie z.B. der SysML.

[171] Vgl. Abschnitt „3.2 *Anforderungen an eine Kompatibilitätsmodellierungssprache*“ ab Seite 109.
[172] Das Akronym SYSML steht für **Sys**tems **M**odelling **L**anguage.

Stammbaum der SysML
Die SysML ist aus der aus der Softwareentwicklung stammenden Unified Modelling Language – UML 2.1 – entstanden. Aufbauend auf den Grundeigenschaften der UML/UML2 wurden zusätzliche Diagrammarten mit der SysML eingeführt, mit deren Hilfe technische Systeme besser beschrieben werden können, als dies mit Hilfe der für die Softwareentwicklung konzipierten UML/UML2 möglich ist. Die Abbildung 3.49 zeigt den oben beschriebenen Zusammenhang zwischen der UML/UML2 auf der einen Seite und der SysML auf der anderen.

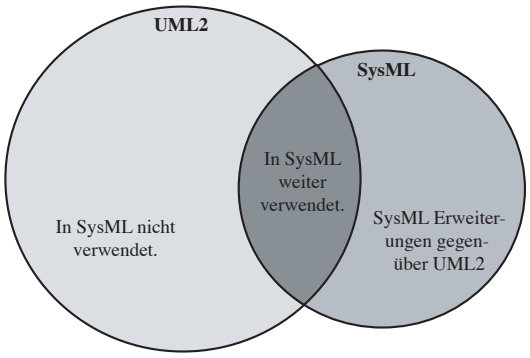


Abbildung 3.49.: Zusammenhang zwischen SysML und UML/UML2 nach [OMG, 7].

In den beiden folgenden Abschnitten diese Kapitels werden zunächst die wichtigsten Sprachelemente der SysML eingeführt und erläutert, die für die Modellierung technischer Systeme mit dem Fokus auf Kompatibilität notwendig sind. Daran anschließend wird ein konkretes System mit Hilfe der SysML modelliert und auf Kompatibilität untersucht.

3.4.4.1. Die wichtigsten Diagrammarten der SysML für die Modellierung von Kompatibilität

In diesem Abschnitt werden die wichtigsten Diagrammarten der SysML eingeführt, mit deren Hilfe ein technisches System, bestehend aus Hard- und Software, modelliert und dargestellt werden kann. Dabei werden vor allem die für die Modellierung von technischen Systemen notwendigen Diagrammarten für die *Strukturbeschreibung* bzw. die *Verhaltensmodellierung* kurz angerissen. Der Fokus der SysML-Einführung liegt dabei insbesondere auf der Beschreibung und Bestimmung von Kompatibilität von eingebetteten Systemen.
Eine ausführliche Beschreibung der Systemmodellierungssprache SysML finden Sie unter [Sys06][Wei06][PT] sowie eine knappe Einführung unter [FMS].

Aufbau und Struktur der SysML
Die Abbildung 3.50 zeigt den grundsätzlichen Aufbau sowie die Struktur der SysML. In dieser Darstellung sind sämtliche Diagrammarten, die die SysML unterstützt dargestellt. Zusätzlich zur Beschreibung der vorhandenen Diagrammarten der SysML ist in dieser Darstellung der „Stammbaum“ der einzelnen Diagrammarten eingetragen. So ist z.B. das *Sequenzdiagramm* unverändert von der UML/UML2 übernommen worden, während das *Block Definition Diagram* ein modifiziertes Klassen- bzw. Objektdiagramm der UML/UML2 ist.

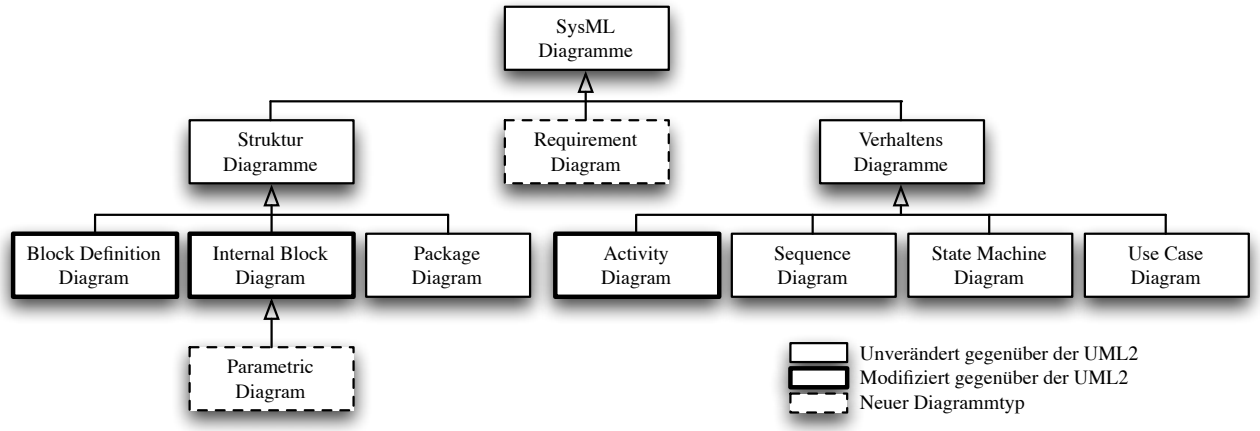


Abbildung 3.50.: Die Grundstruktur der SysML nach [OMG, 11].

Gegenüber der UML/UML2 wurde das *Klassendiagramm* in SysML in *Block Definition Diagram* umbenannt und dessen Eigenschaften an die Bedürfnisse des Systems Engineering angepasst. Ebenso wie das UML/UML2-Klassendiagramm wurde auch das *Kompositionsstrukurdiagramm* in SysML in *Internal Block Diagram* umbenannt und erweitert.

Mit Hilfe der in der Abbildung 3.50 dargestellten Diagramme ist es möglich, komplexe technische Systeme, die sowohl aus Hard- als auch aus Software bestehen, komplett zu modellieren.

Im Anschluss an den Überblick über die unterschiedlichen SysML-Diagrammarten folgt in den nächsten beiden Abschnitten zunächst die Einführung dreier Strukturdiagramme – *Block Definition Diagram*, *Internal Block Diagram* sowie das *Parametric Diagram*. Abgeschlossen wird die Einführung der SysML-Diagramme mit dem bereits im Abschnitt 3.4.3.1 eingeführten Sequenzdiagramm zur Modellierung des Verhaltens eines Systems.

3.4.4.1.1. SysML-Strukturdiagramme

Die Systemmodellierungssprache SysML stellt für die Modellierung der Struktur eines Systems vier unterschiedliche Diagrammarten zur Verfügung. Dabei wird das *Block Definition Diagram* zur abstrakten Beschreibung der Komponenten des Systems (in SysML-Blöcke genannt) sowie zur Modellierung der Zusammenhänge zwischen den Blöcken verwendet. Mit Hilfe des *Internal Block Diagrams* werden die Blöcke des *Block Definition Diagrams* verfeinert und ihre Interaktion beschrieben. Das *Package Diagram* dient zur Strukturierung des Systems. Abgerundet werden die Strukturdiagramme mit dem *Parametric Diagram*, in dem Bedingungen (auch Zusicherungen genannt) und Eigenschaften des Systems bzw. einzelner Blöcke definiert werden können.

In den nachfolgenden Unterpunkten werden die drei für die Modellierung von Kompatibilität wichtigsten Strukturdiagrammarten kurz angerissen.

Block Definition Diagram (bdd)[173]

In einem *Block Definition Diagram* wird die Struktur des Systems – also die Blöcke (Systembestandteile) und ihre Beziehungen untereinander – modelliert und dargestellt. Dazu werden weitgehend die gleichen Symbole verwendet, die auch im UML/UML2-Klassendiagrammen zum Einsatz kommen. Als wichtigste Änderung ist zu bemerken, dass die aus der UML/UML2 bekannte Klasse durch den SysML-Block ersetzt wurde. Im Vergleich zur Klasse sind die Eigenschaften des Blocks leicht unterschiedlich. Die Abbildung 3.51 zeigt ein einfaches *Block Definition Diagram*, in dem zwei Blöcke *Block1* und *Block2* dargestellt sind, die mittels einer Komposition verbunden sind. Zusätzlich ist an der Komposition die Kardinalität sowie die Bezeichnung/Beschreibung der Komposition angegeben.

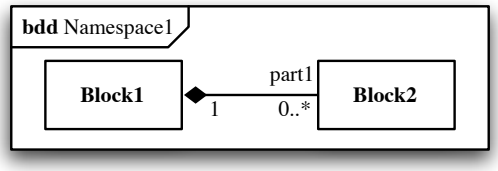


Abbildung 3.51.: SysML-Block Definition Diagram nach [OMG, 35].

Die Abbildung 3.52 zeigt den schematischen Aufbau sowie die Struktur eines Blocks in SysML. Dabei unterscheidet sich der SysML-Block von einer UML/UML2-Klasse im Wesentlichen durch drei neu hinzugekommene Felder – *constraints, parts* und *references*. Mit Hilfe der *constraints* können Bool´sche Bedingungen beschrieben werden, die gelten müssen, damit der Block verwendet werden darf. Die beiden Felder *parts* und *references* dienen zur strukturellen Beschreibung eines Systems (z.B. ein Block ist in einem anderen Block enthalten).

Die übrigen beiden Felder *operations* und *values* wurden aus der UML/UML2-Klassendefinition übernommen. Allerdings können im Moment keine Sichtbarkeiten für operations und values wie in UML/UML2 definiert werden. In SysML sind somit alle Einträge in einem Block stets öffentlich zugänglich (vgl. [OMG, 20]).

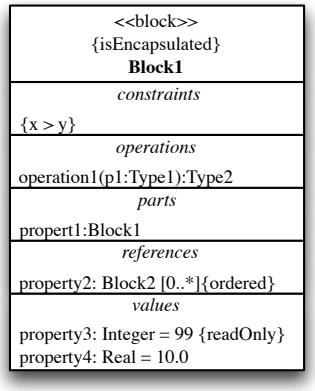


Abbildung 3.52.: Definition eines SysML-Blocks nach [OMG, 35].

[173] Im Deutschen wird das *Block Definition Diagram* mit *Block-Definitionsdiagramm* übersetzt.

In einem *Block Definition Diagram* können Blöcke als Rechtecke dargestellt werden, die nur den Namen des Blocks enthalten (vgl. Abbildung: 3.51 „Block1“) oder in einer ausführlichen Darstellung (vgl. Abbildung: 3.52). Auch Mischvarianten sind in einem *Block Definition Diagram* möglich. Ihre Verwendung ist abhängig davon, ob der Inhalt eines Blocks für dessen Verständlichkeit notwendig ist oder nicht.

Nachdem in einem *Block Definition Diagram* sämtliche Blöcke eines Systems festgelegt sind, können mit Hilfe des *Internal Block Diagrams* ihre Schnittstellen (in SysML-Ports genannt) modelliert und dargestellt werden.

Internal Block Diagram (ibd)[174]
Zusätzlich zur strukturellen Beschreibung eines Systems in einem *Block Definition Diagram* werden in einem *Internal Block Diagram* die Schnittstellen (Ports) zwischen den Blöcken eines Systems bzw. innerhalb eines Blocks genauer spezifiziert. In der Abbildung 3.53 ist ein *Internal Block Diagram* dargestellt. Es zeigt die interne Struktur des *Block1* aus dem *Block Definition Diagramm* aus der Abbildung 3.51.

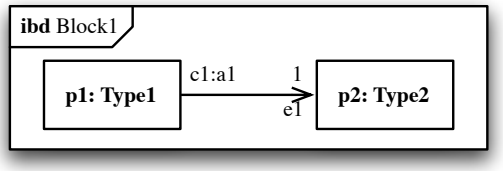


Abbildung 3.53.: SysML-Internal Block Diagram nach [OMG, 39].

In einem *Internal Block Diagram* wird zwischen zwei Arten von Ports an einem Block unterschieden. Die Abbildung 3.54 zeigt auf der linken Seite der Abbildung die bereits aus der UML/UML2 bekannte „Lollipop-Notation“ eines SysML-*Standard Ports*. Dabei repräsentiert der Kreis den Schnittstellenanbieter und der halbrunde Anschluss den Konsumenten einer bestimmten Schnittstelle.

Abbildung 3.54.: SysML-Ports [OMG, 57].

Ähnlich wie in UML/UML2 werden in dieser Darstellung der Schnittstelle keine weiteren Angaben über die Ausprägung der Schnittstelle gemacht (Methoden, Attribute, Zusammenhang zwischen Schnittstellenanbieter und Empfänger etc.). Es wird lediglich spezifiziert, dass der entsprechende Block eine Schnittstelle zur Verfügung stellt bzw. eine Schnittstelle zu einem anderen Block benötigt. Auf der rechten Seite der Abbildung 3.54 ist ein SysML-Block mit einem so genannten *Flow Port* abgebildet. Der *Flow Port* symbolisiert einen gerichteten Fluss von einem Port zu einem anderen. Dabei ist es unerheblich, was über den Port transportiert wird. Die Abbildung 3.55 zeigt einen gerichteten *Item Flow* vom Block *Engine* zum Block *Transmission*. Hier wird beispielsweise mechanische Kraft vom Block *Engine* zum Block *Transmission* übertragen.

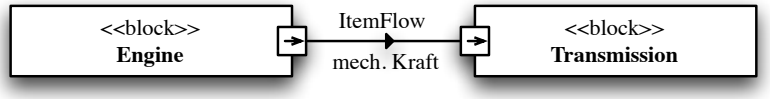


Abbildung 3.55.: SysML-Item Flow (nach [OMG, 58]).

Mit Hilfe der SysML-*Flow Ports* kann abstrakt der Fluss zwischen zwei Blöcken modelliert werden. Außerdem ist es möglich, die Art des Flusses genauer zu spezifizieren, als dies in der Lollipop-Notation möglich ist. Für die formale Kompatibilitätsbeschreibung des Flusses reicht dies jedoch bei weitem nicht aus, da zu wenig Information für die Kompatibilitätsprüfung vorhanden ist (siehe hierzu: Kapitel „3.4.4.2 *Modellierung des Beispielsystems „KompTest“ in SysML*“ ab Seite 150).

Um dieses Problem zu umgehen, bietet die SysML die Möglichkeit, zusätzliche Informationen aller Art textuell in einem Kommentarfeld zu hinterlegen. Dafür gelten allerdings die selben Einschränkungen, die auch für die UML/UML2-Kommentarfelder gelten.

Anmerkung:
In einem *Internal Block Diagram* können auch Interaktionen zwischen zwei Blöcken über beliebig viele Hierarchieebenen hinweg dargestellt werden, ohne dass dafür die gesamte Struktur des Systems dargestellt werden muss. Durch diese Art der vereinfachten Darstellung wird zwar das Konzept der Kapselung durchbrochen, jedoch ist diese Darstellungsart im Allgemeinen näher an

[174] Im Deutschen wird das *Internal Block Diagram* mit *Internes Blockdiagramm* übersetzt.

der Realität eines Systems[175] (vgl. [Wei06, 186]). Außerdem wird durch den Wegfall der expliziten Darstellung der Grenzen zwischen den einzelnen Hierarchieebenen die Kompatibilitätsbestimmung erheblich erschwert, da es für die Bestimmung der Schnittstellenkompatibilität zwischen Blöcken zwingend notwendig ist, eineindeutige Grenzen zwischen Blöcken zu modellieren. Dieser Nachteil bei der Modellierung eines Systems in SysML lässt sich jedoch leicht umgehen, indem in einem *Internal Block Diagram* stets alle Hierarchieebenen eingezeichnet werden, so dass nicht „versehentlich" Schnittstellen des Systems übersehen oder verletzt werden.

Parametric Diagram (par)[176]
Das *Parametric Diagram* ist eine Erweiterung des bereits beschriebenen *Block Definition Diagrams* (vgl. Abbildung: 3.51 auf Seite 146 sowie die Abbildung 3.53). Im Allgemeinen werden *Parametric Diagrame* in SysML zur formalen Definition von Bedingungen[177] bzw. während des Entwicklungsprozesses zur Auslegung von Blöcken (Baugruppen) und ihrem Zusammenspiel verwendet ([Wei06, 195ff]). Die Abbildung 3.56 zeigt ein einfaches SysML-Parametric Diagram zur Beschreibung von Bedingungen in SysML.

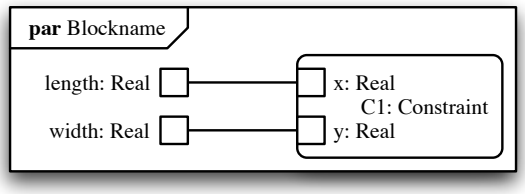


Abbildung 3.56.: SysML-Parametric Diagram nach [OMG, 73].

Bedingungen können jedoch nicht nur in einem *Parametric Diagram* modelliert werden, sondern bereits im *Block Definition Diagram*. Sollen jedoch ausschließlich Bedingungen modelliert werden, wird in der Regel ein *Parametric Diagram* verwendet.

Beispiel 39: Newtons Welt
Wie bereits oben erwähnt, können Zusicherungen sowohl im *Block Definition Diagram* als auch im *Parametric Diagram* modelliert und dargestellt werden. Das gezeigte Modell (Abbildung 3.57) ist der Legende nachempfunden, dass I. Newton durch den Fall eines Apfels auf die Formel der Gravitation ($f = m * a$) stieß.

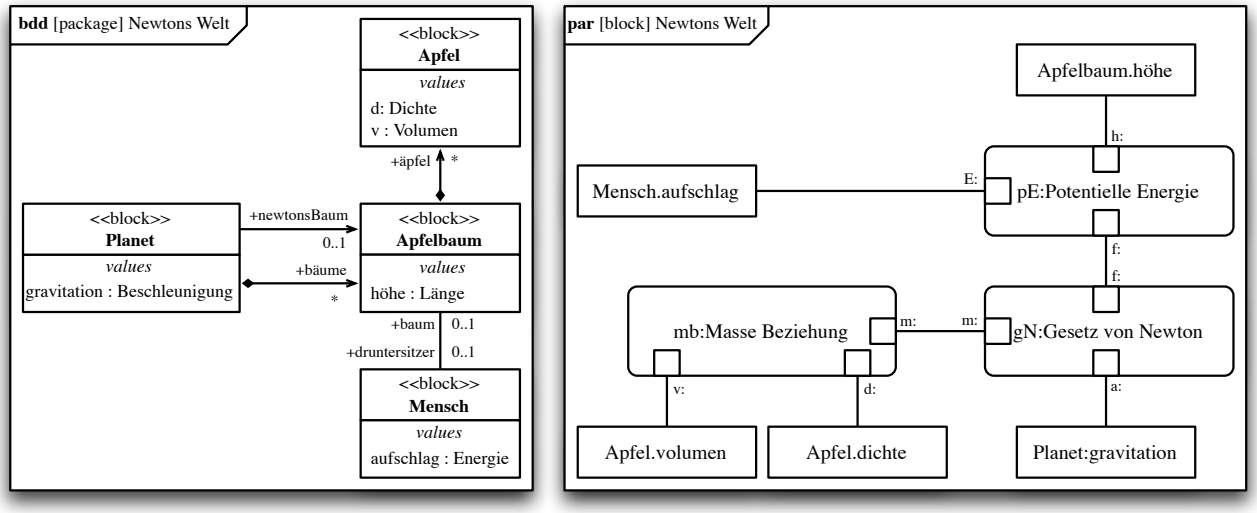


Abbildung 3.57.: Modellierung von Zusicherungen im *Block Definition Diagram* sowie im *Parametric Diagram* nach [Wei06, 195, 197].

In der Abbildung 3.57 ist auf der linken Seite das *Block Definition Diagram* und auf der rechten Seite das dazugehörige *Parametric Diagram* abgebildet. Beide Diagramme stellen den selben Sachverhalt auf unterschiedliche Arten dar. So sind im *Block Definition Diagram* (links) die Blöcke des Systems sowie deren Zusammenhang modelliert. In den Rümpfen der Blöcke sind ihre typisierten Attribute (Eigenschaften) hinterlegt. Im Gegensatz dazu sind im *Parametric Diagram* der Fluss der Attribute zwischen den Blöcken des Systems sowie die physikalischen Zusammenhänge (Formeln[178]) eingetragen. Die Folge der getrennten Modellierung von Struktur und Bedingungen führt dazu, dass stets beide Diagrammarten notwendig sind, um ein System zu beschreiben. Wie bereits oben erwähnt, ist es in SysML möglich, *Constraints* innerhalb von Blöcken zu beschreiben, dabei geht jedoch der Zusammenhang, der im *Parametric Diagram* zwischen den Blöcken explizit modelliert wird, verloren. □

[175] Diese Aussage bezieht sich nur auf technische Systeme, nicht auf Software!
[176] Im Deutschen wird das *Parametric Diagram* mit *Zusicherungsdiagramm* übersetzt.
[177] Bedingungen bzw. Zusicherungen werden im Englischen oft als *Constraints* übersetzt.
[178] Die eigentlichen mathematischen Zusammenhänge werden normalerweise nicht im *Parametric Diagram* direkt hinterlegt sondern in tabellarischer Form, um damit rechnen zu können. Diese Funktionalität des *Parametric Diagram* wird jedoch ausschließlich durch ein Modellierungswerkzeug zur Verfügung gestellt und ist nicht Teil der SysML.

3.4.4.1.2. SysML-Verhaltensdiagramme

Zur Beschreibung des Verhaltens eines Systems stellt die SysML vier unterschiedliche Diagrammarten zur Verfügung. Auf der obersten Ebene kann das Verhalten bzw. die Interaktion des Systems mit der Systemumwelt mit Hilfe des *Use Case Diagrams* beschrieben werden. Das *State Machine Diagram* dient zur Modellierung von Zuständen und Übergängen zwischen den Zuständen. Zusätzlich kann mit einem *Activity Diagrams* bzw. einem *Sequence Diagram* das zeitliche Verhalten des Systems modelliert und beschrieben werden.

Für die Modellierung und Bestimmung der Verhaltenskompatibilität reicht es aus, das Verhalten des Systems mit Hilfe eines *Sequence Diagrams* zu beschreiben (vgl. [Koß09] und [Eck07]). Aus diesem Grund wird hier lediglich auf das *Sequence Diagram* eingegangen.

Sequence Diagram (sd)[179]

Das SysML-*Sequence Diagram* entspricht im Wesentlichen dem bereits im Kapitel „3.4.3.1 *Die wichtigsten Diagrammarten der UML/UML2 für die Modellierung von Kompatibilität*“ ab Seite 132 eingeführten UML/UML2-*Sequence Diagram*. Aus diesem Grund wird es an dieser Stelle nicht noch einmal erläutert.

3.4.4.1.3. Erweiterte SysML-Modellierungskonzepte

Aufbauend auf den, in den vorangegangenen Abschnitten vorgestellten Diagrammarten bietet die SysML verschiedene weitere Konzepte, die für die Modellierung von Kompatibilität hilfreich sind. So können in SysML z.B. Einheiten direkt modelliert und dargestellt werden.

Modellierung von Einheiten

Aufgrund der Tatsache, dass im Systems Engineering häufig Werte und Einheiten modelliert werden müssen, sind in SysML die Grundtypen der SI[180] als Blöcke modelliert eingeführt worden (vgl. [Wei06, 189ff]). Die Abbildung 3.58 zeigt im oberen Teil, wie eine Einheit in SysML modelliert werden kann. Im unteren Abschnitt der Abbildung ist ein Ausschnitt aus dem SI-Typensystem in einem SysML-*Block Definition Diagram* (als so genannte Modellbibiliothek) dargestellt.

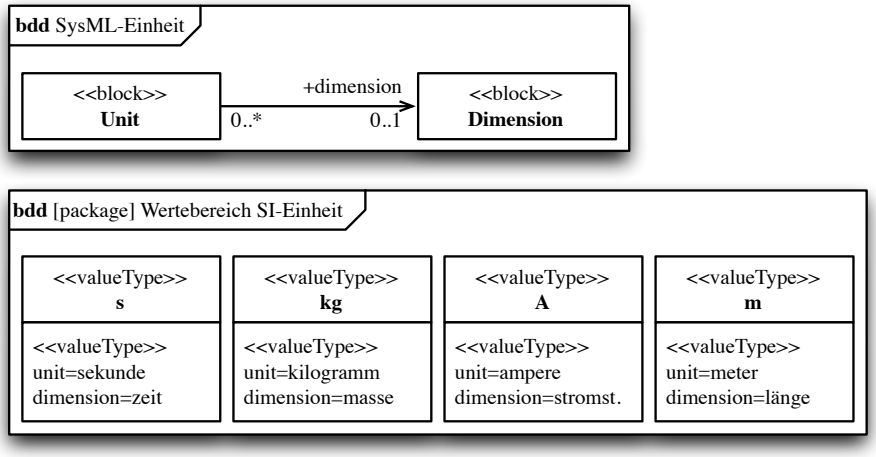


Abbildung 3.58.: Modellierung von Einheiten in SysML ([Wei06, 189ff]).

Mit Hilfe des *bdd SysML-Einheit* können beliebig viele weitere Einheiten definiert und der Einheitenbibiliothek hinzugefügt werden. Durch die direkte Modellierung von Einheiten in SysML kann das auf Seite 58 dargestellte Einheitenproblem nicht auftreten, falls die Kompatibilität der Einheiten überprüft wird. Zusätzlich zur Modellierung von Einheiten können in SysML auch Wertetypen direkt angegeben werden (vgl. [Wei06, 191ff]).

Modellierung optionaler Aktivitäten

Ein weiteres, für die Modellierung von Kompatibilität wichtiges Konzept ist die Modellierung von optionalen Aktivitäten ([Wei06, 203ff]). Durch die Modellierung von optionalen Aktivitäten in einem Sequenz- bzw. Aktivitätsdiagramm ist es möglich, das Verhalten eines Systems abhängig von Eingaben zu beschreiben. Insbesondere ist die Modellierung von Optionalität für den Austausch einer Komponente eines Systems mit verändertem Verhalten wichtig, um zu beschreiben, ob die neue Komponente kompatibel zum alten System ist oder nicht (Austausch-/Ersetzungskompatibilität).

Die Modellierung von optionalem Verhalten reicht jedoch für die Kompatibilitätsmodellierung nicht aus. Zusätzlich zur Modellierung des optionalen Verhaltens eines Systems ist es für die Unterstützung der Austausch-/Ersetzungskompatibilität notwendig, auch die Struktur eines Systems modellieren zu können. Es ist jedoch in SysML nicht möglich, Strukturelemente wie z.B. Ports oder Schnittstellen eines Blocks als optional zu kennzeichnen. Aus diesem Grund kann das Wegfallen bzw. Hinzukommen eines Ports bzw. einer Schnittstelle in SysML nicht direkt modelliert werden. Diese Eigenschaft ist jedoch für die Modellierung von Austausch- und Ersetzungskompatibilität zwingend notwendig.

[179] Im Deutschen wird das *Sequence Diagram* mit *Sequenzdiagramm* übersetzt.

[180] Vgl. Abschnitt „2.3.2.2.1 *Modellierung von Einheiten, dekadischen und nichtdekadischen Präfixen sowie von Gültigkeitsintervallen in der erweiterten formalen Eigenschaftsnotation für die Modellierung und Bestimmung von Kompatibilität*“ ab Seite 69.

3.4.4.2. Modellierung des Beispielsystems „KOMPTEST“ in SysML

In diesem Kapitel wird das im Kapitel 3.3 vorgestellte Beispielsystem „KOMPTEST“ in SysML modelliert und im Anschluss daran das Modell auf Kompatibilität untersucht. Abgeschlossen wird die Kompatibilitätsuntersuchung mit der Bewertung der SysML für die Modellierung und Bestimmung der Kompatibilität von eingebetteten Systemen.

Szenario 1: Modellierung der Struktur des Beispielsystems „KOMPTEST“
Die Abbildung 3.59 zeigt das *Block Definition Diagram* des Beispielsystems „KOMPTEST“, das bereits in der Einleitung beschrieben wurde (vgl. Abbildung: 3.1 auf Seite 110) in SysML. In diesem Diagramm sind sämtliche Strukturelemente des Beispielsystems in SysML-Blöcke überführt worden. Außerdem wurde der strukturelle Zusammenhang der einzelnen Blöcke des Systems – ähnlich wie im vorangegangenen Kapitel – modelliert und dargestellt.

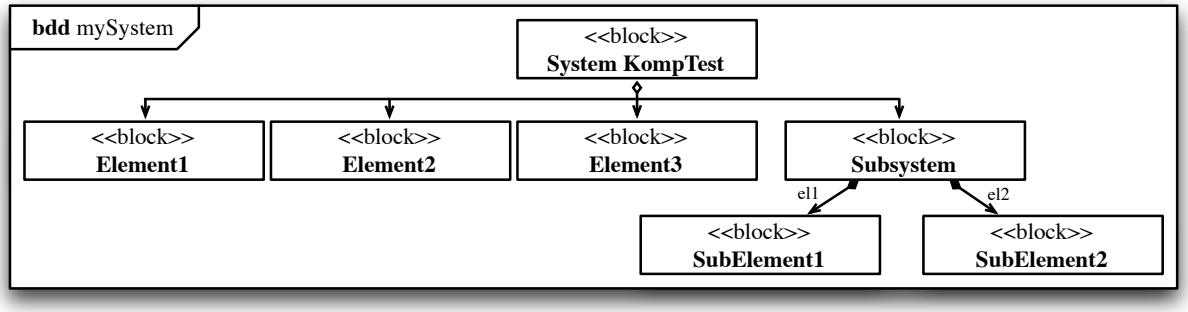


Abbildung 3.59.: Block Definition Diagram des Beispielsystems „KOMPTEST“.

Nachdem die Struktur des Systems im *Block Definition Diagram* abstrakt dargestellt wurde, folgt nun beispielhaft die genaue Spezifikation eines Blocks des Systems. In der Abbildung 3.60 ist die innere Struktur des Block *Element1* ausführlich dargestellt. Der Block beinhaltet fünf *operations* ($SetI1 \ldots SetI3, GetO1, GetO2$) sowie zwei *values* (*O1Spannung* und *I1Strom*), mit deren Hilfe die Struktur des Block vollständig beschrieben wird.

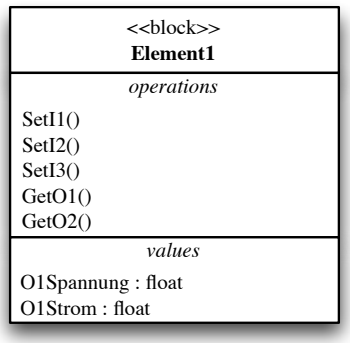


Abbildung 3.60.: Ausführliche Darstellung eines Blocks aus Abbildung 3.59.

Die Abbildung 3.61 zeigt das *Internal Block Diagram* des Blocks *SubSystem* aus Abbildung 3.59. In diesem Diagramm sind sowohl der Aufbau als auch die Schnittstellen (Ports) der Blöcke inklusive der Verbindungen zwischen den Schnittstellen der Blöcke dargestellt.

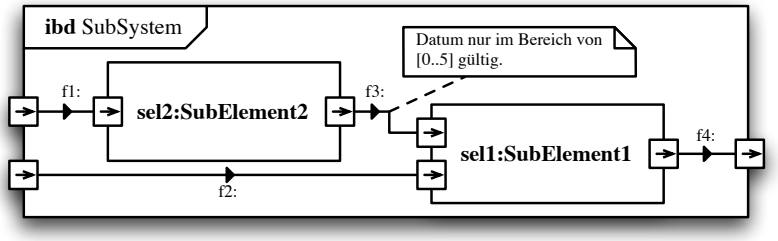


Abbildung 3.61.: Internal Block Diagram des Blocks *SubSystem* aus der Abbildung 3.59.

In der Abbildung 3.61 ist zwischen dem Block *SubElement2* und dem Block *SubElement1* die *Item Flow*-Verbindung *f3* modelliert. Über diese gerichtete Verbindung tauscht der Block *SubElement2* ein Datum mit dem Block *SubElement1* aus. In SysML ist es nicht möglich das zu übertragende Datum genauer als durch seinen Namen und Typ zu spezifizieren. Weitere Eigenschaften wie z.B. ein Gültigkeitsintervall, können nicht direkt angegeben werden. Um zusätzliche Eigenschaften dennoch in SysML abbilden zu können, können diese in Kommentarfelder eingetragen werden. In der Abbildung 3.61 ist rechts oben ein Kommentarfeld abgebildet, das den *Item Flow f3* genauer spezifiziert.

Des Weiteren sind in der Abbildung 3.61 drei externe Ports, die das Subsystems mit dem restlichen System verbinden, dargestellt. Auch diese könnten mit Hilfe von Kommentarfeldern mit zusätzlichen Informationen versehen werden, die für die Kompatibilitätsprüfung wichtig sind. Im *Parametric Diagram* des Beispielsystems sind sämtliche Bedingungen (*Constraints*) eingetragen, die für die ordnungsgemäße Funktion des Systems notwendig sind. Die im *Parametric Diagram* hinterlegten Beziehungen und *Constraints* können ebenfalls für die Kompatibilitätsbestimmung herangezogen werden. Es gilt auch für das *Parametric Diagram*, dass dort nicht alle Informationen, die für die Kompatibilitätsbestimmung notwendig sind (wie z.B. Intervalle) direkt hinterlegt werden können. Auch hier kann wieder der Umweg über die Kommentarfelder genommen werden.

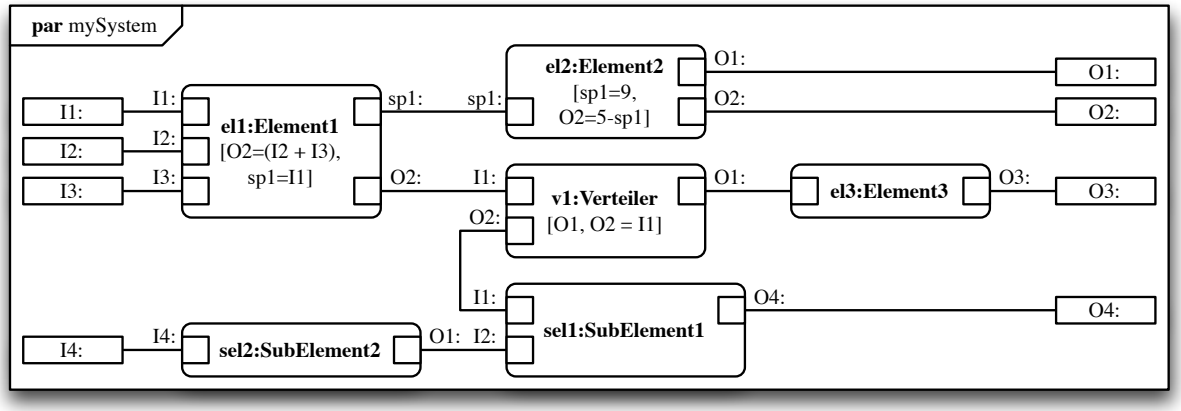


Abbildung 3.62.: Parametric Diagram des Beispielsystems.

Ergebnis: Die Struktur des Beispielsystems „KOMPTEST“ kann mit Hilfe der drei beschriebenen SysML-Strukturdiagramme vollständig modelliert werden. Für die Bestimmung der Kompatibilität reichen die Informationen in den Diagrammen jedoch nicht aus. Zum Beispiel können weder im *Internal Block Diagram* noch im *Parametric Diagram* Gültigkeitsintervalle direkt angegeben werden[181]. Dies ist jedoch eine sehr häufige Fehlerquelle bei der Modellierung von eingebetteten Systemen (vgl. Abschnitt „2.3.2.2.1 *Modellierung von Einheiten, dekadischen und nichtdekadischen Präfixen sowie von Gültigkeitsintervallen in der erweiterten formalen Eigenschaftsnotation für die Modellierung und Bestimmung von Kompatibilität*“ ab Seite 69).

Szenario 1: Modellierung des Verhaltens des Beispielsystems „KOMPTEST“
Nachdem in den letzten drei SysML-Diagrammen die Struktur des Beispielsystems „KOMPTEST“ modelliert wurde, folgt nun die Modellierung des Verhaltens des Systems. Beispielhaft für die Verhaltensmodellierung des Systems wurde hier die Interaktion zwischen dem Block *Element1* und *Element2* herausgegriffen. Die Abbildung 3.63 zeigt das *Sequence Diagram*, das die Interaktion zwischen den beiden Blöcken *Element1* und *Element2* beschreibt.

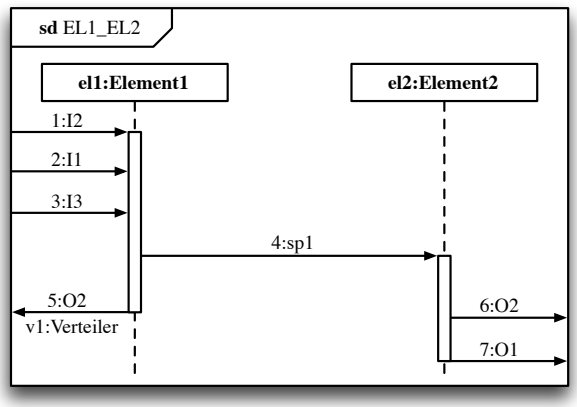


Abbildung 3.63.: Sequenzdiagramm der beiden Elemente *Element1* und *Element2*.

[181] In SysML ist es stets möglich, Intervalle als Kommentare anzugeben. Diese können jedoch nur durch Erweiterungen automatisch für die Kompatibilitätsprüfung herangezogen werden.

Ergebnis: Mit Hilfe des SysML-*Sequence Diagrams* wird das Verhalten des Systems beschrieben. Für die Kompatibilitätsbestimmung reichen die Informationen, die im Sequenzdiagramm angegeben sind aus, um eine Aussage über die Verhaltenskompatibilität zu machen. Lediglich kontinuierliche Signale, wie z.B. elektrischer Strom, lassen sich mittels eines Sequenzdiagramms nur unvollständig beschreiben.

Szenario 2: Austausch der Komponente *Element2* gegen die neue Komponente *ElementX*
Nachdem das Grundmodell des Beispielsystems „KOMPTEST“ in SysML modelliert worden ist (vgl. Abbildung: 3.59 auf Seite 150), wird nun – wie in der Einleitung angesprochen – eine Komponente des Systems gegen eine andere mit leicht veränderter Schnittstelle ausgetauscht und das neue System anschließend ebenfalls auf Kompatibilität untersucht. Die Abbildung 3.64 zeigt auf der linken Seite den Ausschnitt aus dem *Internal Block Diagram*, in dem die beiden relevanten Blöcke *Element1* und *Element2* enthalten sind. Die restlichen Blöcke werden hier nicht betrachtet, weil jeder Block eine in sich abgeschlossene Einheit bildet (vgl. Kapselung), die die anderen nicht verändert, solange keine Verbindung zwischen den einzelnen Blöcken durch einen explizit modellierten *Item Flow* besteht.

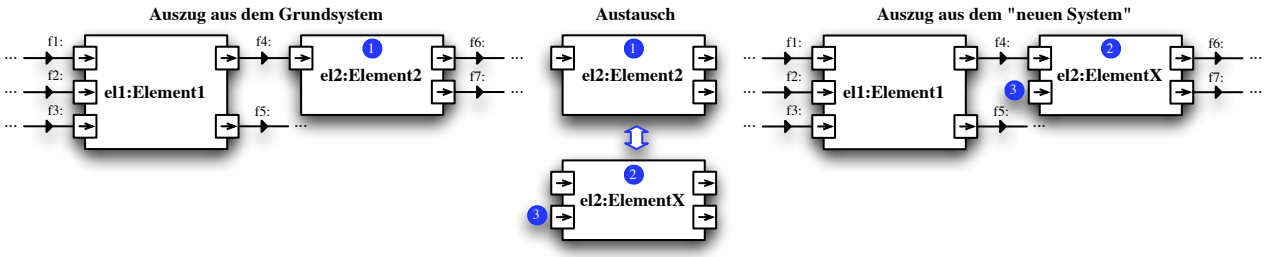


Abbildung 3.64.: Austausch des Blocks *Element2* gegen den Block *ElementX*.

In der Bildmitte ist der Austausch des Blocks *Element2* (1) gegen den neuen Block *ElementX* (2) dargestellt. Der neue Block *ElementX* hat im Vergleich zum ursprünglichen Block einen weiteren optionalen Eingangsport. Auf der rechten Seite der Abbildung 3.64 ist das neu entstandenen System abgebildet, in dem der Block *Element2* gegen den neuen Block *ElementX* ausgetauscht wurde.

Die Tatsache, dass der neu hinzugekommene Port (3) optional ist und nicht angeschlossen werden muss kann aus dem Diagramm 3.64 (rechts) nicht entnommen werden. Um dies zu verdeutlichen, kann jedoch zusätzlich ein Kommentarfeld in das Diagramm eingefügt werden, in dem die Optionalität des Ports explizit verbal angegeben ist. Aufgrund der Tatsache, dass Kommentarfelder nicht automatisch ausgewertet werden können, kann die Eigenschaft „Optionalität des Ports (3)“ nicht überprüft werden.

Ein weiterer möglicher Lösungsansatz, die Optionalität von Ports in SysML zu modellieren, ist die Erweiterung der Portdefinition um die Eigenschaft *optional*. Dann muss für jeden Port angegeben werden ob er im Modell angeschlossen werden muss. Zusätzlich muss dann eine logische Verknüpfung zwischen dem Port und der Verbindung mit einem *Item Flow* mitmodelliert werden. Die Abbildung 3.65 zeigt auf der linken Seite einen erweiterten SysML-Standardport sowie einen optionalen Port, an dem kein *Item Flow* angeschlossen sein muss. Rechts wird die Anwendung der beiden erweiterten Portarten, in einem einfachen Beispielsystem, dargestellt.

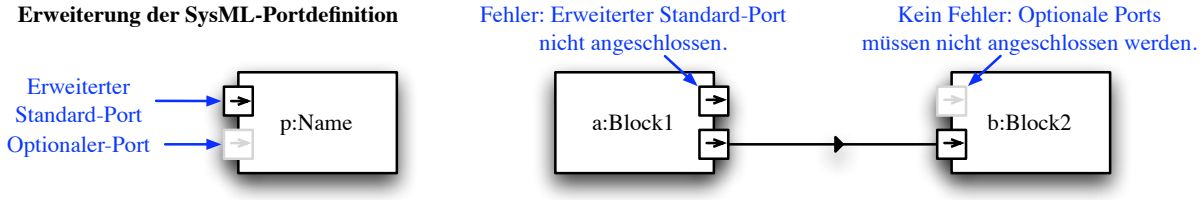


Abbildung 3.65.: Erweiterung der SysML-Portdefinition um Optionalität.

Die Lollipop-Schnittstellennotation (vgl. Abbildung: 3.66 auf Seite 153 links) ist für die Kompatibilitätsbestimmung ungeeignet, da hier nicht explizit modelliert ist, wie zwei Blöcke miteinander verbunden sind, sondern nur dass ein Block eine Schnittstelle anbietet bzw. eine benötigt. Erst durch die explizite Modellierung der Verbindung zwischen zwei Blöcken in einem *Internal Block Diagram* kann sichergestellt werden, dass beide Blöcke miteinander verbunden sind (rechts). Aus diesem Grund sollte auf die Modellierung von Schnittstellen in Lollipop-Notation verzichtet werden.

Ergebnis: In SysML ist es zum gegenwärtigen Zeitpunkt nicht möglich, optionale Ports direkt zu modellieren. Außerdem fehlt die Verbindung zwischen Port und „Flusspfeil“. Beide Eigenschaften sind jedoch zwingend notwendig, um die Kompatibilität, genauer die Schnittstellenkompatibilität, exakt modellieren und testen zu können. Zuletzt ist es in SysML nicht möglich Modellverletzungen graphisch darzustellen.

Die Systemmodellierungssprache SysML ist zwar für die Modellierung von neuen Systemen gut geeignet, ein späterer Austausch von Komponenten (Blöcken) wird nicht explizit unterstützt. Es ist jedoch möglich, auch den Austausch von Komponenten in SysML zu simulieren, jedoch tritt hier vor allem die Schwierigkeit auf, dass in „alten“ Modellen so gut wie keine kompatibilitätsrelevanten Informationen enthalten sind und somit der Austausch von alten Komponenten gegen neue nur bedingt unterstützt wird.

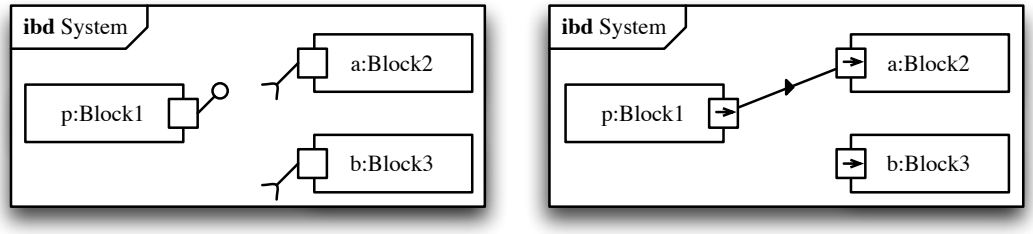


Abbildung 3.66.: Unterschied zwischen der Lollipop-Notation und der *Item Flow*-Darstellung aus Kompatibilitätssicht.

3.4.4.3. Einsetzbarkeit und Bewertung von SysML für die Bestimmung der Kompatibilität von eingebetteten Systemen

Die Systemmodellierungssprache SysML ist zum gegenwärtigen Zeitpunkt ungeeignet für die Modellierung und Bestimmung von Kompatibilität von eingebetteten softwarelastigen Systemen. Dies liegt zum einen daran, dass Schnittstellen (Ports) von Blöcken – ähnlich wie die Schnittstellen von Klassen in UML/UML2 – nicht formal beschrieben werden können. Zum anderen kann kein Regelwerk definiert werden, mit dessen Hilfe sich Kompatibilitätsregeln beschreiben lassen. Mit Hilfe des *Parametric Diagrams* bzw. der *Constraints* im *Block Definition Diagram* können zwar Kompatibilitätsbedingungen definiert und einfache Berechnungen durchgeführt werden. Dies reicht für eine Bestimmung der Kompatibilität bei weitem nicht aus, da z.B. weder ein global gültiges Regelwerk hinterlegt werden kann, noch die Beschreibung oder Schachtelung von Intervallen erlaubt ist.

Eine weitere Schwachstelle der SysML ist die fehlende graphische Darstellung der Kompatibilitätsergebnisse. So kann beispielsweise eine Bedingung (Constraint) fehlschlagen, ohne dass dies in einem Diagramm als Fehler angezeigt wird.

3.5. Bewertung der existierenden Modellierungssprachen für die Modellierung von Kompatibilität

In den letzten Kapiteln wurden vier unterschiedliche existierende Modellierungssprachen und -methoden vorgestellt und die Einsetzbarkeit für die interdisziplinäre Modellierung und Bewertung der Kompatibilität von eingebetteten softwarelastigen Systemen anhand eines einfachen Beispielsystems gezeigt. Das Ergebnis dieses Vergleichs ist: Sämtliche Modellierungssprachen lassen sich nur bedingt für die Modellierung und Bewertung von Kompatibilität einsetzen. Jede der vorgestellten Modellierungssprachen hat zum Teil erhebliche Einschränkungen bei der Kompatibilitätsmodellierung. Außerdem werden bei weitem nicht alle Anforderungen an eine domänenübergreifende Modellierungssprache für Hard- und Software erfüllt. So kann zum Beispiel in keinem existierenden Modellierungsansatz ein Kompatibilitätsregelwerk hinterlegt werden, anhand dessen ein Modell auf Kompatibilität untersucht und bewertet werden kann. Ohne die Existenz eines projektunabhängigen, global gültigen Kompatibilitätsregelwerks kann ein Modell eines Systems jedoch nicht auf Kompatibilität untersucht werden, da im Allgemeinen innerhalb eines Modells nicht festgelegt wurde, unter welchen Bedingungen zwei Komponenten eines Systems zueinander bzw. zum System kompatibel sind. Des Weiteren unterstützt kein Modellierungsansatz die Modellierung von Austauschkompatibilität.

Im nachfolgenden Abschnitt wird aufbauend auf Stärken und Schwächen der vorgestellten Modellierungsansätzen und -methoden, eine neue, speziell für die Modellierung und Bewertung von Kompatibilität von eingebetteten softwarelastigen Systemen entwickelte interdisziplinäre Modellierungssprache vorgestellt.

3.6. Einführung in die Kompatibilitätsmodellierungssprache (U)CML

Nachdem in den letzten beiden Abschnitten des Kapitels „*Modellierungssprachen und -techniken für technische Systeme*“ eine kleine Auswahl an Systementwurfs- und Modellierungssprachen für komplexe technische Systeme vorgestellt wurde, folgt in diesem Kapitel die Einführung in die Kompatibilitätsmodellierungssprache (U)CML. Das Akronym *(U)CML* steht für (**U**nified) **C**ompatibility **M**odelling **L**anguage[182]. Die (U)CML ist eine speziell für die Bedürfnisse der Kompatibilitätsmodellierung und -bestimmung ausgelegte domänenunabhängige graphische Modellierungssprache.

Einleitung und Motivation
Wie bereits in der Einleitung der Dissertation erwähnt, wurde die (U)CML während des vom BMBF geförderten Forschungsvorhabens MOKOMA mit dem Ziel entwickelt, eine domänenunabhängige Kompatibilitätsmodellierungssprache für eingebettete softwarelastige Systeme, bestehend aus Elektrik/Elektronik, Mechanik und Software, mit sehr langer Lebenszeit zu entwickeln. Im Kapitel „3.2 *Anforderungen an eine Kompatibilitätsmodellierungssprache*“ ab Seite 109 wurden die wichtigsten Anforderungen an eine domänenübergreifende Kompatibilitätsmodellierungssprache formuliert.

Nachdem die essentiellen Anforderungen an eine generische Kompatibilitätsmodellierungssprache festgelegt wurden, wurden sechs existierende Modellierungssprachen bzw. -konzepte auf ihre Anwendbarkeit und Erweiterbarkeit für die Modellierung und Bewertung von Kompatibilität untersucht. In der nachfolgenden Aufzählung sind die während des MOKOMA-Projekts untersuchten Sprachen und Konzepte aufgeführt:

[182] Anmerkung: Das „U“ von Unified steht absichtlich in Klammern, aufgrund der Tatsache, dass die (U)CML kein einheitlicher (Industrie-) Standard für die Kompatibilitätsmodellierung ist.

- Systems Engineering Element-Konzept
- UML/UML2 [BRJ98]
- SysML [Sys06]
- IPO
- AutoFOCUS [HEB08]
- Altova *XML Spy* bzw. Altova *MapForce* [Alt07]

Warum eine neue Modellierungssprache?
Eine existierende Modellierungssprache (bzw. eine Modellierungsmethode), die sowohl den Haupt- als auch den Nebenanforderungen genügt, gab es während der Laufzeit des MOKOMA Projekts nicht. Bis heute gibt es keine Modellierungssprache, die alle Anforderungen erfüllt. Die meisten Anforderungen wurden von UML/UML2[183] und SysML[184] erfüllt. Die Anpassung einer oder beider Modellierungssprachen an die im MOKOMA-Projekt festgelegten Anforderungen wurde verworfen, aufgrund der Tatsache, dass beide Modellierungssprachen die folgenden entscheidenden Schwachstellen aufweisen:

- Mit keiner existierenden Modellierungssprache kann Kompatibilität direkt modelliert bzw. das Modell auf Kompatibilität untersucht und getestet werden.
- In keiner Modellierungssprachen können Kompatibilitätsregeln hinterlegt werden.
- Keine Modellierungssprache unterstützt benutzerdefinierte Datentypen für Hard- und Software.
- Keine „Verbindung“ zwischen Datentypen und Einheiten.
- Weder in UML/UML2 noch in SysML können alle Sprachelemente sowohl graphisch, als auch textuell beschrieben werden.

Bei SysML kam noch erschwerend hinzu, dass es eine „sehr junge“ Modellierungssprache ist, die während der Projektlaufzeit noch nicht vollständig durch die OMG spezifiziert und freigegeben war. Aus den angegebenen Gründen wurde im MOKOMA-Projekt entschieden, eine neue Modellierungssprache („Referenzsprache für die Modellierung und Bewertung von Hard- und Software“) zu entwickeln, die allen obigen Anforderungen gerecht wird – (U)CML; (U)CML integriert die wichtigsten Methoden und Konzepte aller untersuchten Modellierungssprachen, die für die Modellierung von Kompatibilität notwendig sind. Im nächsten Abschnitt dieses Kapitels wird die Abstammung der (U)CML erläutert.

Anmerkung:
Eine ausführliche Bewertung der unterschiedlichen Modellierungssprachen und ihre Anwendung für die Kompatibilitätsmodellierung und -bewertung finden Sie im Abschnitt „3.7 *Vergleich und Bewertung der vorgestellten Modellierungssprachen*“ ab Seite 253.

Stammbaum der (U)CML
Stammväter der (U)CML sind verschiedene in der Informatik und dem klassischen Maschinenbau beheimatete Modellierungssprachen und -konzepte. Aus der Informatik stammt die weltweit verbreitete Softwaremodellierungssprache UML bzw. die erweiterte Version UML2, mit deren Hilfe komplexe Softwaresysteme objektorientiert modelliert und beschrieben werden können. Aus der UML/UML2 wurden vor allem die statischen Sprachelemente wie z.B. die Pakete und Klassen übernommen. Weiterhin wurden Teile der UML/UML2-Sequenzdiagramme für die Modellierung des Verhaltens eines Systems in die Kompatibilitätsmodellierungssprache (U)CML integriert. Um das Verhalten eines Systems bzw. der Komponenten des Systems zu beschreiben wurden zusätzlich Teile aus der Automatentheorie sowie eine spezielle Art von Sequenzdiagrammen – die so genannten nachrichtenbasierten Sequenzdiagramme (MSC [Krü00][Ker05]) – in (U)CML aufgenommen. Durch die Einbindung von Automaten bzw. Sequenzdiagrammen kann das Verhalten des gesamten Systems modelliert, simuliert und dargestellt werden.

Aus dem klassischen Maschinenbau und insbesondere der Raumfahrt stammen sowohl das Systems Engineering Element-Konzept[185] als auch das IPO-Konzept[186]. Mit Hilfe dieser beiden Modellierungssprachen und -konzepte lassen sich sowohl komplexe technische Systeme als auch Prozesse für die Auslegung (Phasen 0-A) von Systemen modellieren. Aus dem Systems Engineering Element-Konzept wurde das Konzept der Relationen (Verbindungen) zwischen Elementen und eine im Systems Engineering weit verbreitete Darstellungsform – die Matrixdarstellung (DSM) – für die Modellierung komplexer Systeme entnommen. Aus dem IPO-Konzept stammen die Pfeile und Flüsse, während aus der Systemmodellierungssprache SysML das Komponentenkonzept sowie die Ports und Teile der Sequenzdiagramme in (U)CML übernommen wurden.

Eine Sonderstellung im Abstammungsbaum der (U)CML nimmt die Modellierungssprache (v)Sys bzw. das damit eng verbundene Modellierungswerkzeug (v)Sys-ed[187] ein. (v)Sys bzw. (v)Sys-ed wurde wie (U)CML am Lehrstuhl für Raumfahrttechnik an der Technischen Universität München entwickelt, so dass eine sehr starke gegenseitige Beeinflussung stattfand. Es wurden einige Konzepte wie z.B. die automatische Berechnung von Attributen (Eigenschaften) aus (v)Sys in (U)CML übernommen, während andere Konzepte, wie beispielsweise das Paketkonzept in (v)Sys integriert wurden. Zusätzlich wurde ein gemeinsames XML-basiertes Datenaustauschformat festgelegt, so dass ein Systemmodell (Phase 0-A) in (v)Sys-ed ausgelegt und entwickelt und anschließend im *(U)CML-ed*[188] auf Kompatibilität untersucht werden kann. Durch die enge Kopplung der beiden Modellierungskonzepte bzw. der Werkzeuge (v)Sys-ed und *(U)CML-ed* kann der komplette Lebenszyklus eines technischen Systems von der Planungs- bis zur Wartungsphase modelliert werden.

[183] Siehe hierzu: Kapitel „3.4.3 *Die Unified Modelling Language – UML/UML2*“ ab Seite 131.
[184] Siehe hierzu: Kapitel „3.4.4 *Die System Modelling Language – SysML* “ ab Seite 144.
[185] Siehe hierzu: Kapitel „3.4.2 *Das Systems Engineering Element-Konzept – „Die Münchner Schule““* ab Seite 116.
[186] Siehe hierzu: Kapitel „3.4.1 *Input-Process-Output Modell*“ ab Seite 113.
[187] Siehe hierzu [DIS08], sowie [DIS07].
[188] Siehe hierzu: Kapitel „3.8 *(U)CML-ed, ein graphischer Editor für die Kompatibilitätsmodellierungssprache (U)CML*“ ab Seite 254.

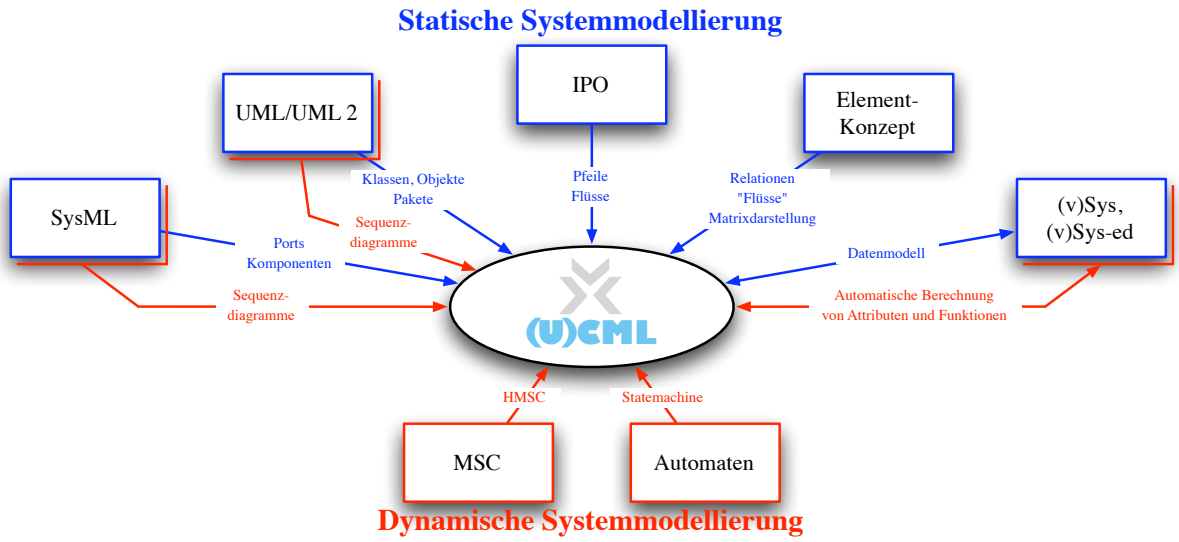


Abbildung 3.67.: Stammbaum der Kompatibilitätsmodellierungssprache (U)CML.

Die „Essenz“ aller Modellierungskonzepte, die für die Modellierung von Kompatibilität notwendig sind, wurden zur (U)CML verschmolzen mit dem Ziel eine leicht zu erlernende domänenunabhängige Modellierungssprache zu kreieren, die sowohl von Informatikern als auch von klassischen Ingenieuren leicht erlernt und verstanden werden kann. In der Abbildung 3.67 sowie der Tabelle 3.7 sind der „Stammbaum“ der (U)CML und die in (U)CML integrierten Konzepte kompakt dargestellt.

Modellierungssprache	dynamisch	statisch	In (U)CML übernommene Konzepte
UML/UML2	X	X	Sequenzdiagramme; Pakete, Klassen, Objekte
SysML	X	X	Sequenzdiagramme; Komponenten, Ports
IPO		X	Pfeile, Flüsse
Element-Konzept		X	Relationen, Flüsse, Matrixdarstellung
(v)Sys bzw. (v)Sys-ed	X	X	Automatische Berechnung; Datenmodell
MSC	X		MSC, HMSC
Automaten	X		Zustandsautomaten

Tabelle 3.7.: In (U)CML übernommene statische und dynamische Modellierungskonzepte.

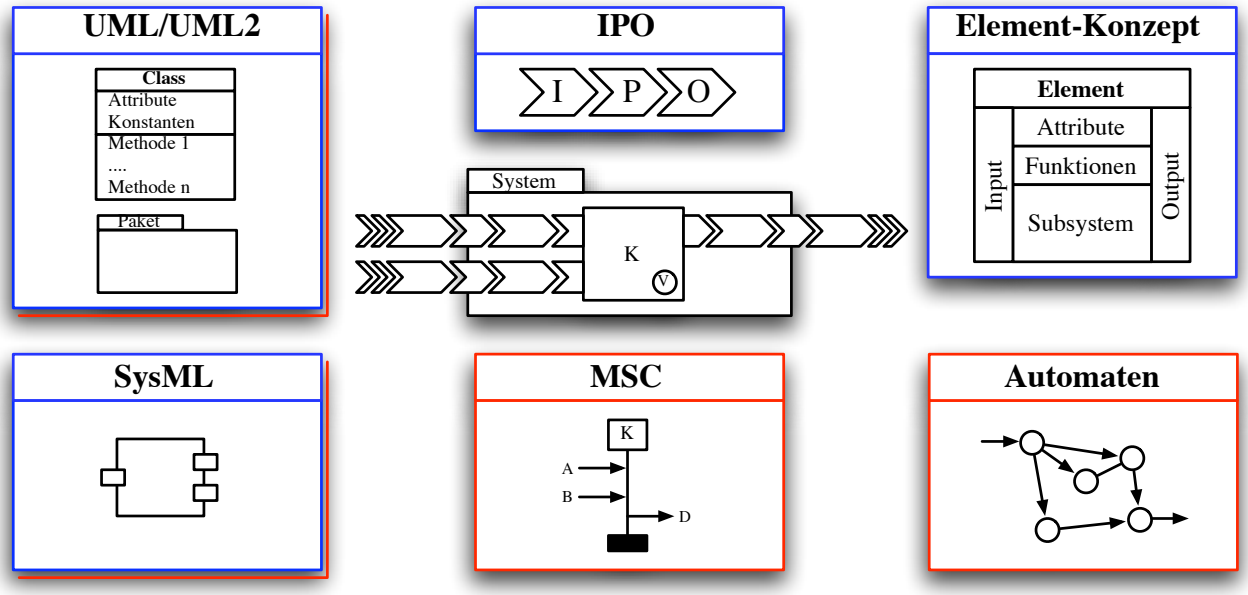


Abbildung 3.68.: Verschmelzung der verschiedenen Modellierungskonzepte – *UML/UML2* , *SysML* , *IPO*, *Element-Konzept*, *MSC* und *Automaten* – zur Kompatibilitätsmodellierungssprache (U)CML.

Die Abbildung 3.68 auf der vorherigen Seite illustriert die Verschmelzung der verschiedenen Stammväter der (U)CML noch einmal in anderer Darstellungsform. In der Mitte der Abbildung ist die statische Struktur eines einfachen (U)CML-Systems, die aus dem Systempaket *System* und einer Komponente *K* besteht, abgebildet. Die statische Struktur des Systems wird mit Hilfe von Paketen und Komponenten, die aus UML/UML2 bzw. SysML übernommen wurden, beschrieben. Die Schnittstellen am Paket *System*, die Buchse und Stecker an der Komponente *K* sowie die Pfeile dazwischen sind aus der IPO-Modellierung bzw. dem Systems Engineering Element-Konzept übernommen. Mit Hilfe der MSC und Automaten wird sowohl das innere Verhalten der Komponente *K* als auch das Verhalten der Stecker und Buchsen der Komponente *K* beschrieben.

3.6.1. Aufbau und Struktur der (U)CML

Die Modellierungssprache (U)CML ist aus vier unterschiedlichen Diagrammarten, dem *Fluss-*, *Baum-*, *Graphen-* und dem *Matrixdiagramm* sowie einer textuellen Notation (MIDL – **M**odel and **I**nterface **D**escription **L**anguage) aufgebaut. In der nachfolgenden Abbildung 3.69 ist der grundsätzliche Aufbau und die Struktur der (U)CML schematisch dargestellt.

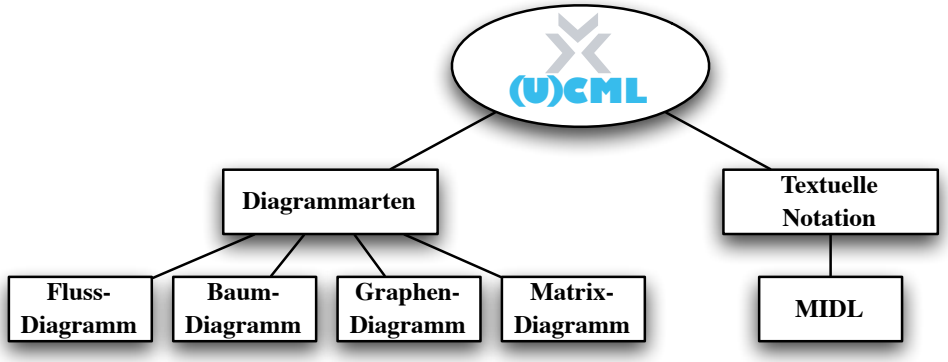


Abbildung 3.69.: Struktur der Modellierungssprache (U)CML.

Jede in der Abbildung 3.69 dargestellte Diagrammart wird genutzt, um einen speziellen Aspekt/Ansicht[189], auf das System darzustellen. Mit Hilfe der beschriebenen vier Diagrammarten wird

- die hierarchische Struktur des Systems
- die Verbindungen zwischen Komponenten des Systems
- die Verbindung vom System zur Umwelt
- die Beschreibung der Komponenten des Systems

sowie das

- statische und dynamische Verhalten des Systems

beschrieben. Zusätzlich zu den hier vorgestellten vier Diagrammarten, mit deren Hilfe ein System graphisch modelliert und dargestellt werden kann, stellt die (U)CML die textuelle Notationssprache MIDL zur Verfügung, mit deren Hilfe sowohl das System als auch das Kompatibilitätsregelwerk vollständig textbasiert beschrieben werden können. Im Abschnitt „E.1 *Einführung in* MIDL“ ab Seite 337 dieses Kapitels folgt eine kurze Einführung in die textuelle Modellierungssprache MIDL der (U)CML.

Anmerkung:
Die vier Diagrammarten der (U)CML repräsentieren jeweils eine spezielle Ansicht auf das System. Aus diesem Grund müssen sie stets synchron gehalten werden. Ohne die Synchronisierung der vier Diagrammarten kann es zu Inkonsistenzen und „Verwirrungen“ beim Modellieren kommen. So kann zum Beispiel ein Paket *P* im Flussdiagramm drei Komponenten *A, B* und *C* enthalten, in der Baumansicht jedoch kann bereits die Komponente *C* aus dem Paket *P* gelöscht worden sein. Somit stellen die vier Diagramme nicht mehr das selbe System dar. Um diesem negativen Effekt bei der Modellierung von Systemen entgegenzuwirken, ist der Einsatz eines integrierten Modellierungswerkzeugs, dass stets alle vier Diagrammarten synchronisiert zwingend erforderlich. Die Einschränkung, dass die (U)CML nur mit Hilfe eines Werkzeugs effizient benutzt werden kann, trifft auch auf viele andere graphische Modellierungssprachen zu. Zum Beispiel muss in UML/UML2 das Klassendiagramm stets synchron mit dem Objektdiagramm gehalten werden, da sonst Klasse und Objekt unterschiedlich sein können. Die Synchronisierung von Hand ist bei der Modellbildung von realen Systemen nicht möglich, da reale Systeme zu komplex sind und eine Synchronisation von Hand zu fehleranfällig ist. Aus diesem Grund muss ein Werkzeug eingesetzt werden, das den Ingenieur bei seiner Arbeit unterstützt und leitet. Für die Kompatibilitätsmodellierungssprache (U)CML existiert so ein prototypisches Werkzeug – *(U)CML-ed*[190] – mit dessen Hilfe die unterschiedlichen Sprachelemente der (U)CML gezeichnet und das Modell automatisch auf Kompatibilitätsfehler untersucht werden kann.

[189] Die Ansichten auf ein System werden auch als *views* bezeichnet.
[190] Eine knappe Beschreibung des graphischen Werkzeugs *(U)CML-ed* finden Sie unter „3.8 *(U)CML-ed, ein graphischer Editor für die Kompatibilitätsmodellierungssprache (U)CML*“ ab Seite 254.

In diesem Kapitel und den folgenden Unterabschnitten werden ausschließlich die unterschiedlichen Modellierungskonzepte der (U)CML bzw. die Modellierungssprache (U)CML an sich vorgestellt und anhand von einfachen Beispielen erläutert ohne, auf die in der vorangegangenen Anmerkung dargestellten Probleme bei der Modellierung näher einzugehen. Im Kapitel „3.8 *(U)CML-ed, ein graphischer Editor für die Kompatibilitätsmodellierungssprache (U)CML*“ ab Seite 254 wird das Modellierungswerkzeug *(U)CML-ed* vorgestellt, mit dessen Hilfe die Modellierungssprache (U)CML umgesetzt werden kann. Durch die Verwendung des Werkzeugs treten die erwähnten Synchronisierungsprobleme nicht auf.

(U)CML-Diagrammarten
In der nachfolgenden Aufzählung werden die grundlegenden Eigenschaften sowie die Anwendung der vier (U)CML-Diagrammarten kurz beschrieben und erläutert. Eine ausführliche Beschreibung der einzelnen Diagrammarten entnehmen Sie bitte [BK05].

- **Flussdiagramm**
  Das Flussdiagramm ist die wichtigste Diagrammart der (U)CML. Im Flussdiagramm wird sowohl die hierarchische Struktur des Systems mit Hilfe von Paketen modelliert und dargestellt als auch die Verbindungen (Pfeile) zwischen den einzelnen Komponenten des Systems bzw. des Systems und der Umwelt. Des Weiteren werden im Flussdiagramm die Beschreibungsfelder jedes einzelnen Elements (Pakete, Komponenten, Pfeile etc.) dargestellt. Die hierarchische Struktur sowie die Verbindungen zwischen den Komponenten des Systems und die Beschreibungsfelder bilden zusammen die statische Struktur des Systems (vgl. Abbildung 3.70 links).

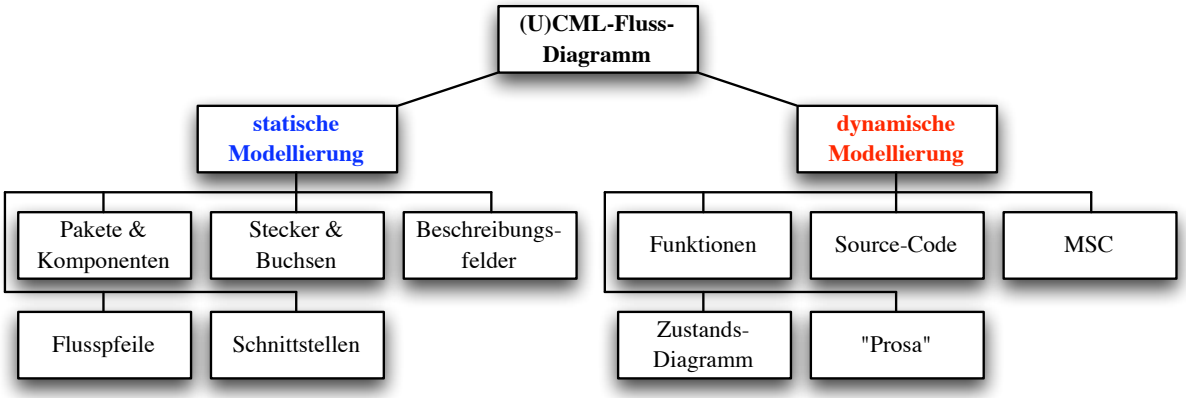


Abbildung 3.70.: Aufbau und Struktur des (U)CML-Flussdiagramms.

Zusätzlich zur statischen Struktur eines Systems wird im Flussdiagramm das dynamische Verhalten des Systems modelliert und in einem eigenen, dem Flussdiagramm untergeordneten Diagramm dargestellt (vgl: Abbildung 3.70 rechts). Das Gesamtverhalten setzt sich dabei aus dem Verhalten der einzelnen Komponenten des Systems zusammen. Dabei wird zwischen dem inneren Verhalten einer Komponenten und dem Verhalten der Komponente an der Schnittstelle unterschieden. Für die Beschreibung des Verhaltens einer Komponente werden unterschiedliche Beschreibungsmethoden verwendet (vgl. Abschnitt „3.6.2.3 *(U)CML-Komponente*“ ab Seite 165).

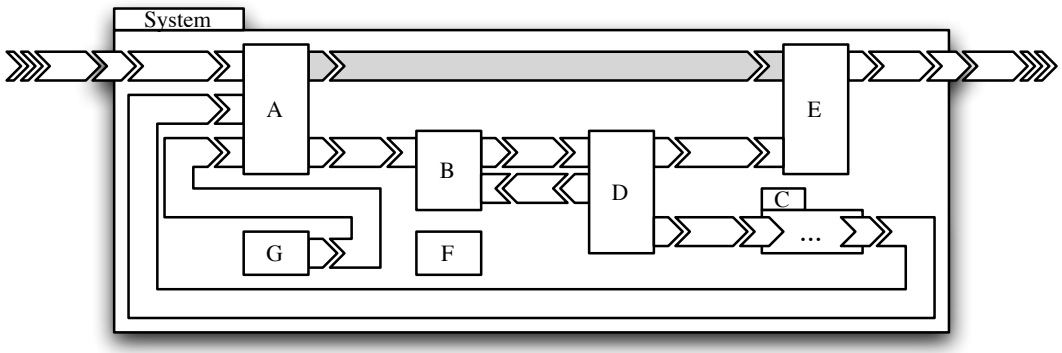


Abbildung 3.71.: (U)CML-Flussdiagramm: Darstellung der *Flüsse* (durch Pfeile) zwischen den einzelnen Komponenten des Systems und der Systemumwelt.

Beispiel 40: Flussdiagramm
In der Abbildung 3.71 ist ein einfaches (U)CML-Modell eines Systems als Flussdiagramm (Flussansicht) dargestellt. Das hier modellierte System besteht aus sechs Komponenten (*A*,*B*,*D*,*E*,*G* und *H*) und zwei Paketen (*System* und *C*). Im Flussdiagramm werden zusätzlich zu den Paketen und Komponenten des Systems die Verbindungen zwischen ihnen dargestellt. Zwingende

Stecker, Buchsen und Verbindungen werden weiß, optionale hingegen grau hinterlegt dargestellt[191]. In der Abbildung 3.71 beispielsweise sind die Komponenten *D* und *E* mit einem zwingenden Flusspfeil (weiß) verbunden, die Komponenten *A* und *E* sind hingegen durch einen optionalen Flusspfeil (grau) verbunden.

In der Abbildung 3.71 nimmt das Paket *C* eine Sonderstellung ein. Dieses Paket ist im Gegensatz zum Systempaket in der so genannten „Black-Box[192]“ Notation dargestellt, in der der Inhalt des Pakets nicht explizit (in dieser Hierarchieebene) dargestellt wird. Soll der Inhalt des Pakets *C* dargestellt werden, so ist eine Verfeinerung dieses Pakets *C* notwendig. Die Abbildung 3.72 zeigt die Verfeinerung des Pakets *C*. In der verfeinerten Darstellung des Pakets *C* ist zu erkennen, dass das Paket *C* eine weitere Komponente *H* enthält, die in der obigen Black-Box Ansicht nicht zu sehen war.

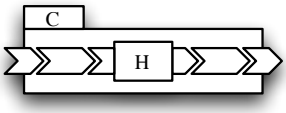


Abbildung 3.72.: (U)CML-Flussdiagramm: Verfeinerung des Pakets *C* aus der Abbildung 3.71.

Daneben sind im Flussdiagramm in Abbildung 3.71 zwei Komponenten (*G* und *F*) enthalten, die nicht der (U)CML-Sprachdefinition entsprechen. Nach der (U)CML-Sprachdefinition muss jedes Paket und jede Komponente eines Systems mindestens einen beschalteten Ein- und einen Ausgang (Buchse, Stecker) haben. Die beiden Komponenten *G* und *F* widersprechen dieser Definition: Komponente *G* hat nur einen beschalteten Ausgang, während die Komponente *F* weder einen beschalteten Ein- noch Ausgang aufweist. Eine Aufzählung aller (U)CML-Sprachregeln und Definitionen finden Sie unter „3.6.5.1 *(U)CML-Sprachregelwerk*“ ab Seite 236. □

Anmerkung:
Alle hier eingeführten (U)CML-Bestandteile wie z.B. Pakete, Komponenten, Stecker, Pfeile etc., werden in den nachfolgenden Kapiteln ausführlich erläutert (siehe hierzu: Kapitel „3.6.2 *Sprachelemente der (U)CML*“ ab Seite 161).

- **Baumdiagramm**
  In einem Baumdiagramm ist die hierarchische Struktur des Systems kompakt dargestellt. Um dies zu erreichen werden im Gegensatz zum Flussdiagramm im Baumdiagramm weder Flusspfeile zwischen den einzelnen Pakten und Komponenten noch Beschreibungsfelder dargestellt, sondern lediglich die hierarchische Struktur des Systems, also die Pakete und Komponenten des Systemmodells. In der nachfolgenden Abbildung 3.73 sind die einzelnen Elemente, aus denen ein Baumdiagramm besteht dargestellt.

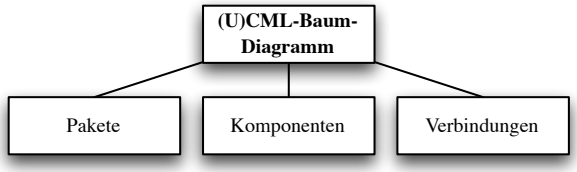


Abbildung 3.73.: Aufbau und Struktur des (U)CML-Baumdiagramms.

Im Baumdiagramm werden die Pakete als Rechtecke mit abgerundeten Ecken und die Komponenten als Ellipsen dargestellt. Verbunden werden die Pakete und Komponenten in Abhängigkeit von der Systemstruktur durch Pfeile, die vom „Vater“ zum „Kind“ zeigen. Als oberstes Element jedes Baumdiagramms wird stets das Systempaket dargestellt, in dem allen anderen Pakete und Komponenten des Systems liegen. Im Unterschied zum Flussdiagramm gibt es im Baumdiagramm keine „Black-Box“ Darstellung von Paketen. Deshalb wird stets das gesamte System mit allen Paketen und Komponenten dargestellt. In der Abbildung 3.74 ist ein einfaches System mit drei Hierarchieebenen modelliert.

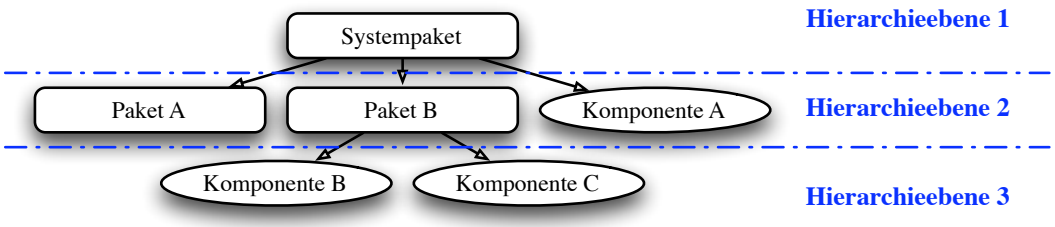


Abbildung 3.74.: (U)CML-Baumdiagramm: Kompakte Darstellung der hierarchischen Struktur eines Systems.

[191] Siehe hierzu: Kapitel „3.6.3 *Das Optionalitätsprinzip der (U)CML*“ ab Seite 218.
[192] Siehe hierzu: Kapitel „1.10 *Grundlegende Begriffe und Definitionen aus der Systems Engineering Welt*“ ab Seite 17.

Das abgebildete System (Abbildung 3.74) besteht aus drei Hierarchieebenen. Auf Ebene eins befindet sich das Systempaket. Auf der Hierarchieebene zwei sind zwei Pakete *Paket A* und *Paket B* und eine Komponente *Komponente A* dargestellt. Das Paket *Paket B* besteht wiederum aus den beiden Komponenten *Komponente B* und *Komponente C*, die sich auf der dritten Hierarchieebene des Systems befinden. Darüber hinaus kann aus dem dargestellten Baumdiagramm entnommen werden, dass das Paket *Paket A* keine weiteren Pakete oder Komponenten enthält.

Beispiel 41: Baumdiagramm
In der nachfolgenden Abbildung 3.75 ist das selbe System, das in den beiden Abbildungen 3.71 und 3.72 als Flussdiagramm abgebildet ist, in einem Baumdiagramm dargestellt.

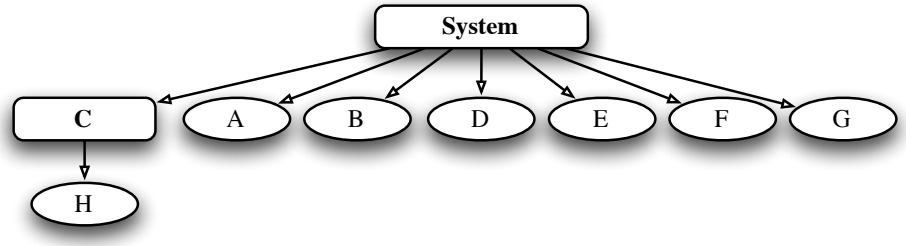


Abbildung 3.75.: (U)CML-Baumdiagramm des in Abbildung 3.71 dargestellten Systems.

Im Baumdiagramm wird das System in seine Hierarchieebenen unterteilt dargestellt. Das System aus der Abbildung 3.75 besteht aus drei Hierarchieebenen. Auf der ersten Ebene befindet sich das Systempaket. Auf der zweiten Ebene befindet sich das Paket *C* sowie die sechs Komponenten *A*,*B*,*D*,*E*,*F* und *G*. Auf der Hierarchieebene drei befindet sich die Komponente *H*. □

- **Graphendiagramm**
  In einem Graphendiagramm werden sämtliche Pakete und Komponenten des Systems mit ihren zusammengefassten Verbindungen kompakt dargestellt. Dabei werden zwei Arten von Verbindungen zwischen Paketen bzw. Komponenten unterschieden. Zum einen die so genannten *zwingenden* und zum anderen die *optionalen Verbindungen*[193]. In der Abbildung 3.76 sind sämtliche Elemente dargestellt, aus denen ein (U)CML-Graphendiagramm aufgebaut ist.

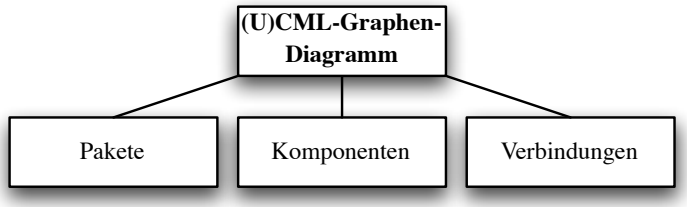


Abbildung 3.76.: Aufbau und Struktur des (U)CML-Graphendiagramms.

Jedes (U)CML-Graphendiagramm besteht aus Paketen, Komponenten sowie den Verbindungen (Flusspfeilen) zwischen diesen. Sowohl Pakete als auch Komponenten werden im Graphendiagramm als Rechtecke mit abgerundeten Ecken dargestellt. Pakete haben zusätzlich noch einen „Reiter“ links oben, in dem der Name des Pakets angeführt wird. Zusätzlich werden Verbindungen (Pfeile) zwischen Paketen und Komponenten im Graphendiagramm zusammengefasst dargestellt. Wenn beispielsweise zwischen der Komponente *A* und Komponente *B* ein oder mehrere Flusspfeile (im Flussdiagramm) existieren, werden diese im Graphendiagramm zusammengefasst als ein gemeinsamer ausgefüllter Pfeil zwischen der Komponente *A* und der Komponente *B* dargestellt (vgl. Abbildung 3.77 oben). Sollte eine Verbindung zwischen den beiden Komponenten eine optionale Verbindung sein (vgl. Abbildung 3.77 unten), wird der gemeinsame Pfeil mit einer nicht ausgefüllten Spitze dargestellt.

Ähnlich wie im Flussdiagramm können im Graphendiagramm Pakete als „Black-Box Pakete“ dargestellt werden, um die Darstellungskomplexität eines Systems im Graphendiagramm zu verringern bzw. in dieser Hierarchieebene den Inhalt unwichtiger Pakete auszublenden.

Beispiel 42: Graphendiagramm
In der Abbildung 3.78 auf der nächsten Seite ist das selbe System in der Graphendarstellung abgebildet, das bereits in Abbildung 3.71 als Flussdiagramm gezeigt wurde. Die Pakete und Komponenten des Systems werden im Graphendiagramm als Rechtecke mit abgerundeten Ecken dargestellt. Die Pakete haben zusätzlich ein kleines Fähnchen links oben, in dem der Name des Pakets vermerkt ist. Die Verbindungspfeile zwischen den Paketen und Komponenten werden im Gegensatz zur Flussdarstellung zusammengefasst (aggregiert) dargestellt. Ist mindestens eine Verbindung zwischen zwei Paketen oder Komponenten optional (grau im Flussdiagramm), wird der Pfeil mit einer nicht ausgefüllten Spitze dargestellt, sonst mit einer ausgefüllten.

[193] Nähere Informationen zu den hier beschriebenen Verbindungsarten entnehmen sie bitte dem Kapitel „3.6.2.5 *(U)CML-Flusspfeilarten*“ ab Seite 184 bzw. dem Kapitel „3.6.3 *Das Optionalitätsprinzip der (U)CML*“ ab Seite 218.

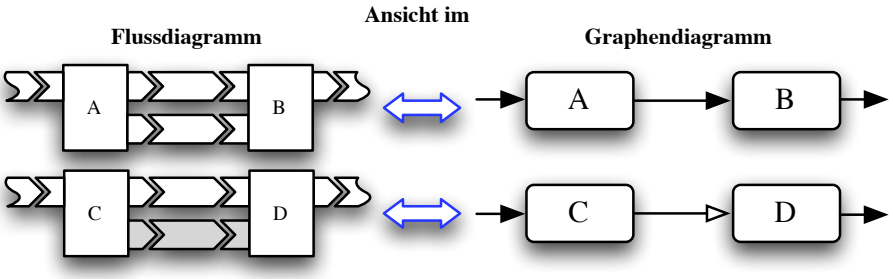


Abbildung 3.77.: Pfeile im Graphendiagramm.

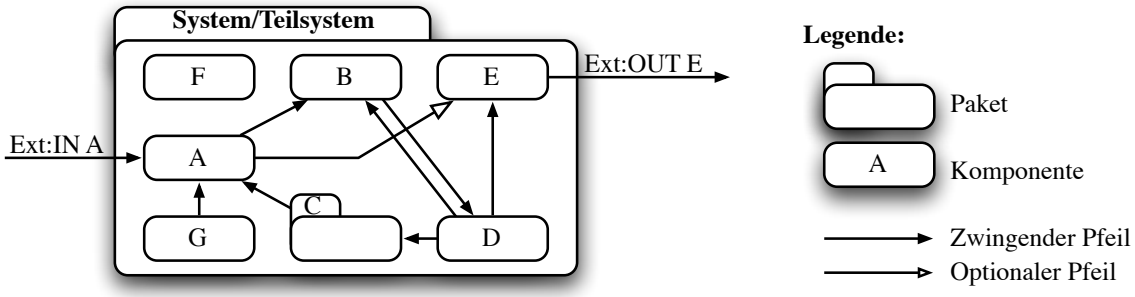


Abbildung 3.78.: (U)CML-Graphendiagramm: Vereinfachte Darstellung eines Systems und deren Bestandteile (Pakete und Komponenten).

In der Abbildung 3.78 ist eine optionale Verbindung zwischen der Komponente *A* und der Komponente *E* abgebildet. Alle anderen Verbindungen in diesem System sind zwingend. Das Paket C ist auch hier wieder in „Black-Box“ Darstellung modelliert. Der Inhalt ist auf dieser Hierarchieeben nicht sichtbar. □

- **Matrixdiagramm**
  Im Matrixdiagramm liegt der Fokus auf einer möglichst kompakten Darstellung der Verbindungen (Flusspfeile) zwischen den Paketen und Komponenten des Systems sowie dem System und der Umwelt. Das (U)CML-Matrixdiagramm entspricht dabei im Wesentlichen der bereits im Kapitel „3.4.2.1 *Das ursprüngliche Systems Engineering Element-Konzept aus dem Jahre 1993*“ ab Seite 122 eingeführten Matrixdarstellung des Systems Engineering Element-Konzepts. Das (U)CML-Matrixdiagramm wurde gegenüber dem ursprünglichen SE Matrixdiagramm um eine weitere Spaltenart – der Systemumwelt Spalte – erweitert, die die Verbindung des Systems mit der Umwelt repräsentiert. Die Systemumwelt Spalten werden am rechten Rand der „normalen“ Matrix für jeden Ein- und Ausgang des Systems einzeln hinzugefügt. Anschließend wird in jeder Spalte der Systemumwelt Sektion ein Quadrat oder Kreis eingetragen, wenn das entsprechenden Paket oder die Komponenten eine Verbindung mit der Umwelt hat. Ein Quadrat bedeutet, dass die entsprechende Verbindung zwingend ist, während ein Kreis eine optionale Verbindung repräsentiert.

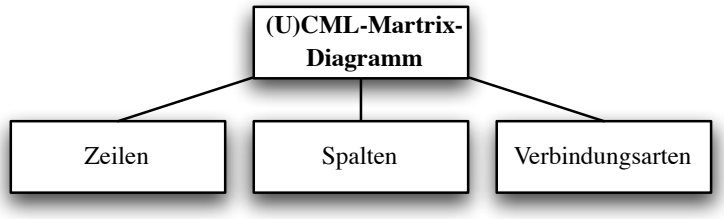


Abbildung 3.79.: Aufbau und Struktur des (U)CML-Matrixdiagramms.

In der Abbildung 3.79 sind die wesentlichen Bestandteile des Matrixdiagramms dargestellt. Der Hauptbestandteil jeder Matrix sind ihre Zeilen und Spalten. Dort werden die Pakete und Komponenten, aus denen das System besteht, eingetragen. Daran anschließend werden die System-Umwelt Spalten rechts an die quadratische Matrix angefügt. Nun werden alle Verbindungen zwischen den Paketen und Komponenten des Systems und dem System und der Umwelt in die Matrix eingetragen.

Beispiel 43: Matrixdiagramm
Als Grundlage für dieses Beispiel dient wieder das System, das in der Abbildung 3.71 als Flussdiagramm dargestellt ist. Alle Pakete (*C*) und Komponenten (*A*, *B*, *D*, *E*, *F*, *G* und *H*) des Systems werden in die Zeilen und Spalten der Matrix eingetragen. Das Systempaket wird nicht explizit in die Matrix eingetragen, sondern entspricht der gesamten quadratischen Matrix. Nach

diesem Schritt werden die Systemumwelt Spalten rechts an die quadratische Matrix angefügt. Daran anschließend werden die einzelnen Verbindungen der Pakete und Komponenten und des Systems an den entsprechenden Stellen eingetragen.

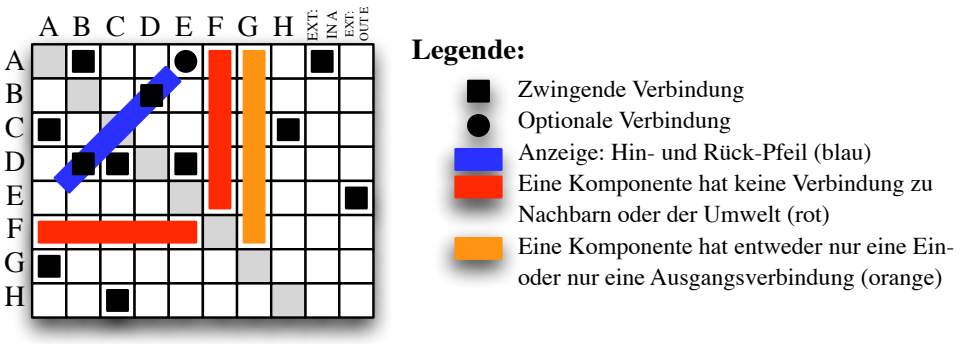


Abbildung 3.80.: (U)CML-Matrixdiagramm: Kompakte Darstellung des Aufbaus- und der Struktur eines Systems.

Nachdem alle Verbindungen in die Matrix eingetragen wurden, kann diese ausgewertet werden. Wenn ein Paket oder eine Komponente keine Verbindung zu einer anderen aufweist, wird die entsprechende Zeile bzw. Spalte rot markiert (vgl. Zeile/Spalte *F*). Hat ein Paket oder eine Komponente nur einen Ein- oder Ausgang, wird die entsprechende Zeile orange eingefärbt (vgl. Spalte *G*). Mit Hilfe des Matrixdiagramms können nicht nur Fehler einfach lokalisiert werden, sondern zum Beispiel auch bestimmte Abhängigkeiten zwischen einzelnen Paketen und Komponenten. So wird z.B. eine Kommunikationsverbindung zwischen zwei Paketen oder Komponenten als Diagonale dargestellt (vgl. *D*) und die entsprechenden Zellen durch eine diagonale blaue Linie verbunden. □

Nachdem nun die wichtigsten vier Diagrammarten der (U)CML eingeführt wurden, folgt in den nächsten Unterabschnitten diese Kapitels die Einführung der wichtigsten Sprachelemente der (U)CML. Besonderer Wert wird dabei auf die unterschiedlichen Sprachelemente des Flussdiagramms gelegt weil sich die anderen Diagrammarten aus dem Flussdiagramm ableiten lassen.

3.6.2. Sprachelemente der (U)CML

Nachdem im vorangegangenen Abschnitt die vier unterschiedlichen Diagrammarten der (U)CML sowie einige Sprachelemente kurz vorgestellt wurden, folgt in diesem Kapitel eine ausführliche Beschreibung aller Sprachelemente der (U)CML. Das Hauptaugenmerk liegt dabei sowohl auf der Beschreibung der Funktionen als auch der Darstellung der einzelnen Elemente im Flussdiagramm. In der nachfolgenden Aufzählung ist der Inhalt dieses Kapitels tabellarisch aufgeführt:

- Eineindeutigkeit der Namen – (U)CML-Namensräume
- (U)CML-Pakete
- (U)CML-Komponente
- (U)CML-Paketschnittstellen und Komponentenanschlüsse
- (U)CML-Flusspfeilarten
- Externer (U)CML-Systemeingangs- und -ausgangspfeil
- (U)CML-Beschreibungsfelder
- (U)CML-BUS-System

3.6.2.1. Eineindeutigkeit der Namen – (U)CML-Namensräume

In (U)CML ist es zwingend notwendig, dass sämtliche Pakete und Komponenten innerhalb eines Modells einen eineindeutigen Namen (Bezeichner/Identifikator) besitzen, anhand dessen sie im gesamten Modell eineindeutig identifiziert werden können[194]. Das heißt, dass jeder Name eines Pakets oder einer Komponente in einem System nur ein Mal vergeben werden darf. Diese Einschränkung ist notwendig, damit jedes Paket oder Komponente des Systems zweifelsfrei anhand ihres Namens identifiziert werden kann.

Im Gegensatz dazu dürfen die Namen von (U)CML-Steckern, -Buchsen und -Schnittstellen mehrfach in einem System verwendet werden, da sie anhand ihrer Zugehörigkeit zu einem Paket oder einer Komponente stets zweifelsfrei identifiziert werden können (vgl. Kapselung bzw. Namensräume in C++). Die Pakete bzw. Komponenten bilden einen lokalen Namensraum für die Schnittstellen bzw. Stecker und Buchsen. Innerhalb eines Pakets bzw. einer Komponente müssen sie jedoch ebenfalls exklusiv sein – dürfen also nicht zwei Mal vergeben werden, da es sonst ebenfalls zu unerwünschten Verwechselungen/Mehrdeutigkeiten kommen kann.

[194] Die Eineindeutigkeit von Bezeichnern ist z.B. auch in sämtlichen Programmiersprachen gegeben. Außerdem wird die Eineindeutigkeit der Bezeichner auch im SE Element-Konzept verwendet (vgl. Abschnitt 3.4.2 *Das Systems Engineering Element-Konzept – „Die Münchner Schule“* ab Seite 116), auf dem die (U)CML zum Teil aufsetzt.

Das folgende Beispiel illustriert diesen verbal beschriebenen Sachverhalt anhand eines einfachen in (U)CML modellierten Beispielsystems, in dem mehrere Fehler enthalten sind.

Beispiel 44: Eineindeutigkeit von Paket- und Komponentennamen eines in (U)CML modellierten Systems
Die Abbildung 3.81 zeigt ein in (U)CML modelliertes System in der Flussansicht. In dem System wurde sowohl gegen die Eineindeutigkeit von Paket- bzw. Komponentennamen, als auch gegen die eineindeutige Bezeichnung von Schnittstellen innerhalb des Namensraums eines Pakets verstoßen. Die Komponente *K2* im Systempaket hat den selben Namen wie die Komponente *K2*, die sich innerhalb des Pakets *P1* befindet (2). Dies ist in (U)CML nicht erlaubt. In (U)CML ist es ebenfalls nicht gestattet, den Namen einer Komponente gleich dem Namen eines Pakets zu wählen (1).

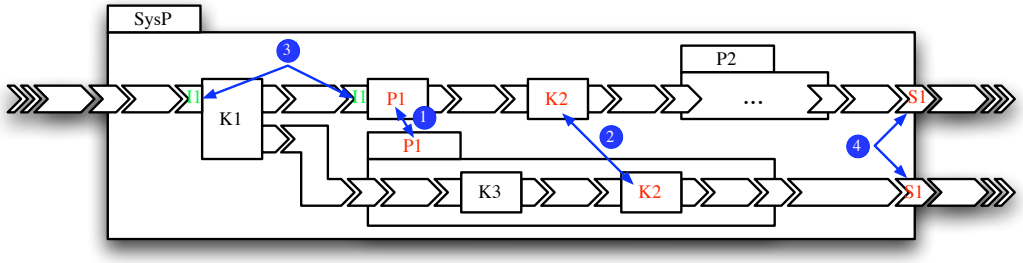


Abbildung 3.81.: Eineindeutigkeit der Namen von Paketen und Komponenten.

Außerdem ist es in (U)CML verboten zwei Schnittstellen eines Pakets mit dem selben Namen zu versehen (4). Im Gegensatz dazu ist es in (U)CML erlaubt, z.B. die Eingänge (Buchsen) von zwei unterschiedlichen Komponenten gleich zu bezeichnen (3), da diese anhand der eineindeutigen Komponentenbezeichnung jederzeit zweifelsfrei identifiziert werden können. □

3.6.2.2. (U)CML-Pakete

In (U)CML repräsentiert ein Paket eine logische und/oder physikalische, in sich abgeschlossene Einheit ohne eigene Funktionalität – im Gegensatz zu einer (U)CML-Komponente (siehe Kapitel 3.6.2.3 auf Seite 165). Pakete können ihrerseits beliebig viele weitere Pakete bzw. Komponenten enthalten. Durch die Schachtelung der Pakete entsteht die hierarchische Struktur des Systemmodells.

System und Systempaket
Basierend auf dem allgemein gültigen Systembegriff nach Dr. Negele (vgl. Definition: 1.2 auf Seite 12) hat jedes (U)CML-System auf der obersten Hierarchieebene genau ein ausgezeichnetes Systempaket, das die Grenze zwischen dem System auf der einen Seite und der Umwelt auf der anderen repräsentiert. Jedes Systempaket besitzt Schnittstellen (genauer: Systempaketschnittstellen), mit deren Hilfe das Systempaket die Systemgrenze überwinden kann – es ihm also ermöglicht, Daten und Informationen mit der Umwelt kontrolliert auszutauschen (*Schnittstelle* in der Abbildung 1.11)[195].

Definition 3.4 ***(U)CML-System und Systempaket***
Jedes in (U)CML modellierte System S hat auf der obersten Hierarchieebene – Ebene 1 – ***genau ein*** *ausgezeichnetes* ***Systempaket*** $\mathbf{P_S}$*. Das Systempaket P_S stellt die Schnittstelle, genauer die Systempaketschnittstelle, zwischen dem System S und der es umgebenden Umwelt U mit Hilfe von externen Systemeingangs- bzw. Systemausgangspfeilen her:*

$$P_S \underset{input}{\overset{output}{\rightleftharpoons}} U$$

Dabei bedeutet: $P_S \xrightarrow{output} U$ den Fluss vom Systempaket P_S in die Umwelt U, während $P_S \xleftarrow{input} U$ den Fluss von der Umwelt in das Systempaket beschreibt.

Beispiel 45: (U)CML-Systempaket
Die Abbildung 3.82 auf der nächsten Seite zeigt ein einfaches in (U)CML modelliertes und in der Flussansicht dargestelltes System. Das System hat auf der obersten Hierarchieebene genau ein ausgezeichnetes Paket (das Systempaket *SysP*) mit einem externen Systemeingangspfeil (1) und zwei externen Systemausgangspfeilen (2). Das Systempaket repräsentiert die Grenze zwischen dem System auf der einen Seite und der Umwelt auf der anderen. Dabei wird die Umwelt nicht explizit modelliert und dargestellt. Es werden lediglich die Schnittstellen (3) des Systems zur Umwelt sowie die Kommunikation des Systems mit der Umwelt explizit modelliert und dargestellt. □

Anmerkung zur Definition 3.4 sowie der Abbildung 3.82:
In einem in (U)CML modellierten System gibt es *genau* ein ausgezeichnetes *Systempaket* (in Abb. 3.82 *SysP*) in dem alle weiteren Systembestandteile (Pakete, Komponenten, Flusspfeile etc.) des Systems enthalten sind (in Abb. 3.82 Komponente *K* mit einer Eingangsbuchse und zwei Ausgangssteckern sowie drei Flusspfeilen). Das Systempaket stellt die Schnittstelle (3) zwischen dem

[195] Nähere Informationen zur Kommunikation eines in (U)CML modellierten Systems mit seiner Umwelt finden Sie im Kapitel „3.6.2.5.3 *Externer (U)CML-Systemeingangs- und -ausgangspfeil*“ ab Seite 194.

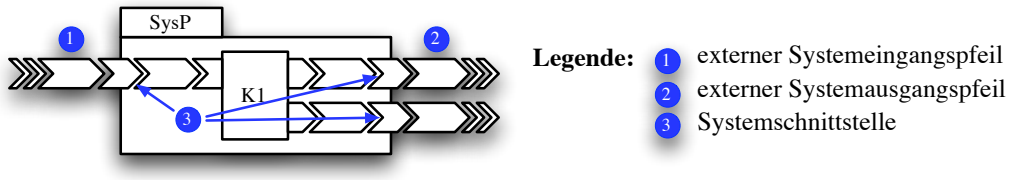


Abbildung 3.82.: (U)CML-Systempaket mit einem externen Systemeingangspfeil und zwei externen Systemausgangspfeilen.

Systempaket *SysP* auf der einen und der Umwelt auf der anderen Seite zur Verfügung. Die Systempaketschnittstelle hat keine eigene Funktionalität. Das System kann ausschließlich über diese explizit modellierte Schnittstelle mit der Umwelt kommunizieren (vgl. Kapselung). Dadurch wird eine Beeinflussung des Systems durch nicht modellierte Verbindungen ausgeschlossen. Diese Eigenschaft ist besonders für die Bestimmung der Kompatibilität entscheidend.

Eigenschaften der (U)CML-Pakete sowie der Paketschnittstellen
Jedes (U)CML-Paket repräsentiert einen Container, in dem beliebig viele Systembestandteile (Pakete, Komponenten und Flusspfeile) enthalten sein können. In den beiden nachfolgenden Definitionen 3.5 und 3.6 sind die grundlegenden Eigenschaften jedes (U)CML-Pakets definiert. Abgeschlossen wird die Beschreibung der Eigenschaften eines Pakets mit der Definition 3.7 auf der nächsten Seite, in der die Schachtelung von Paketen definiert ist. Durch die Schachtelung kann ein System hierarchisch gegliedert werden (Siehe hierzu: Kapitel „2.2 *Modellbildung*“ ab Seite 31.). Die hierarchische Gliederung eines Systems wird auch als Dekomposition eines Systems in handhabbare Teile bezeichnet.

Anmerkung:
Alle hier definierten Eigenschaften eines Standardpaketes gelten insbesondere auch für das ausgezeichnete Systempaket. Eine Ausnahme bildet die Schachtelung von Paketen, da es in einem System lediglich ein Systempaket geben darf (vgl. Def.: 3.4).

Definition 3.5 ***(U)CML-Standardpaket***
*Ein (U)CML-**Standardpaket** stellt einen **Container** dar, der beliebig viele weitere **Pakete** oder andere Systembestandteile wie z.B. **Komponenten** oder **Flusspfeile** enthalten kann. Des Weiteren gilt: (U)CML-Pakete haben **keine eigene Funktionalität**.*

Für alle (U)CML-Paket gilt: Ein Paket muss mindestens einen Eingang und einen Ausgang haben. Die Paketeingänge- bzw. -ausgänge bilden die Schnittstelle, genauer die Paketschnittstelle, des Pakets. In der nachfolgenden Definition ist die Schnittstelle eines Pakets formal definiert.

Definition 3.6 ***Schnittstellen eines (U)CML-Pakets***
*Sei $i, j \in \mathbb{N}$ mit $1 \leq i, j < \infty$. Des Weiteren sei P ein Paket, $I := \{IN_1, IN_2, \ldots, IN_i\}$ die Menge aller Eingänge (Eingangsschnittstelle) des Pakets P und $O := \{OUT_1, OUT_2, \ldots, OUT_j\}$ die Menge aller Ausgänge (Ausgangsschnittstelle) des Pakets P. Also gilt: Jedes Paket P muss **mindestens einen** Eingang I **und mindestens einen** Ausgang O besitzen:*

$$Q \xrightarrow{I} P \xrightarrow{O} S$$

Mit Q = Quelle, S = Senke und $I \neq \{\emptyset\}$ und $O \neq \{\emptyset\}$.

Anmerkungen zur Definition 3.6:

- Die Quelle bzw. die Senke kann entweder ein anderes Paket – genauer die Schnittstelle dieses Pakets – oder eine Komponente – genauer der Ein- bzw. Ausgang der Komponente – sein. Die Verbindung wird mit einem Flusspfeil hergestellt, der den Ein- bzw. Ausgang entsprechend verbindet. Flusspfeile werden im Kapitel „3.6.2.5 *(U)CML-Flusspfeilarten*“ ab Seite 184 erläutert.
- Pakete dienen zur Abgrenzung und Gliederung von komplexen Systemen. Dabei bilden die Paketschnittstellen die „Grenze“ zwischen dem Inneren eines Pakets – dem Paketinhalt – wie z.B. weiteren Paketen und Komponenten und dem äußeren Umfeld eines Pakets – der Paketumwelt.

Nachdem nun sämtliche Grundeigenschaften der (U)CML-Pakete definiert wurden, folgt nun die Definition sowie die Anwendung der hierarchischen Strukturierung bzw. der Schachtelung der Pakete. Um ein System „überschaubar“ zu gestalten, ist es ratsam, logisch oder physikalisch zusammengehörige Teilkomponenten des Systems (auch Subsysteme oder Module genannt) in jeweils ein Paket zu legen. Durch die konsequente Anwendung der hierarchischen Strukturierung von Systemen kann die Systemstruktur bzw. die „wahrgenommene Komplexität“ eines Systems entscheidend beeinflusst werden. Im Systems Engineering bzw. der Softwareentwicklung ist das Prinzip der hierarchischen Dekomposition von Systemen ein möglicher Ansatz, komplexe Systeme beherrschbar zu gestalten [PDII06][Ver06a]. Zusätzlich zur hierarchischen Strukturierung des Systems durch die Schachtelung von Paketen bietet die (U)CML das Konzept der unterschiedlichen Ansichten auf ein Paket[196].

[196] Siehe hierzu: „Ansichten auf ein (U)CML-Paket“ auf Seite 164 bzw. das Kapitel „3.8 *(U)CML-ed, ein graphischer Editor für die Kompatibilitätsmodellierungssprache (U)CML*“ ab Seite 254.

Definition 3.7 ***Schachtelung von Paketen***
Sei S das System, P_S das Systempaket, P ein Standardpaket und $i, j \in \mathbb{N}$ mit $1 \leq i, j < \infty$. Das Systempaket P_S bzw. das Standardpaket P kann beliebig viele weitere Pakete P_i und Komponenten K_j enthalten.

Im nachfolgenden Beispiel ist die Schachtelung von (U)CML-Paketen anhand eines einfachen Systems dargestellt. Zusätzlich zur Schachtelung der Pakete im Flussdiagramm ist die hierarchische Struktur des Systems im Baumdiagramm dargestellt.

Beispiel 46: Schachtelung von Paketen
Die Abbildung 3.83 zeigt auf der linken Seite die *Schachtelung* von sechs Paketen. Auf der obersten Hierarchieebene befindet sich das ausgezeichnete Systempaket *System*. Auf der zweiten Hierarchieebene befinden sich die Pakete *Paket B* und *Paket E*. Das Paket *Paket B* enthält die Pakete *Paket B* und *Paket C*. Auf der untersten Hierarchieebene des modellierten Systems befindet sich das Paket *Paket D*.

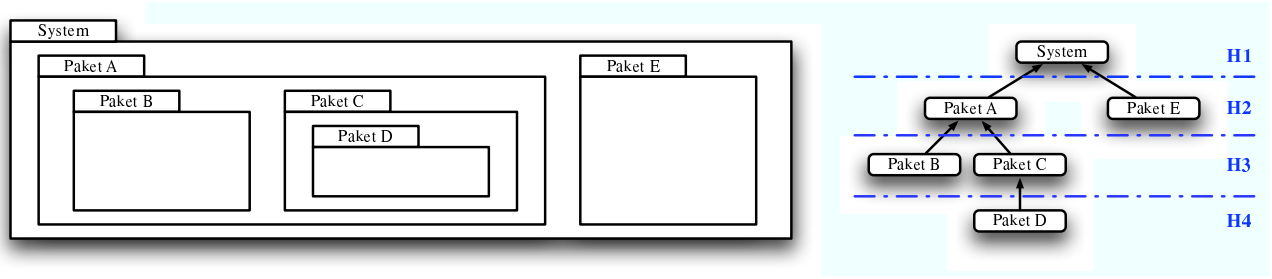


Abbildung 3.83.: Schachtelung von (U)CML-Paketen.
Links: Darstellung des Systems in Flussansicht
Rechts: Darstellung des Systems in Baumansicht

Auf der rechten Seite der Abbildung ist das selbe System in der Baumansicht dargestellt. Dort lässt sich die hierarchische Struktur eines Systems kompakter darstellen als in einem Flussdiagramm, da lediglich die Struktur des Systems dargestellt ist. □

Darstellung von (U)CML-Paketen im Flussdiagramm
In einem (U)CML-Fluss- bzw. -Graphendiagramm gibt es, wie bereits bei der Vorstellung der (U)CML-Diagrammarten erwähnt, unterschiedliche Arten der Darstellung der verwendeten Pakete. Zum einen können Pakete in der so genannten „Black-Box Darstellung“ (vgl. Abbildung 3.84 links) gezeigt werden, in der lediglich die äußere Schnittstelle des Pakets dargestellt wird, während der Inhalt des Pakets dabei verborgen[197] bleibt. Und zum anderen in der „Glass-Box Darstellung“ (Abbildung 3.84 rechts) in der der gesamte Inhalt eines Pakets und nicht nur dessen Schnittstelle nach außen dargestellt wird.

Definition 3.8 ***Ansichten auf ein (U)CML-Paket***
*Jedes (U)CML-Paket kann sowohl in der **Glass-Box Darstellung**, als auch in der **Black-Box Darstellung** dargestellt werden. In der Black-Box Darstellung eines Pakets ist lediglich die Schnittstelle des Pakets dargestellt, dessen Inhalt nicht. Dies steht im Gegensatz zur Glass-Box Darstellung, die sowohl die Schnittstelle des Pakets als auch dessen Inhalt darstellt. Dabei gilt: Ungeachtet von der jeweiligen Darstellungsart ist der Inhalt des Pakets identisch.*

In (U)CML werden Pakete als Rechtecke mit einem kleinen Fähnchen links oben dargestellt, in dem der eineindeutige Name des Pakets eingetragen wird. Die Abbildung 3.84 zeigt die Grundstruktur jedes (U)CML-Pakets in der Fussdiagrammansicht.

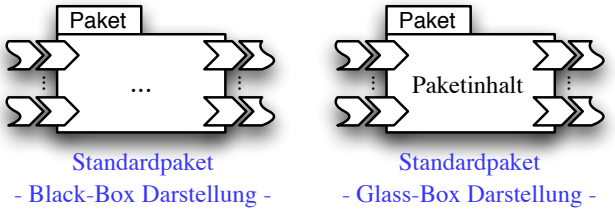


Abbildung 3.84.: Graphische Darstellung von (U)CML-Paketen – im Flussdiagramm.

Beispiel 47: Ansichten auf ein (U)CML-Paket im Flussdiagramm
Die Abbildung 3.85 auf der nächsten Seite zeigt auf der linken Seite ein in (U)CML modelliertes Teilsystem in Black-Box Darstellung. In der Black-Box Darstellung werden lediglich die Schnittstellen des Pakets dargestellt, an denen weitere Systembestandteile angeschlossen sein können.

[197] Vgl. Definition 2.8 *Information Hiding* auf Seite 49.

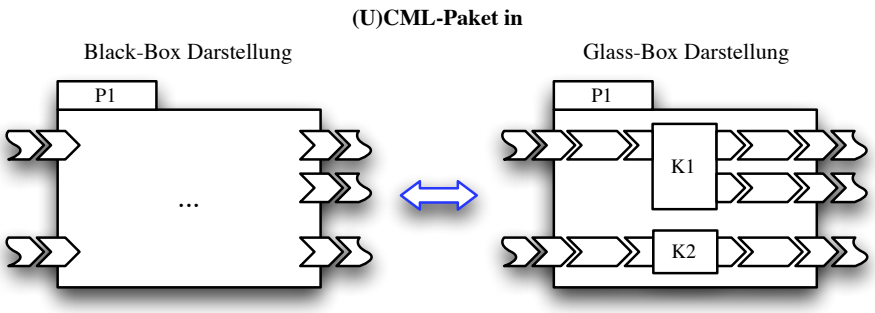


Abbildung 3.85.: Ansichten eines (U)CML-Pakets.

Auf der rechten Seite ist das selbe Teilsystem in der Glass-Box Darstellung abgebildet. Hier ist der Inhalt des Pakets *P1*, also die Komponenten *K1* und *K2* sowie die Flusspfeile zwischen den Paketschnittstellen und den Steckern und Buchsen der Komponenten explizit dargestellt. □

Wichtig für das weitere Verständnis der beiden unterschiedlichen Ansichten auf ein Paket ist es, dass sich unabhängig von der jeweiligen Darstellungsart des Pakets, weder dessen Inhalt noch innere Struktur verändert. Die innere Struktur und der Aufbau des Pakets ist in (U)CML vollständig entkoppelt von dessen gegenwärtiger Repräsentation[198]. Es ist also unerheblich, ob ein Paket *mit* oder *ohne* Inhalt dargestellt wird, sofern das Paket den (U)CML-Sprachregeln[199] genügt. Um die beiden unterschiedlichen Darstellungsarten eines Pakets zu realisieren, wird das so genannte „kontextsensitive Zooming“ verwendet, das dafür Sorge trägt, dass stets nur so viele Details einer Hierarchieebene dargestellt werden wie notwendig. Nähere Informationen zum kontextsensitiven Zooming finden Sie im Kapitel 3.8.2 auf Seite 258 und bei der Beschreibung des (U)CML-Editors *(U)CML-ed* ab Seite 254.

Darstellung von (U)CML-Paketen im Baum-, Graphen- und Matrixdiagramm

Nachdem in der Abbildung 3.84 ein (U)CML-Paket im Flussdiagramm dargestellt wurde, folgt in der Abbildung 3.86 die Repräsentation eines Pakets durch die restlichen (U)CML-Diagrammarten, also das Baum-, Graphen- und Matrixdiagramm.

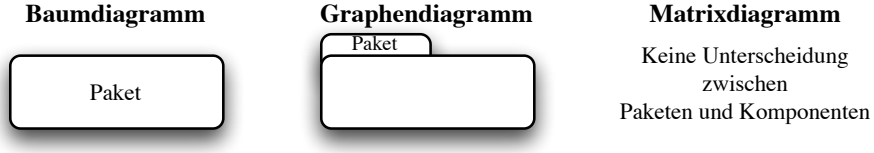


Abbildung 3.86.: Graphische Darstellung (U)CML-Pakete – Baum-, Graphen- und Matrixdiagramm.

Sowohl im Baum- als auch im Graphendiagramm werden Pakete als Rechtecke mit abgerundeten Ecken dargestellt. Im Graphendiagramm hat jedes Paket zusätzlich einen Reiter in der linken oberen Ecke, in dem der eineindeutige Name des Pakets eingetragen ist. Im Baudiagramm wird der Paketname in der Mitte des Rechtecks notiert.

Das Matrixdiagramm stellt eine Ausnahme in der Modellierung von Pakten dar. In einem Matrixdiagramm gibt es keine Unterscheidung zwischen Paketen und Komponenten, da der Fokus in einem Matrixdiagramm auf der einfachen und kompakten Darstellung von Verbindungen zwischen Paketen und Komponenten liegt.

Zusammenfassung der Eigenschaften von (U)CML-Paketen

- In jedem System S gibt es genau ein ausgezeichnetes Systempaket P_S
- Jedes Paket hat einen eineindeutigen Namen, mit dessen Hilfe es im System identifiziert werden kann
- Pakete können ineinander geschachtelt sein – dies dient der Hierarchiebildung
- Jedes Paket muss mindestens einen Eingang (Buchse) und mindestens einen Ausgang (Stecker) haben
- Ein Paket kann sowohl in Black-Box, als auch in Glass-Box Darstellung gezeichnet werden

3.6.2.3. (U)CML-Komponente

Eine Komponente repräsentiert in (U)CML die kleinste logisch oder physikalisch modellierbare abgeschlossene Funktionseinheit. Wie schon die Bezeichnung „Funktionseinheit“ assoziiert, wird die Gesamtfunktionalität eines Systems ausschließlich durch die Summe der im System enthaltenen Komponenten gebildet. Dabei wird in (U)CML zwischen dem *statischen* und dem *dynamischen Verhalten* der *Anschlüsse*[200] und dem *inneren Verhalten* einer Komponente unterschieden. Sowohl das statische als auch das dynami-

[198] Siehe hierzu: Kapitel „3.8.2 *Kontextsensitives Zooming und Black-Box Ansicht des Werkzeugs (U)CML-ed* “ ab Seite 258.

[199] Vgl. Abschnitt „3.6.5.1 *(U)CML-Sprachregelwerk*“ ab Seite 236

[200] Die Anschlüsse einer Komponente, die Stecker und Buchsen (vgl. Kapitel „3.6.2.4.1 *Standardpaketschnittstellen und Standardkomponentenanschlüsse*“ ab Seite 176), werden manchmal auch als *Komponentenschnittstelle* bezeichnet.

sche Verhalten der Komponentenanschlüsse (d.h. der Stecker und Buchsen einer Komponente) wird für die Kompatibilitätsprüfung herangezogen; das interne Verhalten der Komponente jedoch nicht. Sämtliche Eigenschaften einer Komponente werden nicht direkt in den „Komponentenrumpf“ eingetragen sondern werden zentral im Beschreibungsfeld[201] der Komponente abgelegt.

Im Gegensatz zu (U)CML-Paketen können Komponenten nicht weiter verfeinert werden. Aus diesem Grund werden Komponenten auch als *atomare Einheiten* bzw. *logische* oder *physikalische Entitäten* bezeichnet. Beispiele für einfache Komponenten sind: eine mechanische Schraube, ein elektrischer Schaltkreis oder eine C++ Methode. In (U)CML ist es auch möglich, komplexe Systeme bzw. Subsysteme als atomare Einheiten abhängig von der Abstraktion des Systems zu modellieren. Beispiele für komplexe Komponenten sind integrierte Schaltkreise (IC), eingebettete Systeme (Steuergeräte) oder ein Softwaremodule. Die Einschätzung, wann eine logische oder physikalische Einheit als atomare Komponente modelliert werden soll, hängt von der Granularität des zu modellierenden Systems bzw. den gegebenen Systemgrenzen ab (Siehe hierzu: Kapitel „2.2 *Modellbildung*“ ab Seite 31.).

Eigenschaften einer (U)CML-Komponente

Definition 3.9 ***(U)CML-Komponente***
Eine Komponente repräsentiert in (U)CML die ***kleinste logisch*** *oder* ***physikalisch abgeschlossene atomare Funktionseinheit****. Dabei gilt: Jede (U)CML-Komponente muss mindestens einen Eingang (Buchse) und einen Ausgang (Stecker) besitzen – die statische Komponentenschnittstelle (auch Komponentenanschlüsse oder einfach Anschlüsse genannt).*

Außerdem kann eine Komponente sowohl ein ***internes dynamisches****, als auch ein* ***dynamisches Anschluss-*** *bzw.* ***Schnittstellenverhalten*** *haben.*

Anmerkung zur Definition 3.9:

- Die Buchsen und Stecker der (U)CML-Komponente bilden zusammen die statischen Komponentenanschlüsse (auch Komponentenschnittstelle genannt). Für die statischen Komponentenanschlüsse gelten die gleichen Regeln wie für die Paketschnittstelle eines (U)CML-Pakets (vgl. Definition 3.6).

- Die Existenz der statischen Stecker und Buchsen einer Komponente ist die Voraussetzung für die Modellierung der dynamischen Eigenschaften der Stecker und Buchsen der Komponente.

- Eine (U)CML-Komponente muss kein explizit modelliertes internes Verhalten bzw. kein Komponentenanschlussverhalten besitzen. Für eine umfassende Kompatibilitätsprüfung ist die Modellierung des Komponentenanschlussverhaltens jedoch von Vorteil, weil dadurch sowohl die statischen Stecker und Buchsen der Komponente als auch das dynamische Verhalten der Anschlüsse verifiziert werden kann.

Darstellung von (U)CML-Komponenten im Flussdiagramm
Im Flussdiagramm werden Komponenten als Rechtecke mit einem eineindeutigen Namen in der Mitte des Rechtecks dargestellt. In der Abbildung 3.87 ist auf der linken Seite eine (U)CML-Komponente ohne inneres oder Komponentenanschlussverhalten dargestellt. Auf der rechten Seite der Abbildung ist eine Komponente mit Verhalten abgebildet. Dabei symbolisiert ein kleiner Kreis mit einem „V“ in der Mitte (in der rechten unteren Ecke der Komponente) die Existenz eines verhaltensbeschreibenden MSCs oder Automaten. Das *V* symbolisiert dabei lediglich, dass die Komponente ein dezidiert modelliertes Verhalten besitzt. Es wird keine Aussage darüber getroffen, ob die Komponente ein modelliertes Anschluss- oder internes Verhalten enthält. Im nächsten Abschnitt dieses Kapitels wird sowohl das innere Verhalten einer Komponente (ab Seite 170ff) als auch das Anschlussverhalten (ab Seite 171ff) eingeführt und erläutert.

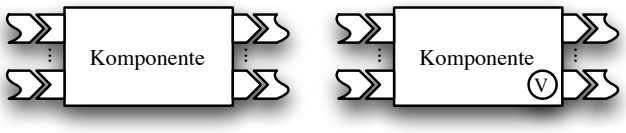


Abbildung 3.87.: (U)CML-Komponente mit Anschlüssen, Buchsen (links) und Steckern (rechts).

Zur Erleichterung der Navigation innerhalb eines komplexen Systems bzw. zur besseren Übersicht über die einzelnen Komponenten in einem System ist es möglich, Komponenten, Stecker/Buchsen/Schnittstellen und Pfeile, je nach ihrem Typ – *Keine Zuordnung*, *Elektrik/Elektronik*, *Mechanik* oder *Software* – einzufärben. Die nachfolgende Definition 3.10 auf der nächsten Seite beschreibt das Farbschema für eingebettete softwarelastige Systeme.

[201] Siehe hierzu: Kapitel „3.6.2.6 *(U)CML-Beschreibungsfelder*“ ab Seite 200.

Definition 3.10 ***Farbschema einer (U)CML-Komponente***
Jede (U)CML-Komponente kann in Abhängigkeit ihres Typs (Keine Zuordnung, Elektrotechnik, Mechanik oder Softwaretechnik) farbig eingefärbt werden. Anhand der Farbcodierung kann die Zugehörigkeit der Komponente zu einer der vier folgenden Gruppen festgelegt werden. Mögliche Farben einer Komponente sind:

- ***Schwarz – Keine Zuordnung***
  Die Komponente ist keiner Gruppe zugeordnet. Sie hat kein domänenspezifisches Beschreibungsfeld.
- ***Rot – Elektrische- bzw. Elektronische Komponenten***
  Die Komponente repräsentiert ein elektrisches oder elektronisches Bauteil oder Baugruppe.
- ***Grün – Mechanische Komponenten***
  Die Komponente repräsentiert ein mechanisches Bauteil oder eine Baugruppe.
- ***Blau – Softwarekomponenten***
  Die Komponente repräsentiert ein Softwaremodul oder eine Bibliothek.

Das eingeführte Farbschema wird beispielsweise in der Abbildung 4.36 auf Seite 290 für ein komplexes eingebettetes System verwendet, um die Übersichtlichkeit zu verbessern.

Anmerkungen zur Definition 3.10:

- Das Farbschema wird auch für die Färbung der Stecker und Buchsen eines Pakets bzw. einer Komponente verwendet. Außerdem werden die Flusspfeile (siehe Seite 184) und Paketschnittstellen nach dem gleichen Farbschema eingefärbt, sofern die Stecker-Buchse-Kombination den gleichen Farbcode haben und damit vom gleichen Typ sind. Ansonsten wird eine Warnung ausgegeben, da der Typ der Stecker-Buchse-Kombination nicht übereinstimmt (Siehe hierzu: Kapitel „3.6.5 *(U)CML-Sprach- und Kompatibilitätsregelwerk*“ ab Seite 236.).
- Das hier definierte Farbschema gilt für eingebetete Systeme, die aus Elektrotechnik, Mechanik und Softwaretechnik bestehen. Es lässt sich jedoch einfach für beliebige andere Domänen anpassen. Im Abschnitt „3.6.8 *Anpassung und Erweiterung der (U)CML auf andere Fachdisziplinen*“ ab Seite 251 wurde das Farbschema auf die Domäne *Anforderungsmanagement* erweitert.

In der nachfolgenden Abbildung 3.88 sind vier Komponenten, die dem obigen Farbschema (Definition 3.10) genügen, abgebildet. Von links nach rechts: eine Komponente ohne feste Zuordnung, eine elektrische/elektronische Komponente, eine mechanische Komponente sowie eine Softwaretechnik Komponente.

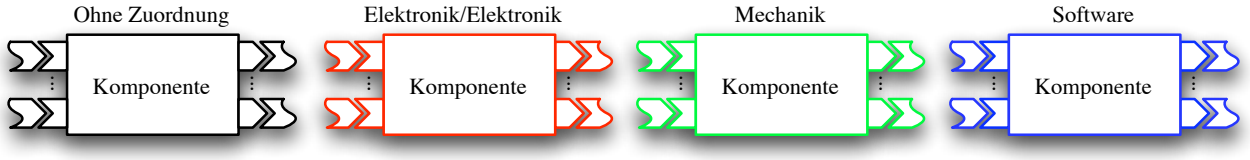


Abbildung 3.88.: Farbschema einer (U)CML-Komponente

Zusätzlich zum Farbschema einer Komponente kann ein Symbol innerhalb einer Komponente angezeigt werden. Dadurch kann dieses, z.B. bei einem elektrischen Bauteil dargestellt werden, um die Funktionalität der Komponente „auf einen Blick“ zu ermöglichen. In der Abbildung 3.89 ist die Komponente *NAND* mit einem elektronischen Schaltzeichen abgebildet.

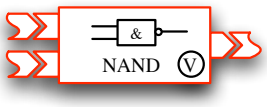


Abbildung 3.89.: Symbolische Darstellung der Funktionalität einer (U)CML-Komponente

Wird ein Symbol innerhalb einer Komponente angezeigt, so wandert der Name der Komponente in die Mitte des unteren Randes der Komponente.

Darstellung von (U)CML-Komponenten im Baum-, Graphen- und Matrixdiagramm
Die Abbildung 3.90 auf der nächsten Seite zeigt eine Komponente in der Baum-, Graphen- und Matrixansicht. In der Baumansicht (links) wird eine Komponente als Ellipse mit dem Namen der Komponente in der Mitte gezeichnet, während im Graphendiagramm (Mitte) die Komponente als Rechteck mit abgerundeten Ecken und dem Komponentennamen in der Mitte dargestellt wird. Im Matrixdiagramm (rechts) wird zwischen der Darstellung einer Komponente und eines Pakets nicht unterschieden.

Anmerkung:
Das eingeführte Farbschema für Komponenten (Definition 3.10) wird auch im Baum- bzw. Graphendiagramm zur leichteren Identifikation der Komponententypen verwendet.

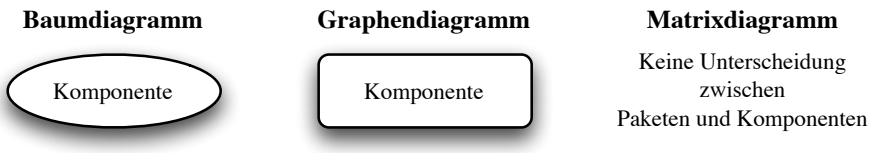


Abbildung 3.90.: Graphische Darstellung (U)CML-Komponente – Baum- Graphen- und Matrixdiagramm

3.6.2.3.1. Innere Aufbau und Struktur einer (U)CML-Komponente

Eine (U)CML-Komponente repräsentiert ähnlich wie ein (U)CML-Paket einen Container, in dem alle für die Beschreibung der *Funktionalität* bzw. das *Verhalten* der Komponente wichtigen Inhalte – *Methoden, Parameter, Diagramme, Tabellen* etc. – abgelegt werden können. Zusätzlich zur Verhaltensbeschreibung einer Komponente können beliebige *Zusatzinformationen – Schaltzeichen, CAD Zeichnungen, verbale Beschreibungen* etc. – im Komponentenrumpf abgelegt werden.

Im Gegensatz zu einem strukturbildenden Paket darf eine Komponente keine weiteren (U)CML-Strukturen wie z.B. Pakete, Flusspfeile oder Schnittstellen enthalten[202]. In der nachfolgenden Definition ist der innere Aufbau und die Struktur einer (U)CML-Komponente definiert:

Definition 3.11 ***Innerer Aufbau und Struktur einer (U)CML-Komponente***
Eine (U)CML-Komponente repräsentiert einen Container, in dem alle für die ***funktionale Modellierung*** *bzw. für die* ***Verhaltensbeschreibung*** *der Komponente notwendigen Informationen hinterlegt werden. Des Weiteren können* ***Zusatzinformationen****, die die Komponente beschreiben, eingetragen werden.*

Anmerkung zur Definition 3.11:
Um die Kompatibilität zu anderen Modellierungssprachen bzw. -werkzeugen wie z.B. UML/UML2 oder (v)Sys ((v)Sys-ed) sicher zu stellen, wird der Inhalt einer Komponente nicht in der Komponente selbst, sondern im korrespondierenden Beschreibungsfeld[203] abgelegt. Durch die „Verschiebung“ des Inhalts von der Komponente in das korrespondierende Beschreibungsfeld sind alle für die Spezifikation der Komponente notwendigen Informationen und Daten an einer zentralen Stelle abgelegt. Außerdem ist durch die zentrale Speicherung aller Daten im Beschreibungsfeld eine einfache Erweiterung des „Inhalts“ einer Komponente jederzeit möglich, ohne den inneren Aufbau der (U)CML-Komponente verändern zu müssen.

Beispiel 48: Innerer Aufbau einer (U)CML-Komponente
Die Abbildung 3.91 zeigt die Verschiebung des Inhalts einer Komponente in das korrespondierende Beschreibungsfeld. Auf der linken Seite der Abbildung ist die Komponente *NAND* mit zwei Eingängen ($I1$ und $I2$) und einem Ausgang (O) abgebildet. Auf der rechten Seite ist das korrespondierende Beschreibungsfeld dargestellt. Im unteren Teil des Beschreibungsfelds *Funktionalität* (blauer Rahmen) ist die „innere Struktur“ der Komponente *NAND* abgelegt. Im Beispiel hat die Komponente *NAND* drei Inhalte:

- ein Schaltzeichen
- ein MSC – Message Sequence Chart
- eine Funktion $O = \neg(I_1 \wedge I_2)$

Durch diese drei Eintragungen im Beschreibungsfeld ist sowohl der innere Aufbau, die Struktur, als auch die Funktionalität (Verhalten) der Beispielkomponente *NAND* vollständig beschrieben.

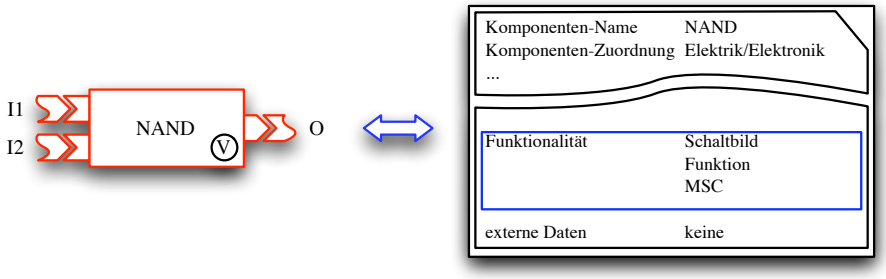


Abbildung 3.91.: (U)CML-Komponente mit korrespondierendem Beschreibungsfeld.

□

Anmerkung:
Im weiteren Verlauf dieser Arbeit wird stets vom „Inhalt einer Komponente“, also den inneren, verhaltensbeschreibenden Funktionen der Komponente gesprochen, obwohl der eigentliche Inhalt der Komponente im Beschreibungsfeld der Komponente gespeichert wird.

[202] Siehe hierzu: Kapitel „3.6.5.1 *(U)CML-Sprachregelwerk*“ ab Seite 236.
[203] Siehe hierzu: Kapitel „3.6.2.6 *(U)CML-Beschreibungsfelder*“ ab Seite 200.

Inhalt einer (U)CML-Komponente
Im Allgemeinen kann der Inhalt einer (U)CML-Komponente grob in die vier Kategorien – *Elektronik/Elektrik, Mechanik, Software* und *Sonstiges* – unterteilt werden. Für die Bestimmung der Kompatibilität von eingebetteten softwarelastigen Systemen werden jedoch nur die Felder *Elektrik/Elektonik, Mechanik* und *Informatik* benötigt. Alle Eintragungen im Feld *Sonstiges* werden nicht zur Bestimmung der Kompatibilität herangezogen. In der Abbildung 3.92 ist eine (U)CML-Komponente mit verschiedenen Inhalten dargestellt. Für die Bestimmung der Kompatibilität eines Systems ist diese Einteilung jedoch noch zu ungenau.

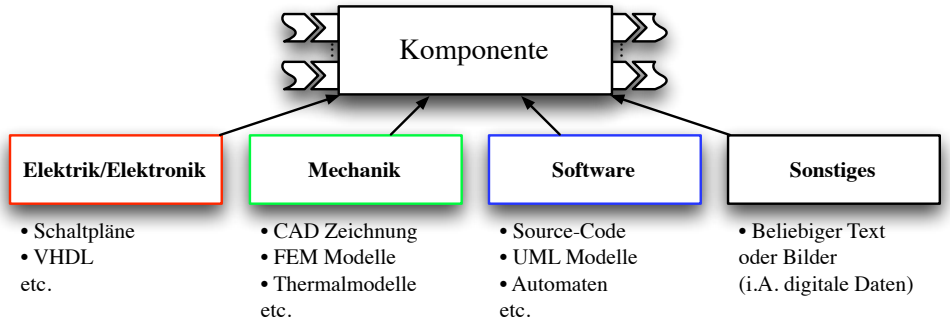


Abbildung 3.92.: Verschiedene Inhalte einer (U)CML-Komponente.

Aus diesem Grund ist es zwingend notwendig, eine eineindeutige strukturierte Beschreibung des Inhalts einer Komponente vorzunehmen.

Strukturelle Beschreibung des Inhalts einer (U)CML-Komponente
Der Inhalt einer (U)CML-Komponente kann in verschiedenen Ausprägungen vorliegen. Angefangen von einer technischen Zeichnung, verbal abgefassten Texten („Prosa“) und Tabellen bis hin zu speziellen Kenngrößen (Attributen) aus Datenblättern. Für die Modellierung und Bestimmung von Kompatibilität ist eine exakte, formale Notation der Eigenschaften (Attribute) und Funktionen (Methoden) einer Komponente zwingend notwendig. In den beiden Kapiteln *„Modellierung der Eigenschaften eines Objekts bzw. einer Klasse“* und *„Modellierung des Verhaltes eines Objekts/Klasse“* wurde gezeigt, wie sich Attribute (Eigenschaften) und verhaltensbeschreibende Funktionen formal spezifizieren lassen. Die dort vorgestellte formale Notation ist jedoch sehr mathematisch, so dass in einer domänenübergreifenden Modellierungssprache wie der (U)CML eine „benutzerfreudlichere“ Notationsart gewählt wurde, die fast so leistungsfähig wie eine mathematisch Beschreibung der Attribute und Funktionen ist – die Beschreibungsfelder. In den standardisierten Beschreibungsfeldern werden alle für die Bestimmung der Kompatibilität notwendigen Eigenschaften eingetragen. Die Abbildung 3.91 auf der vorherigen Seite zeigt rechts das Beschreibungsfeld einer Komponente, in dem sowohl die statischen Eigenschaften der Komponente, als auch das dynamische Verhalten (MSC, Funktionen) eingetragen sind. Zusätzlich ist ein elektrisches Schaltzeichen hinterlegt. Dieses kann allerdings für die Bestimmung der Kompatibilität nicht genutzt werden, da es sich nicht in Form von Eigenschaften oder Funktionen beschreiben lässt[204].

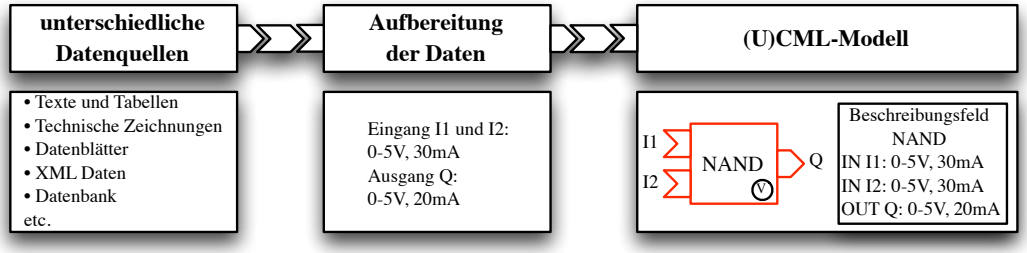


Abbildung 3.93.: Transformation unterschiedlicher Datenquellen in ein (U)CML-Modell.

Im Allgemeinen lassen sich komplexe zusammengesetzte Inhalte wie z.B. technische Zeichnungen, Stromlaufpläne, Tabellen etc. nicht weiter formalisieren, da die Attribute und Funktionen „gebunden“ sind. Beispielsweise lässt sich eine technische Zeichnung oder ein Bild (s.o. Schaltzeichen) nicht in einzelne Attribute und Funktionen zerlegen, so dass die enthaltenen Daten nicht für die Bestimmung der Kompatibilität herangezogen werden können. Lassen sich die Attribute und Funktionen jedoch extrahieren wie z.B. bei einer technischen Zeichnung die Abmessungen eines Bauteils, so können diese entsprechend „aufbereitet“ und ebenfalls in das Beschreibungsfeld eingetragen werden. Die Aufbereitung der Daten ist nicht immer möglich, so dass ein Vergleich der Eigenschaften (z.B. bei Bildern) nicht automatisch durchgeführt werden kann und der Mensch diese Aufgabe erledigen muss.

Der Umwandlungsprozess – angefangen von den unterschiedlichen Datenquellen über die eigentliche Umwandlung und Anpassung der Daten bis hin zum (U)CML-Modell – ist in Abbildung 3.93 beispielhaft dargestellt.

[204] Im Projekt INTERXPHASE bzw. der Dissertation von C. Eckl [Eck12] wird gezeigt (wie aus nahezu beliebigen Daten, wie beispielsweise CAD Zeichnungen oder Stromlaufplänen) Eigenschaften extrahiert werden können, die für die Modellierung und insbesondere für die Kompatibilitätsbestimmung notwendig sind. Siehe auch [BE08].

Eine ausführliche Beschreibung des Transformationsprozesses zur Überwindung der „semantischen Lücke“ zwischen den unterschiedlichen Datenquellen auf der einen, und den (U)CML-Beschreibungsfeldern auf der anderen Seite, wird für die kompatibilitätsrelevanten Daten eines Systems im Abschnitt „3.6.6 *Der (U)CML-Modellbildungsprozess*“ ab Seite 243 genauer erläutert.

3.6.2.3.2. Verhalten einer (U)CML-Komponente

Wie bereits bei der Einführung der (U)CML-Komponente angedeutet wurde, repräsentiert eine Komponente in (U)CML die kleinste, nicht weiter unterteilbare Funktionseinheit der (U)CML. Die Funktionalität bzw. das Verhalten einer Komponente kann dabei in das *statische* und das *dynamische Verhalten* der Anschlüsse (Komponentenschnittstelle) und das *innere Verhalten* einer Komponente unterteilt werden. In der Abbildung 3.94 ist in der Mitte eine elektrische/elektronische (U)CML-Komponente *NAND* mit zwei Eingängen und einem Ausgang dargestellt. Auf der linken Seite ist ein Stecker-/Buchsen-Beschreibungsfeld auszugsweise abgebildet, in dem sowohl der statische Verhaltensteil (blaue Box) als auch das Anschlussverhalten des Steckers/Buchse (rote Box) eingetragen ist. Rechts ist das innere Verhalten der Komponente *NAND* dargestellt (rote Box). Das innere Verhalten der Komponente steuert sowohl das statische als auch das dynamische Verhalten der Anschlüsse der Komponente.

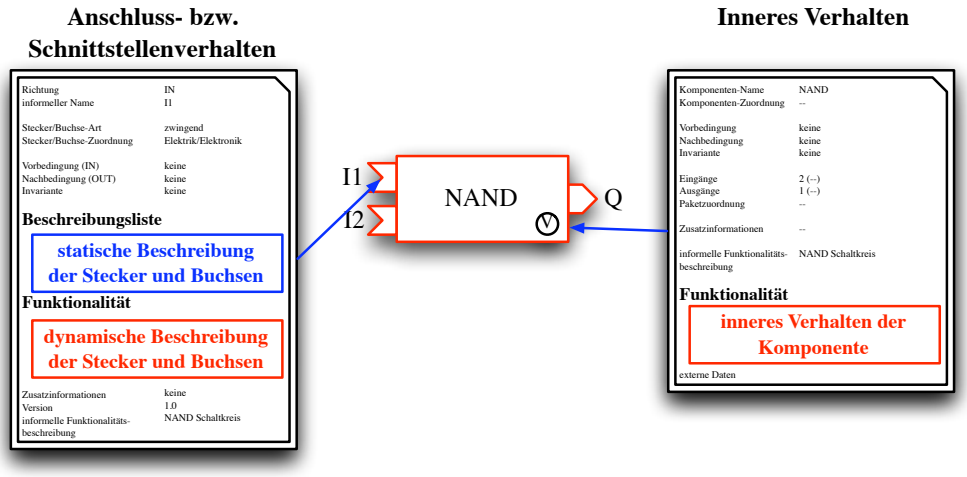


Abbildung 3.94.: (U)CML-Komponente mit Anschlussverhalten und einem innerem Verhalten.

Definition 3.12 ***Verhalten einer (U)CML-Komponente***
Das Verhalten einer (U)CML-Komponente wird festgelegt durch das ***statische*** *und optionale* ***dynamische*** *Verhalten der Anschlüsse der Komponente sowie dem optionalen* ***inneren Verhalten*** *der Komponente. Dabei gilt: Sowohl das dynamische Anschlussverhalten, als auch das innere Verhalten einer Komponente sind optional und müssen nicht für jede Komponente festgelegt werden.*

Nach Definition 3.12 besitzt jede (U)CML-konforme Komponente mindestens ein modelliertes statisches Anschlussverhalten. Zusätzlich zum statischen Anschlussverhalten kann ein dynamisches Verhalten der Anschlüsse (Stecker und Buchsen) hinterlegt werden. Sowohl das statische wie auch das dynamische Anschlussverhalten wird dabei durch das innere Verhalten der Komponente festgelegt. Das innere Verhalten muss jedoch nicht explizit modelliert werden, da für die Kompatibilitätsprüfung das Verhalten der Anschlüsse, nicht jedoch das innere Verhalten einer Komponente wichtig ist. Aus diesem Grund wird das innere Verhalten einer Komponente meistens auf das Verhalten der Anschlüsse projiziert und nicht weiter in (U)CML modelliert (siehe auch Seite 171: Dynamisches Anschlussverhalten einer Komponente).

Das Verhalten eines in (U)CML modellierten Systems setzt sich aus dem Verhalten aller Komponenten des Systems zusammen. In der Definition 3.13 ist das Systemverhalten eines in (U)CML modellierten Systems definiert.

Definition 3.13 ***Verhalten eines in (U)CML modellierten Systems***
Sei $i, k \in \mathbb{N}$ mit $0 < i < \infty$ und i gleich der Anzahl aller Komponenten des Systems S. Des Weiteren sei V_S das Verhalten des Gesamtsystems und V_{K_i} das Verhalten der i-ten Komponente des Systems, dann gilt: Das Verhalten des Gesamtsystem $V_S := \sum_{k=1}^{i} V_{K_i}$.

Vereinfacht bedeutet die Definition 3.13: Das Verhalten eines Systems setzt sich aus der Summe des Verhaltens seiner Komponenten zusammen.

In Definition 3.12 wurde das statische und dynamische Anschlussverhalten wie auch das innere Verhalten einer Komponente eingeführt. Zusätzlich wurde der Zusammenhang zwischen dem inneren Verhalten und dem Verhalten der Komponentenanschlüsse kurz angerissen. In der nachfolgenden Aufzählung werden die drei Verhaltensvarianten einer (U)CML-Komponente noch einmal aufgegriffen und weiter aufgeschlüsselt:

- **Statisches und dynamisches Anschlussverhalten einer (U)CML-Komponente**
  Das Anschlussverhalten einer (U)CML-Komponente, also das Verhalten der Stecker und Buchsen der (U)CML-Komponente, lässt sich weiter in das statische und dynamische Verhalten der Anschlüsse herunterbrechen.

Definition 3.14 ***Anschlussverhalten einer (U)CML-Komponente***
Jede (U)CML-Komponente hat sowohl ein ***statisches****, als auch ein optionales* ***dynamisches*** *Anschlussverhalten. Das statische Verhalten einer Komponente wird mit Hilfe der Beschreibungsfelder beschrieben, während das dynamische Verhalten der Komponente mittels Message Sequence Charts (MSCs) formal spezifiziert wird.*

Die Abbildung 3.95 zeigt eine Verfeinerung der Abbildung 3.94. In der Bildmitte ist die elektrische/elektronische Komponente *NAND* mit zwei Eingangsbuchsen *I1* und *I2* und einem Ausgangsstecker *Q* dargestellt. Die Stecker und Buchsen der Komponente bilden zusammen den Komponentenanschluss (bzw. die Komponentenschnittstelle). Das statische Verhalten des Komponentenanschlusses *NAND* wird mit Hilfe der einzelnen Stecker- und Buchsenbeschreibungsfeldern der Komponente festgelegt (links), während das dynamische Verhalten der Anschlüsse mit Hilfe von MSCs beschrieben wird (rechts). Die MSCs sind ebenfalls in den Beschreibungsfeldern der Stecker und Buchsen hinterlegt.

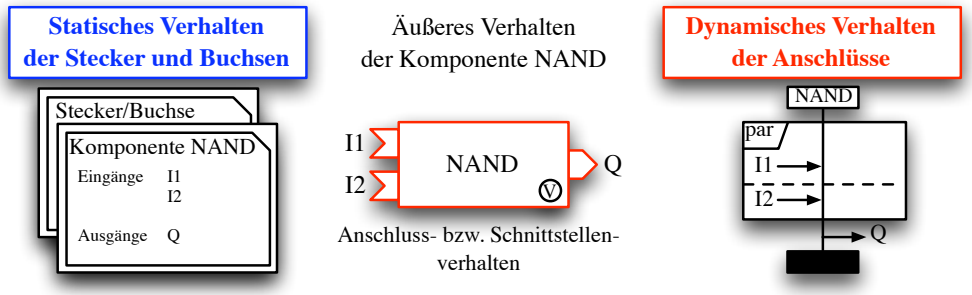


Abbildung 3.95.: (U)CML-Komponente mit statischem und dynamischem Anschlussverhalten.

– *Statisches Anschlussverhalten einer Komponente*
Das statische Anschlussverhalten einer Komponente wird durch die unveränderlichen (konstanten) Eigenschaften der *Stecker und Buchsen* der Komponente festgelegt. Beispielsweise hat die Buchse *I1* aus der Abbildung 3.95 die konstante Eigenschaft, Strom von $0 \ldots 5V$ verarbeiten zu können, wobei der Wertebereich von $0 \ldots 2$ als logische 0 und $2 \ldots 5$ als logische 1 interpretiert wird. Dabei fungieren die Stecker und Buchsen der Komponente als statische *Interaktionsschnittstelle* der Komponente mit ihrer „Umwelt" – ähnlich wie die Paketschnittstellen eines Pakets. Nur über die ausgezeichneten Stecker und Buchsen der Komponente kann diese mit ihrer Umwelt interagieren. Eine unabsichtliche, nicht explizit modellierte Beeinflussung der Komponente kann so wirkungsvoll verhindert werden. Die Stecker und Buchsen einer Komponente sind eine zwingende Voraussetzung für die Modellierung des dynamische Anschlussverhaltens, weil die Stecker und Buchsen der Komponente den Datenaustausch (Verbindung) mit der Komponentenumwelt erst ermöglichen.

– *Dynamisches Anschlussverhalten einer Komponente*
Mit Hilfe des dynamischen Anschlussverhaltens wird das Verhalten der Komponente nach außen festgelegt. Dies betrifft zum Beispiel wann die Komponente an einem bestimmten Anschluss ein Signal sendet. Das Anschlussverhalten wird mit Hilfe der, in der Informatik weit verbreiteten, Message Sequence Charts (MSC) modelliert[205]. Das innere Verhalten der Komponente spielt dabei eine untergeordnete Rolle – es wird auf das Verhalten der Anschlüsse projiziert.

Definition 3.15 ***Dynamisches Anschlussverhalten einer (U)CML-Komponente***
Das dynamische Anschlussverhalten einer (U)CML-Komponente setzt sich aus dem dynamischen Verhalten der einzelnen Stecker und Buchsen der Komponente zusammen.

Die Abbildung 3.96 zeigt die oben eingeführte elektrische/elektronische Komponente *NAND* mit den beiden Eingängen *I1* und *I2* sowie dem Ausgang *Q* (links). Jeder Ein- und Ausgang der Komponente verfügt über ein eigenes dynamisches Verhalten. In der Mitte der Abbildung ist das zusammengesetzte MSC dargestellt, mit dessen Hilfe das dynamische Anschlussverhalten der Komponente *NAND* beschrieben wird. Auf der rechten Seite der Abbildung ist das vereinfachte MSC der Komponente *NAND* abgebildet.

In einem (U)CML-Modell eines Systems wird auf die explizite Modellierung der einzelnen Verhalten der Stecker und Buchsen verzichtet und nur das Gesamtverhalten der Komponente modelliert und dargestellt. Dies ist vor allem bei komplexen Systemen hilfreich, da ansonsten sehr umfangreiche MSC Diagramme entstehen können.

Beispiel 49: Komponente mit statischem und dynamischem Anschlussverhalten
In der Abbildung 3.97 auf der nächsten Seite ist das Softwaresystem *SysP*, das aus zwei Komponenten *K1* und *K2* besteht, den entsprechenden Verbindungen sowie den Beschreibungsfeldern für statisches und dynamisches Anschlussverhalten der Komponenten *K1* und *K2* auszugsweise dargestellt.

[205] Eine ausführliche Beschreibung des Anschluss-Verhaltens einer (U)CML-Komponente finden sie in [Koß09] und [Eck07].

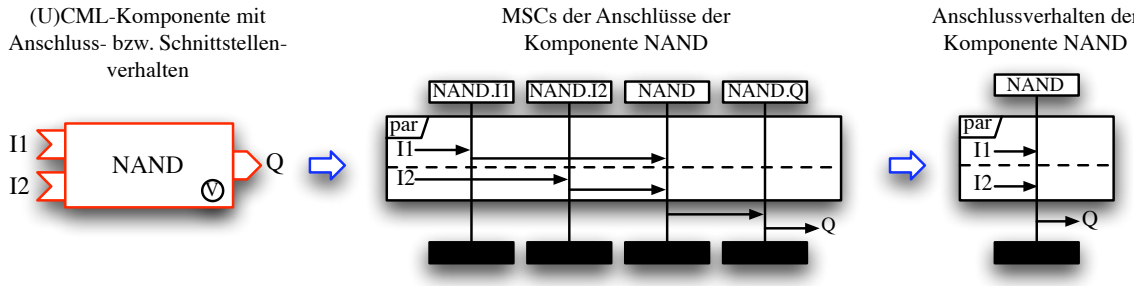


Abbildung 3.96.: (U)CML-Komponente mit dynamischem Verhalten der Anschlüsse.

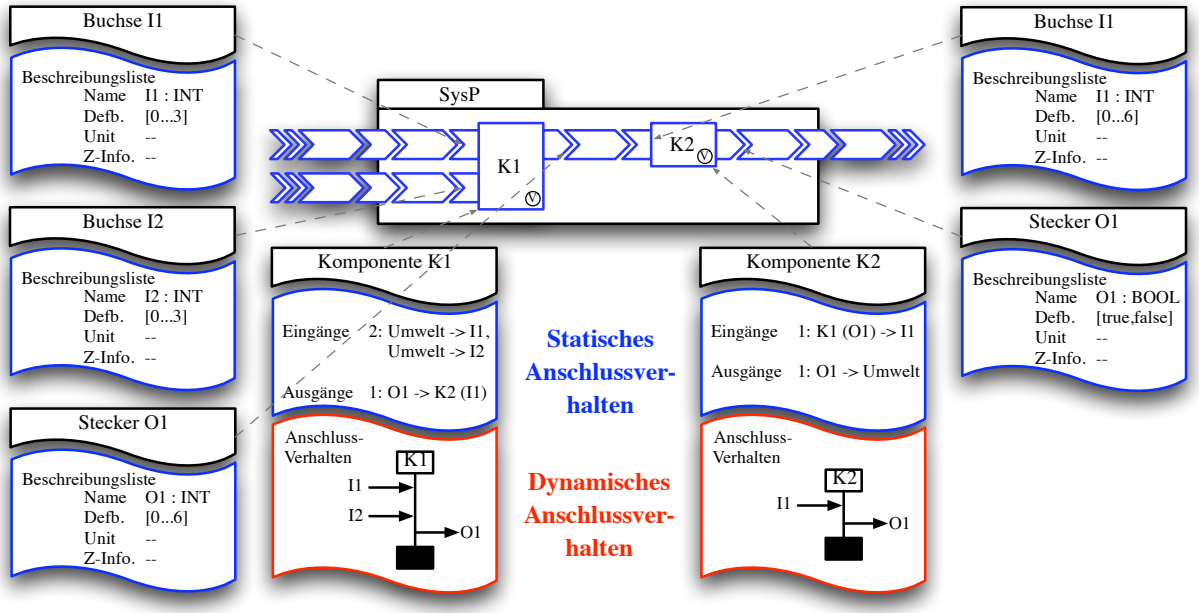


Abbildung 3.97.: (U)CML-System, bestehend aus zwei Komponenten mit statischem und dynamischem Anschlussverhalten.

Für die beiden Komponenten *K1* und *K2* sowie ihre Anschlüsse (Stecker und Buchsen) sind die wichtigsten Bestandteile der entsprechenden Beschreibungsfelder stark vereinfacht dargestellt. Die blau umrandeten Teilbereiche sind für die Beschreibung der statischen, die rot markierten für die Beschreibung des dynamischen Verhaltens der Komponentenanschlüsse zuständig.

Mit Hilfe der Beschreibungsfelder kann nun festgestellt werden ob das modellierte System sowohl statisch als auch dynamisch kompatibel ist[206]. So ist z.B. der Ausgangsstecker *O1* der Komponente *K1* mit der Eingangsbuchse *I1* der Komponente *K2* durch einen Flusspfeil verbunden. Durch den Vergleich der entsprechenden beiden Beschreibungsfelder kann nun überprüft werden, ob die Verbindung kompatibel ist. Im modellierten Beispiel ist die Verbindung vollständig und richtig, weil sowohl der (statische) Name, Typ und Definitionsbereich als auch die beiden (dynamischen) MSCs übereinstimmen. □

- **Inneres Verhalten einer (U)CML-Komponente**
  In der praktischen Anwendung der (U)CML hat sich gezeigt, dass in vielen Fachdisziplinen, wie z.B. der Raumfahrt, der Elektrotechnik oder der Informatik, nicht nur die statische bzw. dynamische Kompatibilität der Komponentenanschlüsse, sondern das gesamte dynamische Verhalten einer Komponente eine wichtige Rolle bei der Bewertung des Systems spielt. Um diesem erweiterten Einsatzgebiet Rechnung zu tragen, ist es in (U)CML möglich, neben dem statischen bzw. dynamischen Verhalten der Komponentenanschlüsse auch das interne Verhalten zu modellieren.

  Das interne Verhalten einer Komponente beschreibt die Funktionalität bzw. das Verhalten der gesamten Komponente. Dabei besteht eine sehr enge Kopplung zwischen dem internen Verhalten und dem Anschlussverhalten der Komponente. Für die Bestimmung der Kompatibilität ist das interne Verhalten einer Komponente nicht wichtig, da sich das interne Verhalten einer Komponente an den Anschlüssen nach außen widerspiegelt. In der Abbildung 3.98 auf der nächsten Seite ist der Zusammenhang zwischen dem internen Verhalten einer *NAND*-Komponente und dem Verhalten an den Steckern und Buchsen der Komponente exemplarisch dargestellt. Für die Modellierung des inneren Verhaltens einer Komponente können sowohl MSCs, Quellcode, mathematische Funktionen als auch Zustandsautomaten verwendet werden.

[206] Siehe hierzu: Kapitel „3.6.5 *(U)CML-Sprach- und Kompatibilitätsregelwerk*“ ab Seite 236.

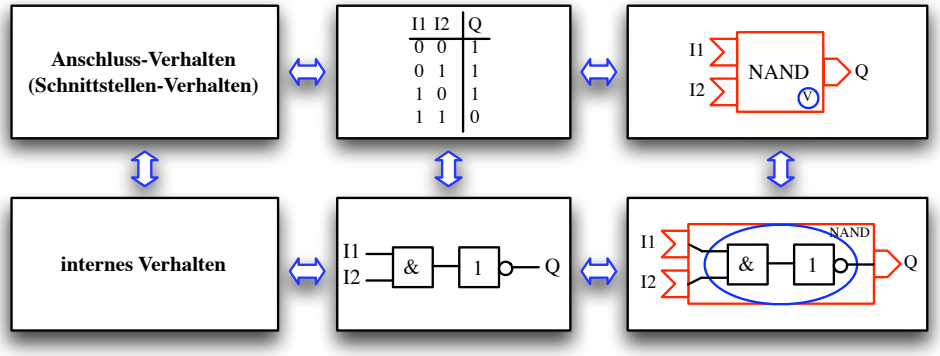


Abbildung 3.98.: Zusammenhang zwischen internem Verhalten und Anschlussverhalten einer (U)CML-Komponente.

Das innere Verhalten der Komponente *NAND* wird durch die beiden Eingangsbuchsen *I1* und *I2* sowie den Ausgangsstecker *Q* nach außen weitergereicht. Das innere Verhalten legt demnach das Verhalten der statischen und dynamischen Anschlüsse der Komponente fest.

Definition 3.16 ***Inneres Verhalten einer (U)CML-Komponente***
Jede (U)CML-Komponente kann ein optionales inneres Verhalten haben. Das innere Verhalten der Komponente ist dabei sehr eng mit dem Verhalten der Anschlüsse der Komponente verbunden. Das innere Verhalten einer Komponente kann z.B. mittels mathematischer Funktionen, Quellcode, MSCs, Zustands-Diagrammen und Prosa beschrieben werden.

Beispiel 50: Anwendung einer (U)CML-Komponente mit innerem Verhalten
In der nachfolgenden Abbildung 3.99 ist ein einfaches System „*SysP*“ bestehend aus zwei Softwarekomponenten *K1* und *K2* modelliert, die miteinander durch einen Flusspfeil verbunden sind. Genauer: Der Ausgang der Komponente *K1* ist mit einem Flusspfeil mit dem Eingang der Komponente *K2* verbunden. Die Komponente *K1* hat zwei Eingänge *IN1* und *IN2*, die von der Umwelt kommen und über die sie ihre Eingangswerte übergeben bekommt. Die Komponente *K2*, deren innere Struktur unbekannt ist, reicht ihren Ausgangswert an die Umwelt weiter. In der nachfolgenden Tabelle ist der oben verbal beschriebene Sachverhalt mit den jeweils gültigen Wertebereichen jedes Ein- bzw. Ausgangs noch einmal kompakt aufgelistet.

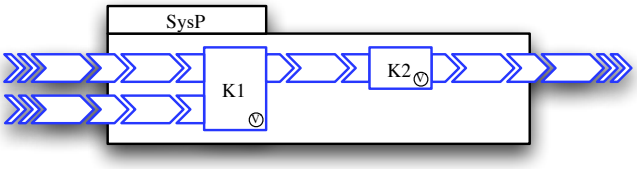


Abbildung 3.99.: (U)CML-Komponente mit innerem Verhalten.

Statisches Verhalten der Komponenten *K1* und *K2*:

K1		**K2**	
Eingang/Ausgang	Wertebereich	Eingang/Ausgang	Wertebereich
IN1	0-3		
IN2	0-3		
OUT	0-6	IN	0-6
		OUT	boolean

Die Softwarekomponente *K1* hat ein explizit modelliertes inneres Verhalten. Sie berechnet aus den beiden Eingangswerten *IN1* und *IN2* den Ausgangswert *OUT*.

Im unten abgedruckten C++ Listing ist das innere Verhalten der Komponente *K1* vereinfacht dargestellt.

Internes Verhalten der Komponente *K1*:

```
1  int Add(int IN1, int IN2)
2  {
3    return IN1 + IN2;
4  }
```

In Abhängigkeit von den beiden Eingangswerten *IN1* und *IN2* berechnet die Komponente *K1* das Ergebnis und gibt dieses an den Ausgang *OUT* weiter. Dadurch ist eine Simulation des Systems *SysP* möglich. Ohne das explizit modellierte innere Verhalten der Komponente *K1* wäre die Simulation des Systems nicht möglich. □

Anwendungsgebiete für die Modellierung des inneren Verhaltens einer (U)CML-Komponente
Die nachfolgende Aufzählung beschreibt exemplarisch drei Domänen, in denen eine Modellierung des inneren Verhaltens einer (U)CML-Komponente von entscheidender Bedeutung ist:

- **Elektrotechnik**
  In der Elektrotechnik ist es für die Modellierung eines Systems essentiell sowohl das innere als auch das Komponentenanschlussverhalten (Baugruppe, Schaltkreis etc.) genau zu kennen, um unerwünschte Fehler oder Seiteneffekte zu vermeiden. Eines dieser Probleme ist beispielsweise das Auftreten von unerwünschten Laufzeitfehlern in digitalen Schaltungen (so genannte Hazards oder Glitches[207]). Solche Laufzeitfehler können auf Modellebene nur durch Simulation (z.B. Auswertung von KV-Diagrammen[208]) des internen Verhaltens einer Komponente bzw. durch reale Tests an der Hardware entdeckt werden.

- **Maschinenbau – Raumfahrttechnik**
  Besonders wichtig ist die Simulation und Verifikation von komplexen technischen Modellen in der Raumfahrttechnik und insbesondere bei der Entwicklung von Satelliten. Mit Hilfe der relativ neuen Simulationstechnik – **M**odel-based **D**evelopment & **V**erification **E**nvironment (MDVE) [DE07][Die06] – ist es möglich, einen Satelliten bereits in einer sehr frühen Entwicklungsphase zu simulieren, noch bevor „reale Hardware" des Satelliten existiert. Anschließend werden, wenn Teile des Systems in realer Hardware vorliegen (z.B. ein Subsystem des Satelliten), dieses Subsystem in das virtuelle Modell „eingesetzt" um die korrekte Funktionsweise des Systems zu verifizieren. Der Prozess des Ersetzens von modellierter Hardware durch reale (Sub-) Systeme wird auch als „Hardware in the Loop (HiL)" [Wik07d][Rot07] bezeichnet.

  In der MDVE-Simulationsumgebung kann die (U)CML zur Überprüfung der modellierten Systeme auf Kompatibilität eingesetzt werden, noch bevor reale Hardware erzeugt wird.

- **Informatik**
  In der Softwareentwicklung ist es heute üblich aus einem Modell des Systems Code zu generieren. Diese Technik wird vor allem bei im **M**odel **D**riven **A**rchitecure (MDA) Ansatz benutzt[209]. So kann z.B. aus dem UML/UML2-Modell eines Softwaresystems automatisch das Programmgerüst generiert werden. Dazu wird in den Klassen neben der Methodendeklaration auch der entsprechende Methodenrumpf eingetragen. Nach dem gleichen Prinzip kann in (U)CML in einer Komponente der Rumpf einer Methode hinterlegt werden. Auf diese Art kann aus einem in (U)CML modellierten System ebenfalls Code generiert werden.

Klassifikation des inneren Verhaltens einer Komponente
Zusätzlich zu den fünf Beschreibungsarten des inneren Verhaltens einer Komponente (vgl. Def.: 3.16) ist es sinnvoll einen „externen Verweis" auf ein externes Simulationswerkzeug in die Liste Verhaltensbeschreibungsmethoden aufzunehmen. Die fünf bzw. sechs Beschreibungsmethoden lassen sich in zwei unterschiedliche Klassen einteilen. Zum einen gibt es die Beschreibungsarten, mit deren Hilfe es möglich ist, das Verhalten einer Komponente formal so genau zu beschreiben, dass sie simuliert werden kann. Zu dieser Klasse zählen: die mathematischen Funktionen, der Quellcode, MSCs sowie die Zustandsautomaten. Zum anderen bleibt die Klasse der nicht simulierbaren Beschreibungsmethoden – „Prosa". Die „externen Verweise" auf ein Simulationswerkzeuge nehmen dabei in der Aufzählung eine Sonderstellung ein. Mit ihrer Hilfe kann das Verhalten einer Komponente nur so genau beschrieben werden, wie es das externe Beschreibungswerkzeug zulässt. Aus diesem Grund kann a priori keine Aussage über die „Güte" der Simulation bzw. der Simulationsergebnisse gemacht werden.

In der Abbildung 3.100 auf der nächsten Seite (oben) ist die (U)CML-Komponente *K* mit innerem Verhalten abgebildet. Die Komponente *K* hat drei Eingänge *A, B, C* und einen Ausgang *D*, wobei der Eingang *C* optional[210] ist. In der Bildmitte ist eine Tabelle dargestellt, in der die oben eingeführten fünf Beschreibungsmethoden für das innere Verhalten einer Komponente als Spaltenindex dienen. In der ersten Zeile der Tabelle ist das Verhalten der Komponente *K* ohne den optionalen Eingang *C* eingetragen, während in der zweiten Zeile das Verhalten mit dem optionalen Eingang eingetragen ist.

In den beiden Zeilen der Tabelle ist stets die gleiche Funktionalität (das gleiche Verhalten) in den unterschiedlichen Beschreibungsmethoden, die die (U)CML zur Verfügung stellt modelliert.

Anmerkung:
Wie das einfache Beispiel zeigt, ist durch die Berücksichtigung des inneren Verhaltens einer Komponente des Systems, die Simulation möglich. So kann z.B. der „Fluss" von Daten oder Material innerhalb des Systems modelliert und somit beispielsweise die Ausbreitung eines Signals beschrieben werden. In (v)Sys-ed wird ein ähnliches Konzept für die Berechnung von Werten benutzt, um z.B. die Gesamtmasse eines Systems zu ermitteln.

[207] Nähere Informationen zu Hazards bzw. Glitches finden Sie unter [Wik07c][PMA99][Kni].
[208] Das Akronym *KV* steht für „Karnaugh-Veitch". Nähere Informationen zu KV-Diagrammen entnehmen Sie bitte [Wik07i] bzw. [JR85, 264ff].
[209] Weitere Ansätze, aus einem formal spezifizierten Modell Code zu generieren, werden unter dem Sammelbegriff „Modellgetriebene Softwareentwicklung" (englisch Model-Driven Software Development, MDSD) zusammengefasst.
[210] Siehe hierzu: Kapitel „3.6.3 *Das Optionalitätsprinzip der (U)CML*" ab Seite 218.

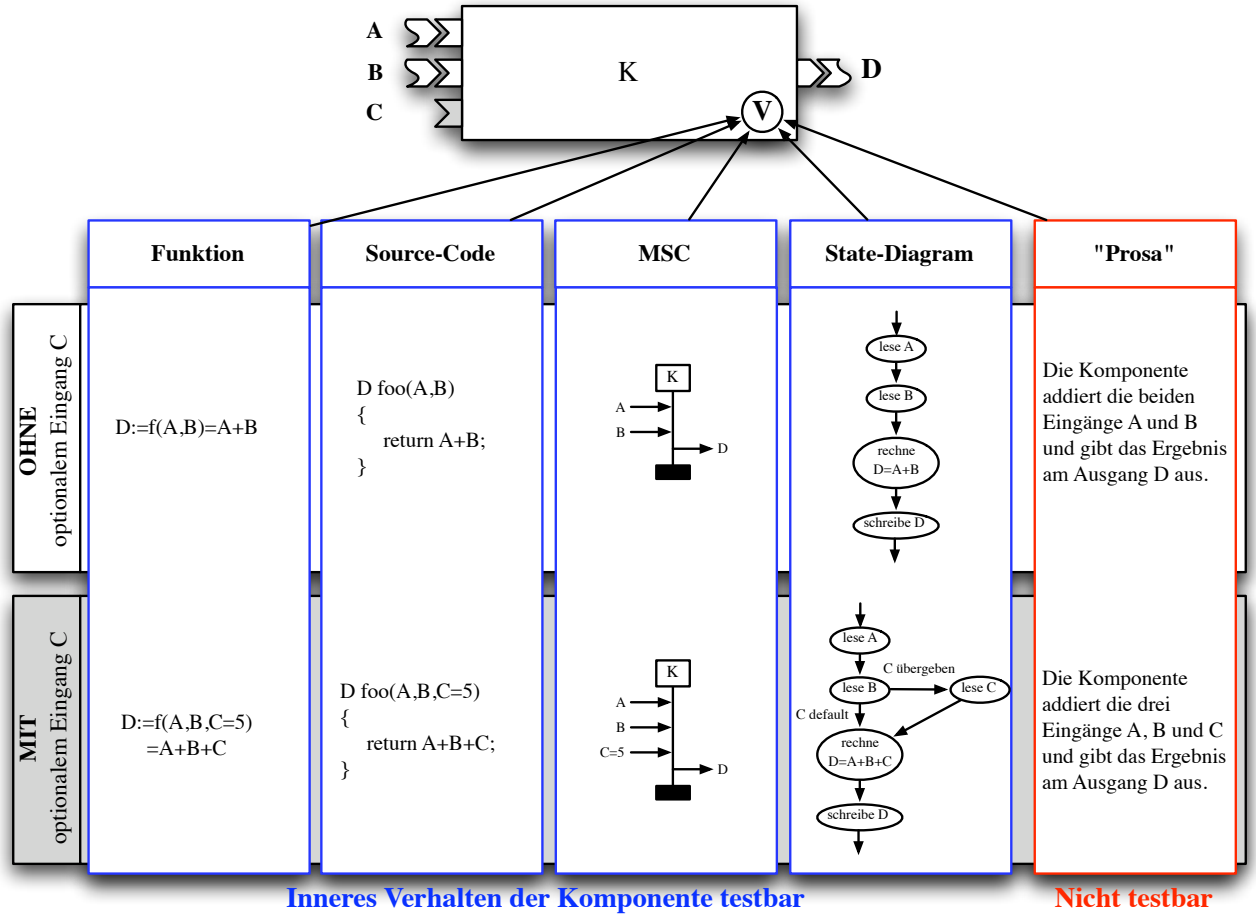


Abbildung 3.100.: (U)CML-Komponente mit innerem Verhalten.

3.6.2.3.3. Zusammenfassung der Eigenschaften einer (U)CML-Komponente

Nachdem die Einführung der (U)CML-Komponente abgeschlossen ist, folgt noch einmal eine Zusammenfassung der wichtigsten Eigenschaften einer Komponente:

- Komponenten sind atomare Einheiten.
- Komponenten können ein inneres Verhalten und ein Anschlussverhalten haben.
- Komponenten können ein inneres Verhalten haben, dieses wird zur Kompatibilitätsbestimmung jedoch nicht herangezogen.
- Jede Komponente muss sich innerhalb eines Pakets befinden.
- Jede Komponente hat einen eineindeutigen Namen, mit dessen Hilfe sie im System identifiziert werden kann.
- Jede Komponente muss mindestens einen Eingang (Buchse) und mindestens einen Ausgang (Stecker) haben .
- Komponenten können mit einem Farbschema versehen werden.
- Komponenten können ein (Funktions-) Symbol enthalten.

3.6.2.4. (U)CML-Paketschnittstellen und Komponentenanschlüsse

Nachdem in den letzten beiden Abschnitten „(U)CML-Pakete“ sowie „(U)CML-Komponente“ des Kapitels „Einführung in die Kompatibilitätsmodellierungssprache (U)CML“ die Grundelemente der (U)CML – die Pakete und Komponenten – eingeführt und erläutert wurden, folgt nun die Beschreibung der unterschiedlichen Anschluss- und Schnittstellenarten, die die (U)CML für die Modellierung eines Systems im Flussdiagramm zur Verfügung stellt.

In der nachfolgenden Tabelle 3.8 auf Seite 177 sind sämtliche (U)CML-Anschluss- und Schnittstellenarten nach ihrem Verwendungszweck geordnet aufgelistet. Alle Anschluss- und Schnittstellenarten der (U)CML lassen sich dabei in drei Hauptkategorien mit jeweils zwei Untergruppen einteilen. Die Hauptkategorien sind: *Standardschnittstellen und -anschlüsse, Kommunikationsschnittstellen und -anschlüsse* sowie *BUS-Systemschnittstellen*[211]. Jede der drei Hauptkategorien hat zwei Untergruppen: *Standard* und *zusammengefasst*. Die Kategorie „BUS-System“ nimmt dabei eine Sonderstellung innerhalb der (U)CML und auch in der Tabelle ein. Zum einen haben BUS-Systeme keine Unterteilung in Untergruppen wie alle anderen Schnittstellenarten und zum anderen werden sie hauptsächlich für die Modellierung von elektrischen Signalen verwendet. Sämtliche (U)CML-BUS-Systeme lassen sich in zwei disjunkte Untergruppen aufteilen: die unidirektionalen und die bidirektionalen BUS-Systeme.

[211] Siehe hierzu: Kapitel „3.6.2.7 *(U)CML-BUS-System*“ ab Seite 211.

Die in der Tabelle 3.8 aufgelisteten Paketschnittstellen sowie die Komponentenanschlüsse lassen sich formal beschreiben. Die formale Notation der unterschiedlichen Schnittstellen- und Anschlussarten, die die (U)CML zur Modellierung im Flussdiagramm bereitstellt, sind in der nachfolgenden Definition zusammengefasst.

Definition 3.17 ***Notation der Paketschnittstellen und Komponentenanschlüsse***
Sei ***Paket_Name*** *der eineindeutige Name eines Pakets, sowie* ***Komponenten_Name*** *der eineindeutige Name einer Komponente. Dann lassen sich* ***sämtliche Schnittstellenarten*** *der (U)CML schreiben als:*

$$Paket\_Name_{Art:Schnittstellenart:Name:(Cond)}$$

sowie sämtliche ***Komponentenanschlussarten*** *(Stecker und Buchsen):*

$$Komponenten\_Name_{Art:Anschlussart:Name:(Cond)}$$

mit

Art	*Z (zwingend), OPT (optional)*
Schnittstellenart	*Standardschnittstelle (SS_{Std})*
	Kommunikationsschnittstelle (SS_{Kom})
	Zusammengefasste Standard- bzw. Kommunikationsschnittstelle ($SS_{Std.Zus}, SS_{Kom.Zus}$)
Anschlussart	*Standardstecker/-Buchse (St_{Std}, Bu_{Std})*
	Kommunikationsstecker/-Buchse (St_{Kom}, Bu_{Kom})
	Zusammengefasste Standard- bzw. Kommunikationsstecker/-Buchsen ($St_{Std.Zus}, Bu_{Std.Zus}, St_{Kom.Zus}, Bu_{Kom.Zus}$)
Name	*informeller Name*
Cond	*Vorbedingung, Nachbedingung, Invariante*

In den einzelnen Unterabschnitten dieses Kapitels wird die in der Definition 3.17 eingeführte Notation noch einmal aufgegriffen und anhand von einfachen Beispielen erläutert. Die beiden Bezeichner „Art“ und „Cond“ werden in den beiden Kapiteln „3.6.3 *Das Optionalitätsprinzip der (U)CML*“ ab Seite 218 und „3.6.5 *(U)CML-Sprach- und Kompatibilitätsregelwerk*“ ab Seite 236 genauer erläutert. Für die einfache Beschreibung der Paketschnittstellen sowie der Komponentenanschlüsse sind sie an dieser Stelle noch nicht relevant. In der Tabelle 3.8, Spalte „Abkürzung“ sind die Standardbezeichner für sämtliche Paketschnittstellen, Komponentenanschlüsse sowie das BUS-System, basierend auf der Notationsdefinition 3.17, eingetragen. Außerdem gilt für alle Paketschnittstellen bzw. Komponentenanschlüsse der (U)CML: Der informelle Name eines Eingangs, Ausgangs oder einer Paketschnittstelle ist stets eineindeutig innerhalb seines Namensraums, also z.B. der Komponente oder des Pakets.

Bevor mit der Erläuterung der einzelnen Anschluss- und Schnittstellenarten, die die (U)CML für die Modellierung von Systemen in der Flussansicht zur Verfügung stellt begonnen wird, folgt hier noch einmal eine kurze Wiederholung der wichtigsten Eigenschaften der (U)CML-Pakete und Komponenten, an denen die unterschiedlichen Anschlüsse und Schnittstellen angebracht werden können.

- Jedes regelkonforme (U)CML-Paket bzw. jede regelkonforme (U)CML-Komponente[212] besitzt mindestens einen explizit modellierten *Eingang* und mindestens einen *Ausgang*. Bei Paketen sind dies Schnittstellen, während an Komponenten Anschlüsse, also Stecker und Buchsen, angeschlossen werden. Es ist in (U)CML nicht erlaubt, eine Paketschnittstelle an einer Komponenten anzubringen. Dies gilt auch für den umgekehrten Fall.
- Über die ausgezeichneten Ein- und Ausgänge stellt ein Paket bzw. eine Komponente eine Verbindung mit anderen Komponenten bzw. der Umwelt her. Für alle Pakete und Komponenten gilt: Ein Datenaustausch ist ausschließlich über die explizit modellierten Stecker, Buchsen oder Paketschnittstellen möglich.
- Eingänge werden stets links, Ausgänge stets rechts an einem Paket oder einer Komponente angebracht. Ausnahmen bilden die Kommunikationsschnittstellen eines Pakets sowie die Kommunikationsanschlüsse einer Komponente und sämtliche BUS-System Anschlüsse.

Nachdem nun die wichtigsten Eigenschaften der Pakete und Komponenten wiederholt wurden, folgt die ausführliche Beschreibung der einzelnen Schnittstellenarten der (U)CML.

3.6.2.4.1. Standardpaketschnittstellen und Standardkomponentenanschlüsse

Die (U)CML unterscheidet grundsätzlich zwischen zwei unterschiedlichen Standardanschlussarten: Zum einen die so genannten *Standardpaketschnittstellen*, mit deren Hilfe ein Paket eine Verbindung von außerhalb des Pakets, der Paketumwelt, mit dem Inhalt (untergeordneten Paketen und Komponenten) des Pakets ermöglicht. Und zum anderen die *Standardkomponentenanschlüsse – Eingangsbuchsen* und *Ausgangsstecker* – mit deren Hilfe eine Komponente mit anderen Komponenten des Systems Daten austauschen kann.

Standardpaketschnittstelle

Ein Paket stellt, wie im Kapitel „3.6.2.2 *(U)CML-Pakete*“ ab Seite 162 beschrieben, einen Container ohne eigene Funktionalität dar. Um eine Verbindung (Datenaustausch) zwischen dem Inhalt des Pakets und der Paketumwelt zu ermöglichen, muss an jedem (U)CML-Paket mindestens eine Standardpaketeingangsschnittstelle und eine Standardpaketausgangsschnittstelle vorhanden sein. Die nachfolgende Definition beschreibt, wie eine Paketschnittstelle in (U)CML formal geschrieben werden kann. Daran anschließend folgt die Beschreibung der graphischen Notation einer Paketschnittstelle im Flussdiagramm.

[212] Siehe hierzu: Kapitel „3.6.5 *(U)CML-Sprach- und Kompatibilitätsregelwerk*“ ab Seite 236. bzw. vgl. „Definition: C.9“ auf Seite 329.

	Schnittstellenart	Abkürzung	Paket	Komponente
Standard-schnittstellen und -anschlüsse	(Ausgangs-) Stecker	St_{Std}		
	(Eingangs-) Buchse	Bu_{Std}		
	Paketschnittstelle	SS_{Std}		
	Zusammengefasste Standardschnittstellen und -anschlüsse			
	Stecker	$St_{Std.Zus}$		
	Buchse	$Bu_{Std.Zus}$		
	Paketschnittstelle	$SS_{Std.Zus}$		
Kommunikations-schnittstellen und -anschlüsse	Stecker/Buchse	St_{Kom}; Bu_{Kom}		
	Schnittstelle	SS_{Kom}		
	Zusammengefasste Kommunikationsschnittstellen und -anschlüsse			
	Stecker/Buchse	$St_{Kom.Zus}$; $Bu_{Kom.Zus}$		
	Schnittstelle	$SS_{Kom.Zus}$		
BUS-System-Schnittstellen und -anschlüsse	*Unidirektional*			
	Stecker	St_{BUS}		
	Buchse	Bu_{BUS}		
	Paketschnittstelle	$SS_{BUS:Uni}$		
	Bidirektional			
	Anschluss	$Ansch_{BUS}$		
	Paketschnittstelle	$SS_{BUS:Bi}$		

Tabelle 3.8.: Paketschnittstellen und Komponentenanschlussarten der (U)CML.

Definition 3.18 ***Standardpaketschnittstellen***
Sei o.B.d.A. ***A*** *ein (U)CML-Paket. Dann kann eine* ***Standardpaketschnittstelle*** *formal als*

$$A_{Art:SS_{Std}:Name:(Cond)}$$

geschrieben werden.

Die in der Definition 3.18 eingeführte formale Notation einer Standardpaketschnittstelle basiert auf der allgemeinen Definition 3.17.

Definition 3.19 ***Standardpaketschnittstellen (graphische Darstellung)***
Standardpaketschnittstellen sind stets an (U)CML-Paketen angebracht. Sie dienen als dezidierte Schnittstelle zwischen der Paketumwelt und dem Inhalt des Pakets. Standardpaketschnittstellen werden in (U)CML sowohl in der Glass-Box, als auch in der Black-Box Darstellung als kleine Pfeile dargestellt, die über den Rand eines Pakets hinweg gezeichnet werden. Dabei wird zwischen ***Standardpaketeingangs-*** *und* ***-ausgangsschnittstellen*** *unterschieden. Standardpaketeingangsschnittstellen werden stets links, Standardpaketausgangsschnittstellen stets rechts an einem Paket angebracht.*

Anmerkung:
Die Wortvorsatz „Standard“ kann weggelassen werden, wenn eindeutig klar ist, das es sich um eine Standardschnittstelle handelt.

Beispiel 51: Standardpaketschnittstellen in Glass-Box sowie Black-Box Darstellung
Die Abbildung 3.101 zeigt auf der linken Seite ein Paket in Black-Box und auf der rechten Seite der Abbildung, das selbe Paket in Glass-Box Darstellung.

In der Black-Box Darstellung (links) wurde das Paket *P1* mit fünf Standardpaketschnittstellen (*SS1* . . . *SS5*) abgebildet. In der Mitte des Pakets *P1* sind drei Punkte eingetragen, als Zeichen, dass dieses Paket sich in Black-Box Darstellung befindet (also der Inhalt des Pakets nicht dargestellt ist). Rechts ist das selbe Paket *P1* in Glass-Box Darstellung mit ebenfalls fünf Standardpaketschnittstellen abgebildet. In dieser Darstellungsart ist der Inhalt des Pakets vollständig sichtbar. Nach Definition 3.8 sind beide dargestellten Pakete identisch. Sie unterscheiden sich lediglich durch die Art ihrer Darstellung, nicht jedoch in ihrem Inhalt.

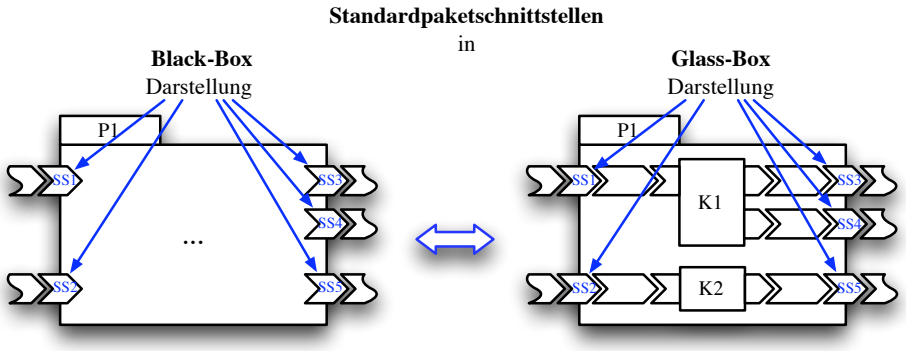


Abbildung 3.101.: Paketschnittstellen in Black-Box sowie Glass-Box Darstellung.

Nach Definition 3.18 lassen sich sämtliche Standardpaketschnittstellen zusätzlich zur graphischen Repräsentation im Flussdiagramm auch formal beschreiben. Zum Beispiel kann die Schnittstelle in der Abbildung 3.101 links oben (*SS1*), mit Hilfe der formalen Darstellung

$$P1_{Z:SS_{Std}:SS1}$$

geschrieben werden. Dabei bedeutet das „Z“, dass es sich um eine zwingende Paketschnittstelle mit dem informellen Namen „SS1“ handelt. Die genaue Bedeutung des Terminus *zwingend* wird im Kapitel „3.6.3 *Das Optionalitätsprinzip der (U)CML*“ ab Seite 218 beschreiben. □

Standardkomponentenanschlüsse (Standardkomponentenschnittstellen)
Die Eingangsbuchsen und Ausgangsstecker[213] einer Komponente bilden zusammen die Komponentenanschlüsse, auch Komponentenschnittstelle genannt, über die eine Komponente Daten mit ihrer Umwelt austauschen kann.

Definition 3.20 ***Standardkomponentenanschlüsse (Stecker und Buchsen)***
Sei o.B.d.A. ***B*** *eine (U)CML-Komponente. Dann kann ein* ***Standardkomponentenanschluss****, also die Stecker und Buchsen, formal*

Standardstecker	$B_{Art:St_{Std}:Name:(Cond)}$
Standardbuchse	$B_{Art:Bu_{Std}:Name:(Cond)}$

geschrieben werden.

Definition 3.21 ***Standardkomponentenanschlüsse (Stecker und Buchsen) (graphische Darstellung)***
Die Ausgangsstecker (Stecker) ▷ *und Eingangsbuchsen (Buchsen)* ▷| *bilden die Standardkomponentenanschlüsse, an denen Standardflusspfeile andocken können. Dabei werden Stecker stets rechts, Buchsen stets links an einer Komponente angebracht.*

Beispiel 52: Standardkomponentenanschlüsse (Stecker und Buchsen)
Die Abbildung 3.102 zeigt eine (U)CML-Standardkomponente mit zwei Standardeingangsbuchsen (*I1* und *I2*) und drei Standardausgangssteckern (*O1* . . . *O3*).

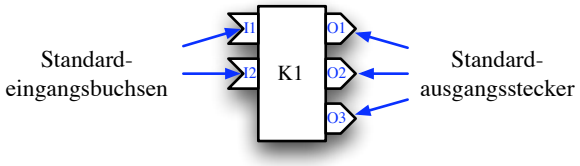


Abbildung 3.102.: (U)CML-Komponente mit Standardsteckern und -buchsen.

Mit Hilfe der zwei (Eingangs-) Buchsen sowie der drei (Ausgangs-) Stecker kann die Komponente *K1* mit anderen Paketen bzw. Komponenten kommunizieren. Genauer: Die Kommunikation findet „über“ die Verbindungspfeile statt. Die Stecker und Buchsen bilden lediglich die „Andockstelle“ für die Verbindungspfeile.

Auch die Stecker und Buchsen einer Komponente lassen sich mit Hilfe der oben eingeführten formalen Notation beschreiben. Der Stecker *O2* der Komponente *K1* lässt sich formal mit

$$K1_{Z:St_{Std}:O1}$$

bezeichnen. □

[213] Anmerkung: Im nachfolgenden Text werden die *Eingangsbuchsen* sowie die *Ausgangsstecker* einer Komponente stets als „Stecker“ bzw. „Buchsen“ bezeichnet.

3.6.2.4.2. Paketkommunikationsschnittstellen und Komponentenkommunikationsanschlüsse

Soll in (U)CML ein *Regelkreis*, ein *Protokoll* oder ein *Funktions-* bzw. *Methodenaufruf* modelliert werden, kann dies sowohl mit den *Standardpaketschnittstellen* bzw. *Standardkomponentenanschlüssen*, als auch mittels einer *Paketkommunikationsschnittstelle* bzw. mit einem *Komponentenkommunikationsanschluss* modelliert werden. In der Abbildung 3.103 ist auf der linken Seite ein Methodenaufruf mit einer getrennten Hin- und Rückverbindung (1,2) und auf der rechten Seite der Abbildung durch eine zusammengesetzte Hin- und Rückverbindung, dem *Komponentenkommunikationsanschluss* (3), modelliert worden.

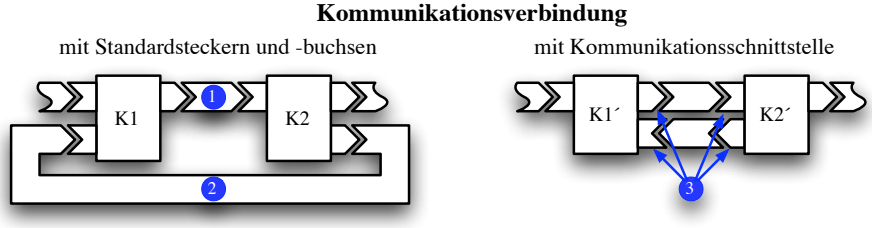


Abbildung 3.103.: Gegenüberstellung: Standardverbindungen kontra Kommunikationsverbindung.

Auf der linken Seite ist die „enge Kopplung“, die bei einem Methodenaufruf zwischen der Komponente *K1* und *K2* vorliegt nicht zu erkennen, so dass bei der Modellierung eine der beiden Verbindungen leicht „vergessen“, oder während der Systemmodellierungsphase wieder gelöscht werden kann, so dass es zu einer unbeabsichtigten Inkompatibilität im Modell kommt. Bei der Modellierung auf der rechten Seite der Abbildung 3.103 hingegen wurde der Methodenaufruf mit Hilfe eines Komponentenkommunikationsanschlusses zwischen den beiden Komponenten $K1'$ und $K2'$ realisiert. Hier kann bereits während der Modellierung erkannt werden, dass es sich um eine enge Kopplung von zwei Signalen handelt. Ein versehentliches Löschen einer der beiden Verbindungsteile (Hin- oder Rückkanalanschluss) kann nicht vorkommen, da die beiden Verbindungen eine atomare Einheit bilden und somit nicht einzeln angelegt oder gelöscht werden können.

In der nachfolgenden Definition 3.22 ist der oben verbal geschilderte Sachverhalt zusammengefasst dargestellt.

Definition 3.22 ***Kommunikationsschnittstellen und Kommunikationsanschlüsse***
Eine ***Paketkommunikationsschnittstelle*** *bzw. ein* ***Komponentenkommunikationsanschluss*** *in (U)CML ist eine atomare Einheit. Sie besteht aus einem Hin- und einem Rückkanal. Außerdem gilt: An einer Paketkommunikationsschnittstelle bzw. an einem Komponentenkommunikationsanschluss kann nur ein* ***Kommunikationspfeil***[214] *angeschlossen werden.*

Paketkommunikationsschnittstelle
Eine Kommunikationsschnittstelle an einem Paket wird in (U)CML mit Hilfe einer in beide Richtungen zeigenden Standardpaketschnittstelle dargestellt. Dabei ist der Abstand zwischen den beiden Schnittstellen kleiner als zwischen zwei „normalen“ Schnittstellen.

Definition 3.23 ***Paketkommunikationsschnittstelle***
Sei o.B.d.A. ***A*** *ein (U)CML-Paket. Dann kann eine* ***Paketkommunikationsschnittstelle*** *formal durch*

$$A_{Art:SS_{Kom}:Name:(Cond)}$$

beschrieben werden.

Definition 3.24 ***Paketkommunikationsschnittstelle (graphische Darstellung)***
Die ***Paketkommunikationsschnittstelle*** *eines Pakets kann sowohl links als auch rechts an einem Paket angebracht werden. Dabei gilt: Die beiden Bestandteile der Schnittstelle (der Hin- und Rückkanal) liegen eng beieinander und bilden eine untrennbare Einheit. Sie werden sowohl in der Glass-Box, als auch in der Black-Box Darstellung als zwei Paketschnittstellen dargestellt, wobei der untere Teil der Schnittstelle um 180° gegenüber dem oberen gedreht dargestellt wird.*

Anmerkung zur Definition 3.22 bzw. Definition 3.24:

- An einer Paketkommunikationsschnittstelle darf nur ein Kommunikationspfeil angeschlossen werden. Es ist nicht gestattet, einen Standardpfeil an einer Paketkommunikationsschnittstelle anzuschließen. Nähere Informationen über die möglichen Verbindungsarten der (U)CML finden Sie unter „3.6.5 *(U)CML-Sprach- und Kompatibilitätsregelwerk*“ ab Seite 236.
- Paketkommunikationsschnittstellen können sowohl rechts als auch links an einem Paket angebracht werden. Dies gilt insbesondere für das ausgezeichnete Systempaket.

[214] Siehe hierzu: Kapitel „3.6.2.5 *(U)CML-Flusspfeilarten*“ ab Seite 184.

Komponentenkommunikationsanschlüsse
Die enge Kommunikationskopplung, z.B. bei einem Methodenaufruf, wird in (U)CML mit Hilfe des *Komponentenkommunikationsanschlusses* modelliert (vgl. Abbildung 3.103 auf der vorherigen Seite (3)). Dabei setzt sich der Komponentenkommunikationsanschluss aus zwei eng aneinander liegenden Steckern und Buchsen zusammen, wobei einer der beiden um 180° gegenüber dem anderen gedreht ist. Außerdem ist es beim Komponentenkommunikationsanschluss unerheblich, ob er links oder rechts an einer Komponenten angebracht ist.

Die Definition 3.25 beschreibt die formale Notation eines Komponentenkommunikationsanschlusses, während die Definition 3.26 die graphische Darstellung im Flussdiagramm beschreibt.

Definition 3.25 ***Komponentenkommunikationsanschluss***
Sei o.B.d.A. ***B*** *eine (U)CML-Komponente. Dann kann ein* ***Komponentenkommunikationsanschluss*** *formal als*

Kommunikationsstecker	$B_{Art:St_{Kom}:Name:(Cond)}$
Kommunikationsbuchse	$B_{Art:Bu_{Kom}:Name:(Cond)}$

geschrieben werden.

Definition 3.26 ***Komponentenkommunikationsanschluss (graphische Notation)***
Mit Hilfe des ***Komponentenkommunikationsanschlusses*** *wird die „enge Kopplung“ zwischen einem Eingang und einem Ausgang modelliert. Dabei können Kommunikationsanschlüsse sowohl links als auch rechts an einer Komponente angebracht werden.*

links:  *rechts:* 

Beispiel 53: Paketkommunikationsschnittstellen und Komponentenkommunikationsanschlüsse
In der Abbildung 3.104 sind zwei Pakete *P1* und *P2* dargestellt. Zwischen den beiden Paketen *P1* und *P2* ist eine Paketkommunikationsschnittstelle abgebildet. Die Paketschnittstelle gehört zum Paket *P2* und lässt sich mit Hilfe der formalen Notation für Paketkommunikationsschnittstellen

$$P2_{Z:SS_{Kom}:IOSS1}$$

beschreiben. Des Weiteren haben die beiden Komponenten *K1* und *K2* jeweils einen Komponentenkommunikationanschluss, über den sie kommunizieren:

$$K1_{Z:St_{Kom}:IO1} \quad und \quad K2_{Z:Bu_{Kom}:IO1}.$$

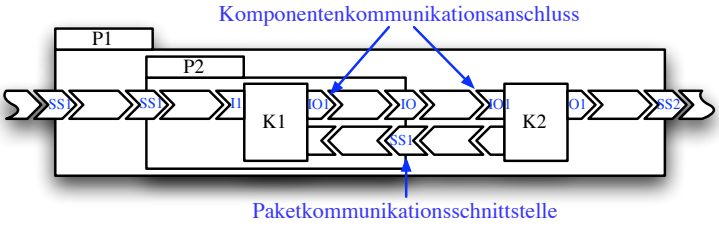


Abbildung 3.104.: Paketkommunikationsschnittstelle und Komponentenkommunikationsanschluss.

Beispielsweise ist die Kommunikation zwischen der Komponente *K1* und der Komponente *K2* ein C++ Methodenaufruf `int Add (int IN1, int IN2)`. Dabei übergibt die Komponente *K1* auf dem Hinweg die beiden Werte *IN1* und *IN2* der Komponente *K2*. Diese berechnet das Ergebnis aus den beiden Eingangsgrößen und sendet dieses über den Rückkanal an die Komponente *K1* zurück.

Außerdem kann aus der obigen Abbildung entnommen werden, dass eine Kommunikationsverbindung, die über eine Paketgrenze geht, über eine Paketkommunikationsschnittstelle erfolgt. Es ist nicht gestattet, eine „normale“ Verbindung über eine Kommunikationsschnittstelle zu verbinden. Sie hierzu das Kapitel 3.6.5 *(U)CML-Sprach- und Kompatibilitätsregelwerk* ab Seite 236, in dem das Regelwerk der (U)CML erläutert wird. □

3.6.2.4.3. Zusammengefasste Paketschnittstellen und zusammengefasste Komponentenanschlüsse

Eine weitere Modellierungsart der (U)CML für die Schnittstellen von Paketen bzw. die Anschlüsse von Komponenten ist die so genannte *zusammengefasste Paketschnittstelle* bzw. der *zusammengefasste Komponentenanschluss*. Wenn bei der Modellierung sehr viele unterschiedliche Signale (Verbindungen) von einer Komponente zu einer anderen geführt werden, können diese Verbindungen zu einer „virtuellen Verbindung“ (ähnlich den Verbindungen im Graphendiagramm) zusammengefasst werden. Auch hier gilt: Die virtuell zusammengefassten Einzelverbindungen sind nur eine besondere Darstellungsform im Flussdiagramm. Die modellierten Einzelverbindungen gehen dabei nicht verloren und können jederzeit einzeln angezeigt werden.

Definition 3.27 ***Zusammengefasste Paketschnittstellen und zusammengefasste Komponentenanschlüsse***
Sei $Anz, i \in \mathbb{N}; 0 < i < \infty$ *und* $Anz \geq 2$. *Des Weiteren sei:* PK_{Std} *eine Standardpaketschnittstelle oder ein Standardkomponentenanschluss. Dann gilt: Eine* ***zusammengefasste Standardpaketschnittstelle*** *bzw. ein* ***zusammengefasster Standardkomponentenanschluss*** $PK_{Std.Zus}$ *besteht aus* i *Standardpaketschnittstellen oder Standardkomponentenanschlüssen, also*

$$PK_{Std.Zus} = \sum_{i=1}^{Anz} PK_{Std_i}$$

mit $Anz =$ *Anzahl der* ***Standardpaketschnittstellen*** *bzw.* ***Standardkomponentenanschlüsse*** *die zusammengefasst werden sollen.*

Nach Definition 3.27 können Standardpaketschnittstellen bzw. Standardkomponentenanschlüsse mit $Anz \geq 2$ zu einer zusammengefassten Standardpaketschnittstelle bzw. einem zusammengefassten Standardkomponentenanschluss gebündelt werden. Es ist in den meisten Fällen jedoch nicht sinnvoll, „nur“ zwei Schnittstellen oder Anschlüsse zusammenzufassen, weil sich dadurch die Lesbarkeit des Systems nur geringfügig verbessert.

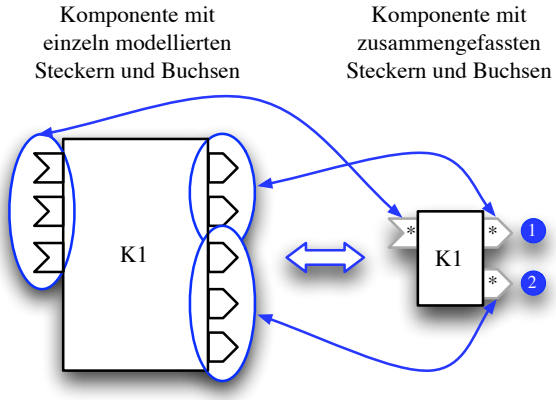


Abbildung 3.105.: Zusammengefasste Buchsen und Stecker einer (U)CML-Komponente.

In der Abbildung 3.105 ist auf der linken Seite die Komponente *K1* mit einzeln modellierten Standardbuchsen und -steckern dargestellt. Auf der rechten Seite der Abbildung ist die selbe Komponente *K1* mit zusammengefassten Buchsen und Steckern abgebildet. Dabei wurden die Ausgangsstecker der Komponente *K1* in zwei unterschiedliche Gruppen, mit zwei (1) und drei (2) Steckern, zusammengefasst. So kann z.B. die Steckergruppe 1 über den zusammengefassten Verbindungspfeil zu einer anderen Komponente als die der Gruppe 2 geführt werden, da die „Gabelung“ einer Verbindung nicht gestattet ist.

Beispiel 54: Anwendung zusammengefasster Komponentenanschlüsse
Die Modellierungsart mit zusammengefassten Komponentenanschlüssen ist besonders hilfreich, wenn zwei Komponenten viele Signale austauschen wollen wie z.B. der Hauptprozessor (CPU[215]) mit dem Hauptspeicher (RAM[216]) eines Rechners, es jedoch für das Verständnis des Systems unerheblich ist, ob eine oder viele Verbindungen zwischen den beiden Komponenten dargestellt sind (Abbildung 3.106). Der in der Abbildung (rechts) dargestellte zusammengefasste Komponentenanschluss *I0_32* kann nach Definition 3.27 als

$$CPU_{Z:St_{Std.Zus}:I0\_32}$$

geschrieben werden.

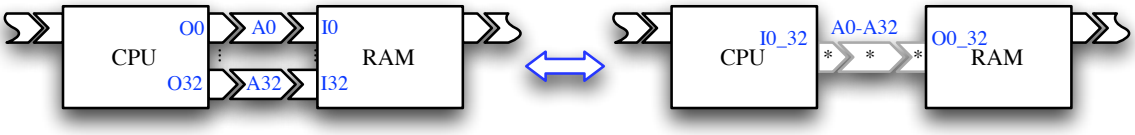


Abbildung 3.106.: Anwendung: Zusammengefasster Komponentenanschluss.

Hier ist es für das Verständnis der Funktionsweise unerheblich, ob es eine oder viele Verbindungen ($A_0 \ldots A_{32}$) zwischen der CPU und dem RAM gibt. Wichtig ist nur, dass es eine (zusammengefasste) Flussverbindung gibt, über die die CPU mit dem RAM unidirektional kommuniziert. □

[214] Siehe hierzu: Kapitel „3.6.2.5 *(U)CML-Flusspfeilarten*“ ab Seite 184.
[215] Das engl. Akronym *CPU* steht für **C**entral **P**rocessing **U**nit – Hauptprozessor [Wik07e][das07a].
[216] Das engl. Akronym *RAM* steht für **R**andom **A**ccess **M**emoy – Wahlfreier Schreib-/Lese-Speicher [Wik07r][das07b].

Zusammengefasste Paketschnittstellen
Eine *zusammengefasste Paketschnittstelle* dient zum Anschluss von zusammengefassten Verbindungen (Pfeilen) von innerhalb des Pakets über die Paketgrenze hinweg in die Paketumwelt bzw. umgekehrt.

Definition 3.28 ***Zusammengefasste Paketschnittstelle***
Sei o.B.d.A. ***A*** *ein (U)CML-Paket. Dann kann eine* ***zusammengefasste Paketschnittstelle*** *formal mit*

$$A_{Art:SS_{Std.Zus}:Name:(Cond)}$$

beschrieben werden.

Definition 3.29 ***Darstellung: Zusammengefasste Paketschnittstellen***
Mit Hilfe der ***zusammengefassten Paketschnittstelle*** *werden zusammengefasste Verbindungen von innerhalb des Pakets über die Paketgrenze hinweg in die Paketumwelt und umgekehrt realisiert. Zusammengefasste Paketschnittstellen sind mit einem grauen Rand umrandet und haben einen Stern in der Mitte.*

Beispiel 55: Zusammengefasste Paketschnittstelle
In der nachfolgenden Abbildung 3.107 ist sowohl eine *zusammengefasste Paketschnittstelle* in Black-Box Ansicht (1), als auch in Glass-Box Ansicht (2) dargestellt. Zusätzlich zu den beiden zusammengefassten Paketschnittstellen ist in (3) ein *zusammengefasster Komponentenanschluss* abgebildet.

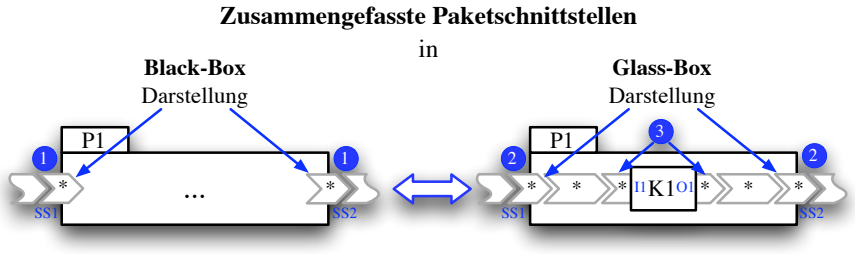


Abbildung 3.107.: Zusammengefasste Paketschnittstelle.

In der Abbildung 3.107 (rechts) ist außerdem dargestellt, dass eine zusammengefasste Paketschnittstelle stets nur an einem zusammengefassten Komponentenanschluss mittels eines zusammenfassten Verbindungspfeils verbunden werden kann. Die zusammengefasste Paketschnittstelle *SS2* lässt sich formal durch

$$P1_{Z:SS_{Std.Zus}:SS2}$$

beschreiben. □

Zusammengefasste Komponentenanschlüsse
Zusammengefasste Komponentenanschlüsse dienen zum Anschluss von zusammengefassten Pfeilen an eine Komponente. In der Abbildung 3.108 auf der nächsten Seite sind zwei zusammengefasste Eingangsbuchsen sowie zwei zusammengefasste Ausgangsstecker dargestellt, die an die Komponente *K1* angeschlossen sind.

Definition 3.30 ***Zusammengefasste Komponentenanschlüsse***
Sei o.B.d.A. ***B*** *eine (U)CML-Komponente. Dann kann ein* ***zusammengefasster Komponentenanschluss*** *formal als*

$$\begin{array}{ll} \textit{zusammengefasster Stecker} & B_{Art:St_{Std.Zus}:Name:(Cond)} \\ \textit{zusammengefasste Buchse} & B_{Art:Bu_{Std.Zus}:Name:(Cond)} \end{array}$$

geschrieben werden.

Definition 3.31 ***Zusammengefasste Komponentenanschlüsse (graphische Darstellung)***
Ein ***zusammengefasster Komponentenanschluss*** *wird als Standard Eingangsbuchse*  *bzw. Standard Ausgangsstecker* *mit grauer Umrandung und einem Stern in der Mitte dargestellt. Dabei werden die Buchsen stets links und die Stecker stets rechts an der Komponente angebracht.*

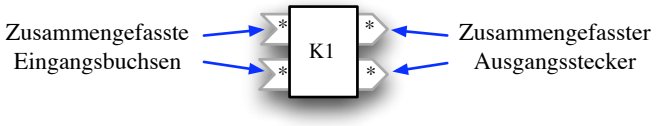


Abbildung 3.108.: Zusammengefasste Komponentenanschlüsse.

3.6.2.4.4. Zusammengefasste Paketkommunikationsschnittstelle sowie -Komponentenkommunikationsanschluss

Die letzten beiden Anschlussarten, die die (U)CML für die Modellierung von Systemen zur Verfügung stellt, sind die so genannten *zusammengefassten Paketkommunikationsschnittstellen* bzw. die *zusammengefassten Komponentenkommunikationsanschlüsse*. Sie stellen eine Kombination aus den oben vorgestellten Paketkommunikationsschnittstellen bzw. Komponentenkommunikationsanschlüssen dar.

Hat ein Paket oder eine Komponente mehr als zwei Kommunikationsschnittstellen/-Anschlüsse, die von einem Paket/Komponente zu einem gemeinsamen anderen Paket oder Komponente geführt werden, so können die Paketschnittstellen oder die Komponentenanschlüsse zusammengefasst werden.

Definition 3.32 ***Zusammengefasste Paketkommunikationsschnittstelle und Komponentenkommunikationsanschlüsse***
Sei $Anz, i \in \mathbb{N}; 0 < i < \infty$ *und* $Anz \geq 2$. *Des Weiteren sei* PK_{Kom} *eine Paketkommunikationsschnittstelle bzw. ein Komponentenkommunikationsanschluss. Dann gilt: Eine* ***zusammengefasste Kommunikationsschnittstelle/-anschluss*** $PK_{Kom.Zus}$ *besteht aus* i *Paketkommunikationsschnittstellen bzw. Komponentenkommunikationsanschlüssen, also*

$$PK_{Kom.Zus} = \sum_{i=1}^{Anz} PK_{Kom_i}$$

mit $Anz =$ *Anzahl der* ***Paketkommunikationsschnittstellen*** *bzw.* ***Komponentenkommunikationsanschlüssen*** *die zusammengefasst werden sollen.*

Zusammengefasste Paketkommunikationsschnittstelle
Die *zusammengefasste Paketkommunikationsschnittstelle* ist, wie der Name andeutet, eine Kombination aus vielen Standardpaketkommunikationsschnittstelle, die zu einer virtuellen Paketkommunikationsschnittstelle zusammengefasst worden sind.

Definition 3.33 ***Zusammengefasste Paketkommunikationsschnittstelle***
Sei o.B.d.A. ***A*** *ein (U)CML-Paket. Dann kann eine* ***zusammengefasste Kommunikationspaketschnittstelle*** *formal mit*

$$A_{Art:SS_{Kom.Zus}:Name:(Cond)}$$

geschrieben werden.

Definition 3.34 ***Zusammengefasste Paketkommunikationsschnittstelle (graphische Darstellung)***
Die ***zusammengefasste Paketkommunikationsschnittstelle*** *kann sowohl links als auch rechts an einem Paket angebracht werden. Dabei gilt: Die beiden Bestandteile der Schnittstelle also der Hin- und Rückkanalanschluss, sind atomar und werden stets eng aneinander liegend mit einem grauen Rand und einem Stern in der Mitte gezeichnet*[217].

Die Abbildung 3.109 auf der nächsten Seite zeigt auf der linken Seite das Paket *P1* in Black-Box Darstellung mit einer *zusammengefassten Paketkommunikationsschnittstelle* an seiner rechten Seite (1).

Auf der rechten Seite der Abbildung 3.109 ist das selbe Paket *P1* in Glass-Box Ansicht dargestellt. Auch hier ist die zusammengefasste Paketkommunikationsschnittstelle auf der rechten Seite des Pakets *P1* gezeichnet (1). Außerdem ist in der Glass-Box Darstellung die Komponente *K1* sichtbar. Sie hat einen *zusammengefassten Komponentenkommunikationsanschluss* (2) an der rechten Seite, der durch einen zusammengefassten Kommunikationspfeil mit der zusammengefassten Paketkommunikationsschnittstelle verbunden ist.

Zusammengefasster Komponentenkommunikationsanschluss
Ein *zusammengefasster Komponentenkommunikationsanschluss* wird in einem Modell verwendet, wenn zwischen zwei (U)CML-Komponenten viele einzelne Kommunikationsverbindungen existieren, es aber für das Verständnis der Funktionalität unerheblich ist, ob eine oder viele Verbindungen dargestellt werden (vgl. Zusammengefasste Paketkommunikationsschnittstelle).

[217] Vgl. Definition 3.24 auf Seite 179.

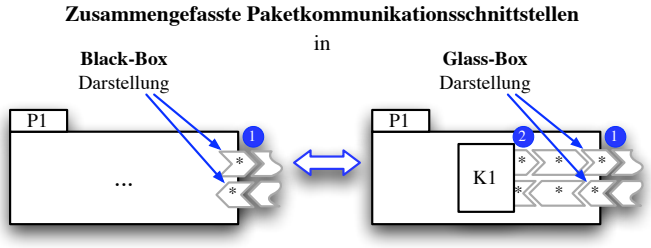


Abbildung 3.109.: Zusammengefasste Paketkommunikationsschnittstelle.

Definition 3.35 ***Zusammengefasste Komponentenkommunikationsanschluss***
Sei o.B.d.A. ***B*** *eine (U)CML-Komponente. Dann kann ein* ***zusammengefasster Komponentenkommunikationsanschluss****, formal als*

zusammengefasster Kommunikationsstecker	$B_{Art:St_{Kom.Zus}:Name:(Cond)}$
zusammengefasste Kommunikationsbuchse	$B_{Art:Bu_{Kom.Zus}:Name:(Cond)}$

geschrieben werden.

Definition 3.36 ***Zusammengefasster Komponentenkommunikationsanschluss (graphische Darstellung)***
Der ***zusammengefasste Komponentenkommunikationsanschluss*** *dient zur virtuell zusammengefassten Modellierung vieler einzelner Komponentenkommunikationsanschlüsse, die von einer Komponente zu einer anderen gehen.*

links: *rechts:*

In der Abbildung 3.110 ist die Komponente *K1* aus Abbildung 3.109 noch einmal einzeln dargestellt. Auf der rechten Seite der Komponente *K1* befindet sich der zusammengefasste Komponenten-Kommunikationsanschluss.

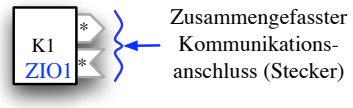


Abbildung 3.110.: Zusammengefasster Komponentenkommunikationsanschluss.

Der zusammengefasste Komponentenkommunikationsanschluss *ZIO1* kann mit Hilfe der formalen Notation, aus Definition 3.35, wie folgt geschrieben werden:

$$K1_{Z:St_{Kom.Zus}:ZIO1}.$$

3.6.2.4.5. Erweiterung des Farbschemas auf Paketschnittstellen und Komponentenanschlüsse

Im Abschnitt „3.6.2.3 *(U)CML-Komponente*“ ab Seite 165 wurde ein Farbschema für die erweiterte Darstellung von Komponenten eingeführt. Das dort eingeführte Farbschema kann auch auf alle Stecker, Buchsen und Schnittstellen, die die (U)CML zur Modellierung im Flussdiagramm zur Verfügung stellt, angewendet werden.

Erweiterung der Farbschemadefinition 3.10 auf Seite 167 für Stecker, Buchsen und Schnittstellen:

Definition 3.37 ***Farbschema von Stecker, Buchsen und Schnittstellen***
Stecker, Buchsen und Schnittstellen können abhängig von ihrem Typ – keine Zuordnung, Elektrik/Elektronik, Mechanik und Software – farblich markiert dargestellt werden.

3.6.2.5. (U)CML-Flusspfeilarten

Nachdem nun sämtliche Paket- und Komponentenarten mit den dazugehörigen Paketschnittstellen und Komponentenanschlüssen, die die (U)CML zur Modellierung der Struktur von Systemen zur Verfügung stellt, in den letzten Kapiteln eingeführt und erläutert wurden, folgt in diesem Kapitel eine ausführliche Erklärung der Eigenschaften von (U)CML-Flusspfeilen anhand von einfachen Beispielen.

Flusspfeile dienen in (U)CML zur Verbindung von Paketen und Komponenten, genauer: zur Verbindung von Paketschnittstellen und Komponentenanschlüssen. Dabei gilt insbesondere, dass die Flusspfeile *genau einen Ausgang* (i.A. die Quelle) mit *genau einem Eingang* (i.A. die Senke) eines Pakets oder einer Komponente verbinden. Flusspfeile stellen somit eine „Punkt-zu-Punkt“

Verbindung von der Quelle zur Senke her. Insbesondere repräsentieren die Flusspfeile einen *virtuellen Kanal*, über den die Pakete und Komponenten Daten austauschen können. Dabei ist es unerheblich, ob elektrische Signale, mechanische Kräfte oder Softwaredaten über einen Flusspfeil transportiert werden sollen.

In der nachfolgenden Abbildung 3.111 sind zwei Komponenten *K1* und *K2* mit je einem Ein- und einem Ausgang dargestellt, die mittels eines Standardflusspfeils (1) miteinander verbunden sind.

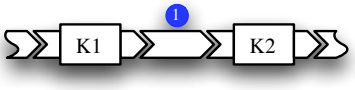


Abbildung 3.111.: Verbindung zwischen zwei Komponenten mit einem Standardflusspfeil.

Mit Hilfe des Standardflusspfeils (1) ist der Ausgang der Komponente *K1* mit dem Eingang der Komponente *K2* statisch verbunden, so dass nun ein Datum oder ein Signal „über" den Flusspfeil in Pfeilrichtung gesendet werden kann. Welche Daten zwischen dem Ein- und Ausgang ausgetauscht werden können wird durch das *Beschreibungsfeld*[218] der Quelle festgelegt. Eine weitere Eigenschaft aller (U)CML-Flusspfeile besteht darin, dass sie das zu übertragende Datum nicht ändern und zwischen dem Sendezeitpunkt und dem Empfangszeitpunkt keine Zeit vergeht, also die Daten instantan am Eingang anliegen, nachdem der Ausgang das Datum gesendet hat[219].

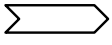
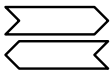
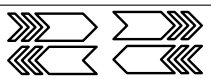
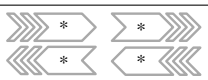
Flusspfeilart	Standard	Zusammengefasst
Standardflusspfeil		
Kommunikationsflusspfeil		
Externer Systemeingangspfeil Externer Systemausgangspfeil		
Externer Systemkommunikationspfeil		

Tabelle 3.9.: Flusspfeilarten der (U)CML.

In (U)CML gibt es jedoch nicht nur *Standardflusspfeile* sondern für jede Paketschnittstellenart bzw. Komponentenanschlussart einen speziellen Flusspfeil, mit dessen Hilfe je zwei Schnittstellen/Anschlüsse (Quelle → Senke) miteinander verbunden werden können. Die Tabelle 3.9 zeigt alle Flusspfeilarten, die die (U)CML für die Modellierung im Flussdiagramm zur Verfügung stellt ohne die speziellen BUS-System-Pfeilarten. Die BUS-System-Pfeilarten sind in der Tabelle 3.11 auf Seite 201 zusammengefasst dargestellt.

Grundeigenschaften aller (U)CML-Flusspfeilarten
Alle in (U)CML vorhandenen Flusspfeilarten besitzen die gleichen Grundeigenschaften. Diese sind in den nachfolgenden beiden Definitionen (3.38, 3.39) zusammengefasst.

Definition 3.38 ***(U)CML-Flusspfeile***
(U)CML-Flusspfeile *dienen zur* ***Punkt-zu-Punkt Verbindung*** *von Paketschnittstellen bzw. Komponentenanschlüssen mit- und untereinander. Dabei gilt:*

- *Flusspfeile transportieren ein Datum von der* ***Quelle*** *zur* ***Senke*** *ohne dieses zu verändern („virtueller Kanal").*
- *Das zu übertragende Datum ist im Beschreibungsfeld des Senders genau spezifiziert.*
- *Die Kommunikation erfolgt instantan, sofern kein Anschlussverhalten der Komponente definiert ist.*
- *Flusspfeile sind gerichtete Verbindungen.*
- *Rekursive Verbindungen*[220] *sind nicht erlaubt.*
- *Flusspfeile können selbst keine Paketgrenzen „überwinden".*
- *Flusspfeile verbinden stets „gleiche" Schnittstellen/Anschlussarten miteinander.*
- *Flusspfeile sind Punkt-zu-Punkt Verbindungen. Eine Gabelung ist nicht zulässig.*
- *Flusspfeile können „virtuell" zu einem zusammengefassten Flusspfeil zusammengefasst werden (betrifft nur Ansicht).*

[218] Siehe hierzu: Kapitel „3.6.2.6 *(U)CML-Beschreibungsfelder*" ab Seite 200.
[219] Anmerkung:
Die Eigenschaft der *instantanen Kommunikation* zwischen Quelle und Senke gilt nur für das statische Modell. Wenn eine Komponente ein externes Anschlussverhalten hat, wird die „Kommunikationszeit" ebenfalls berücksichtigt.
[220] Siehe hierzu: Kapitel „A.2 *Relationen*" ab Seite 317.

In der obigen Definition 3.38 auf der vorherigen Seite sind die Grundeigenschaften aller (U)CML-Flusspfeilarten aufgelistet. In der nachfolgenden Abbildung 3.112 wird der sechste Punkt – Flusspfeile können keine Paketgrenzen überwinden – aus der obigen Definition noch einmal aufgegriffen.

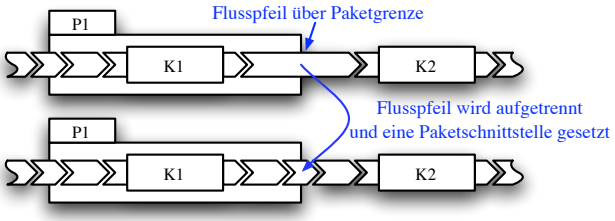


Abbildung 3.112.: Flusspfeil über eine Paketgrenze hinweg.

Soll in (U)CML ein Flusspfeil über eine Paketgrenze hinweg ein Datum transportieren, so muss das Paket eine entsprechende Paketschnittstelle besitzen. Ist keine entsprechende Paketschnittstelle vorhanden, kann entweder eine neue eingeführt oder der Flusspfeil nicht gezeichnet werden.

Beispiel 56: (U)CML-Flusspfeile
In der Abbildung 3.113 sind verschiedene in (U)CML erlaubte und **nicht** erlaubte Verbindungsarten abgebildet. Im oberen Teil der Abbildung sind zwei **erlaubte Verbindungsarten** dargestellt. Zum einen (links) eine *Standardverbindung* zwischen den beiden Komponenten *K1* und *K2* mittels eines Standardflusspfeils. Auf der rechten oberen Seite ist eine *Kommunikationsverbindung*, bestehend aus einem Hin- und einem Rückkanal, zwischen den beiden Komponenten *K3* und *K4* modelliert.

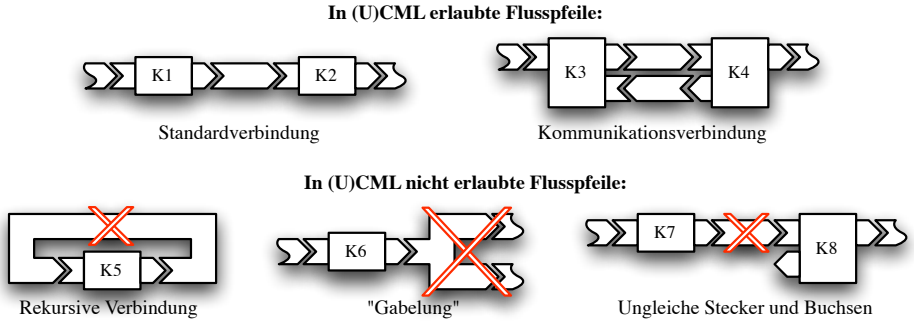


Abbildung 3.113.: In (U)CML zulässige und unzulässige Verbindungsarten.

Im unteren Bereich der Abbildung 3.113 sind drei in (U)CML **unzulässige Verbindungsarten** dargestellt. Auf der linken unteren Seite der Abbildung ist eine, in (U)CML nicht erlaubte *rekursive Verbindung*, die vom Ausgang (Quelle) der Komponente *K5* zum Eingang (Senke) der Komponente *K5* geht, dargestellt. In der unteren Bildmitte ist eine unzulässige *Gabelung* eines Flusspfeils abgebildet, der vom Ausgangsstecker der Komponente *K6* ausgeht und sich anschließend verzweigt. Eine Gabelung muss über eine Komponente realisiert werden, die die Verteilung des Signals übernimmt. Auf der rechten Seite ist eine *ungültige Verbindung*, die von einem Standard Ausgangsstecker über einen Standardflusspfeil hin zu einer Kommunikationsschnittstelle geht modelliert. □

Definition 3.39 ***(U)CML-Flusspfeile***
Sei $A, B \in \{Paket, Komponente\}$ und ***D***[221] *das über den Flusspfeil F zu übertragenden Datum. Des Weiteren sei o.B.d.A.* ***A*** *die Quelle und* ***B*** *die Senke einer Verbindung. Dann lassen sich die* ***Standardflusspfeile***, ***zusammengefassten Flusspfeile***, ***Kommunikationsflusspfeile*** *sowie* ***zusammengefasste Kommunikationsflusspfeile*** *formal schreiben als:*

$$A_{Art:Steckerart:Name:(Cond)} \xrightarrow{F_{Art:Flusspfeilart:Datum}} B_{Art:Buchsenart:Name:(Cond)}$$

mit

Art	*Z (zwingend), OPT (optional)*
Steckerart	*Standardstecker/-buchse/-schnittstelle ($St_{Std}, Bu_{Std}, SS_{Std}$)* *Kommunikationsstecker/-buchse/-schnittstelle ($St_{Kom}, Bu_{Kom}, SS_{Kom}$)* *Zusammengefasste Standard- bzw. Kommunikationsstecker/-buchse/-schnittstelle ($St_{Std.Zus}, Bu_{Std.Zus}, SS_{Std.Zus}; St_{Kom.Zus}, Bu_{Kom.Zus}, SS_{Kom.Zus}$)*
Name	*informeller Name*
Cond	*Vorbedingung, Nachbedingung und Invariante (pre, post, inv)*
Flusspfeilart	*Standardflusspfeil, Kommunikationsflusspfeil, zusammengefasster Standardflusspfeil sowie zusammengefasster Kommunikationsflusspfeil (S, K, SZ, KZ)*
Datum	*Das zu übertragende Datum*

[221] Das zu übertragende Datum D ist im Sender (Quelle) der Nachricht genau spezifiziert (Siehe hierzu: Kapitel „3.6.2.6 *(U)CML-Beschreibungsfelder*“ ab Seite 200.).

Die nachfolgende Definition 3.40 dient zur Vereinfachung der allgemeinen Flusspfeilnotation (vgl. Definition 3.39).

Definition 3.40 ***Vereinfachung der allgemeinen Flusspfeilnotation***
Die in der Definition 3.39 eingeführte formale Notation für Flusspfeile in (U)CML kann vereinfachte werden, sofern eine oder mehrere der nachfolgenden Bedingungen zutreffen:

- *Sind sämtliche Bestandteile eines Flusspfeils* ***zwingend*** *bzw.* ***optional****, so kann das „Z" für zwingend bzw. das „OPT" für optional weggelassen werden.*
- *Die in der Definition 3.41 eingeführte Notation lässt sich verkürzt*

$$A \xrightarrow{F_{Art:Flusspfeilart_{Name}:Datum}} B$$

schreiben, sofern die exakte Beschreibung der Quelle bzw. der Senke der Verbindung für das Verständnis der Verbindung unwichtig ist. Um dennoch verschiedene Flusspfeile zwischen einer Quelle und Senke identifizieren zu können, wird der informelle Name der Verbindung als Index an die Flusspfeilart angehängt.
- *Sind für einen Flusspfeil keine Bedingungen vorhanden, so kann das Feld Cond ebenfalls weggelassen werden.*

Beispiel 57: Notation von (U)CML-Flusspfeilen
Die Abbildung 3.114 zeigt die Komponente *NAND*, die sich im Systempaket *System* befindet[222]. In der Abbildung sind außerdem sämtliche Bezeichnungen eingetragen, die für die formale Beschreibung des Systems notwendig sind.

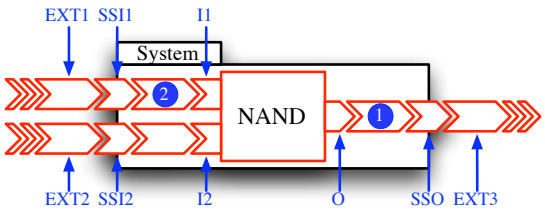


Abbildung 3.114.: Elektrisches/elektronisches Beispielsystem mit drei Flusspfeilen.

1. Flusspfeil zwischen dem Stecker *O* der Komponente *NAND* und der Paketschnittstelle *SSO* des Systempakets

$$NAND_{Z:St_{Std}:O:(post:O\ in\ [0\ldots5]V)} \xrightarrow{F_{Z:S:D}} System_{Z:SS_{Std}:SSO:(post:SSO\ in\ [0\ldots5]V)}$$

2. Flusspfeil zwischen der Paketschnittstelle *SSI1* und der Buchse *I1* der Komponente *NAND*

$$System_{Z:SS_{Std}:SSI1:(post:SSI1\ in\ [0\ldots5]V)} \xrightarrow{F_{Z:S:D}} NAND_{Z:Bu_{Std}:I1:(pre:I1\ in\ [0\ldots5]V)}$$

Vereinfachung der Notation 1:
Da sämtliche Bestandteile des Flusspfeils zwingend sind, kann das „Z" nach Definition 3.40 weggelassen werden.

$$NAND_{St_{Std}:O:(post:O\ in\ [0\ldots5]V)} \xrightarrow{F_{S:D}} System_{SS_{Std}:SSO:(post:SSO\ in\ [0\ldots5]V)}$$

Diese Schreibweise kann weiter zu

$$NAND \xrightarrow{F_{S:D}} System$$

vereinfacht werden, sofern die genaue Beschreibung der Stecker und Buchsen für das Verständnis der Verbindung unwichtig ist. □

Anmerkung:
Die externen Systemeingangs- und -ausgangspfeile sind in der obigen Definition nicht enthalten, da sie eine spezielle Ausprägung eines Flusspfeils repräsentieren. Sie werden im Abschnitt „3.6.2.5.3 *Externer (U)CML-Systemeingangs- und -ausgangspfeil*" ab Seite 194 eingeführt und erläutert.

Die Tabelle 3.10 auf der nächsten Seite zeigt eine kompakte Aufstellung aller in (U)CML möglichen Verbindungsarten mit Ausnahme der BUS-System-Verbindungsarten. Diese werden im Kapitel 3.6.2.5.6 *BUS-System-Pfeil* ab Seite 200 erläutert.

Ansichten auf einen Flusspfeil im Flussdiagramm
Im (U)CML-Flussdiagramm gibt es nach Definition 3.39 auf der vorherigen Seite zwei Arten von Flusspfeilen. Zum einen die *normalen* Flusspfeile (Standard- bzw. Kommunikationsflusspfeile), die genau eine Quelle mit genau einer Senke verbinden. Zum anderen die so genannten *zusammengefassten Flusspfeile*. Zusammengefasste Flusspfeile werden in (U)CML verwendet, wenn es zwischen einem Paket oder einer Komponente viele gleichartige Flusspfeile gibt, es für das Verständnis der Verbindung jedoch unerheblich ist ob ein oder mehrere Verbindungspfeile modelliert sind. In diesem Fall können die einzelnen Flusspfeile zu einem

[222] Die Abbildung 3.114 basiert größtenteils auf der Abbildung 3.95 sowie den dort im Text angegebenen statischen Bedingungen.

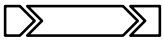
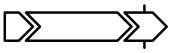
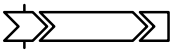
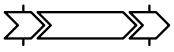
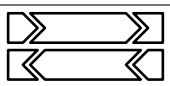
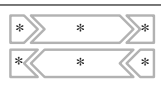
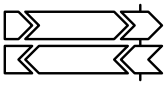
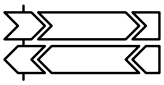
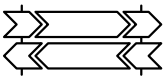
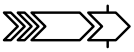
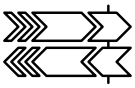
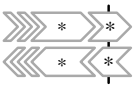
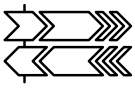
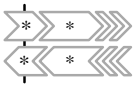
Quelle		Ziel	Beispiele	
			Standard	Zusammengefasst
Stecker	→	Buchse		
Stecker	→	Schnittstelle		
Schnittstelle	→	Buchse		
Schnittstelle	→	Schnittstelle		
Kommunikations-stecker	⇌	Kommunikations-buchse		
Kommunikations-stecker	⇌	Kommunikations-schnittstelle		
Kommunikations-schnittstelle	⇌	Kommunikations-buchse		
Kommunikations-schnittstelle	⇌	Kommunikations-schnittstelle		
Umwelt	→	Systempaket-schnittstelle		
Systempaket-schnittstelle	→	Umwelt		
Umwelt	⇌	Systempaket Kommunikationsschnittstelle		
Systempaket Kommunikationsschnittstelle	⇌	Umwelt		

Tabelle 3.10.: Zulässige Verbindungsarten in (U)CML.

einzelnen zusammengefassten Flusspfeil zusammengefasst werden. Die Zusammenfassung passiert jedoch nur auf der graphischen Zeichenebene, ähnlich wie bei der Black-Box und Glass-Box Darstellung der Pakete. Im dahinter liegenden Datenmodell werden alle zusammengefassten Flusspfeile einzeln repräsentiert und nur in der Ansicht entweder als einzelne Flusspfeile oder als zusammengefasster Flusspfeil dargestellt.

Um die zusammengefassten Flusspfeile effektiv nutzen zu können, ist eine Werkzeugunterstützung zwingend notwendig, da sonst jede Zusammenfassung von einzelnen Flusspfeilen mit zusätzlichem Zeichenaufwand verbunden ist (vgl. Abschnitt „3.8 *(U)CML-ed, ein graphischer Editor für die Kompatibilitätsmodellierungssprache (U)CML*“ ab Seite 254).

Beispiel 58: Zusammengefasste Flusspfeile
Wie bereits im Kapitel „3.6.2.4.3 *Zusammengefasste Paketschnittstellen und zusammengefasste Komponentenanschlüsse*“ ab Seite 180 gezeigt, können Paketschnittstellen und Komponentenanschlüsse in (U)CML zusammengefasst werden, wenn sie die *gleichen Grundeigenschaften* besitzen. Im dortigen Beispiel wurde gezeigt, dass wenn es für das Verständnis eines Systems/Teilsystems unerheblich ist, wie viele Verbindungen zwischen zwei Paketen bzw. Komponenten gezeichnet werden, die einzelnen Flusspfeile zu einem zusammengefassten Flusspfeil zusammengefasst werden können. Im dahinter liegenden Modell bleiben die einzelnen Flusspfeile unverändert erhalten. □

Anmerkungen:

- Eine ausführliche Aufzählung aller in (U)CML zulässigen Verbindungsarten und Regeln finden Sie im Kapitel „3.6.5 *(U)CML-Sprach- und Kompatibilitätsregelwerk*“ ab Seite 236.
- Alle im Beispiel auf Seite 186 modellierten fehlerhaften Verbindungen können mit Hilfe des graphischen Werkzeugs „*(U)CML-ed* “ (Siehe hierzu: Kapitel „3.8 *(U)CML-ed, ein graphischer Editor für die Kompatibilitätsmodellierungssprache (U)CML*“ ab Seite 254.) bereits während der Modellierungsphase gefunden, bzw. gar nicht gezeichnet werden.
- (U)CML-Pakete bzw. die Paketschnittstellen besitzen keine „aktiven“ Eigenschaften. Sie können nicht als Sender (Quelle) bzw. Empfänger (Senke) einer Verbindung dienen.
- Im nachfolgenden Text wird aus Vereinfachungsgründen meistens von der Kommunikation zwischen Paketen und Komponenten gesprochen, obwohl die eigentliche Kommunikation stets über Paketschnittstellen bzw. Komponentenanschlüsse geführt werden muss.

In den nachfolgenden Abschnitten dieses Kapitels werden sämtliche Flusspfeilarten, die die (U)CML zur Modellierung von Verbindungen zur Verfügung stellt, eingeführt und anhand von einfachen Beispielen erläutert. Begonnen wird mit dem Standardflusspfeil.

3.6.2.5.1. Standardflusspfeil

In (U)CML werden mit Hilfe der *Standardflusspfeile* unidirektionale Punkt-zu-Punkt Verbindungen zwischen *genau einer Quelle* und *genau einer Senke* hergestellt. Genauer: Standardflusspfeile verbinden genau einen *Standardausgangsstecker* einer Komponente mit genau einer *Standardeingangsbuchse*[223] einer anderen Komponente (Quelle $\neq$ Senke). Des Weiteren kann über einen Standardflusspfeils genau ein *Standardausgangsstecker* einer Komponente mit genau einer *Standardpaketschnittstelle*, bzw. eine *Standardpaketschnittstelle* mit genau einer *Standardeingangsbuchse* einer Komponente verbunden werden. Sollen zwei Komponenten durch einen Standardflusspfeil verbunden werden und liegt auf dem Verbindungsweg zwischen den beiden zu verbindenden Komponenten eine Paketgrenze, so wird eine Standardpaketschnittstelle eingefügt (vgl. Abbildung: 3.112 auf Seite 186).

Ist es für die Modellierung eines Signals wichtig, einen dezidierten Hin- und Rückweg zu modellieren, kann dies mit Hilfe von zwei unidirektionalen Flusspfeilen oder mit Hilfe des *Kommunikationsflusspfeils* realisiert werden. Im Kapitel „3.6.2.4.2 *Paketkommunikationsschnittstellen und Komponentenkommunikationsanschlüsse*“ ab Seite 179 (Abbildung: 3.103 auf Seite 179) wurde gezeigt, dass eine *Kommunikationsverbindung* zwischen zwei Komponenten sowohl aus zwei einfachen Standardflusspfeilen als auch mit Hilfe eines *Kommunikationsflusspfeils* modelliert werden kann. Die speziellen Kommunikationsflusspfeile werden im nächsten Abschnitt erläutert.

Definition 3.41 ***Standardflusspfeil***
Sei $A, B \in \{Paket, Komponente\}$ und D das über den ***Standardflusspfeil*** *F_S von der Quelle A zur Senke B zu übertragende Datum. Dann lässt sich ein Flusspfeil zwischen A und B formal schreiben als:*

$$A_{Art:Steckerart:Name:(Cond)} \xrightarrow{F_{Art:S:D}} B_{Art:Buchsenart:Name:(Cond)}$$

bzw. nach Definition 3.40 vereinfacht

$$A \xrightarrow{F_{Art:S_{Name}:D}} B$$

mit $Steckerart = \{St_{Std}, SS_{Std}\}, Buchsenart = \{Bu_{Std}, SS_{Std}\}$.

Anmerkungen zur Definition 3.41:

- Wenn es für das Verständnis eines Systems unwichtig ist, welches Datum zwischen zwei Kommunikationspartnern übertragen werden soll, kann der Parameter D in der formalen Notation der Verbindung weggelassen werden, da die eigentliche Definition des Datums im Beschreibungsfeld des Senders bzw. Empfängers enthalten ist.
- Nach Definition 3.40 kann der Parameter *Art* bei der Flusspfeilbeschreibung weggelassen werden, sofern es eineindeutig ist, dass alle Bestandteile des Flusspfeils zwingend oder optional sind.

In der nachfolgenden Definition 3.42 wird die graphische Darstellung des oben definierten Standardflusspfeils im Flussdiagramm festgelegt.

Definition 3.42 ***Standardflusspfeil (graphische Darstellung)***
Ein ***Standardflusspfeil*** *verbindet genau einen Standardausgangsstecker mit genau einer Standardeingangsbuchse. Standardflusspfeile werden ebenfalls für die Verbindung von Steckern bzw. Buchsen mit einer Standardpaketschnittstelle verwendet.*

Beispiel 59: Standardflusspfeile
In der Abbildung 3.115 ist ein kleiner Ausschnitt aus einem in (U)CML modellierten System, welches aus zwei Komponenten *K1*, *K2* und einem Paket *P1* besteht, in Black-Box Ansicht dargestellt.

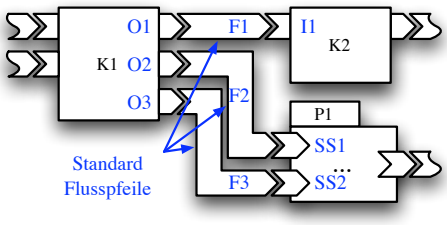


Abbildung 3.115.: Anwendung: Standardflusspfeile.

[223] Siehe hierzu: Kapitel „3.6.2.4.1 *Standardpaketschnittstellen und Standardkomponentenanschlüsse*“ ab Seite 176.

Die beiden Komponenten *K1* und *K2* des Teilsystems (genauer der Stecker *O1* der Komponente *K1* ist mit der Buchse *I1* der Komponente *K2*) durch den *Standardflusspfeil* $F1$ verbunden. Dieser informelle Sachverhalt lässt sich mit Hilfe der Definition 3.41 formal schreiben als:

$$K1_{Z:St_{Std}:O1} \xrightarrow{F_{Z:S:D}} K2_{Z:Bu_{Std}:I1}$$

bzw. in der vereinfachten Notation

$$K1 \xrightarrow{F_{Z:S_{F1}:D}} K2.$$

Über den Flusspfeil $F1$ kann die Komponente *K1* der Komponente *K2* das Datum D_1 zuschicken. Das Datum wird dabei in Pfeilrichtung (unidirektional) von der Komponente *K1* ausgehend zur Komponente *K2* geschickt. Außerdem ist die Komponente *K1* über zwei Standardflusspfeile $F2$ und $F3$ mit dem Paket *P1* verbunden. Für die beiden Standardflussrelationen $F2$ und $F3$, die ebenfalls von der Komponente *K1* ausgehen, gilt der gleiche Sachverhalt wie für den Flusspfeil $F1$. Also verbinden die beiden Standardflusspfeile $F2$ und $F3$ die Komponente *K1* mit dem Paket *P1* ($K1 \xrightarrow{F_{Z:S_{F2}:D_2}} P1$ sowie $K1 \xrightarrow{F_{Z:S_{F3}:D_3}} P1$).

Über den weiteren Datenfluss innerhalb des Pakets *P1* kann a priori keine Aussage getroffen werden, da dieses Paket in Black-Box Darstellung abgebildet ist. Sollte z.B. die obere Paketschnittstelle (SS1) innerhalb des Pakets nicht angeschlossen sein so würde die gesamte Kommunikation über den Flusspfeil $F2$ nicht funktionieren und ein Fehler angezeigt werden.

Ist es für die Modellierung bzw. das Verständnis eines Systems unwichtig, welches Datum zwischen zwei Kommunikationspartnern übertragen werden soll, so kann es in der formalen Notation der Verbindung weggelassen werden. Die Beschreibung des zu übertragenden Datums ist weiterhin im Beschreibungsfeld der Senders enthalten. Im Beispiel kann die Verbindung zwischen den beiden Komponenten $K1$ und $K2$ somit auch vereinfacht als $K1 \xrightarrow{F_{Z:S_{F1}}} K2$ ohne den Parameter D geschrieben werden, sofern das zu übertragende Datum D im Beschreibungsfeld des Steckers der Komponente $K1$ enthalten ist. □

Zusammengefasster Standardflusspfeil
Sind zwischen zwei Paketen oder Komponenten viele einzelne *Standardflusspfeile* modelliert, so können diese, sofern sie die gleichen Domäne repräsentieren – Elektrik/Elektronik, Mechanik, Software – , zu einem *zusammengefassten Standardflusspfeil* zusammengefasst werden. Genauer: Es werden nicht nur die Verbindungen, also die Standardflusspfeile, sondern auch die dazugehörigen Anschlusspunkte, die Buchsen/Stecker der Komponenten sowie die Paketschnittstellen, zu einer zusammengefassten Verbindung[224]. Dabei ist zu beachten, dass die Zusammenfassung der einzelnen Standardflusspfeile zu einem zusammengefassten Standardflusspfeil nur „virtuell" ist. Das heißt, die Zusammenfassung von einzelnen Standardflusspfeilen zu einem zusammengefassten Standardflusspfeil passiert *nur* in der graphischen Repräsentation des Modells im Flussdiagramm. Im dahinter liegenden Datenmodell werden die an dem zusammengefassten Standardflusspfeil beteiligten Standardflusspfeile weiter einzeln gespeichert, so dass diese wieder aufgetrennt werden können.

In den beiden nachfolgenden Definitionen 3.43 und 3.44 wird die formale Notation der zusammengefassten Standardflusspfeile festgelegt und in der Definition 3.45 auf der nächsten Seite ihre graphische Repräsentation im Flussdiagramm.

Definition 3.43 ***Zusammengefasster Standardflusspfeil***
Sei $Anz, i \in \mathbb{N}; 1 \leq i < \infty$ und $Anz \geq 2$. Des Weiteren sei $F_{Art:S:D}$ ein Standardflusspfeil, der das Datum D von der Quelle zur Senke transportiert. Dann gilt: Ein ***zusammengefasster Flusspfeil*** *$F_{Art:SZ:D_{ges}}$ besteht aus Anz Standardflusspfeilen, also*

$$F_{Art:SZ:D_{ges}} = \sum_{i=1}^{Anz} F_{Art:S_i:D_i}$$

mit Anz = Anzahl der ***Standardflusspfeile*** *die zusammengefasst werden sollen. D_{ges} ist die Menge aller Daten der Standardflusspfeile $F_{Art:S:D_i}$, also $D_{ges} = \{D_1, D_2, \ldots, D_{Anz}\}$.*

Definition 3.44 ***Zusammengefasster Standardflusspfeil***
Sei $A, B \in \{Paket, Komponente\}$ und D_{ges} das über den ***zusammengefassten Standardflusspfeil*** *$F_{SZ:D_{ges}}$ von der Quelle A zur Senke B zu übertragende zusammengefasste Datum. Dann lässt sich ein zusammengefasster Standardflusspfeil zwischen A und B formal schreiben als:*

$$A_{Art:Steckerart:Name:(Cond)} \xrightarrow{F_{Art:SZ:D_{ges}}} B_{Art:Buchsenart:Name:(Cond)}$$

bzw. nach Definition 3.40 vereinfacht

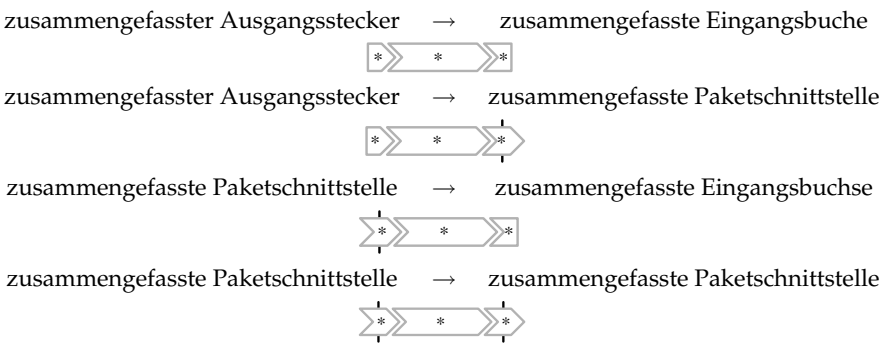
$$A \xrightarrow{F_{Art:SZ_{Name}:D_{ges}}} B$$

mit $Steckerart = \{St_{Std.Zus}, SS_{Std.Zus}\}$, $Buchsenart = \{Bu_{Std.Zus}, SS_{Std.Zus}\}$. D_{ges} ist die Menge aller Daten der involvierten Standardflusspfeile $F_{Art:S:D_i}$, also $D_{ges} = \{D_1, D_2, \ldots, D_{Anz}\}$.

Anmerkungen zur Definition 3.43 und 3.44:

- Ein *zusammengefasster Standardflusspfeil* dient zur Verbindung von zusammengefassten Komponentenanschlüssen und zusammengefassten Paketschnittstellen. Andere Kombinationen aus Steckern/Buchsen und Paketschnittstelle sind in (U)CML nicht zulässig.

[224] Siehe hierzu: Kapitel „3.6.2.5.5 *Zusammengesetzter (U)CML-Pfeil*" ab Seite 198.



- In (U)CML können beliebig viele (i.A. mehr als zwei) Standardflusspfeile zu einem „virtuell“ zusammengefassten Standardflusspfeil verbunden werden, sofern alle Einzelflusspfeile
  - den gleichen Anfang (Quelle) und
  - das gleiche Ziel (Senke)

  haben. Außerdem sollten die Flusspfeile der gleichen Domäne – Elektrik/Elektronik, Mechanik oder Software – angehören. Diese Forderung ist jedoch nicht zwingend, da es manchmal sinnvoll sein kann, auf einer abstrakten Systemmodellierungsebene alle Verbindungen zwischen zwei Systembestandteilen abstrakt zusammenzufassen.
- Bei der Zusammenfassung mehrerer einzelner Standardflusspfeile $F_{Art:S:D}$ zu einem zusammengefassten Standardflusspfeil $F_{Art:SZ:D_{ges}}$ werden die Daten der einzelnen Flusspfeile, die über den Flusspfeil transportiert werden sollen, ebenfalls „virtuell“ zu einer neuen Menge $D_{ges} = \{D_1, D_2, \ldots, D_n\}$ zusammengefasst. Im darunter liegenden Datenmodell werden die einzelnen Daten von jedem virtuell zusammengefassten Flusspfeil einzeln gespeichert und übertragen.
- Um die zusammengefassten Flusspfeile in (U)CML effizient nutzen zu können, gilt die gleiche Aussage wie für die zusammengefassten Paketschnittstellen bzw. die zusammengefassten Komponentenanschlüsse – eine Werkzeugunterstützung ist für die Modellierung notwendig.

Nachdem die Eigenschaften des zusammengefassten Standardflusspfeils in den beiden Definitionen 3.43 und 3.44 eingeführt wurden, folgt in der nachfolgenden Definition die Darstellung eines zusammengefassten Standardflusspfeils im Flussdiagramm.

Definition 3.45 ***Zusammengefasster Standardflusspfeil (graphische Darstellung)***
Ein ***zusammengefasster Standardflusspfeil*** *verbindet genau einen* ***zusammengefassten Standardausgangsstecker*** *mit genau einer* ***zusammengefassten Standard Eingangsbuchse****. Zusammengefasste Standardflusspfeile werden ebenfalls für die Verbindung von zusammengefassten Steckern bzw. Buchsen mit einer* ***zusammengefassten Standardpaketschnittstelle*** *verwendet.*

Beispiel 60: Zusammengefasster Standardflusspfeil
In der Abbildung 3.115 auf Seite 189 war ein Teilsystem abgebildet in dem zwischen der Komponente *K1* und dem Paket *P1* zwei Standardflusspfeile modelliert waren. Diese beiden Verbindungen wurden in der nachfolgenden Abbildung 3.116 zu einem *zusammengefassten Standardflusspfeil* zusammengefasst. Der Flusspfeil $F1$ hingegen kann nicht zusammengefasst werden.

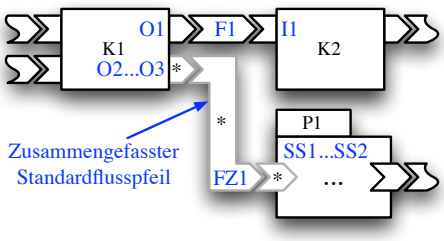


Abbildung 3.116.: Anwendung: Zusammengefasster Standardflusspfeil.

Durch die virtuelle Zusammenfassung der beiden einzelnen Standardflusspfeile ($F2$ und $F3$) bzw. der Kombination aus Ausgangsbuchsen und Paketschnittstellen, im vorangegangenen Beispiel zu einem *zusammengefassten Standardflusspfeil FZ1* wurden die statischen und dynamischen Schnittstellen des Systems nicht verändert, da die einzelnen Standardflusspfeile im Modell als einzelne Flusspfeile vorhanden sind und die Zusammenfassung der Einzelpfeile nur in der graphischen Repräsentation stattfindet ohne das darunter liegende Datenmodell zu verändern. Formal lässt sich der zusammengefasste Flusspfeil *FZ1* schreiben als

$$K1_{Z:St_{Std.Zus}:O2...O3} \xrightarrow{F_{Z:SZ:D_{ges}}} P1_{Z:SS_{Std.Zus}:SS1...SS2}$$

bzw. in der vereinfachten Schreibweise

$$K1 \xrightarrow{F_{Z:SZ_{FZ1}:D_{ges}}} P1$$

wenn alle Bedingungen für die vereinfachte Schreibweise gegeben sind und $D_{ges} = \{D2, D3\}$ ist. □

3.6.2.5.2. Kommunikationsflusspfeil

Um eine direkte Kommunikation, z.B. einen *Funktions-* bzw. *Methodenaufruf* oder einen *Regelkreis* zwischen zwei Paketen oder Komponenten zu modellieren, gibt es in (U)CML den so genannten *Standardkommunikationsflusspfeil*. Mit Hilfe des atomaren Standardkommunikationsflusspfeils wird genau eine *Paketkommunikationsschnittstelle* oder ein *Komponentenkommunikationsanschluss*[225] (Quelle) mit genau einer *Paketkommunikationsschnittstelle* oder einem *Komponentenkommunikationsanschluss* (Senke) verbunden. Dabei besteht der Standardkommunikationsflusspfeil aus einem *Hin-* und einem *Rückpfeil*, die untrennbar miteinander verbunden sind (siehe Abbildung 3.117 auf der nächsten Seite). Über den Hinpfeil des Standardkommunikationsflusspfeils wird das zu übertragende Datum von der Quelle zur Senke (Quelle $\xrightarrow{\text{Hinkanal}}$ Senke) transportiert. Das Ergebnis wird anschließend über den Rückpfeil (auch Rückkanal genannt) von der Senke zur Quelle (Quelle $\xleftarrow[\text{Rückkanal}]{}$ Senke) übertragen. Zusammen bilden sie den atomaren *Standardkommunikationsflusspfeil*, der eine Verbindung von der Quelle zur Senke und umgekehrt herstellt (Quelle $\underset{\text{Rückkanal}}{\overset{\text{Hinkanal}}{\rightleftharpoons}}$ Senke).

In den nachfolgenden beiden Definition wird zuerst der Standardkommunikationsflusspfeil definiert (Def. 3.46) und in der zweiten Definition die graphische Darstellung im Flussdiagramm eingeführt (Def. 3.47).

Definition 3.46 ***Standardkommunikationsflusspfeil***
Sei $A, B \in \{Paket, Komponente\}$ und D[226] das über den ***Standardkommunikationsflusspfeil*** *F_K von der Quelle A zur Senke B und umgekehrt zu übertragende Datum. Dann lässt sich ein Standardkommunikationsflusspfeil zwischen A und B formal schreiben als:*

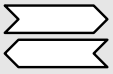
$$A_{Art:Steckerart:Name:(Cond)} \overset{F_{Art:K:D}}{\rightleftharpoons} B_{Art:Buchsenart:Name:(Cond)}$$

bzw. nach Definition 3.40 vereinfacht

$$A \overset{F_{Art:K_{Name}:D}}{\rightleftharpoons} B$$

mit $Steckerart = \{St_{Kom}, SS_{Kom}\}, Buchsenart = \{Bu_{Kom}, SS_{Kom}\}$.

Definition 3.47 ***Standardkommunikationsflusspfeil (graphische Darstellung)***
Ein ***Standardkommunikationsflusspfeil*** *verbindet genau einen* ***Kommunikationsausgangsstecker*** *mit genau einer* ***Kommunikationseingangsbuchse****. Kommunikationsflusspfeile werden ebenfalls für die Verbindung von* ***Kommunikationssteckern/-Buchsen*** *mit einer* ***Paketkommunikationsschnittstelle*** *verwendet.*



Beispiel 61: Standardkommunikationsflusspfeil
Die Abbildung 3.117 auf der nächsten Seite zeigt ein Teilsystem, bestehend aus zwei Komponenten *K1* und *K2* und einem Paket *P1*. Zwischen den beiden Komponenten *K1* und *K2* ist ein *Standardkommunikationsflusspfeil* $F1$ dargestellt. Genauer: Der Standardkommunikationsflusspfeil $F1$ verbindet genau den *Komponentenkommunikationsausgangsstecker IO1* der Komponente *K1* mit der *Komponentenkommunikationseingangsbuchse IO1* der Komponente *K2*

$$K1_{Z:St_{Kom}:IO1} \overset{F_{Z:K:D_1}}{\rightleftharpoons} K2_{Z:Bu_{Kom}:IO1}$$

bzw. in der vereinfachten Schreibweise

$$K1 \overset{F_{K_{F1}:D_1}}{\rightleftharpoons} K2.$$

Die Komponente *K1* hat außerdem zwei Standardkommunikationsflusspfeile ($F2$ und $F3$), die zu Paket *P1* gehen

$$K1 \overset{F_{K_{F2}:D_2}}{\rightleftharpoons} P1 \text{ und } K1 \overset{F_{K_{F3}:D_3}}{\rightleftharpoons} P1.$$

Mit Hilfe der Standardkommunikationsflusspfeile $F1, F2$ und $F3$ kann die Komponente *K1* das Datum D_1 mit der Komponente *K2* und das Datum D_2 und D_3 mit dem Paket *P1* austauschen. Durch die Modellierung der Verbindungen als Standardkommunikationsflusspfeile wird deutlich, dass zwischen der Komponente *K1* und der Komponente *K2* bzw. dem Paket *P1* eine enge Kopplung

[225] Siehe hierzu: Kapitel „3.6.2.4.2 *Paketkommunikationsschnittstellen und Komponentenkommunikationsanschlüsse*“ ab Seite 179.

[226] Bei einem Kommunikationsflusspfeil wird dabei stets zwischen dem von A nach B und dem von B nach A zu übertragenden Datum, also $A \xrightarrow{D_{\text{Hinkanal}}} B$ und $A \xleftarrow[D_{\text{Rückkanal}}]{} B$ unterschieden. Aus Vereinfachungsgründen wird jedoch stets von D gesprochen.

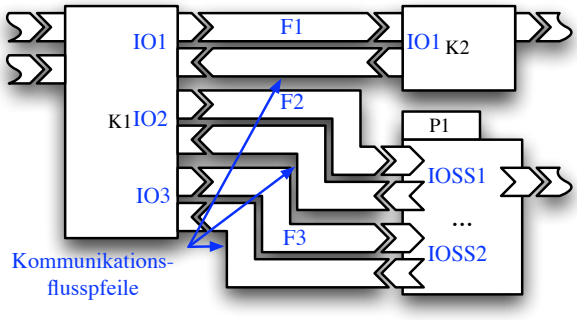


Abbildung 3.117.: Anwendung: Standardkommunikationsflusspfeil.

besteht. Die enge Kopplung könnte z.B. ein Funktions- oder Methodenaufruf, ein Protokoll oder ein Regelkreis sein, für die stets gilt: Dem Senden eines Datums von der Quelle zur Senke (Hinpfeil des Kommunikationsflusspfeils) folgt stets eine Antwort der Senke, die auf dem Rückkanal des Kommunikationsflusspfeils von der Senke zur Quelle übermittelt wird. □

Zusammengefasster Kommunikationsflusspfeil
Sind zwischen zwei Paketen oder Komponenten mehrere (i.A. mehr als zwei) Standardkommunikationsflusspfeile vorhanden, so können diese zu einem virtuell *zusammengefassten Kommunikationsflusspfeil* vereinigt werden. Für die Zusammenfassung gelten die gleichen Bedingungen und Regeln wie für zusammengefasste Standardflusspfeile.

In den nachfolgenden beiden Definitionen wird zunächst die formale Notation der zusammengefassten Kommuniktionsflusspfeile definiert (Def. 3.48) und anschließend ihre graphische Repräsentation im Flussdiagramm (Def. 3.49).

Definition 3.48 ***Zusammengefasster Kommunikationsflusspfeil***
*Sei $A, B \in \{Paket, Komponente\}$ und D_{ges} das über den **zusammengefassten Kommunikationsflusspfeil** F_{KZ} von der Quelle A zur Senke B und umgekehrt zu übertragende Datum. Dann lässt sich ein zusammengefasster Kommunikationsflusspfeil zwischen A und B formal schreiben als:*

$$A_{Art:Steckerart:Name:(Cond)} \overset{F_{Art:KZ:D_{ges}}}{\rightleftharpoons} B_{Art:Buchsenart:Name:(Cond)}$$

bzw. nach Definition 3.40 vereinfacht

$$A \overset{F_{Art:KZ_{Name}:D_{ges}}}{\rightleftharpoons} B$$

mit $Steckerart = \{St_{Kom.Zus}, SS_{Kom.Zus}\}$, $Buchsenart = \{Bu_{Kom.Zus}, SS_{Kom.Zus}\}$. D_{ges} ist die Menge aller Daten der einzelnen Kommunikationsflusspfeile $F_{Art:K:D_i}$, also $D_{ges} = \{D_1, D_2, \ldots, D_{Anz}\}$.

In der Definition 3.49 ist die graphische Darstellung eines zusammengefassten Kommunikationsflusspfeils im Flussdiagramm definiert.

Definition 3.49 ***Zusammengefasster Kommunikationsflusspfeil (graphische Darstellung)***
*Ein **zusammengefasster Kommunikationsflusspfeil** verbindet genau einen **zusammengefassten Kommunikationsausgangsstecker** mit genau einer **zusammengefassten Kommunikationseingangsbuchse**. Zusammengefasste Kommunikationsflusspfeile werden ebenfalls für die Verbindung von Steckern bzw. Buchsen mit einer **zusammengefassten Kommunikationspaketschnittstelle** verwendet.*

Beispiel 62: Zusammengefasster Kommunikationsflusspfeil
In der Abbildung 3.118 wird das gleiche Teilsystem noch einmal aufgegriffen, das im vorangegangenen Beispiel eingeführt wurde (Abbildung 3.117), mit dem Unterschied, dass die beiden Kommunikationsflusspfeile $F2$ und $F3$ zu einem *zusammengefassten Kommunikationsflusspfeil* $FZ1$ zusammengefasst wurden, also

$$K1_{Z:St_{Kom.Zus}:IO2\ldots IO3} \overset{F_{Z:KZ:D_{ges}}}{\rightleftharpoons} P1_{Z:SS_{Kom.Zus}:IOSS1\ldots IOSS2}$$

bzw. in der vereinfachten Schreibweise

$$K1 \overset{F_{Z:KZ_{FZ1}:D_{ges}}}{\rightleftharpoons} P1.$$

Mit Hilfe des zusammengefassten Kommunikationsflusspfeils $FZ1$ wurden die beiden einzelnen Standardkommunikationsflusspfeile $F2$ und $F3$ virtuell ersetzt, ohne dass ihre Informationen im darunter liegenden Datenmodell verloren gehen.

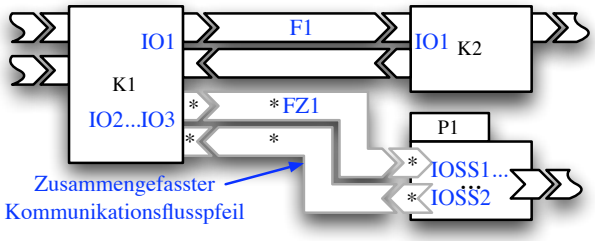


Abbildung 3.118.: Anwendung: Zusammengefasster Kommunikationsflusspfeil.

Des Weiteren wird durch die Zusammenfassung mehrerer einzelner Standardkommunikationsflusspfeile die Übersichtlichkeit des modellierten Systems verbessert, da nicht so viele Verbindungen zwischen einzelnen Pakten bzw. Komponenten bestehen. Für die Zusammenfassung von Standardkommunikationsflusspfeilen sollte jedoch der Grundsatz gelten: *Logik geht vor Übersichtlichkeit*. Ist eine Zusammenfassung einzelner Standardkommunikationsflusspfeile aus logischer Sicht nicht zu empfehlen, da dadurch die Verständlichkeit vermindert wird, sollte darauf verzichtet werden, auch wenn das System dadurch unübersichtlicher wird[227]. □

3.6.2.5.3. Externer (U)CML-Systemeingangs- und -ausgangspfeil

Das (U)CML-Systempaket bildet die Grenze zwischen der *Systemumwelt* auf der einen Seite, und dem *System* und dessen innerem Aufbau und Struktur auf der anderen. Um trotz der Kapselung des Systems gegenüber der Umwelt mit dieser Daten auszutauschen, werden in (U)CML die so genannten *externen Systemeingangs-* bzw. *-ausgangspfeile* verwendet. Mit ihrer Hilfe wird eine dezidierte Punkt-zu-Punkt Verbindung zwischen der Systemumwelt und dem System hergestellt, über die das System mit der Umwelt und diese mit dem System kommunizieren kann.

Elektrische/elektronische Signale, mechanische Größen oder Softwaredaten, die von der Umwelt kommen, werden durch *externe Systemeingangspfeile* an das System übermittelt. Ein *externer Systemeingangspfeil* wird genau mit einer *Paketschnittstelle* des Systempakets verbunden. In dem Beschreibungsfeld des externen Systemeingangspfeils wird spezifiziert, welche Signale oder Daten von der Umwelt an das System übermittelt werden sollen. Über den *externen Systemausgangspfeil* werden Daten und Signale des Systems an seine Umwelt übertragen. Auch in diesem Fall werden die zu übertragenden Daten und Signale mit Hilfe der Beschreibungsfelder beschrieben. Außerdem gilt: In (U)CML dürfen externe Systemeingangs- bzw. -ausgangspfeile nur an die Schnittstellen des Systempakets angeschlossen werden.

In der nachfolgenden Definition 3.50 ist ein *externer Systemeingangs-* bzw. *-ausgangspfeil* formal beschrieben. In der Definition 3.51 wird die graphische Repräsentation im Flussdiagramm spezifiziert.

Definition 3.50 ***Externer Systemeingangs- und -ausgangspfeil***
*Sei S das **System** und U die **Umwelt** des Systems. Des Weiteren sei D das zu übertragende Datum. Dann gilt:*

Externer Systemeingangspfeil:

$$U_{Art:EXT\_Flusspfeilart:Name:(Cond)} \xrightarrow{F_{Art:U_{IN}:D}} S_{Art:EXT\_Steckerart:Name:(Cond)}$$

Externer Systemausgangspfeil:

$$U_{Art:EXT\_Flusspfeilart:Name:(Cond)} \xleftarrow{F_{Art:U_{OUT}:D}} S_{Art:EXT\_Steckerart:Name:(Cond)}$$

mit

EXT_Flusspfeilart	*Externer Standardflusspfeil, externer Kommunikationsflusspfeil,*
	externer zusammengefasster Standard und Kommunikationsflusspfeil (ES, EK, ESZ, EKZ)
EXT_Steckerart	*Standardschnittstelle, Kommunikationsschnittstelle (SS_{Std}, SS_{Kom})*
	Zusammengefasste Standard- und Kommunikationsschnittstelle ($SS_{Std.Zus}, SS_{Kom.Zus}$)

Definition 3.51 ***Externer Systemeingangs- und -ausgangspfeil (graphische Darstellung)***
***Externe Systemeingangs-** und **-ausgangspfeile** dienen in (U)CML zur Verbindung der **Umwelt** mit der externen Schnittstelle des ausgezeichneten **Systempaket**.*

externer Systemeingangspfeil 

externer Systemausgangspfeil 

Beispiel 63: Externer Systemeingangs- und -ausgangspfeil
In der Abbildung 3.119 ist ein kleines System, welches aus zwei Komponenten *K1* und *K2* sowie vier *externen Systemeingangspfeilen* ($F1$ bis $F4$) und drei *externen Systemausgangspfeilen* ($F5$ bis $F7$) besteht, dargestellt.

[227] Diese Anmerkung gilt für alle zusammengefassten Verbindungen in (U)CML.

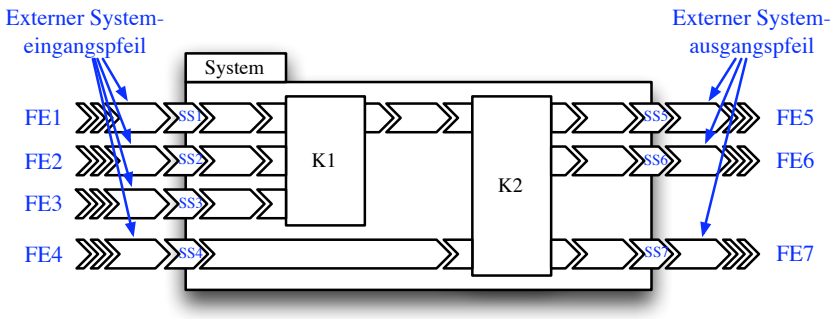


Abbildung 3.119.: Anwendung: Externer Systemeingangs- und -ausgangspfeil.

Mit Hilfe der vier externen Systemeingangspfeile (linke Seite der Abbildung 3.119) werden Daten, die von der Umwelt kommen über die Paketschnittstellen des Systempakets ($SS1 \ldots SS7$) an das System und dessen innere Struktur – Pakete und Komponenten – weitergereicht. Im Beispiel übergibt der externe Systemeingangspfeil $FE1$ das Datum D_1 von der Umwelt an die Systempaketschnittstelle $SS1$, also

$$U_{Z:ES:FE1} \xrightarrow{F_{Z:U_{IN}:D_1}} System_{Z:SS_{Std}:SS1}$$

bzw. in der verkürzten Schreibweise

$$U \xrightarrow{F_{Z:U_{IN}:FE1:D_1}} System.$$

Auf der rechten Seite der Abbildung sind drei externe Systemausgangspfeile ($FE5 \ldots FE7$) modelliert, über die das System Daten an die Umwelt übergibt. Der externe Systemausgangspfeil $FE5$, der das Datum $D5$ vom System an die Umwelt übergibt, wird in der ausführlichen Notation folgendermaßen dargestellt:

$$U_{Z:ES:FE5} \xleftarrow{F_{Z:U_{OUT}:D_5}} System_{Z:SS_{Std}:SS5}$$

Die sieben externen Systemeingangs- und -ausgangspfeile bilden zusammen die externe Paketschnittstelle zwischen der Systemumwelt auf der einen Seite und dem System auf der anderen. Eine Kommunikation zwischen dem System und der Umwelt erfolgt ausschließlich über die explizit modellierten externen Systemeingangs- und -ausgangspfeile. □

Zusammengefasster externer Systemeingangs- und -ausgangspfeil

Zusammengefasste externe Systemeingangs- und *-ausgangspfeile* werden in (U)CML verwendet, wenn zwischen dem System und der Umwelt viele (i.A. mehr als zwei) *externe Systemeingangs-* oder *-ausgangspfeile* vorhanden sind. Dabei gilt insbesondere eine Zusammenfassung unterschiedlicher externer Systemeingangs- oder -ausgangspfeile ist nur dann sinnvoll, wenn durch die Zusammenfassung die Logik bzw. das Verständnis des Systems nicht beeinträchtigt wird.

Definition 3.52 ***Zusammengefasster externer Systemeingangs- und -ausgangspfeil***

Sei $Anz_{IN}, Anz_{OUT}, i, j \in \mathbb{N}; 0 < i, j < \infty$ *und* $2 \leq Anz_{IN}, Anz_{OUT} < \infty$. *Des Weiteren sei* $F_{Art:U_{IN}:D_i}$ *ein* ***externer Systemeingangspfeil****, der das Datum* D_{IN_i} *von der Quelle (Umwelt) zur Senke (System) transportiert, und* $F_{Art:U_{OUT}:D_j}$ *ein* ***externer Systemausgangspfeil****, der das Datum* D_{OUT_j} *von seiner Quelle (System) zur Senke (Umwelt) transportiert.*

Dann gilt: Ein ***zusammengefasster externer Systemeingangspfeil*** $F_{Art:UZ_{IN}:D_{ges_{IN}}}$ *besteht aus* Anz_{IN} *externen Systemeingangspfeilen, also*

$$F_{Art:UZ_{IN}:D_{ges_{IN}}} = \sum_{i=1}^{Anz_{IN}} F_{Art:U_{IN_i}:D_i}$$

mit Anz_{IN} *= Anzahl der* ***externen Systemeingangspfeile****, die zusammengefasst werden sollen. Ein* ***zusammengefasster externer Systemausgangspfeil*** $F_{UZ_{OUT}:D_{ges_{OUT}}}$ *besteht aus* Anz_{OUT} *externen Systemausgangspfeilen:*

$$F_{Art:UZ_{OUT}:D_{ges_{OUT}}} = \sum_{j=1}^{Anz_{OUT}} F_{Art:U_{OUT_j}:D_j}$$

mit Anz_{OUT} *= Anzahl der* ***externen Systemausgangspfeile****, die zusammengefasst werden sollen.* $D_{ges_{IN}}$ *ist die Menge aller Daten der einzelnen externen Systemeingangspfeile* $F_{Art:U_{IN_i}:D_i}$*, also* $D_{ges_{IN}} = \{D_{IN_1}, D_{IN_2}, \ldots, D_{IN_{Anz_{IN}}}\}$ *und* $D_{ges_{OUT}}$ *die Menge aller Daten der externen Systemausgangspfeile.*

Nachdem in Definition 3.52 der zusammengefasste externe Systemeingangs- und -ausgangspfeil definiert wurde, folgt in der nachfolgenden Definitionen 3.53 auf der nächsten Seite zuerst die formale Notation der zusammengefassten externen Systemeingangs- und -ausgangspfeile und in der Definition 3.54 die graphische Darstellung im Flussdiagramm.

Definition 3.53 ***Zusammengefasster externer Systemeingangs- und -ausgangspfeil***
Sei S das ***System*** *und U die* ***Umwelt*** *des Systems. Des Weiteren sei D das zu übertragende* ***Datum****. Dann gelten folgende Formeln.*
Zusammengefasster externer Systemeingangspfeil:

$$U_{Art:ESZ:Name:(Cond)} \xrightarrow{F_{Art:UZ_{IN}:D_{ges_{IN}}}} S_{Art:SS_{Std.Zus}:Name:(Cond)}$$

Zusammengefasster externer Systemausgangspfeil:

$$U_{Art:ESZ:Name:(Cond)} \xleftarrow{F_{Art:UZ_{OUT}:D_{ges_{OUT}}}} S_{Art:SS_{Std.Zus}:Name:(Cond)}$$

mit $D_{ges_{IN}}$ ist die Menge aller Daten der einzelnen externen Systemeingangspfeile und $D_{ges_{OUT}}$ die Menge aller Daten von externen Systemausgangspfeilen.

Die zusammengefassten externen Systemeingangs- und -ausgangspfeile lassen sich nach Definition 3.40 vereinfacht als

$$\begin{array}{ll} \text{Zusammengefasster externer Systemeingangspfeil} & U \xrightarrow{F_{Art:UZ_{IN}:Name:D_{ges_{IN}}}} S \\ \text{Zusammengefasster externer Systemausgangspfeil} & U \xleftarrow{F_{Art:UZ_{OUT}:Name:D_{ges_{OUT}}}} S \end{array}$$

schreiben.

Definition 3.54 ***Zusammengefasster externer Systemeingangs- und -ausgangspfeil (graphische Darstellung)***
Zusammengefasste externe Systemeingangs- und -ausgangspfeile *dienen in (U)CML zur Verbindung der* ***Umwelt*** *mit der externen zusammengefassten Schnittstelle des ausgezeichneten* ***Systempakets****.*

zusammengefasster externer Systemeingangspfeil 

zusammengefasster externer Systemausgangspfeil 

Beispiel 64: Zusammengefasster externer Systemeingangs- und -ausgangspfeil
Die Abbildung 3.120 zeigt das Beispielsystem aus der Abbildung 3.119 auf der vorherigen Seite noch einmal mit zusammengefassten externen Systemeingangs- und -ausgangspfeilen.

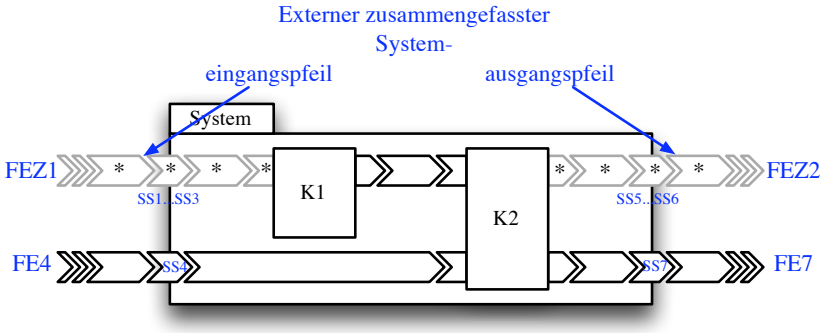


Abbildung 3.120.: Anwendung: Zusammengefasster externer Systemeingangs- und -ausgangspfeil.

Gegenüber der Abbildung 3.119 sind die externen Systemeingangspfeile *FE1* bis *FE3* zu einem externen zusammengefassten Systemeingangspfeil *FEZ1* zusammengefasst worden, also

$$U_{Z:ESZ:FEZ1} \xrightarrow{F_{Z:UZ_{IN}:D_{ges_{IN}}}} System_{Z:SS_{Std.Zus}:SS1...SS3}$$

mit $D_{ges_{IN}} = \{D_{IN_1}, D_{IN_2}, D_{IN_3}\}$. Der externe Systemeingangspfeil *FE4* kann nicht mit den anderen zusammengefasst werden, da er eine andere Senke hat.

Für die externen Systemausgangspfeile gelten die gleichen Regeln wie für die zusammengefassten externen Systemeingangspfeile. Im Beispiel wurden außerdem die beiden externen Systemausgspfeile *FE5* und *FE6* (aus Abb. 3.119) zu einem zusammengefassten externen Systemausgangspfeil

$$U \xrightarrow{F_{Z:UZ_{OUT}:FEZ2:D_{ges_{OUT}}}} System$$

zusammengefasst. Der externe Systemausgangspfeil *FE7* wurde nicht mit den anderen zusammengefasst, da er z.B. eine andere Domäne repräsentiert. □

3.6.2.5.4. Externer Systemkommunikationpfeil

Um eine direkte Kommunikation zwischen der Umwelt auf der einen Seite und dem System auf der anderen zu modellieren, gibt es in (U)CML den so genannten *externen Systemkommunikationspfeil*. Mit Hilfe des atomaren externen Systemkommunikationspfeils wird die Umwelt mit der *externen Systemkommunikationsschnittstelle* des Systempakets verbunden. Dabei besteht der externe Systemkommunikationspfeil aus einem *Hin-* und einem *Rückpfeil*, beide sind untrennbar miteinander verbunden. Über den Hinpfeil des externen Systemkommunikationspfeils wird das zu übertragende Datum von der Umwelt in das System (Umwelt $\xrightarrow{\text{Hinkanal}}$ System) transportiert. Das Ergebnis wird anschließend über den Rückpfeil (auch Rückkanal genannt) vom System in die Umwelt (Umwelt $\xleftarrow[\text{Rückkanal}]{}$ System) übertragen. Zusammen bilden sie den atomaren *externen Systemkommunikationspfeil*, der eine Kommunikationsverbindung von der Umwelt in das System und umgekehrt herstellt (Umwelt $\underset{\text{Rückkanal}}{\overset{\text{Hinkanal}}{\rightleftharpoons}}$ System).

In den beiden nachfolgenden Definitionen wird zunächst der externe Kommunikationspfeil formal beschrieben (Def. 3.55) und in der zweiten Definition die graphische Darstellung im Flussdiagramm eingeführt (Def. 3.56).

Definition 3.55 ***Externer Systemkommunikationspfeil***
*Sei S das **System** und U die **Umwelt** des Systems. Des Weiteren sei D das zu übertragende Datum. Dann lässt sich ein **externer Systemkommunikationspfeil** schreiben als:*

$$U_{Art:EK:Name:(Cond)} \overset{F_{Art:U_{IO}:D}}{\rightleftharpoons} S_{Art:SS_{Kom}:Name:(Cond)}$$

bzw. nach Definition 3.40 vereinfacht

$$U \overset{F_{Art:U_{IO}:Name:D}}{\rightleftharpoons} S.$$

Definition 3.56 ***Externer Systemkommunikationspfeil (graphische Darstellung)***
*Ein **externer Systemkommunikationspfeil** dient zur Verbindung der Umwelt mit genau einer Kommunikationsschnittstelle des ausgezeichneten Systempakets.*

links:  *rechts:* 

Beispiel 65: Externer Systemkommunikationspfeil
In der Abbildung 3.121 ist ein System, das aus dem Systempaket *System* und zwei Komponenten *K1* und *K2* besteht, dargestellt. Des Weiteren hat das Systempaket insgesamt vier *externe Systemkommunikationspfeile*: drei auf der linken (*FEK1* bis *FEK3*) und einen auf der rechten Seite (*FEK4*) des Systempakets.

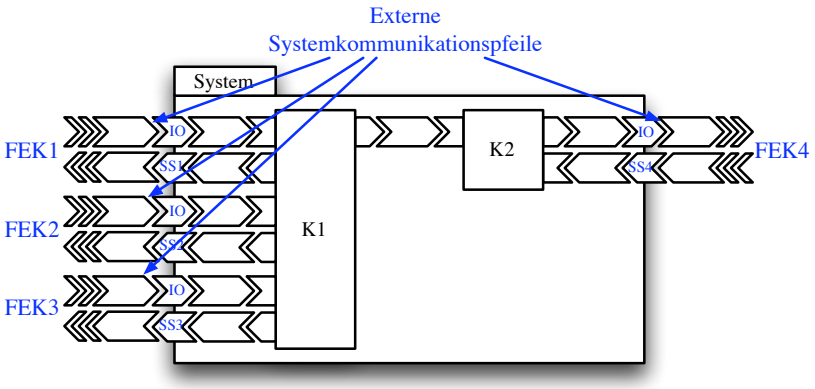


Abbildung 3.121.: Anwendung: Externer Systemkommunikationspfeil.

Mit Hilfe der vier externen Systemkommunikationspfeile kann das System Daten mit der Umwelt austauschen. Die Umwelt U sendet auf dem Hinkanal des externen Systemkommunikationspfeils *FEK1* das Datum D_{IN_1} an das System bzw. die Paketkommunikationsschnittstelle des Systempakets ($U \xrightarrow{F_{Z:U_{IO}:FEK1:D_{IN_1}}} System$). Anschließend wird auf dem Rückkanal das Datum D_{OUT_1} an die Umwelt zurückgegeben ($U \xleftarrow{F_{Z:U_{IO}:FEK1:D_{OUT_1}}} System$). Dabei ist zu beachten, dass es in (U)CML unerheblich ist, ob ein externer Systemkommunikationspfeil auf der linken oder rechten Seite des Systempakets angebracht ist. □

Zusammengefasster externer Systemkommunikationspfeil
Sind zwischen der Umwelt U und dem Systempaket S mehr als zwei externe Kommunikationspfeile mit der gleichen Quelle und Senke vorhanden, so können diese mit Hilfe des *externen zusammengefassten Systemkommunikationspfeils* dargestellt werden.

Definition 3.57 ***Zusammengefasster externer Systemkommunikationspfeil***
Sei $Anz, i \in \mathbb{N}; 0 < i < \infty$ und $2 \leq Anz < \infty$. Des Weiteren sei $F_{Art:U_{IO}:D_i}$ ein ***externer Systemkommunikationspfeil****, der das Datum D_i von der Quelle (Umwelt) zur Senke (System) bzw. umgekehrt transportiert.*

Dann gilt: Ein ***zusammengefasster externer Systemkommunikationspfeil*** *$F_{Art:UZ_{IO}:D_{ges}}$ besteht aus Anz externen Systemkommunikationspfeilen, also*

$$F_{Art:UZ_{IO}:D_{ges}} = \sum_{i=1}^{Anz} F_{Art:U_{IO_i}:D_i}$$

mit $Anz =$ Anzahl der ***externen Systemkommunikationspfeile****, die zusammengefasst werden sollen, und D_{ges} gleich der Menge aller Daten der einzelnen externen Systemkommunikationspfeile $F_{Art:U_{IO}:D_i}$, also $D_{ges} = \{D_1, D_2, \ldots, D_{Anz}\}$.*

Definition 3.58 ***Zusammengefasster externer Systemkommunikationspfeil***
Sei S das ***System*** *und U die* ***Umwelt*** *des Systems. Des Weiteren sei D_{ges} das zu übertragende* ***Datum****. Dann verbindet ein* ***zusammengefasster externer Systemkommunikationspfeil*** *$F_{Art:UZ_{IO}:D_{ges}}$ die Umwelt U mit genau einer* ***zusammengefassten externen Systemkommunikationsschnittstelle*** *des Systempakets S:*

$$U_{Art:EKZ:Name:(Cond)} \overset{Art:UZ_{IO}:D_{ges}}{\rightleftharpoons} S_{Art:SS_{Kom.Zus}:Name:(Cond)}$$

mit $D_{ges} = \{D_1, D_2, \ldots, D_{Anz}\}$ und Anz gleich der Anzahl aller zusammengefassten externen Systemkommunikationspfeile.

In der Definition 3.59 ist die graphische Repräsentation des externen zusammengefassten Kommunikationspfeils im (U)CML-Flussdiagramm definiert.

Definition 3.59 ***Zusammengefasster externer Systemkommunikationspfeil (graphische Darstellung)***
Zusammengefasste externe Systemkommunikationspfeile *dienen zur Verbindung der Umwelt mit dem ausgezeichneten Systempaket.*

Beispiel 66: Zusammengefasster externer Systemkommunikationspfeil
In der Abbildung 3.122 ist das selbe System wie in der Abbildung 3.121 auf der vorherigen Seite dargestellt, nur dass die drei externen Systemkommunikationspfeile *FEK1* bis *FEK3* zu einem *zusammengefassten externen Systemkommunikationspfeil FEKZ1* zusammengefasst wurden. Der externe zusammengefasste Kommunikationspfeil *FEKZ* kann ausführlich als

$$U_{Z:EKZ:FEKZ1} \overset{Z:UZ_{IO}:D_{ges}}{\rightleftharpoons} System_{Z:SS_{Kom.Zus}:IOSS1 \ldots IOSS3}$$

geschrieben werden.

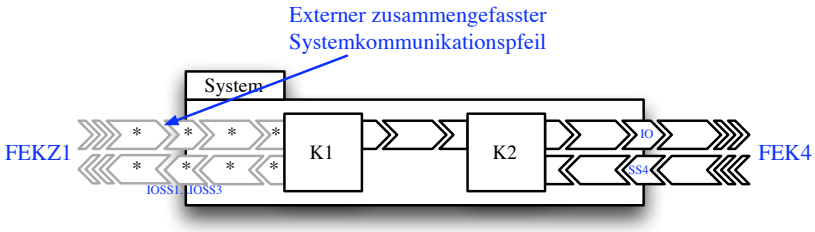


Abbildung 3.122.: Anwendung: Zusammengefasster externer Systemkommunikationspfeil.

Der externe Systemkommunikationspfeil *FEK4* (aus Abb.: 3.121) kann nicht mit den anderen externen Pfeilen (*FEK1* bis *FEK3*) zusammengefasst werden, da er eine andere Quelle (Komponente *K2*) hat. □

3.6.2.5.5. Zusammengesetzter (U)CML-Pfeil

In den letzten beiden Kapiteln wurden zuerst die Paketschnittstellen bzw. die Komponentenanschlüsse und daran anschließend die dazugehörigen Flusspfeile eingeführt. In (U)CML werden Flusspfeile jedoch nie ohne einen dazugehörigen Stecker/Buchse oder eine Paketschnittstelle modelliert, so dass aus Vereinfachungsgründen in (U)CML stets von *zusammengesetzten Pfeilen* bzw. *zusammengesetzten Verbindungen* gesprochen wird. In der Tabelle 3.10 auf Seite 188 wurden bereits alle in (U)CML zulässigen Kombinationen aus Steckern, Buchsen, Schnittstellen und Pfeilen dargestellt. Grundsätzlich besteht jede gültige Verbindung in (U)CML aus einer *Quelle*, einem *Flusspfeil* sowie einer *Senke*. Dabei ist es unerheblich, ob die Quelle eine Paketschnittstelle oder ein Ausgangsstecker einer Komponente ist. Wichtig ist nur, dass die Verbindung den in den letzten beiden Kapiteln eingeführten Grundregeln genügt.

In der nachfolgenden Definition 3.60 wird der oben erläuterte Sachverhalt in Form einer formalen Definition angegeben.

Definition 3.60 ***Zusammengesetzte Verbindung***
Jede ***zusammengesetzte Verbindung*** *in (U)CML besteht aus einer* ***Quelle****, einem* ***Flusspfeil*** *sowie einer* ***Senke****.*

$$Quelle \xrightarrow{Flusspfeil} Senke$$

Dabei gilt: Als Quelle bzw. Senke der Verbindung können alle Stecker-, Buchsen- oder Schnittstellenarten der (U)CML verwendet werden, sofern die entsprechende Verbindungsart in (U)CML wohldefiniert ist.

Nach Definition 3.60 kann eine zusammengesetzte Verbindung sowohl aus Standard-, Kommunikations- als auch zusammengefassten Verbindungselementen bestehen. In der nachfolgenden Abbildung 3.123 sind einige in (U)CML zulässige zusammengesetzte Verbindungen dargestellt.

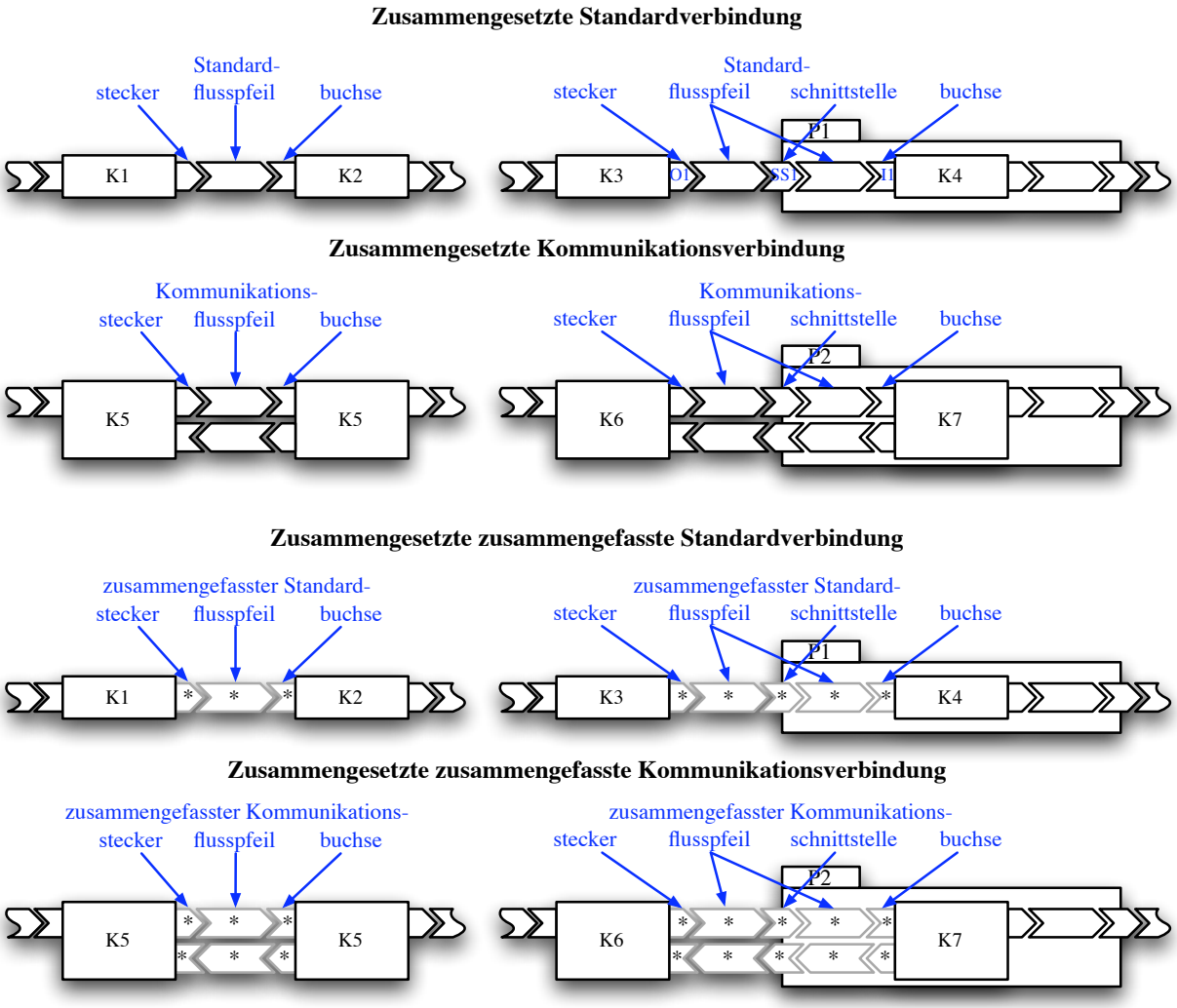


Abbildung 3.123.: Zusammengesetzte Verbindungen.

Im oberen Teil der Abbildung 3.123 sind zunächst die *zusammengesetzten Verbindungen*, die die (U)CML zur Verfügung stellt – Standardverbindung und Standardkommunikationsverbindung – abgebildet. Im unteren Teil sind die gleichen Abbildungen als *zusammengesetzte zusammengefasste Verbindungen* – zusammengefasste Standardverbindung und zusammengefasste Kommunikationsverbindung – dargestellt.

Wie in Abbildung 3.123 (rechts) ersichtlich, kann eine zusammengesetzte Verbindung auch über Paketgrenzen hinweg geführt werden. Wichtig ist nur, dass an jeder Paketgrenze eine Schnittstelle modelliert wird. Wenn diese Forderung erfüllt ist, gilt der Grundsatz Quelle $\xrightarrow{Flusspfeil}$ Senke unabhängig davon, über wie viele Paketschnittstellen die zusammengesetzte Verbindung geführt wird. Zum Beispiel kann die Verbindung zwischen den beiden Komponenten *K3* und *K4* als zusammengesetzte Verbindung $K3 \xrightarrow{Flusspfeil} K4$ bzw. ausführlich mit Einzelverbindungen $K3 \xrightarrow{Flusspfeil} P1 \xrightarrow{Flusspfeil} K4$ geschrieben werden.

Soll zusätzlich zur Beschreibung der Verbindung zweier Pakete/Komponenten der exakte Verlauf von der Quelle zur Senke durch eine zusammengesetzte Verbindung dargestellt werden, so kann die oben eingeführte Notation verwendet werden:

Verbindung des Ausgangs O1 der Komponente K3 mit der Schnittstelle SS1 des Paket P1:

$$K3_{Z:St_{Std}:O1} \xrightarrow{F_{Z:S:D}} P1_{Z:SS_{Std}:SS1}$$

Verbindung der Schnittstelle SS1 des Pakets P1 mit der Buchse I1 der Komponente K4:

$$P1_{Z:SS_{Std}:SS1} \xrightarrow{F_{Z:S:D}} K4_{Z:Bu_{Std}:I1}$$

Oder als eine zusammengesetzte Verbindung

$$K3_{Z:St_{Std}:O1} \xrightarrow{F_{Z:S:D}} P1_{Z:SS_{Std}:SS1} \xrightarrow{F_{Z:S:D}} K4_{Z:Bu_{Std}:I1}$$

geschrieben.

Anmerkungen:

- Die obige Notation kann für alle in der Abbildung 3.123 gezeigten zusammengesetzten Verbindungen gleichermaßen verwendet werden.
- BUS-Systeme können in (U)CML nicht als zusammengesetzte Verbindungen modelliert werden, da BUS-Systeme bereits nach Definition der BUS-Systeme zusammengefasste Verbindungen repräsentieren (vgl.: Kapitel „3.6.2.7 *(U)CML-BUS-System*“ ab Seite 211).

3.6.2.5.6. BUS-System-Pfeil

Die letzte Verbindungsart, die die (U)CML zur Modellierung von Signalen zur Verfügung stellt, ist das so genannte *BUS-System*. Das BUS-System bestehet in (U)CML aus verschiedenen Elementen: dem *BUS-System-Pfeil* als Träger der *BUS-System-Balken* sowie den *BUS-System-Steckern* und *-Buchsen* die am BUS-System-Balken angeschlossen sind. BUS-Systeme werden vor allem in der Elektrotechnik zur Modellierung von elektrischen/elektronischen Verbindungen zwischen einzelnen Elementen einer Schaltung verwendet. Dabei können in einem BUS-System beliebige Signale (auch mit unterschiedlichen Quellen und Senken) zu einem BUS zusammengefasst werden. BUS-Systeme können jedoch auch für die Modellierung von Softwaresignalen bzw. Methodenaufrufe verwendet werden. In der Tabelle 3.11 auf der nächsten Seite sind sämtliche in (U)CML verfügbaren BUS-Systemarten dargestellt.

Anmerkung:
Das BUS-System bzw. die hier eingeführten BUS-System-Pfeile werden im Kapitel „3.6.2.7 *(U)CML-BUS-System*“ ab Seite 211 ausführlich besprochen und werden hier nur der Vollständigkeit halber erwähnt.

3.6.2.5.7. Erweiterung des Farbschemas auf Flusspfeile

Im Abschnitt „3.6.2.4.5 *Erweiterung des Farbschemas auf Paketschnittstellen und Komponentenanschlüsse*“ ab Seite 184 wurde ein Farbschema für die erweiterte Darstellung von Steckern/Buchsen eingeführt. Das dort eingeführte Farbschema kann auch auf alle Flusspfeilarten, die die (U)CML zur Modellierung im Flussdiagramm zur Verfügung stellt, angewendet werden.

Erweiterung der Farbschemadefinition 3.10 auf Seite 167 auf Flusspfeile:

Definition 3.61 ***Farbschema von Flusspfeilen***
Flusspfeile können abhängig von ihrem Typ – keine Zuordnung, Elektrik/Elektronik, Mechanik und Software – farblich markiert dargestellt werden.

Anmerkung zur Definition 3.61:
Die Erweiterung des Farbschemas auf Flusspfeile gilt **nicht** für BUS-System-Stecker, -buchsen und -anschlüsse. Die BUS-System-Elemente werden in (U)CML stets schwarz dargestellt.

3.6.2.6. (U)CML-Beschreibungsfelder

Die Beschreibungsfelder sind die wichtigste Neuerung der (U)CML. Dies zeichnet sie gegenüber allen anderen existierenden Modellierungssprachen und -konzepten zur Beschreibung von Systemen aus. Mit Hilfe der Beschreibungsfelder können alle Bestandteile der Sprache – Pakete, Komponenten, Buchsen/Stecker etc. – mit Eigenschaften (Attributen) versehen werden, die für die Modellierung und Spezifikation eines Systems notwendig sind. Aus diesem Grund sind Beschreibungsfelder modular aufgebaut, so dass jede Domäne (*Hard-* und *Software*) mit der selben Beschreibungstechnik abgedeckt werden kann. Insbesondere sind die Beschreibungsfelder sowohl für die statische als auch die dynamische *Spezifikation* der Hard- und Softwarebestandteile eines Systems sowie die *Kompatibilitätsbeschreibung* und *Kompatibilitätsbewertung* der einzelnen Elemente notwendig.

In den Beschreibungsfeldern können nicht nur Informationen hinterlegt werden, die für die Kompatibilitätsbeschreibung notwendig sind, sondern auch Daten, die für die Spezifikation z.B. eines ICs oder die Protokollierung aller Änderungen am Modell benötig werden. So kann z.B. in einem Komponentenbeschreibungsfeld der Typ des Bauteils (z.B. IC 7400[228]) bzw. die informelle Funktionalitätsbeschreibung („Vier NAND mit je zwei Eingängen.“) hinterlegt werden, wodurch das Modell für den Menschen „lesbarer“ und „verständlicher“ wird.

[228] Siehe hierzu: Kapitel „2.2 *Modellbildung*“ ab Seite 31. bzw. die Abbildung 2.4 auf Seite 34.

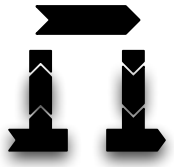
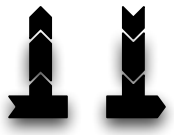
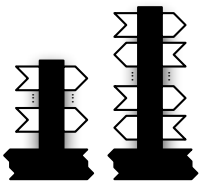
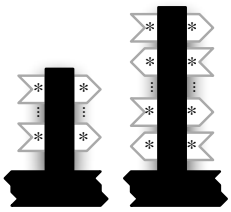
	Pfeilart	Standard	Zusammengefasst
BUS-System-Pfeile	*Unidirektional*		
	BUS-System-Pfeil		
	BUS-System-Pfeil mit BUS-System-Anschluss für Komponenten (Black-Box)		
	BUS-System-Pfeil mit BUS-System-Anschluss für Pakete (Black-Box)		
	BUS-System-Pfeil mit Balken und Anschlüssen (Glass-Box)		
	Bidirektional		
	BUS-System-Pfeil		
	BUS-System-Pfeil mit BUS-System-Anschluss für Komponenten (Black-Box)		
	BUS-System-Pfeil mit BUS-System-Anschluss für Pakete (Black-Box)		
	BUS-System-Pfeil mit Balken und Anschlüssen (Glass-Box)		

Tabelle 3.11.: BUS-System Pfeilarten.

Die Beschreibung der (U)CML-Beschreibungsfelder unterteilt sich vier Abschnitte:

- Grundsätzlicher Aufbau und Struktur der (U)CML-Beschreibungsfelder.
- Beschreibungsfelder und Diagrammarten.
- Wiederkehrende Eigenschaftsfelder der (U)CML-Beschreibungsfelder.
- Beschreibungsfelder der wichtigsten (U)CML-Elemente.

Begonnen wird die Erläuterung der (U)CML-Beschreibungsfelder mit der Beschreibung des grundsätzlichen Aufbaus sowie der Struktur der Beschreibungsfelder[229].

3.6.2.6.1. Grundsätzlicher Aufbau und Struktur der (U)CML-Beschreibungsfelder

Alle Beschreibungsfelder der (U)CML sind modular aufgebaut und haben die gleiche Grundstruktur – unabhängig davon, ob sie für die Beschreibung von Hard- oder Software eingesetzt werden. Um den modularen Charakter der Beschreibungsfelder zu

[229] Anmerkung:
Eine ausführliche Beschreibung aller (U)CML-Beschreibungsfelder finden Sie unter [BK05]. In den folgenden Unterpunkten sind lediglich die wichtigsten Eigenschaften einiger ausgewählter Beschreibungsfelder kurz aufgelistet.

unterstreichen, bestehen alle Beschreibungsfelder aus zwei großen Bereichen: einem *domänenübergreifenden* und einen *domänenspezifischen Teil*. Im domänenübergreifenden Teil sind alle allgemein gültigen Eigenschaften wie z.B. der informelle Name, spezifiziert, während im domänenabhängigen Abschnitt sämtliche spezifischen Eigenschaften, die nur für eine Domäne gelten, hinterlegt sind. Zusätzlich zur domänenspezifischen Unterteilung der Beschreibungsfelder sind auf der linken Seite jedes Beschreibungsfelds dessen fest vorgegebene *Eigenschaftsfelder*[230] aufgelistet, während auf der rechten Seite die *Beschreibung* bzw. die *Spezifikation der Eigenschaften* vom Benutzer eingetragen wird. Die Beschreibung der einzelnen Eigenschaften kann dabei sowohl *qualitativ* als auch *quantitativ* erfolgen, abhängig von den zugrunde liegenden Daten. In der Abbildung 3.124 ist der grundsätzliche Aufbau und die Struktur aller (U)CML-Beschreibungsfelder abgebildet.

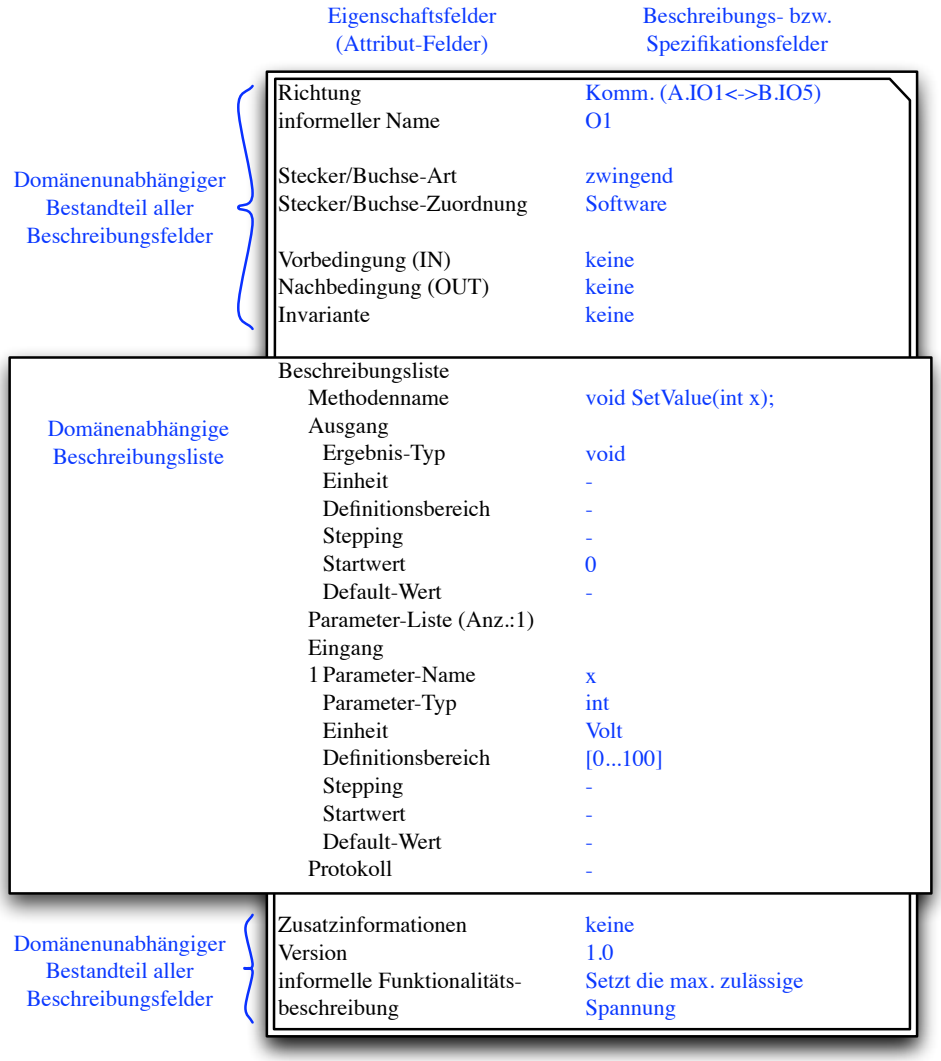


Abbildung 3.124.: Grundsätzlicher Aufbau und Struktur der (U)CML-Beschreibungsfelder.

Die Abbildung 3.124 illustriert den modularen Aufbau sowie die Struktur eines Kommunikationssteckerbeschreibungsfelds für einen Softwarekommunikationsstecker, der den Kommunikationsstecker *IO1* der Komponente *A* mit der Kommunikationsbuchse *IO5* der Komponente *B* verbindet. Im Beschreibungsfeld sind einige der Eigenschaftsfelder mit Werten, einer quantitativen Beschreibung (Definitionsbereich $[0 \ldots 100]$) versehen, andere mit informellen Beschreibungen (*informelle Funktionalitätsbeschreibung: Setzt die max. zulässige Spannung*). Um die Domänenunabhängigkeit der Beschreibungsfelder zu gewährleisten, sind einige Bestandteile der Beschreibungsfelder auf die entsprechenden Domänen angepasst. Im Beispiel ist die *Beschreibungsliste* für die Modellierung eines Softwarekommunikationssteckers dargestellt. Für die Beschreibung beispielsweise eines elektrischen/elektronischen Kommunikationssteckers wird das selbe Beschreibungsfeld wie für die Beschreibung eines Softwarekommunikationssteckers verwendet. Die beiden Beschreibungsfelder unterscheiden sich nur im Feld *Beschreibungsliste*. Das Feld Beschreibungsliste ist für die unterschiedlichen Domänen unterschiedlich strukturiert.

Anmerkung:
Alle mit einem (*) gekennzeichneten Eigenschaftsfelder bzw. die dazugehörigen Spezifikationsfelder werden durch das Werkzeug *(U)CML-ed* automatisch erzeugt und verwaltet. Siehe hierzu: „3.8 *(U)CML-ed, ein graphischer Editor für die Kompatibilitätsmodellierungssprache (U)CML*“ ab Seite 254.

[230] Die Eigenschaftsfelder werden oft auch als Attributfelder bezeichnet.

3.6.2.6.2. Beschreibungsfelder und Diagrammarten

Sämtliche Beschreibungsfelder werden in allen (U)CML-Diagrammarten – Fluss-, Graphen-, Matrix- und Baumdiagramm – gleichermaßen verwendet. Dabei kann es jedoch vorkommen, dass abhängig von der Diagrammart nicht alle Informationen dargestellt werden können. So kann z.B. im Matrixdiagramm keine Information über die Buchsen/Stecker, Schnittstellen und die Verbindungspfeile ausgegeben werden, weil diese nicht explizit im Matrixdiagramm dargestellt werden können. Die meisten Informationen sind im Flussdiagramm hinterlegt, wodurch die Wichtigkeit des Flussdiagramms noch einmal unterstrichen wird.

3.6.2.6.3. Wiederkehrende Eigenschaftsfelder der (U)CML-Beschreibungsfelder

In diesem Abschnitt werden einige der in vielen Beschreibungsfeldern stets wiederkehrenden Eigenschaftsfelder erläutert, um eine spätere Wiederholung zu vermeiden. Begonnen wird mit der Erläuterung der Vor- und Nachbedingungen sowie den Invarianten.

- **Vorbedingung**
  Unter einer Vorbedingung wird eine Bedingung verstanden, die *zwingend* gelten muss, damit ein Datum von einer Komponente korrekt verarbeitet werden kann [PDB98b, 174ff]. Die logische Verknüpfung aller Vorbedingungen aller Stecker- und Buchsen eines Pakets/Komponente werden in das Paket-/Komponentenbeschreibungsfeld eingetragen. In (U)CML wird an der Eingangsbuchse eines Pakets/Komponente geprüft, ob das gesendete Datum des korrespondierenden Ausgangssteckers eines Pakets/Komponente der Vorbedingung entspricht. Ist die Vorbedingung nicht erfüllt, wird ein Fehler ausgegeben.

  Vorbedingungen, Nachbedingungen und Invarianten werden in (U)CML in CCL – Compatibility Constraint Language – geschrieben. Die CCL [Dan09] wurde im Rahmen eines interdiziplinären Projektpraktikums am Lehrstuhl für Raumfahrttechnik (TUM) entwickelt. Die CCL basiert auf OCL [WK06][Cor06][OMG07b] – Object Constraint Language – die von UML/UML2 her bekannt ist. Sie wurde jedoch an die speziellen Anforderungen der (U)CML angepasst. Eine ausführliche Beschreibung der CCL finden Sie unter [Dan09].

  In den Beschreibungsfeldern bzw. der formalen Notation werden die Vor- und Nachbedingungen sowie die Invarianten in vereinfachter Form eingetragen. Zum Beispiel wird die Vorbedingung $I1 > 10$ des zwingenden Standardeingangsbuchse *I1* einer Komponente *A* formal durch

  $$A_{Z:Bu_{Std}:I1:(pre:I1>10)}$$

  oder als CCL Ausdruck

```
1 context A
2 pre {
3   plug("I1").value>10;
4 }
```

  geschrieben. Die gleiche Schreibweise wird auch für die Nachbedingungen bzw. Invarianten verwendet. Im (U)CML-Beschreibungsfeld der Buchse *I1* wird im Eigenschaftsfeld *Vorbedingung* die Bedingung $I1 < 10$ eingetragen.

- **Nachbedingung**
  Nachbedingungen sind alle Bedingungen, die gelten müssen, wenn eine Information von einem Paket/Komponente verarbeitet und an den Ausgangsstecker weitergegeben worden ist [PDB98b, 174ff]. Die Nachbedingung wird in der Paket-/Komponentenspezifikation festgelegt. Nachbedingungen machen nur im Zusammenspiel mit Vorbedingungen (s.o.) Sinn, da sonst keine gesicherten Aussagen über den Initiierungszustand des Pakets/Komponente vorliegen und folglich keine Aussage über die Gültigkeit der Nachbedingungen gemacht werden kann.

- **Invariante**
  Unter einer Invarianten versteht man alle Parameter/Größen eines Pakets/Komponente, die sich innerhalb dieses Pakets/Komponente nicht verändern bzw. nicht durch Bestandteile des Pakets (z.B. Komponenten) verändert werden [PDB98b, 178]. Invarianten sind in (U)CML besonders wichtig, um zu zeigen, dass eine Komponente einen Eingangswert unverändert an einen Ausgang weiterreicht.

- **Beschreibungsliste**
  Im Abschnitt Beschreibungsliste eines (U)CML-Beschreibungsfelds werden die domänenspezifischen Eigenschaftsfelder eingeblendet (vgl. Abb.: 3.124).

- **Version**
  Mit Hilfe des Eigenschaftsfelds *Version* kann eine einfache Versionierung einzelner (U)CML-Bestandteile vorgenommen werden.

- **Zusatzinformationen**
  Fast alle (U)CML-Beschreibungsfelder besitzen das Eigenschaftsfeld *Zusatzinformationen*. Dieses Feld wird vor allem für die Modellierung eines Systems nach dem SIL-Level Standard benötigt, da für eine Zertifizierung nach SIL eine ausführliche Dokumentation der einzelnen Arbeitsschritte zwingend vorgeschrieben ist. In der nachfolgenden Aufzählung werden die einzelnen Felder unter *Zusatzinformationen* einzeln aufgelistet und erläutert:

  - **SIL-Level/Zertifikat**
    Das Feld *SIL-Level/Zertifikat* enthält eine Bezeichnung nach dem SIL-Standard oder ein Zertifikat, nachdem das Element erzeugt bzw. modifiziert wurden. Beispielsweise kann als Zertifikat MISRA[231], ISO[232], SOF-STD-2 [GRe07] etc. verwendet werden.

[231] Das Akronym *MISRA* steht für „The **M**otor **I**ndustry **S**oftware **R**eliability **A**ssociation“ [MIS07][Wik07n].
[232] Das Akronym *ISO* steht für “ **I**nternationale **O**rganisation für **N**ormung“ [Web07].

- **SIL-Level Beschreibung**
  Das Feld *SIL-Level Beschreibung* dient zur Ergänzung des Feldes *SIL-Level/Zertifikat*. In diesem können zusätzliche Informationen und Daten in verbaler Form hinterlegt werden wie z.B. Ergänzungen und Kommentare, die für andere Benutzer hilfreich sein könnten.
- **Typ/ID**
  Das Feld Typ/ID repräsentiert eine Art Typenschild bzw. Produktnummer einer Baugruppe oder eines Softwaremoduls. Mit Hilfe des Typs bzw. der Identifikationsnummer (ID) kann das Auffinden einer Baugruppe in einem Produktkatalog erleichtert werden. Dieses Feld ist vor allem in der Elektrotechnik und Mechanik wichtig, um die Produktbezeichnung einer Baugruppe spezifizieren zu können.
- **Version**
  Mit Hilfe der Versionsnummer kann eine einfache Versionierung der Pakete/Komponenten etc. vorgenommen werden. Dadurch kann der Entwicklungsstand (Reifegrad) des gesamten Systems dokumentiert werden. Des Weiteren ist es möglich, verschiedene Arten (Bauzustände) einer Baugruppe in einem Modell zu verwalten bzw. zu benutzen.
- **Erstellerinformationen und Änderungsliste (*)**
  Die beiden Felder *Erstellerinformation* und *Änderungsliste* dienen zur Erzeuger- und Änderungsverfolgung aller in (U)CML modellierten Elemente. Die beiden Listen enthalten folgende Informationen:
  - *Benutzername:* Name des Benutzers bzw. *(U)CML-ed*-Identitiät.
  - *Firma:* Name und Anschrift der Firma, der der Benutzer angehört.
  - *Datum:* Datum, wann das Element erzeugt bzw. verändert.
  - *Kommentar:* Ein optionaler Kommentar zur Beschreibung der Erstellung bzw. Änderung, falls dies notwendig ist.

  In der Abbildung 3.125 ist das Erstellerinformations- und Änderungslistenbeschreibungsfeld abgebildet.

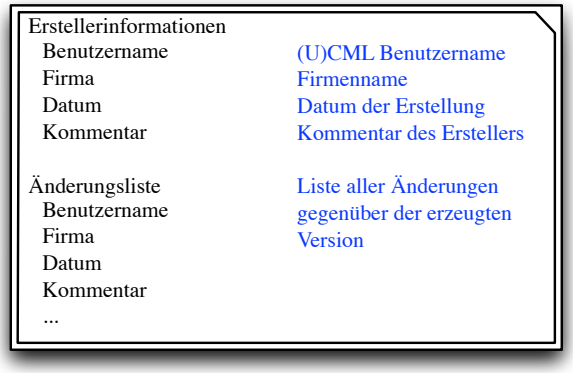


Abbildung 3.125.: Erstellerinformations- und Änderungsbeschreibungsfeld.

Die beiden Listen – Erstellerinformation und Änderungsliste – sind besonders für die Zertifizierung nach SIL notwendig. Sie können jedoch auch allgemein für die Änderungsverfolgung benutzt werden[233]. Die beiden Listen werden automatisch durch das Werkzeug *(U)CML-ed* erzeugt und verwaltet. Das einzige Feld, das vom Benutzer verändert werden kann, ist das optionale Feld *Kommentar*.

Das nachfolgende Beispiel illustriert die Verwendung von Vor- und Nachbedingungen anhand eines einfachen Beispielssystems.

Beispiel 67: Vor- und Nachbedingung einer Komponente
In der Abbildung 3.126 auf der nächsten Seite ist ein Auszug aus einem einfachen, in (U)CML modellierten System, mit zwei Komponenten und den dazugehörigen, auszugsweise dargestellten Beschreibungsfeldern abgebildet. Die beiden Beschreibungsfelder zeigen jeweils die Vor- und Nachbedingungen der Komponenten.

Auf der linken Seite der Abbildung ist die Komponente *K1* mit zwei zwingenden Eingangsbuchsen $K1_{Z:Bu_{Std}:I1:(pre:I1<10)}$ und $K1_{Z:Bu_{Std}:I2:(pre:I2<10)}$ und einem zwingenden Ausgangsstecker $K1_{Z:St_{Std}:O1:(post:O1<20)}$ dargestellt. Im dazugehörigen Beschreibungsfeld sind die jeweiligen Vor- und Nachbedingung der Komponente (der Stecker- und Buchsen der Komponenten) zusammengefasst. Dabei gilt die Vorbedingung für den Eingang der Komponente $I1 < 10 \wedge I2 < 10$ und die Nachbedingung $O1 < 20$ für den Ausgang der Komponente. Da der Ausgang der Komponente $K1_{Z:St_{Std}:O1}$ mittels eines Flusspfeils mit dem Eingang der Komponente $K2_{Z:Bu_{Std}:I1}$ verbunden ist ($K1_{Z:St_{Std}:O1:(post:O1<20)} \xrightarrow{F_{Z:S:D}} K2_{Z:Bu_{Std}:I1:(pre:I1<20)}$), muss die Nachbedingung der Komponente *K1*, genauer die Nachbedingung des Ausgangs der Komponente *K1* mit der Vorbedingung der Komponente *K2* übereinstimmen. Stimmen die beiden Bedingungen nicht überein, wird ein Fehler angezeigt. Im Beispielsystem stimmen die beiden Bedingungen überein und es wird kein Fehler ausgegeben – die beiden Komponenten sind kompatibel. □

[233] Anmerkung:
Aus arbeitsrechtlichen Gründen muss jeder Nutzer des Programms *(U)CML-ed* darüber unterrichtet werden, dass jeder Arbeitsschritt „total" überwacht wird. Stimmt der Nutzer dem nicht zu, darf das Programm nicht eingesetzt werden [HK07].

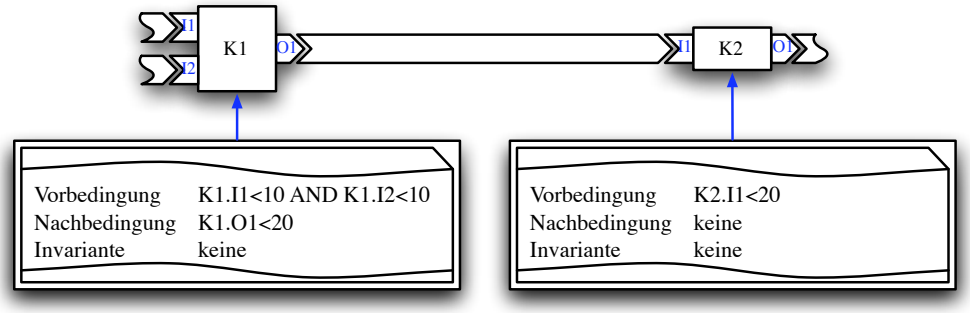


Abbildung 3.126.: Beispiel für Vor- und Nachbedingung.

3.6.2.6.4. Beschreibungsfelder der wichtigsten (U)CML-Elemente

Die nachfolgende Aufzählung enthält eine Auswahl der wichtigsten (U)CML-Beschreibungsfelder, die für die Modellierung von einfachen Systemen benötigt werden. Dabei werden für alle dargestellten Beschreibungsfelder zunächst die graphische Repräsentation und im Anschluss die einzelnen Eigenschaftsfelder erläutert. Begonnen wird mit der Erläuterung des Paket- bzw. Komponentenbeschreibungsfelds.

- **Paket- und Komponentenbeschreibungsfeld**
  Das Paket- bzw. Komponentenbeschreibungsfeld dient zur Spezifikation (Beschreibung) der Eigenschaften eines (U)CML-Pakets bzw. einer (U)CML-Komponente. Die Abbildung 3.127 zeigt auf der linken Seite die graphische Repräsentation eines Paketbeschreibungsfelds, während auf der rechten Seite der Abbildung ein Komponentenbeschreibungsfeld dargestellt ist.

Paket-Name	Name des Pakets
Vorbedingung (IN)	Sammlung aller Vor- und Nachbedingungen aller
Nachbedingung (OUT)	im Paket enthaltenen Buchsen und Stecker
Invariante	
Eingänge	Anzahl und von woher
Ausgänge	Anzahl und wohin
Komponenten	Anzahl und Namen aller Komponenten
Pakete	Anzahl und Namen aller direkten Subpakete
Paketzuordnung	In welchem Paket ist das Paket enthalten
Zusatzinformationen	Daten, die für die Modellierung benötig werden
SIL-Level/Zertifikat	Zertifikate
SIL-Level-Beschreibung	erweiterte Beschreibung
Typ/ID	z.B. IC 7400
Version	Versionsnummer des Pakets
Erzeugerinformationen	Wer hat das Paket erzeugt ...
Änderungsliste	Wer hat das Paket verändert ...
informelle Funktionalitäts-beschreibung	z.B. Logikboard
externe Daten	Verweis auf ein externes Dokument

Komponenten-Name	Name der Komponente
Komponenten-Zuordnung	Elektronik/Elektrik, Mechanik oder Software
Vorbedingung (IN)	Sammlung aller Vor- und Nachbedingungen, die
Nachbedingung (OUT)	für die gesamte Komponente gelten müssen, aus
Invariante	den Steckern und Buchsen
Eingänge	Anzahl und von woher
Ausgänge	Anzahl und wohin
Paketzuordnung	In welchem Paket ist die Komponente enthalten
Zusatzinformationen	Daten, die für die Modellierung benötig werden
SIL-Level/Zertifikat	Zertifikate
SIL-Level-Beschreibung	erweiterte Beschreibung
Typ/ID	z.B. IC 7400
Version	Versionsnummer der Komponente
Erzeugerinformationen	Wer hat die Komponente erzeugt ...
Änderungsliste	Wer hat die Komponente verändert ...
informelle Funktionalitäts-beschreibung	z.B. logische Schaltung (4 NAND)
Funktionalität	z.B. Tabelle oder MSC
externe Daten	Verweis auf ein externes Dokument

Abbildung 3.127.: Paketbeschreibungsfeld (links), Komponentenbeschreibungsfeld (rechts)

Beschreibung der Eigenschaftsfelder des Paket- bzw. Komponentenbeschreibungsfelds:

- *Paket- oder Komponenten-Name*
  Im Feld *Paket-Name* bzw. *Komponenten-Name* wird der eineindeutige Name des Pakets bzw. der Komponente eingetragen. Bei der Vergabe eines Paket- bzw. Komponentennamens sollte auf eine möglichst aussagekräftige Benennung Wert gelegt werden, damit die „Funktionalität“ des Paket bzw. der Komponente bereits am Namen erkannt werden kann.

- *Komponenten-Zuordnung (Komponente)*
  Mit Hilfe der *Komponentenzuordnung* kann eine Komponente einer bestimmten Domäne zugeordnet werden. Zur Auswahl stehen im Moment Elektrik/Elektronik, Mechanik sowie Software.

- *Ein- und Ausgänge (*)*
  Die beiden Felder *Eingänge* bzw. *Ausgänge* dienen zur verbalen Beschreibung sowohl der Anzahl, als auch der Herkunft bzw. des Ziels eines Ein- bzw. eines Ausgangs ($Quelle \rightarrow Senke$). An dieser Stelle wird der gesamte Pfad aller am Paket bzw. der Komponente angeschlossenen Ein- und Ausgänge angegeben, über die eine Verbindung zu einem anderen Paket bzw. Komponente besteht.

- *Pakete und Komponenten (Paket) (*)*
  In den beiden Eigenschaftsfeldern *Pakete* bzw. *Komponenten* werden Anzahl sowie Namen aller Pakete bzw. Komponenten aufgelistet, die sich direkt innerhalb des aktuellen Pakets befinden.
- *Paketzuordnung (*)*
  Im Feld *Paketzuordnung* steht der Name des Vaterpakets des gerade aktuellen Pakets bzw. Komponente. Die Paketzuordnung ist komplementär zu den Feldern Pakete und Komponenten.
- *Funktionalität (Komponente)*
  Mit Hilfe des Felds *Funktionalität* kann die Funktionalität bzw. das Verhalten einer Komponente spezifiziert werden. Nähere Informationen zur Funktionalität von Komponenten finden Sie im Kapitel „3.6.2.3.2 *Verhalten einer (U)CML-Komponente*“ ab Seite 170.
- *Informelle Funktionalitätsbeschreibung*
  In diesem Feld kann der Anwender die Funktionalität des Pakets bzw. der Komponente verbal beschreiben. Das Feld wird jedoch nicht für die Kompatibilitätsbestimmung herangezogen.
- *Externe Daten*
  Im Eigenschaftsfeld *Externe Daten* kann ein Verweis auf eine externe Datenquelle hinterlegt werden. Diese Datenquelle wird nicht für die Bestimmung der Kompatibilität herangezogen.

- **Stecker-, Buchsen- und Kommunikationsstecker/-buchsen-Beschreibungsfeld**
  Die Beschreibungsfelder für Standardstecker/-buchsen sowie für Kommunikationsstecker/-buchsen werden in (U)CML verwendet, um die kompatibilitätsrelevanten Eigenschaften sämtlicher Komponentenanschlüsse zu spezifizieren.

  Die Abbildung 3.128 zeigt auf der linken Seite das generische Standardstecker/-buchsenbeschreibungsfeld, während auf der rechten Seite ein generisches Kommunikationsstecker/-buchsenbeschreibungsfeld dargestellt ist. Das Stecker-/Buchsenbeschreibungsfeld beschreibt einen elektronischen Stecker/Buchse, das Kommunikationsstecker/-buchsebeschreibungsfeld einen Softwarekommunikationsstecker.

Richtung	Eingang/Ausgang (Von wo nach wo)
informeller Name	Name und eineindeutige Zuord. zu einer Komp.
Stecker/Buchse-Art	optional/zwingend
Stecker/Buchse-Zuordnung	Elektronik/Elektrik, Mechanik oder Software
Vorbedingung (IN)	
Nachbedingung (OUT)	
Invariante	
Beschreibungsliste	
Spannung	Spannungswert
Einheit	Einheit z.B. Volt
Definitionsbereich	Definitionsbereich z.B. [0...100]
Strom	Stromwert
Einheit	Einheit z.B. Ampere
Definitionsbereich	Definitionsbereich z.B. [0...1]
Kommentar	Kommentar
Funktionalität	Tabelle oder MSC
Zusatzinformationen	Zusatzinformationen
Version	Version
informelle Funktionalitäts-beschreibung	informelle Beschreibung der Funktionalität des Steckers bzw. der Buchse

Richtung	Kommunikation (Von wo nach wo)
informeller Name	Name und eineindeutige Zuord. zu einer Komp.
Stecker/Buchse-Art	optional/zwingend
Stecker/Buchse-Zuordnung	Elektronik/Elektrik, Mechanik oder Software
Vorbedingung (IN)	
Nachbedingung (OUT)	
Invariante	
Beschreibungsliste	
Methodenname	Funktionsname + Parameterliste
Ausgang	
Ergebnis-Typ	Ergebnis-Typ z.B. int
Einheit	Einheit z.B. Volt
Definitionsbereich	Definitionsbereich z.B. [0...100]
Stepping	Stepping (optional), z.B. 1
Startwert	Startwert (optional)
Default-Wert	Default-Wert (optional)
Parameter-Liste (Anz.: 1)	
Eingang	
1 Parameter-Name	Name des ersten Parameters
Parameter-Typ	Parameter-Typ z.B. float
Einheit	Einheit z.B. Volt
Definitionsbereich	Definitionsbereich z.B. [0...50]
Stepping	Stepping (optional)
Startwert	Startwert (optional)
Default-Wert	Default-Wert (optional)
Protokoll	Protokoll
Funktionalität	Tabelle oder MSC
Zusatzinformationen	Zusatzinformationen
Version	Version
informelle Funktionalitäts-beschreibung	informelle Beschreibung der Funktionalität des Kommunikationssteckers bzw. der -buchse

Abbildung 3.128.: Stecker-/Buchsenbeschreibungsfeld (links), Kommunikationsstecker/-buchsenbeschreibungsfeld (rechts).

Erläuterung der domänenübergreifenden Eigenschaftsfelder der beiden Beschreibungsfeldarten aus der Abbildung 3.128:

- *Richtung (*)*
  Das Eigenschaftsfeld *Richtung* beschreibt die Flussrichtung des Datums relativ zur Komponente, also Stecker – Ausgang (rechts), Buchse – Eingang (links) und Kommunikationsstecker/-buchse (links oder rechts). Zusätzlich zur Richtung wird der vollständige Pfad von der Quelle bis zur Senke hinterlegt.

- *Informeller Name*
  Mit Hilfe des Felds *Informeller Name* kann jedem Stecker, Buchse oder Kommunikationsstecker/-buchse ein Name zugewiesen werden. Dieser Name dient zur Identifikation des Anschlusses an einer Komponente. Dabei darf jeder informelle Name nur einmal pro Komponente vergeben werden.

- *Stecker-/Buchsen-Zuordnung*
  Im Eigenschaftsfeld *Stecker-/Buchsen-Zuordnung* wird festgelegt, welcher Domäne der Stecker/Buchse bzw. der Kommunikationsstecker/-buchse angehört – also Elektrotechnik, Mechanik oder Software.

- *Stecker-/Buchsen-Art*
  Mit Hilfe des Felds *Stecker-/Buchsen-Art* wird festgelegt, ob der Stecker/Buchse bzw. der Kommunikationsstecker/-buchse zwingend oder optional ist.

- *Funktionalität*
  Im Eigenschaftsfeld *Funktionalität* kann die Funktion bzw. das Verhalten des Steckers/Buchse bzw. des Kommunikationssteckers/-buchse hinterlegt werden.

Wie bereits im Kapitel „Grundsätzlicher Aufbau und Struktur der (U)CML-Beschreibungsfelder“ gezeigt wurde, wird mit Hilfe der domänenabhängigen Beschreibungsliste das Beschreibungsfeld an die jeweilige Domäne angepasst. In der nachfolgenden Aufzählung sind einige Eigenschaftsfelder der drei Domänen – Elektrotechnik, Mechanik und Software – beschrieben. Dabei gilt: Das Softwarebeschreibungsfeld ist fest vorgegeben, während die beiden Beschreibungsfelder für Elektrotechnik und Mechanik vom Anwender an seine Bedürfnisse angepasst werden können.

- **Elektrotechnik**
  In diesem Abschnitt wird eine kleine Auswahl an elektrischen/elektronischen Eigenschaftsfeldern beschrieben, mit deren Hilfe es möglich ist, ein elektrisches/elektronisches Signal zu modellieren und anschließend auf Kompatibilität hin zu untersuchen. Als Beispiel dient hier ein Rechtecksignal mit einer Spannungs-Amplitude von 0 bis maximal $100V$ und einer Strom-Amplitude von 0 bis $1A$.

  * *Signalform und Formel*
    Im Feld *Signalform und Formel* kann die Signalform – Sinus, Rechteck, Dreieck etc. – oder die dem elektrischen/elektronischen Signal zugrunde liegende Formel hinterlegt werden. Das im vorangegangenen Beispiel definierte Signal ist ein Rechtecksignal.

  * *Signalart*
    Mit Hilfe des Felds *Signalart* kann die Signalform weiter verfeinert werden, z.B. ob das Signal nur diskrete Zustände oder kontinuierliche Werte annehmen kann. Das Beispielsignal ist ein kontinuierliches Rechtecksignal.

  * *Signaleigenschaften*
    Elektrische/elektronische Signale können auf unterschiedliche Art und Weise spezifiziert werden. Für jede elektrische/elektronische Signaleigenschaft wie z.B. Strom/Spannung oder Frequenz müssen Einheit und Definitionsbereich hinterlegt werden. Das Beispielsignal besitzt zwei Signaleigenschaften: Strom und Spannung. Die Einheit des Stroms ist Ampere und er ist definiert von $0 \ldots 1\mathrm{A}$. Für die Spannung gilt die Einheit Volt und der Definitionsbereich $0 \ldots 100\mathrm{V}$.

  * *Kommentarfeld: Verbindung zu einer Softwareeigenschaft*
    Soll wie im Kapitel „3.6.4.3 *Modellierung von elektrischen/elektronischen Signalen in (U)CML*“ ab Seite 234 gezeigt, die Verbindung einer Softwaremethode mit einem elektrotechnischen Signal modelliert werden, so kann dieser Zusammenhang als Kommentar im Beschreibungsfeld hinterlegt werden.

- **Mechanik**
  Für die Spezifikation von mechanischen Signalen bzw. mechanischen Eigenschaften werden in (U)CML verschiedene Eigenschaftsfelder bereitgestellt. Für alle Eigenschaftsfelder gilt: jede Eigenschaft (Attribut) muss mit einer *Einheit* und einem *Definitionsbereich* spezifiziert werden. Ohne diese Angaben kann das Modell nicht auf Kompatibilität untersucht werden. In der nachfolgenden Aufzählung sind einige Eigenschaftsfelder aufgelistet, um mechanische Eigenschaften einer Baugruppe in einem Beschreibungsfeld zu spezifizieren:

  * *Größe*
    Soll die Größe einer Baugruppe spezifiziert werden sind dafür im Allgemeinen drei Größenangaben notwendig. Diese Größenangaben werden im Feld *Größe* hinterlegt. Die mechanische Baugruppe aus der Abbildung 3.129 hat die drei Abmessungen: $x = 29, y = 10$ und $z = 20$. Alle drei Abmessungen sind in der Einheit „mm“ angegeben.

  * *Form*
    Das Eigenschaftsfeld *Form* beschreibt verbal die Form des zu modellierenden Gegenstandes (z.B. rund, quadratisch oder frei geformt). Dabei gilt, dass stets die Abmessungen des kleinsten Kubus, der den Gegenstand vollständig umfasst, zur Beschreibung der Größe dienen (siehe Abb.: 3.129).

    Im Eigenschaftsfeld *Form* kann außer der verbal beschriebenen Form der Baugruppe auch eine CAD Zeichnung als externe Quelle hinterlegt werden. Beispielsweise könnte die CAD Zeichnung aus der Abbildung 3.129 als externe Quelle angegeben werden.

  * *Masse*
    Das Eigenschaftsfeld *Masse* dient zur Aufnahme des Gewichts eines mechanischen Objekts. Das Beispielwerkstück hat eine Masse von $100\mathrm{kg}$.

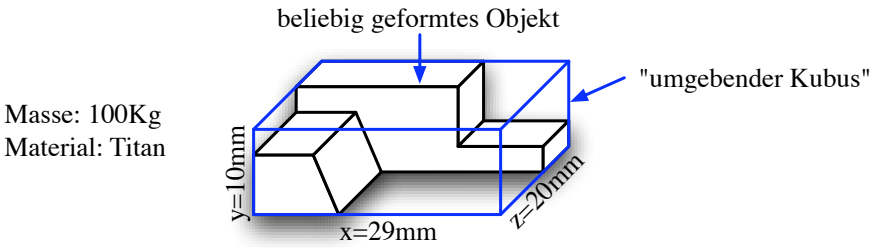


Abbildung 3.129.: Zusammenhang zwischen Form und Größe in (U)CML-Beschreibungsfeldern.

  * *Material*
    Mit Hilfe des Felds *Material* wird das verwendete Material der Baugruppe festgelegt. Die Baugruppe aus der Abbildung 3.129 besteht aus dem Material Titan.

- **Software**
  Aufgrund der „statischen Struktur“ von Methodensignaturen in der Softwaretechnik sind für diese Domäne die Eigenschaftsfelder fest vorgegeben und können durch den Anwender nicht angepasst oder verändert werden.

  Die C++ Methode `int Inc(int IN1)` dient als Grundlage für die Beschreibung des softwarespezifischen Teils des Kommunikationssteckerbeschreibungsfelds (vgl. Abb.: 3.127 rechts).

  * *Methodenname*
    Im Feld *Methodenname* ist die vollständige Signatur der Methode hinterlegt, z.B. `int Inc(int IN1)`. Sämtliche Angaben der Signatur werden in die nachfolgenden Felder einsortiert und um zusätzliche Informationen erweitert.
  * *Ergebnis-Typ*
    Der *Ergebnistyp* beschreibt das Funktionsergebnis einer Methode. Der Ergebnistyp kann jedoch auch leer bzw. nicht spezifiziert (**void**) sein. Zusätzlich zum Ergebniswert muss eine Einheit für das Ergebnis angegeben werden. Die beispielhafte C++ Methode hat als Ergebnistyp **int**. Die Einheit kann aus der C++ Signatur nicht entnommen werden.
  * *Definitionsbereich*
    Mit Hilfe des Eigenschaftsfelds *Definitionsbereich* kann der Definitionsbereich eines Datentyps weiter einschränkt werden. Dies ist besonders wichtig, falls für die Übertragung eines Werts nur ein kleiner Ausschnitt aus dem Zahlenbereich eines Datentyps verwendet werden soll. Aus der oben angegebenen C++ Methode kann kein Definitionsbereich für den Parameter bzw. das Ergebnis entnommen werden.
  * *Stepping, Start- und Default-Wert*
    Mit Hilfe der drei Eigenschaftswerte *Stepping*, *Start-* und *Default-Wert* kann der Definitionsbereich eines Parameters weiter eingeschränkt bzw. an die individuellen Bedürfnisse angepasst werden. Die Beispielmethode verwendet keine der drei Eigenschaften.
  * *Parameter-Name und Parameter-Typ*
    Hat eine Methode einen oder mehrere Parameter, so wird der Name und der dazugehörige Datentyp des einzelnen Parameters hier hinterlegt. Die oben angegebene C++ Methode hat einen Parameter `IN1` vom Typ **int**.
  * *Protokoll*
    Soll mit der Stecker/Buchse Kombination bzw. einem Kommunikationsstecker/-buchse ein komplexer Datenaustausch modelliert werden, kann dies mit Hilfe des Eigenschaftsfelds *Protokoll* erfolgen. Dort kann das gesamte Protokoll hinterlegt werden, das den Datenaustausch spezifiziert.

- **Paketschnittstellenbeschreibungsfeld**
  In (U)CML gibt es zwei unterschiedliche Arten von Paketschnittstellen, zum einen die *Standardpaketschnittstellen* und zum anderen die *Paketkommunikationsschnittstellen*. Mit Hilfe des generischen Paketschnittstellenbeschreibungsfelds werden beide Schnittstellenarten beschrieben.

  Die Abbildung 3.130 zeigt das universell verwendbare Beschreibungsfeld für Paketschnittstellen.

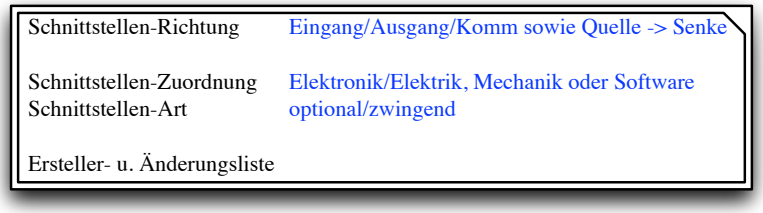


Abbildung 3.130.: Paketschnittstellenbeschreibungsfeld.

Beschreibung der Eigenschaftsfelder des Paketschnittstellenbeschreibungsfelds:

- *Schnittstellen-Richtung (*)*
  Das Feld *Schnittstellenrichtung* legt fest, ob es sich bei der Schnittstelle um eine Eingangs- oder Ausgangsschnittstelle handelt. Im Falle einer Kommunikationsschnittstelle wird als Richtung *Komm* angegeben. Zusätzlich zur Richtung der Schnittstelle wird der komplette Pfad bis zu einem Komponentenstecker/-buchse oder eines Steckers/Buchse angegeben.
- *Schnittstellen-Zuordnung*
  Im Eigenschaftsfeld *Schnittstellenzuordnung* wird festgelegt, welcher Domäne die Schnittstelle angehört (Elektrotechnik, Mechanik oder Software).
- *Schnittstellen-Art*
  Mit Hilfe des Eigenschaftsfelds *Schnittstellenart* wird festgelegt, ob die Schnittstelle zwingend oder optional ist.

- **Flusspfeilbeschreibungsfeld**
  Das generische Flusspfeilbeschreibungsfeld dient zur Beschreibung sowohl des Standard- wie auch des Kommunikationsflusspfeils. Die Abbildung 3.131 zeigt das generische Flusspfeilbeschreibungsfeld.

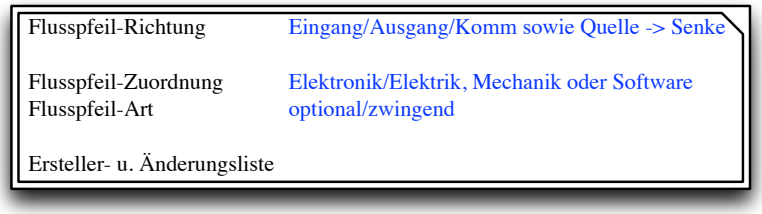


Abbildung 3.131.: Flusspfeilbeschreibungsfeld.

Erläuterung der Eigenschaftsfelder des generischen Flusspfeilbeschreibungsfelds:

- *Flusspfeil-Richtung (*)*
  Im Eigenschaftsfeld *Flusspfeilrichtung* wird der vollständige Name der Quelle (Stecker/Schnittstelle) sowie der Senke des Flusspfeils eingetragen. Der vollständige Name setzt sich aus dem Paket- bzw. Komponentennamen und der Anschlussbezeichnung (informeller Name) zusammen.
- *Flusspfeil-Zuordnung (*)*
  Im Eigenschaftsfeld *Flusspfeilzuordnung* wird festgelegt, welcher Domäne der Flusspfeil angehört (Elektrotechnik, Mechanik oder Software). Die Zuordnung zur Domäne erfolgt aufgrund der Zuordnung der Quelle. Ist beispielsweise die Quelle vom Typ Mechanik, ist der Flusspfeil ebenfalls von Typ Mechanik. Hat die Senke einen anderen Typ, so wird ein Fehler angezeigt.
- *Flusspfeil-Art*
  Mit Hilfe des Eigenschaftsfelds *Flusspfeilart* wird festgelegt, ob der Flusspfeil zwingend oder optional ist.

- **Externes Systemeingangspfeil- und externes Systemausgangspfeilbeschreibungsfeld**
  Mit Hilfe des externen Systemeingangspfeil- bzw. des externen Systemausgangspfeilbeschreibungsfelds werden die Eigenschaften der externen Ein- und Ausgänge des Systems von und zur Umwelt spezifiziert. Die Abbildung 3.132 auf der nächsten Seite zeigt das Beschreibungsfeld eines externen Systemeingangs- bzw. -Systemausgangspfeils. Das abgebildete Beschreibungsfeld gehört der Domäne Elektrotechnik an.

  Sämtliche Eigenschaftsfelder des externen Systemeingangs- bzw. -Ausgangspfeils wurden bereits in den vorangegangen Beschreibungsfeldern ausführlich beschrieben. Aus diesem Grund werden sie an dieser Stelle nicht noch einmal ausgeführt.

- **Zusammengefasste Stecker-/Buchsenbeschreibungsfeld und zusammengefasstes Flusspfeilbeschreibungsfeld**
  Für alle zusammengefassten Stecker/Buchsen und Flusspfeile gilt: Der Inhalt der einzelnen Stecker-/Buchsenbeschreibungsfelder bzw. des Flusspfeilbeschreibungsfeldes wird zu einem großen Beschreibungsfeld zusammengefasst. Der Aufbau und die Struktur entsprechen dabei dem der einzelnen „einfachen“ Beschreibungsfelder.

- **BUS-System Beschreibungsfelder**
  Die Beschreibungsfelder des (U)CML-BUS-Systems werden im Kapitel „3.6.2.7 *(U)CML-BUS-System*“ ab Seite 211 ausführlich beschrieben.

Beispiel 68: (U)CML-System mit Beschreibungsfeldern
In der Abbildung 3.133 ist das selbe System, das bereits in den vorangegangenen Kapiteln[234] in unterschiedlichen Modellierungssprachen dargestellt wurde, in (U)CML modelliert worden.

Das Beispielsystem besteht aus zwei Paketen: dem Systempaket *System* sowie dem Paket *SubSystem*. Des Weiteren enthält das System sechs Komponenten: *Element1*, *Element2*, *Element3*, *Verteiler*, *SubElement1* und *SubElement2*, wobei die beiden Komponenten

[234] Siehe hierzu die Abbildungen 3.7 auf Seite 115, 3.19 auf Seite 125, 3.43 auf Seite 141 sowie 3.59 auf Seite 150.

Richtung	Eingang/Ausgang
informeller Name	Name des externen Ein- bzw. Ausgangs
Stecker-/Buchse-Art	optional/zwingend
Stecker-/Buchse-Zuordnung	Elektronik/Elektrik, Mechanik oder Software
Vorbedingung (IN)	
Nachbedingung (OUT)	
Invariante	
Beschreibungsliste	
Spannung	Spannungswert
Einheit	Einheit z.B. Volt
Definitionsbereich	Definitionsbereich z.B. [0...100]
Strom	Stromwert
Einheit	Einheit z.B. Ampere
Definitionsbereich	Definitionsbereich z.B. [0...1]
Kommentar	Kommentar
Zusatzinformationen	Zusatzinformationen
Version	Version
informelle Funktionalitäts-beschreibung	informelle Beschreibung der Funktionalität des Steckers bzw. der Buchse

Abbildung 3.132.: Externes Systemeingangs- und externes Systemausgangspfeilbeschreibungsfeld.

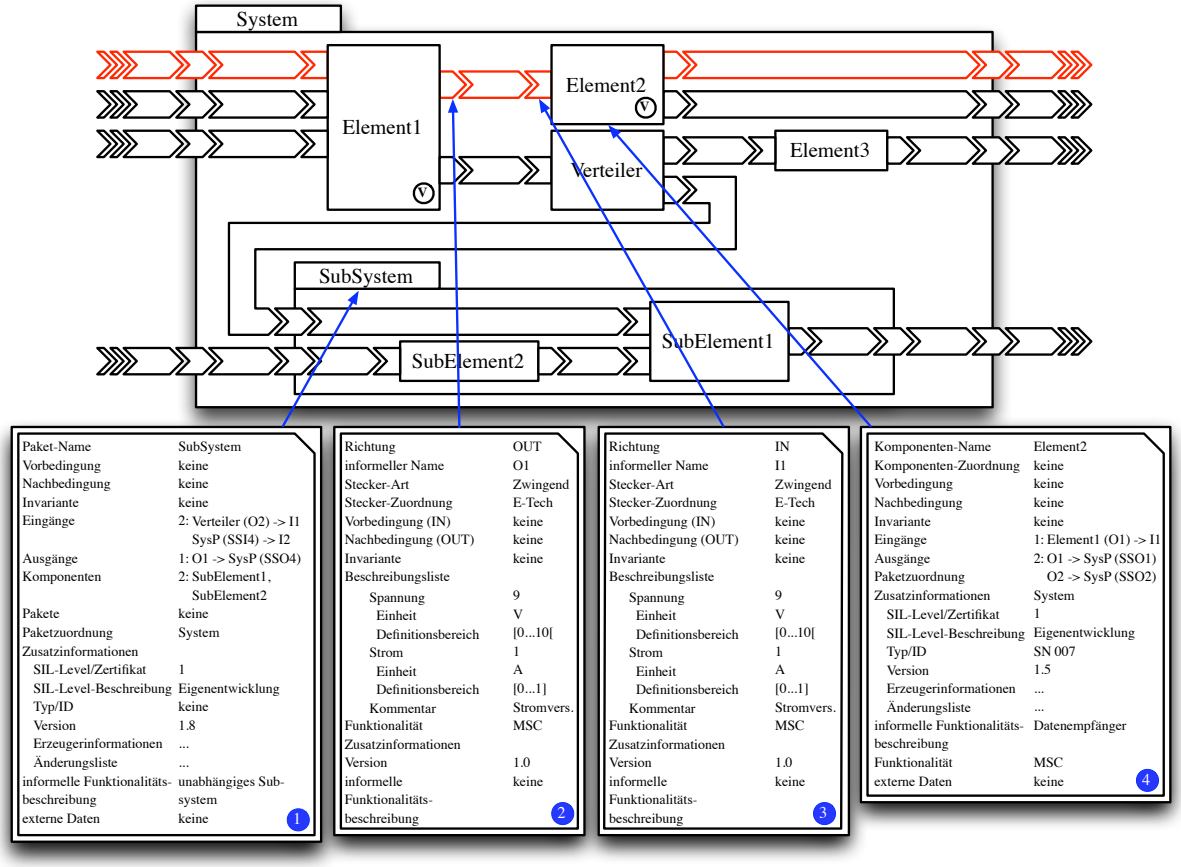


Abbildung 3.133.: (U)CML-Beispielsystem mit ausgefüllten Beschreibungsfeldern.

SubElement1 und *SubElement2* im Paket *SubSystem* enthalten sind. Außerdem besitzt das Beispielsystem vier Eingänge und vier Ausgänge über die es Daten mit der Umwelt austauschen kann. Für die vier Bestandteile des Systems – *Element2*, *SubElement1* sowie den Ausgang *O1* der Komponente *Element1* und den Eingang *I1* der Komponente *Element2* – wurden exemplarisch die entsprechenden Beschreibungsfelder dargestellt.

In den vier Beschreibungsfeldern sind alle Daten enthalten, die für die genaue Spezifikation der Eigenschaften des Elements sowie für die Kompatibilitätsprüfung notwendig sind.

1. **Paketbeschreibungsfeld:**
   Mit Hilfe des *Paketbeschreibungsfelds* wird der Aufbau und die innere Struktur eines (U)CML-Pakets beschrieben. Im ersten Feld wird der Name des Pakets *SybSystem* eingetragen. Daran anschließend folgen drei Felder, in denen die Vor- und Nachbedingungen sowie die Invarianten aller in dem Paket enthaltenen und nach außen geführten Stecker und Buchsen eingetragen werden[235]. Die nächsten fünf Felder des Beschreibungsfelds dienen zur Beschreibung der strukturellen Eigenschaften des Pakets. Hier werden sämtliche Ein- und Ausgänge, die das Paket besitzt, namentlich aufgezählt und mit der Quelle mit der sie verbunden sind annotiert. Das Paket *SubSystem* hat zwei Eingänge *I1* und *I2*. Der Paketeingang *I1* ist mit dem Ausgang *O2* der Komponente *Verteiler*, der Paketeingang *I2* mit der Systempaketschnittstelle *SSI4* verbunden. Der Ausgang *O1* des Pakets *SubSystem* ist mit der Systempaketschnittstelle *SSO4* verbunden. Des Weiteren wird in den beiden Feldern Pakete bzw. Komponenten die Anzahl und der Name der in diesem Paket enthaltenen Pakete und Komponenten eingetragen. Das Beispiel enthält zwei Komponenten *SubElement1* und *SubElement2*, jedoch kein weiteres Paket.

   Das nächste Feld – Zusatzinformationen – des Beschreibungsfelds besteht seinerseits aus sechs Feldern, wobei die letzten beiden Felder zusammengesetzt sind. Sie enthalten folgende Informationen: den SIL-Level sowie die dazugehörige Beschreibung, den Typ und die Version des Pakets. Die beiden zusammengesetzten Felder Erzeugerinformationen und Änderungsliste enthalten sämtliche Daten über die Erzeugung und die Bearbeitung des Pakets.

   Das letzte Feld enthält einen Verweis auf externe Daten, die für die Beschreibung des Pakets notwendig sind. Der Verweis kann auf beliebige Daten verweisen, da diese nicht für die Kompatibilitätsbestimmung herangezogen werden.

2. **Steckerbeschreibungsfeld:**
   Im *Steckerbeschreibungsfeld* werden sämtliche Eigenschaften eines Eingangs einer Komponente festgelegt. Im ersten Feld wird die Richtung des Steckers festgelegt – im Beispiel *OUT*. Anschließend folgt der informelle Name des Steckers *O1*[236] sowie die Beschreibung der Stecker-Art (*zwingend* oder *optional*), die Zuordnung (Elektronik) sowie die Vor- und Nachbedingungen und die Invariante. Der nächste Beschreibungsblock dient zur Beschreibung der domänenspezifischen Eigenschaften des Ausgangs. Der Ausgang hat eine Spannung von $9\mathrm{V}$ mit einem Definitionsbereich von $[0 \ldots 10[\mathrm{V}$ und Strom von $1\mathrm{A}$ mit einem Definitionsbereich von $[0 \ldots 1]\mathrm{A}$. Abgeschlossen wird der Beschreibungsblock mit einer optionalen verbalen Zusatzinformation. Sollte ein Stecker mehr als einen Typnamen übertragen, wird der Definitionsblock für jeden Typnamen wiederholt.

   Im letzten Block des Beschreibungsfelds werden die optionalen Zusatzinformationen, die Funktionalität, die Version sowie die informelle Funktionalitätsbeschreibung des Ausgangs hinterlegt.

3. **Buchsenbeschreibungsfeld:**
   Das *Buchsenbeschreibungsfeld* enthält im Wesentlichen die selben Informationen wie das Steckerbeschreibungsfeld.

4. **Komponentenbeschreibungsfeld:**
   Durch das *Komponentenbeschreibungsfeld* werden die Eigenschaften der Komponente festgelegt. Im Kopf des Beschreibungsfelds wird der Name der Komponente *Element2* eingetragen, gefolgt von den Vor- und Nachbedingungen sowie der Invariante der Komponente. Die Vor- und Nachbedingungen und die Invariante entsteht durch die Sammlung aller Bedingungen der Stecker und Buchsen der Komponente. In den nächsten beiden Feldern werden die Namen und die Verbindungen der Stecker und Buchsen der Komponente automatisch eingetragen. Im nächsten Feld wird die Paketzuordnung, also in welchem Paket sich die Komponente befindet (hier im Paket *System*), beschrieben.

   Abgeschlossen wird das Beschreibungsfeld mit der Beschreibung des SIL-Levels sowie der dazugehörigen Beschreibung, dem Typ, der Version und der informellen Funktionalitätsbeschreibung. Das letzte der beiden Felder des Komponenten-Beschreibungsfelds dient zur formalen Beschreibung der Funktionalität der Komponente (vgl. Abschnitt „3.6.2.3.2 *Verhalten einer (U)CML-Komponente*“ ab Seite 170) sowie zur externen Beschreibung des Inhalts wie z.B. einem Verweis auf ein externes Dokument.

□

3.6.2.7. (U)CML-BUS-System

Das BUS-System stellt in (U)CML eine Besonderheit dar. Es ist ein Zugeständnis an die Elektrotechnik, wo es üblich ist, viele gemeinsame Verbindungen, die von einem Baustein zu einem anderen geführt sind, mittels eines BUS-Systems in einem Schaltplan zu modellieren. In (U)CML sind BUS-Systeme jedoch nicht nur auf die Modellierung von elektrischen/elektronischen Systemen beschränkt sondern können z.B. auch in der Softwaretechnik zur Modellierung von Methodenaufrufen verwendet werden. Sind beispielsweise zwei Softwarekomponenten über viele unterschiedliche Verbindungspfeile – Standard- und Kommunikationsflusspfeile – miteinander verbunden, können diese nicht durch einen zusammengefassten Flusspfeil modelliert werden. Solche Verbindungen werden dann durch ein BUS-Systems modelliert und dargestellt, wodurch das System „logisch geordnet“ und graphisch übersichtlicher wird.

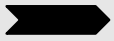
Eine weitere wichtige Grundeigenschaft des (U)CML-BUS-Systems ist die Modellierung einer „$1 : n$“ Verbindung zwischen Paketen und Komponenten. Ohne die Verwendung des BUS-Systems muss für die Realisierung einer $1 : n$ Verbindung zwischen Paketen und Komponenten stets eine „Verteiler-Komponente“ modelliert werden, da in (U)CML die Gabelung von Flusspfeilen[237] unzulässig ist.

[235] Die Felder Vor- und Nachbedingung sowie die Invariante werden durch den (U)CML-Editor *„(U)CML-ed“* automatisch ausgefüllt.
[236] Der Name des Steckers wird durch das Werkzeug *„(U)CML-ed“* vergeben und kann vom Nutzer nicht beeinflusst werden.
[237] Siehe hierzu die Abbildung 3.113 auf Seite 186.

Definition 3.62 ***(U)CML-BUS-System***
Sei $n \in \mathbb{N}$, $2 \leq n < \infty$. Des Weiteren sei n die Anzahl der am BUS angeschlossenen Pakete oder Komponenten. Dann gilt: Jedes ***BUS-System*** *in (U)CML stellt eine $1 : n$ Verbindung zwischen allen am BUS angeschlossenen Paketen und Komponenten her.*

Aufbau und Struktur des (U)CML-BUS-Systems
Jedes BUS-System besteht in (U)CML aus verschiedenen Bausteinen die zusammen das BUS-System bilden. Das Grund- bzw. Trägerelement jedes BUS-Systems ist der *BUS-System-Pfeil*. Dieser gibt für alle angeschlossenen BUS-Teilnehmer verbindlich an, ob das BUS-System uni- oder bidirektional ist. In der Tabelle 3.11 auf Seite 201 wurden sämtliche Grundbausteine eines (U)CML-BUS-Systems eingeführt. In den nachfolgenden Definitionen werden die Bestandteile noch einmal aufgegriffen und erläutert. Abgeschlossen wird diese Einführung mit einem ausführlichen Beispiel.

Definition 3.63 ***BUS-System-Pfeil***
Der BUS-System-Pfeil ist das Trägerelement des (U)CML-BUS-Systems. An ihm werden alle weiteren BUS-Systembestandteile angebracht. Dabei wird zwischen dem ***unidirektionalen*** *und dem* ***bidirektionalen BUS-System-Pfeil*** *unterschieden.*

unidirektionaler BUS-System-Pfeil 

bidirektionaler BUS-System-Pfeil 

Bei einem bidirektionalen BUS-System kann jeder BUS-Teilnehmer sowohl Sender (Quelle) als auch Empfänger (Senke) eines Datums sein. Aus diesem Grund sollten an einem bidirektionalen BUS-System stets *Paketkommunikationsschnittstellen bzw. Komponentenkommunikationsstecker/-buchsen* angeschlossen sein[238]. Im Falle eines unidirektionalen BUS-Systems darf an einem BUS-System-Pfeil nur *genau ein* Sender existieren. Alle anderen BUS-Teilnehmer müssen dann als Empfänger spezifiziert werden.

Definition 3.64 ***BUS-System-Kommunikationsrichtung***
An einem ***bidirektionalen BUS-System*** *kann jeder BUS-Teilnehmer sowohl Sender (Quelle) als auch Empfänger (Senke) eines Datums sein. An einem* ***unidirektionalen BUS-System*** *gibt es* ***genau einen*** *Sender. Alle anderen BUS-Teilnehmer sind dann Empfänger.*

BUS-Systeme können, wie alle anderen Flusspfeile der (U)CML, nicht über Paketgrenzen hinweg verbunden werden. Aus diesem Grund gibt es spezielle *BUS-Systempaketschnittstellen*, mit deren Hilfe ein BUS-System über eine Paketgrenze geführt werden kann.

Definition 3.65 ***BUS-System-Paketschnittstelle***
Für die Verbindung von BUS-Systemen über Paketgrenzen hinweg werden spezielle ***BUS-System-Paketschnittstellen*** *verwendet. Dabei wird zwischen uni- und bidirektionalen BUS-Systemen unterschieden.*

unidirektionale BUS-System-Paketschnittstelle 

bidirektionale BUS-System-Paketschnittstelle 

Anmerkungen zur Definition 3.65:

- In (U)CML können BUS-Systeme nicht direkt mit der Umwelt verbunden werden. Aus diesem Grund gibt es in (U)CML auch keine speziellen *externen BUS-System-Paketschnittstellen* bzw. *externe Flusspfeile*. Ist es für die Modellierung eines Systems dennoch notwendig, ein internes BUS-System mit der Umwelt zu verbinden, so muss dieses explizit mit Hilfe einer Verteilerkomponente modelliert werden. In der Abbildung 3.134 ist solch eine Verbindung eines internen BUS-Systems mit der Umwelt exemplarisch dargestellt.

- Verbindungen zwischen BUS-System-Pfeilen innerhalb eines Pakets sind nicht zulässig. Jeder BUS-System-Pfeil ist innerhalb eines Pakets genau so lang, dass alle angeschlossenen Pakete und Komponenten erreicht werden. Aus diesem Grund ist eine Verbindung von BUS-System-Pfeilen in (U)CML nicht notwendig.

Nachdem nun sowohl die BUS-System-Pfeile als auch die BUS-System-Paketschnittstellen eingeführt wurden, folgt nun die Beschreibung der Verbindung des BUS-Systems mit Paketen und Komponenten.

An dem *BUS-System-Pfeil* können je nach Ansicht entweder *BUS-System-Balken* mit Steckern und Buchsen (Glass-Box Darstellung) oder *BUS-System-Anschlüsse* (Black-Box Darstellung) angeschlossen werden.

[238] Anmerkung:
An ein bidirektionales (U)CML-BUS-System können auch einzelne Stecker und Buchsen angeschlossen sein.

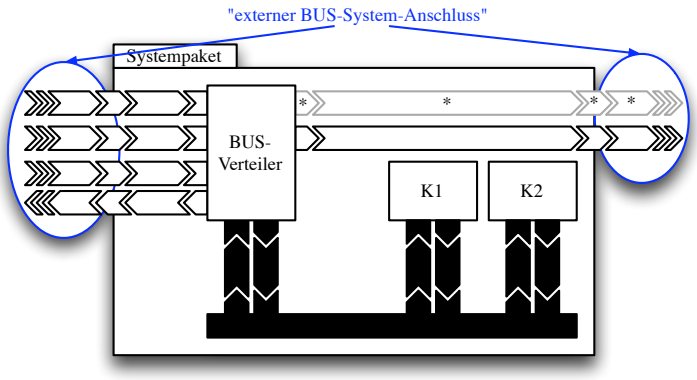
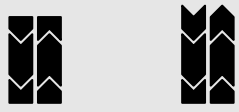


Abbildung 3.134.: Verbindung eines internen BUS-Systems mit der Umwelt.

Definition 3.66 ***BUS-System-Anschluss und BUS-System-Balken***
An einem BUS-System-Pfeil kann, je nach Ansicht, ein ***BUS-System-Anschluss*** *oder ein* ***BUS-System-Balken*** *angeschlossen werden.*

BUS-System-Balken in Black-Box Ansicht



BUS-System-Anschluss in Glass-Box Ansicht

Dabei gilt: An einem BUS-System-Balken muss ***genau ein*** *Paket oder* ***genau eine*** *Komponente angeschlossen sein, während an einem BUS-System-Anschluss* ***mindestens*** *eine Eingangsbuchse oder eine Ausgangsstecker vorhanden sein muss.*

BUS-System-Anschlüsse bzw. BUS-System-Balken werden in (U)CML stets vertikal stehend auf dem BUS-System-Pfeil dargestellt und von unten an das Paket bzw. die Komponente angeschlossen, so dass es zu keiner Verwechselung mit „normalen“ Anschlüssen kommen kann. Die Stecker und Buchsen eines BUS-System-Balkens werden links (Buchsen) und rechts (Stecker) angebracht und durch entsprechende Flusspfeile mit den Schnittstellen der Pakete bzw. Komponente verbunden.
Dabei gilt: Alle Anschlüsse eines (U)CML-BUS-Systems sind *optional*[239] und müssen somit nicht angeschlossen werden.

Definition 3.67 ***BUS-System-Anschlüsse***
Alle Anschlüsse eines ***BUS-System-Balkens****, also die zugehörigen Buchsen und Stecker, sind stets* ***optional****. Das heißt, dass sie nicht angeschlossen müssen.*

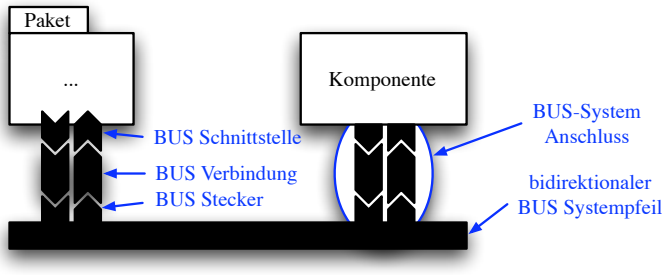


Abbildung 3.135.: (U)CML-BUS-System in Black-Box Ansicht.

In der nachfolgenden Aufzählung werden die unterschiedlichen Ansichten, die in der Definition 3.66 eingeführt worden sind, noch einmal aufgegriffen und anhand eines einfachen Beispiels erläutert.

- **Black-Box Ansicht**
  In der Black-Box Darstellung eines BUS-Systems werden alle Verbindungen zwischen einem Paket bzw. einer Komponente und dem BUS-System „virtuell“ zu einem *BUS-System-Anschluss* zusammengefasst. In der Abbildung 3.135 ist ein bidirektionales BUS-System, an dem ein Paket und eine Komponente angeschlossen sind in Black-Box Ansicht dargestellt.

[239] Eine genauere Beschreibung der optionalen Anschlüsse finden Sie im Kapitel „3.6.3 *Das Optionalitätsprinzip der (U)CML*“ ab Seite 218.

Alle Verbindungen zwischen dem bidirektionalen BUS-System-Pfeil und dem angeschlossenen Paket *Paket* sowie der Komponente *Komponente* werden als BUS-Anschluss modelliert und dargestellt. Dabei ist es unerheblich, ob die einzelnen Verbindung tatsächlich bidirektional ist.

- **Glass-Box Ansicht**
  In der Glass-Box Darstellung eines BUS-Systems werden alle Stecker und Buchsen des BUS-Systems, bzw. des BUS-System-Balkens einzeln dargestellt.

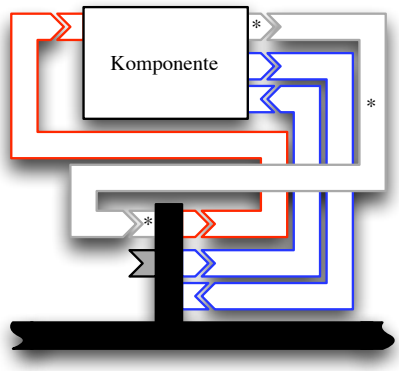


Abbildung 3.136.: (U)CML-BUS-System in Glass-Box Ansicht.

Die Abbildung 3.136 zeigt ein bidirektionales BUS-System mit einem BUS-System-Balken und explizit modellierten Buchsen und Steckern. Dabei zeigt sich, dass an einem BUS-System-Balken alle Arten von Steckern und Buchsen, die die (U)CML zur Verfügung stellt, angeschlossen werden dürfen – mit einer Ausnahme: An einen BUS-System-Balken darf kein anderes BUS-System angeschlossen werden.

Außerdem ist im Beispiel ein Anschluss des BUS-System-Balkens nicht angeschlossen. Dies verursacht jedoch keine Fehler, da nach Definition 3.67 alle Buchsen und Stecker eines BUS-System stets optional sind.

Anmerkung:
In einem (U)CML-BUS-System kann jeder Anschluss innerhalb eines Pakets in einer anderen Ansicht dargestellt werden.

BUS-System-Beschreibungsfelder
Aufgrund des komplexen Aufbaus der BUS-Systeme in (U)CML besitzen diese zwei unterschiedliche Beschreibungsfelder. Zu einen das *logische Beschreibungsfeld* in dem die logischen Eigenschaften des BUS-System-Pfeils bzw. des BUS-System-Balkens vollständig beschrieben sind. Zum anderen das *physikalische Beschreibungsfeld* in dem sowohl die physikalischen Anschlüsse (Pins) des BUS-Systems als auch die Bezeichnungen der Pins einzeln aufgelistet und spezifiziert werden.

- **Logische BUS-System-Beschreibungsfelder**
  Mit Hilfe der *logischen BUS-System-Beschreibungsfelder* werden alle logischen Aspekte des BUS-Systems beschrieben, die für die Kompatibilitätsbestimmung notwendig sind. Dafür wird das logische Beschreibungsfeld in zwei Untergruppen aufgeteilt. Zum einen in das *logische BUS-System-Pfeil-Beschreibungsfeld* in dem sämtliche Anschlüsse des BUS-System-Pfeils beschrieben werden, und zum anderen in das *logische BUS-System-Balken/-Anschluss-Beschreibungsfeld* in dem jeder Stecker bzw. Buchse des BUS-Balkens/-Anschlusses einzeln aufgelistet ist.

  – **BUS-System-Pfeil**
    Im Kopf des Beschreibungsfelds (siehe Abbildung 3.137) steht dessen Bezeichnung (hier: *BUS-System-Pfeil – logische Sicht*). Im darunter liegenden Feld ist die *Kommunikationsrichtung* – uni- oder bidirektional – des gesamten BUS-Systems eingetragen. Daran anschließend folgt die Anzahl der BUS-System-Balken/-Anschlüsse, die an den BUS-System-Pfeil angeschlossen sind.

BUS-System-Pfeil -- logische Sicht --	
Richtung	unidirektional od. bidirektional
Anzahl Balken	Anzahl der Balken bzw. Anschlüsse am BUS
Stecker	Anzahl: Liste aller Stecker-Namen des gesamten BUS-Systems
Buchsen	Anzahl: Liste aller Buchsen-Namen des gesamten BUS-Systems

Abbildung 3.137.: BUS-System-Pfeil-Beschreibungsfeld – logische Sicht.

Die beiden Felder *Stecker* und *Buchsen* repräsentieren eine Liste aller Stecker bzw. Buchsen, die am BUS-System angeschlossen sind, unabhängig von dem Anschluss, an den sie angeschlossen sind. Die Liste besteht aus der Anzahl der Stecker bzw. Buchsen gefolgt von einer Aufzählung aller Stecker- bzw. Buchsennamen. Die Stecker- bzw. Buchsennamen werden aus den einzelnen Beschreibungsfeldern der angeschlossenen BUS-System-Balken/-Anschlüssen entnommen.

– **BUS-System-Balken/-Anschluss**
Mit Hilfe des *BUS-System-Balken/-Anschlussbeschreibungsfelds* werden die einzelnen Stecker und Buchsen des BUS-System-Balkens/-Anschlusses beschrieben. Im Beschreibungsfeldkopf steht der Name des Beschreibungsfelds *BUS-System-Balken/-Anschluss – logische Sicht*. Im darunter liegenden Feld wird die logische Kommunikationsrichtung des gesamten BUS-System-Balkens/-Anschlusses festgelegt. Dabei ist zu beachten: Ist der BUS-System-Pfeil unidirektional, so darf es im gesamten BUS-System nur genau einen Sender geben. Alle anderen BUS-System-Balken/-Anschlüsse müssen dann als Empfänger gekennzeichnet sein. Im bidirektionalen Fall kann jeder BUS-Anschluss sowohl als Sender als auch als Empfänger fungieren. Im letzten Feld des Beschreibungsfeldkopfs wird die Nummer des in Bearbeitung befindlichen BUS-System-Balkens/-Anschlusses eingetragen.

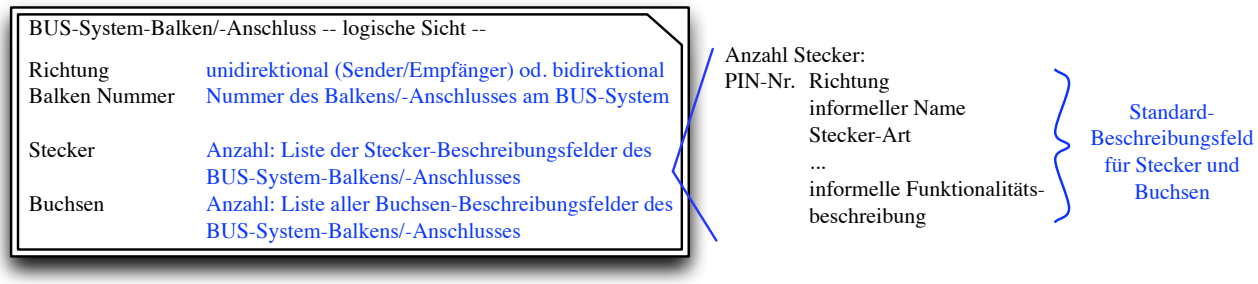


Abbildung 3.138.: BUS-System-Balken/-Anschlussbeschreibungsfeld – logische Sicht.

Die nächsten beiden Felder des Beschreibungsfelds repräsentieren eine Liste aller am BUS-System-Balken/-Anschluss angeschlossenen Stecker und Buchsen. Dabei werden die Informationen aus den Beschreibungsfeldern der einzelnen Stecker und Buchsen, die am BUS-System-Balken/-Anschluss angeschlossen sind, gesammelt und in die beiden Listen eingetragen. Des Weiteren wird hier die Zuordnung zwischen dem logischen Stecker/Buchse und den physikalischen Pins des Anschlusses vorgenommen.

Die Abbildung 3.138 zeigt das soeben beschriebene Beschreibungsfeld (links) sowie Aufbau und Struktur der Stecker- bzw. Buchsenlisten (rechts). Die Stecker- und Buchsenlisten unterscheiden sich von den Standardstecker- und Standardbuchsenbeschreibungsfeldern nur durch den Eintrag der physikalischen Pin-Nummer, die der Zuordnung zwischen der logischen und physikalischen Sicht auf das BUS-System dient.

- **Physikalische BUS-System-Beschreibungsfelder**
Auch das physikalische BUS-System-Beschreibungsfeld lässt sich – ähnlich wie das logische BUS-System-Beschreibungsfeld – in zwei Gruppen unterteilen. Mit Hilfe der beiden physikalischen BUS-System-Beschreibungsfelder werden die realen Eigenschaften der BUS-System-Komponenten, also des BUS-System-Pfeils und der BUS-System-Balken/-Anschlüsse, in (U)CML spezifiziert. In der Abbildung 3.139 ist auf der linken Seite das *BUS-System-Pfeil-Beschreibungsfeld* und auf der rechten Seite das *BUS-System-Balken/-Anschlüsse-Beschreibungsfeld* abgebildet.

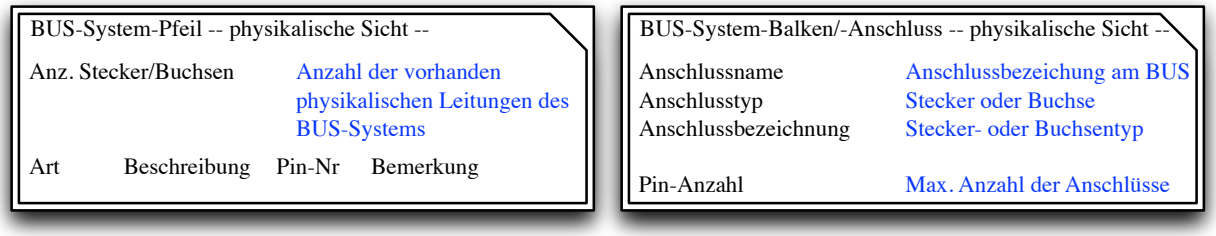


Abbildung 3.139.: BUS-Beschreibungsfelder – physikalische Sicht.

– **BUS-System-Pfeil**
Im Kopf des Beschreibungsfelds wird der Verwendungszweck eingetragen. Anschließend wird die Anzahl aller Stecker und Buchsen bzw. der real vorhandenen Leitungen des gesamten BUS-Systems im Feld *Anz. Stecker/Buchsen* hinterlegt. Die letzten vier Felder des Beschreibungsfelds sind als Liste ausgelegt. Sie werden zur Beschreibung der physikalischen Eigenschaften jedes einzelnen Pins des BUS-System-Pfeils verwendet. Im Feld *Art* wird die physikalische Kommunikationsrichtung jedes einzelnen Pins eingetragen (*IN*, *OUT* bzw. *IN/OUT*). Im Feld *Beschreibung* wird die Anwendung bzw. der Verwendungszweck des Pins verbal beschrieben. In den nächsten beiden Feldern folgt zunächst die *Pin-Nummer* gefolgt von einer kurzen *Bemerkung*.

– **BUS-System-Balken/-Anschluss**
Das *BUS-System-Balken/-Anschluss-Beschreibungsfeld* beginnt ebenfalls mit der Eintragung des Verwendungszwecks. Die nächsten drei Felder dienen der physikalischen Beschreibung des realen Anschlusses. Zuerst wird der *Anschlussname* spezifiziert[240], gefolgt vom *Anschlusstyp*, z.B. Stecker, Pfosten oder Wanne. Im darauf folgenden Feld wird die *Anschlussbezeichnung* des Anschlusstyps festgelegt. Im letzten Feld wird die reale Pin-Anzahl des Anschlusses eingetragen.

Anmerkungen:

- Das physikalische Beschreibungsfeld wird derzeit ausschließlich für die Modellierung von elektrischen BUS-Systemen verwendet. Eine Erweiterung auf andere Domänen kann jedoch leicht vorgenommen werden. Dazu ist lediglich eine Anpassung der entsprechenden Beschreibungsfelder notwendig.
- Die beiden BUS-System-Beschreibungsfelder werden sowohl in der Black-Box als auch in der Glass-Box Ansicht des BUS-Systems gleichermaßen verwendet.
- Bei „kombinierten“ BUS-Systemen werden auch die einzelnen Beschreibungsfelder kombiniert (siehe Beispiel unten).
- Die Stecker und Buchsen an einem BUS-System-Balken werden durch Standardbeschreibungsfelder für Buchsen und Stecker beschrieben.

Nachdem nun alle Bestandteile des (U)CML-BUS-Systems eingeführt wurden, folgt ein ausführliches Bespiel, in dem ein elektrisches Teilsystem eines Computers – das Speichersystem – stark vereinfacht modelliert und mit Hilfe eines (U)CML-BUS-Systems dargestellt ist.

Beispiel 69: Anwendung eines (U)CML-BUS-Systems
In der Abbildung 3.140 ist auf der linken Seite ein Auszug aus einem elektrischen Schaltpan dargestellt, in dem vier RAM-Bausteine und eine CPU enthalten sind. Alle Bausteine sind über drei unterschiedliche BUS-Systeme verbunden – dem *Adress-BUS*, *Daten-BUS* sowie den *Steuer-BUS*. Aus der Schaltplandarstellung (links) ist nicht ersichtlich, ob z.B. das Adress-BUS-System uni- oder bidirektional ist. Diese Information ist jedoch für die Modellierung in (U)CML sehr wichtig, weil nur so das „richtige“ BUS-System modelliert werden kann, und eine Kompatibilitätsprüfung möglich ist. Für das dargestellte Beispiel wird angenommen, dass das Adress- und Steuer-BUS-System unidirektional, das Daten-BUS-System bidirektional ist.

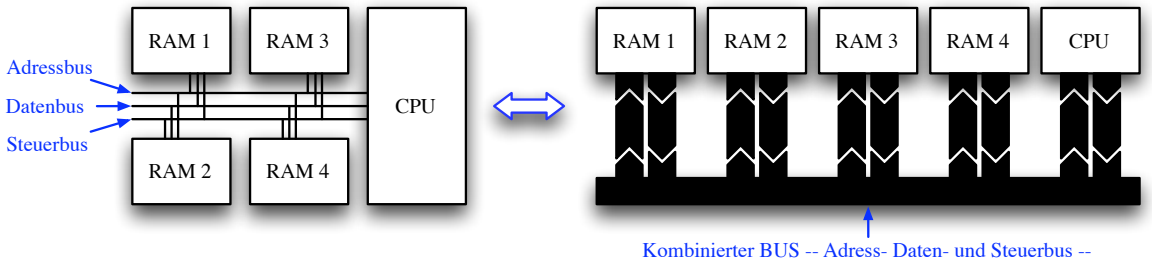


Abbildung 3.140.: Überführung eines elektrischen Schaltplans in ein (U)CML-Flussdiagramm.

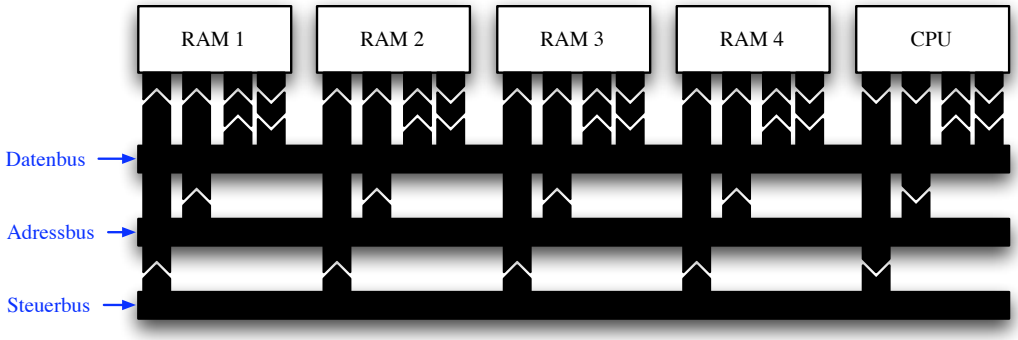


Abbildung 3.141.: Aufgespaltene Darstellung des BUS-Systems aus der Abbildung 3.140.

Auf der rechten Seite der Abbildung 3.140 ist das selbe System im (U)CML-Flussdiagramm dargestellt. Dabei wurden die drei unterschiedlichen BUS-Systeme zu einem gemeinsamen BUS-System zusammengefasst. Es ist jedoch auch möglich, jedes der drei BUS-Systeme (Abb.: 3.140 links) einzeln zu modellieren, je nachdem ob dies für die Kompatibilitätsprüfung notwendig ist oder nicht. So kann in der kompakten Darstellung des BUS-Systems nicht die Datenrichtung der einzelnen BUS-Systeme dargestellt werden. Ist diese Information jedoch für die Kompatibilitätsprüfung wichtig, so muss jedes BUS-System einzeln modelliert und dargestellt werden. In der Abbildung 3.141 ist das kompakte BUS-System aufgespalten in die drei Einzel-BUS-Systeme dargestellt.

[240] Der Anschlussname des BUS-System-Balken/-Anschluss-Beschreibungsfelds muss, wie alle anderen Namen in (U)CML eineindeutig sein, und kann somit im gesamten System nur einmal verwendet werden.

Durch die Auftrennung in drei einzeln modellierte BUS-Systeme ist es möglich, jedes BUS-System einzeln zu betrachten und auf Kompatibilität zu untersuchen. Dies ist besonders dann wichtig, wenn zwischen den Datenrichtungen der einzelnen BUS-Systeme unterschieden werden muss. So kann aus der Abbildung 3.141 leicht entnommen werden, dass der Adress- sowie der Steuer-BUS jeweils ein unidirektionales BUS-System bilden, während der Daten-BUS ein bidirektionales BUS-System ist. Des Weiteren ist in der Abbildung (Abb.: 3.141) zu erkennen, dass die Komponente *CPU* über den Adress- und Steuer-BUS Daten an die angeschlossenen Komponenten *RAM 1-4* unidirektional sendet und die Komponenten *RAM 1-4* ihrerseits unidirektional Daten über den Adress- und Steuer-BUS empfangen.

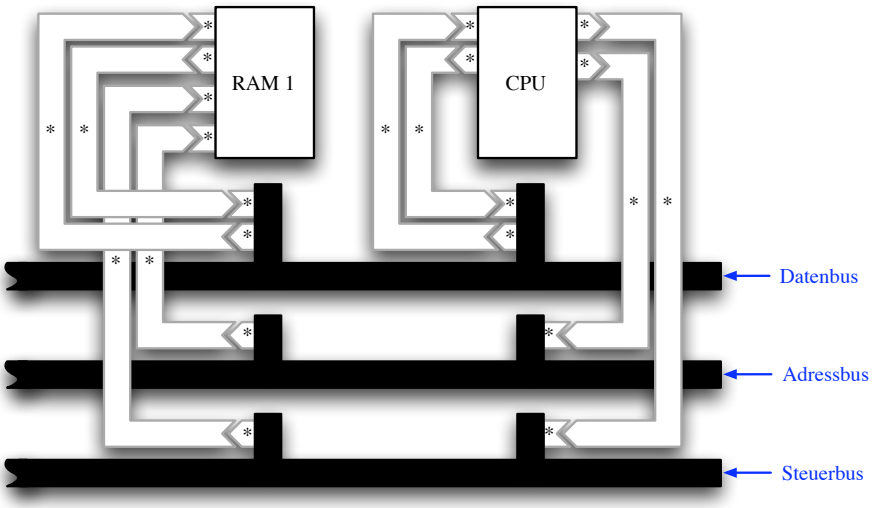


Abbildung 3.142.: Verfeinerung der Verbindungen zwischen der Komponente *RAM 1* und der Komponente *CPU*.

In der Abbildung 3.142 sind die drei BUS-System-Verbindungen zwischen der Komponente *RAM 1* und der Komponente *CPU* weiter verfeinert worden. Je detaillierter die BUS-Verbindung modelliert ist, desto mehr Informationen können später für die Kompatibilitätsprüfung entnommen werden. In der Abbildung 3.142 sind die einzelnen Verbindungen als zusammengefasste Flusspfeile modelliert worden. Die zusammengefassten Flusspfeile können noch weiter verfeinert werden. Dies geschieht in der Abbildung 3.143 beispielhaft für den Daten-BUS.

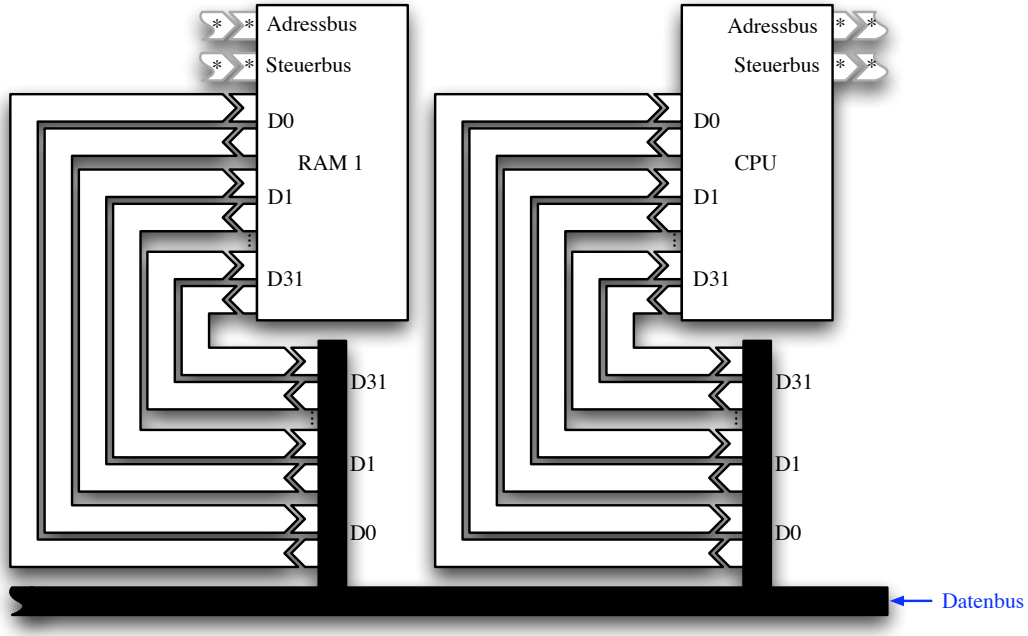


Abbildung 3.143.: Weitere Verfeinerung der Verbindungen zwischen der Komponente *RAM 1* und der Komponente *CPU* (Ohne den Adress- bzw. Steuerbus).

In der verfeinerten Darstellung des Daten-BUSs sind die einzelnen Kommunikationsverbindungen zwischen dem BUS-System auf der einen Seite, und der Komponente *RAM 1* und *CPU* auf der anderen zu erkennen. Die letzte Abbildung dieses Beispiels (Abb.: 3.144 auf der nächsten Seite) zeigt den Daten-BUS sowie die Komponente *RAM 1* inklusive zweier Beschreibungsfelder.

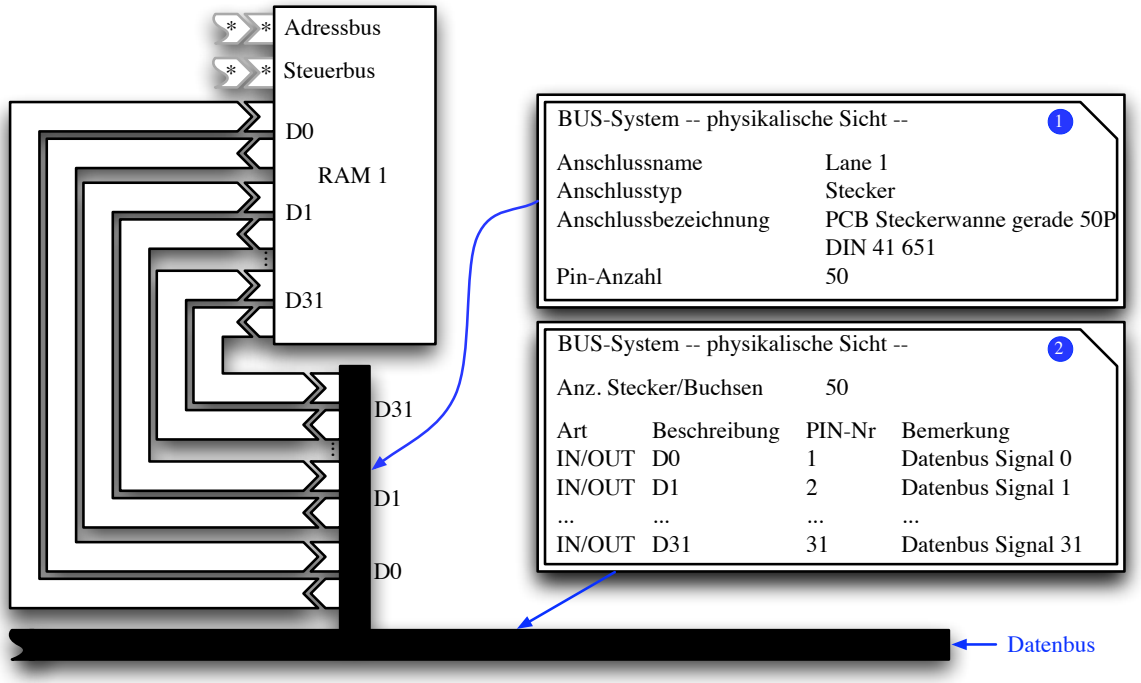


Abbildung 3.144.: Daten-BUS und Komponente *RAM 1* mit Beschreibungsfeldern.

Das physikalische Beschreibungsfeld des BUS-System-Balkens (1) beschreibt den realen Anschluss des BUS-Systems. Er besteht aus einer „PCB Steckerwanne gerade 50P“ nach DIN 41 651 und hat 50 Anschlüsse (Pins). Des Weiteren heißt dieser Anschluss *Lane 1* im Modell. Im zweiten Beschreibungsfeld (2) ist der physikalische Aufbau des BUS-System-Pfeils *Daten-BUS* spezifiziert. In (2) wird die Art, die Beschreibung und eine optionale Bemerkung zu jedem einzelnen Pin des BUS-System-Pfeils definiert. Zusammen bilden die beiden physikalischen Beschreibungsfelder des BUS-Systems die vollständige Beschreibung des physikalischen BUS-Systems. □

3.6.3. Das Optionalitätsprinzip der (U)CML

Das Optionalitätsprinzip der (U)CML ist wie das Konzept der Beschreibungsfelder (vgl. Kapitel „3.6.2.6 *(U)CML-Beschreibungsfelder*“ ab Seite 200) eine neue Kerneigenschaft der Kompatibilitätsmodellierungssprache (U)CML, die sie gegenüber allen anderen existierenden Modellierungssprachen und -konzepten auszeichnet. Mit Hilfe des Optionalitätsprinzips kann bereits auf Modellebene ein System so entworfen und ausgelegt werden, dass es bestimmte gewünschte Systemeigenschaften wie z.B. Flexibilität, Erweiterbarkeit, Vorwärts- oder Rückwärtskompatibilität unterstützt. Besonders hilfreich ist das Konzept der Optionalität für die Modellierung von Austauschkompatibilität (vgl. Abschnitt „1.4 *Der Kompatibilitätsbegriff*“ ab Seite 18) bzw. die spätere Erweiterbarkeit und Anpassung eines Systems an neue Anforderungen.

Das Hauptziel des Optionalitätsprinzips der (U)CML ist es, verschiedene Varianten bzw. Ausbauformen eines Systems in einem gemeinsamen domänenübergreifenden Modell zu vereinigen und dieses Modell auf Kompatibilität untersuchen zu können. Um dieses Ziel zu erreichen, wurde sowohl die Syntax als auch die Semantik verschiedener (U)CML-Elemente an die neuen Anforderungen angepasst. Zum Beispiel wurden die Ausgangsstecker, die Eingangsbuchsen sowie die Paketschnittstellen der (U)CML angepasst. So muss z.B. ein optionaler Stecker nicht mehr zwingend angeschlossen sein um keinen Fehler im Modell zu verursachen (vgl. „3.6.5.1 *(U)CML-Sprachregelwerk*“ ab Seite 236). Um die neu hinzugekommenen Eigenschaften von Steckern, Buchsen und Paketschnittstellen modellieren zu können, wurden die ursprünglichen Definitionen erweitert und angepasst.

In den nachfolgenden Unterabschnitten diese Kapitels werden die einzelnen Erweiterungen gegenüber den bisher erklärten zwingenden Anschlüssen erläutert. Begonnen wird das Kapitel mit der Analyse existierender Modellierungsansätze für Optionalität in den drei Domänen – Elektrotechnik, Mechanik und Software.

3.6.3.1. Optionalität innerhalb von Domänen

Wie bereits in der Einleitung zur Kompatibilitätsmodellierungssprache (U)CML erwähnt, war und ist es eine wesentliche Anforderung an die (U)CML, eine domänenübergreifende Modellierungssprache für Hard- und Software bereitzustellen. Um die unterschiedlichen Anforderungen aus den drei Domänen – Elektrotechnik, Mechanik und Software – in einer Sprache zu vereinen, wurden Teilkonzepte aus ihnen, die für die Modellierung von Systemen benötigt werden, in (U)CML übernommen und integriert. Im Falle des neu eingeführten Optionalitätsprinzips der (U)CML ist es besonders wichtig, eventuell vorhandene ähnliche Modellierungsansätze aus den drei Domänen aufzufinden und diese in (U)CML zu integrieren, um das „Rad nicht neu zu erfinden“.

Softwaretechnik
Das in (U)CML eingeführte Optionalitätsprinzip ist bereits von anderen Domänen her zum Teil bekannt. So kann z.B. in der Programmiersprache *C* bzw. *C++* eine Methode ein oder beliebig viele Argumente haben (Listing Zeile 1). Im Beispiel kann die Methode `int printf(const char *format ...)` genau ein Argument vom Type `const char *` oder beliebig viele weitere Argumente des selben Typs haben (`...`). Diese Eigenschaft wird in *C* bzw. *C++* als *ellipsis* bezeichnet. In *C++* existiert noch eine weitere Möglichkeit optionale Argumente einer Methode zu beschreiben. In Zeile 8 des Listing ist eine *C++* Methode deklariert `void foo(int a, int b = 10)`, in der das zweite Argument `int b = 10` mit einem Default-Wert versehen ist. Dies bedeutet, dass beim Aufruf der Methode `foo` das erste Argument zwingend und das zweite optional ist, also nicht angegeben werden muss. Wird das zweite Argument angegeben, so wird der angegebene Default-Wert mit dem übergebenen Wert überschrieben. Nähere Informationen zu *ellipsis* sowie *default Werten* in C bzw. C++ finden sie in [Str90, 164ff].

```
1  int printf(const char *format ...) {
2    // Die Funktion printf muss mindestens ein Argument haben. Es können beliebig viele
3    // weitere char* angegeben werden, die von der Funktion alle verarbeitet werden.
4  }
5
6  void foo(int a, int b = 10) {
7    // Der Aufrufer der Methode foo() muss nur den Parameter a zwingend
8    // angeben der Parameter b muss nicht angegeben werden -- er ist optional
9  }
```

Elektrotechnik
Auch in der Elektrotechnik ist das Optionalitätskonzept zum Teil bekannt. In der Schaltungstechnik werden zum Beispiel Ein- oder Ausgänge von elektrischen Baugruppen oder einzelne Pins von Steckern und Buchsen nicht beschalten, weil sie in der gegenwärtigen Ausbaustufe des Systems nicht benötigt werden. Ein Beispiel hierfür ist der *SCSI*-BUS[241], bei dem wurden einige Leitungen in der ursprünglichen Definition der Stecker und Buchsen nicht beschalten, so dass sie in späteren Ausbaustufen verwendet werden konnten, ohne den ursprünglichen Anschluss zu verändern.

Mechanik
In der Mechanik werden ebenfalls optionale Eigenschaften von Baugruppen fest vorgesehen, um Baugruppen flexibel einsetzen zu können. So ist es z.B. üblich, bei Steckern und Buchsen zusätzliche Pins vorzusehen, wie dies auch in der Elektrotechnik gemacht wird (s.o.). Meistens werden jedoch zusätzliche Halterungen, Bohrungen etc. als mechanische Optionen bei einer Baugruppe vorgesehen, um diese möglichst flexibel einsetzen zu können. Ein gutes Beispiel dafür sind die vielen unterschiedlichen Bohrungen in einem 19"Gehäuse um dort Platinen oder ganze Einschübe individuell befestigen zu können.

Ergebnis
Die bereits in den unterschiedlichen Domänen vorhandenen „Optionalitätsansätze“ sind zum Teil in die Kompatibilitätsmodellierungssprache (U)CML übernommen worden. So ist z.B. die aus der Softwareentwicklung bekannte Technik der Default-Werte oder die Vorbelegung von Ein- bzw. Ausgängen aus der Elektrotechnik in die (U)CML eingeflossen.

Es wurden jedoch nicht alle vorhandenen Konzepte in den Sprachschatz der (U)CML übernommen. So wurde z.B. auf das Konzept der ellipsis zugunsten einer sicheren und eindeutigen Notation bzw. Modellierung für die Kompatibilitätsbestimmung bewusst verzichtet. Dies stellt jedoch keine „große“ Einschränkung dar, da z.B. die MISRA ebenfalls von der Benutzung der ellipsis abrät[242].

3.6.3.2. Optionale Paketschnittstellen und optionale Komponentenanschlüsse

In (U)CML ist das Konzept der *optionalen Ein-* und *Ausgänge* sowie der *optionalen Paketschnittstellen* durch die Erweiterung der Syntax und Semantik der Stecker, Buchsen und Paketschnittstellen realisiert worden. Bevor jedoch mit der eigentlichen Definition der optionalen Paketschnittstellen sowie der optionalen Komponentenanschlüsse begonnen werden kann, muss zuerst definiert werden, was eine zwingende Paketschnittstelle bzw. ein zwingender Komponentenanschluss in (U)CML ist. Basierend auf den Definitionen der Standard- bzw. Kommunikationsstecker/-buchsen bzw. Paketschnittstellen (vgl. Def.: 3.17) und der in der Tabelle 3.8 auf Seite 177 eingeführten Nomenklatur, folgt die Definition der zwingenden Buchsen, Stecker sowie Paketschnittstellen.

Definition 3.68 ***Zwingende Paketschnittstelle und zwingender Komponentenanschluss***
Sei

$$Paket\_Name_{Art:Schnittstellenart:Name:(Cond)}$$

die formale Notation der Paketschnittstellenarten und

$$Komponenten\_Name_{Art:Anschlussart:Name:(Cond)}$$

die formale Notation der unterschiedlichen Komponentenanschlussarten der (U)CML, dann gilt für alle ***zwingenden Paketschnittstellen*** *bzw.* ***zwingenden Komponentenanschlüsse***

$$Art = Z \text{ (zwingend).}$$

[241] Das Akronym *SCSI* steht für **S**mall **C**omputer **S**ystem **I**nterface. Nähere Informationen finden sie unter [Wik07u][Dud07].
[242] Siehe hierzu: [Sys07, 37] Rule 69 – „Functions with variable number of arguments shall not be used.“.

Anmerkungen und Folgerungen aus der Definition 3.68:

- Der Index „Z“ kann bei der formalen Notation der Pakete und Komponenten weggelassen werden, wenn eindeutig ist, dass es sich um eine zwingende Paketschnittstelle bzw. einen zwingenden Komponentenanschluss handelt.
- Alle bisher vorgestellten Paketschnittstellen, Komponentenanschlüsse sowie sämtliche Flusspfeilarten waren zwingend.
- Kommunikationsstecker/-buchsen sind atomare Einheiten. Aus diesem Grund haben sie stets eine eindeutige Zuordnung – zwingend oder optional. Eine Trennung in z.B. einen optionalen Hin- und einen zwingenden Rückkanal ist nicht erlaubt.

Nachdem in der Definition 3.68 die Notation der zwingenden Standard- bzw. Kommunikationsstecker/-buchsen bzw. Paketschnittstellen definiert wurde, folgt nun die Definition derjenigen Anschlussarten, die die (U)CML zur Modellierung von optionalen Anschlüssen bereitstellt. In der Definition 3.69 ist die Notation für optionale Anschlüsse definiert, während in Definition 3.70 die graphische Repräsentation im Flussdiagramm dargestellt ist.

Definition 3.69 ***Optionale Paketschnittstelle und optionaler Komponentenanschluss***
Sei

$$Paket\_Name_{Art:Schnittstellenart:Name:(Cond)}$$

die formale Notation der Paketschnittstellenarten und

$$Komponenten\_Name_{Art:Anschlussart:Name:(Cond)}$$

die formale Notation der unterschiedlichen Komponentenanschlussarten der (U)CML, dann gilt für alle ***optionalen Paketschnittstellen*** *bzw.* ***optionalen Komponentenanschlüsse***

$$Art = OPT\ (optional).$$

Definition 3.70 ***Optionale Paketschnittstelle und optionaler Komponentenanschluss (graphische Darstellung)***
Optionale Stecker, Buchsen, Paketschnittstellen, Kommunikationsstecker/-buchsen sowie Kommunikationsschnittstellen werden in (U)CML grau gefüllt dargestellt.

Paket	*optionale Schnittstelle*	
	optionale Kommunikationsschnittstelle	
Komponente	*optionaler Stecker*	
	optionale Buchse	
	optionaler Kommunikationsstecker/-buchse	*links:*  *rechts:* 

Zusammengefasste optionale Stecker, Buchsen, Paketschnittstellen sowie Kommunikationsstecker/-buchsen haben zusätzlich einen Stern in der Mitte.

Wie die normalen zwingenden Anschlüsse der (U)CML können auch optionale Anschlüsse zusammengefasst werden. Es gelten hierfür die selben Voraussetzungen wie für zwingende Anschlüsse (vgl.: Definition 3.27).

Unterschied zwischen optionalen und zwingenden Paketschnittstellen und Komponentenanschlüssen
An einem zwingenden Anschluss (einem zwingenden Stecker, Buchse oder einer Paketschnittstelle) muss ein Flusspfeil angeschlossen sein, sonst verursacht der Anschluss einen Fehler im Modell. Optionale Paketschnittstellen bzw. optionale Komponentenanschlüsse hingegen müssen nicht mit einem Flusspfeil verbunden sein und verursachen dennoch keinen Fehler. Die (U)CML-Sprachregel 3.1 beschreibt genau diesen Sachverhalt[243].

(U)CML Kompatibilitätsregel 3.1 – *(U)CML-Sprachregelwerk* –
Für alle ***zwingenden Paketschnittstellenarten*** *bzw. für alle* ***zwingenden Komponentenanschlussarten*** *der (U)CML gilt: Sie müssen stets mit einer entsprechenden Flusspfeilart verbunden sein, um keinen Fehler im Modell zu verursachen. Im Gegensatz dazu müssen* ***optionale Paketschnittstellenarten*** *bzw.* ***optionale Komponentenanschlussarten*** *nicht mit einer (U)CML-Flusspfeilart verbunden sein.*

Beispiel 70: Anwendung zwingende und optionale Komponentenanschlüsse
Die Abbildung 3.145 auf der nächsten Seite zeigt die (U)CML-Komponente A mit drei Eingangsbuchsen ($I1 \ldots I3$) und zwei Ausgangssteckern ($O1$ und $O2$). Alle drei Eingangsbuchsen sind als zwingend modelliert und dargestellt und müssen aus diesem Grund mit einem Flusspfeil verbunden sein, um keinen (U)CML-Sprachregelfehler zu verursachen. Die dritte dieser Buchsen (1: $A_{Z:Bu_{Std}:I3}$) ist nicht mit einem Flusspfeil verbunden und verursacht aus diesem Grund einen (U)CML-Sprachfehler.

[243] Aus Verständlichkeitsgründen werden in diesem Kapitel einige (U)CML-Sprachregeln vorweggenommen. Eine ausführliche Beschreibung des (U)CML-Sprach-Regelwerks finden Sie im Kapitel „3.6.5.1 *(U)CML-Sprachregelwerk*“ ab Seite 236.

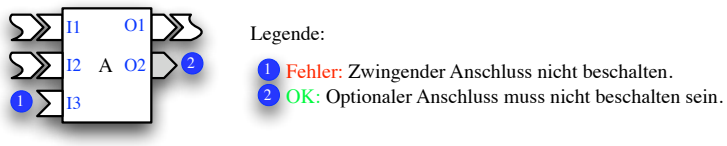


Abbildung 3.145.: Zwingender vs. optionaler Komponentenanschluss.

Die Komponente *A* hat außerdem zwei Ausgangsstecker, wovon einer zwingend und einer optional ist. Der optionale Stecker (2: $A_{OPT:St_{Std}:O2}$) ist nicht mit einem Flusspfeil verbunden und verursacht keinen Fehler, weil optionale Anschlüssen nach der (U)CML-Sprachregel 3.1 nicht beschalten sein müssen. □

Im Anschluss an die Definition der optionalen Paketschnittstellenarten und Komponentenanschlussarten folgt im nächsten Abschnitt die Definition der optionalen Flusspfeile zur Modellierung der Verbindung zwischen den Anschlussarten der (U)CML .

3.6.3.3. Optionale Flusspfeile

Mit Hilfe der optionalen Flusspfeile werden, ähnlich wie bei zwingenden Flusspfeilen, optionale Stecker mit Buchsen bzw. Paketschnittstellen verbunden. Dabei gelten die selben Grundregeln, die auch für die Verbindung von zwingenden Flusspfeilen gelten (vgl.: Kapitel „3.6.2.5 *(U)CML-Flusspfeilarten*“ ab Seite 184 und insbesondere die Definition 3.39). Für die Modellierung und Darstellung der optionalen Flusspfeile kommen jedoch weitere Regeln hinzu. Diese Regeln werden in den nachfolgenden Unterabschnitten dieses Kapitels einzeln erläutert und anhand von einfachen Beispielen anschaulich dargelegt.

Bevor mit der Definition der optionalen Flusspfeile begonnen werden kann, muss ähnlich wie bei der Einführung der optionalen Buchsen/Stecker, die formale Notation der zwingenden Flusspfeile definiert werden. Die Definition 3.71 beschreibt die Notation der zwingenden (U)CML-Flusspfeile aufbauend auf der Grunddefinition für Flusspfeile (Def.: 3.39, sowie Def.: 3.40).

Definition 3.71 ***Zwingende Flusspfeile***
Sei o.B.d.A. ***A*** *die Quelle und* ***B*** *die Senke einer Verbindung. Außerdem sei* ***D*** *das zu übertragende Datum. Dann lassen sich* ***zwingende Standardflusspfeile*** $F_{Z:S_{Name}:D}$ *bzw.* ***zwingende zusammengefasste Flusspfeile*** $F_{Z:SZ_{Name}:D}$ *formal schreiben als*

$$A \xrightarrow{F_{Z:S_{Name}:D}} B \quad mit \quad F_{Z:S_{Name}:D} = zwingender\ Standardflusspfeil$$
$$A \xrightarrow{F_{Z:SZ_{Name}:D}} B \quad mit \quad F_{Z:SZ_{Name}:D} = zwingender\ zusammengefasster\ Flusspfeil$$

und ***zwingende Kommunikationsflusspfeile*** $F_{Z:K_{Name}:D}$ *bzw.* ***zwingende zusammengefassten Kommunikationsflusspfeile*** $F_{Z:KZ_{Name}:D}$

$$A \xrightleftharpoons{F_{Z:K_{Name}:D}} B \quad mit \quad F_{Z:K_{Name}:D} = zwingender\ Kommunikationsflusspfeil$$
$$A \xrightleftharpoons{F_{Z:KZ_{Name}:D}} B \quad mit \quad F_{Z:KZ_{Name}:D} = zwingender\ zusammengefasster\ Kommunikationsflusspfeil$$

in der verkürzten Schreibweise schreiben.

Anmerkungen und Folgerungen aus der Definition 3.71:

- Der Index „*Z*“ kann weggelassen werden, wenn eindeutig ist, dass es sich bei der Verbindung um eine zwingende Verbindung handelt.
- Die Quelle bzw. die Senke einer Verbindung kann ein Paket oder eine Komponente sein.
- Alle bisher eingeführten Verbindungen waren zwingende Verbindungen.
- Für alle zusammengefassten Verbindungen gilt: Es lassen sich nur gleichartige Verbindungen zu einer zusammengefassten Verbindung zusammenfassen. Beispielsweise kann ein zwingender Flusspfeil nicht mit einem optionalen Flusspfeil zusammengefasst werden.

Beispiel 71: Zwingende Verbindungen
Eine zwingende Verbindung zwischen zwei Paketen oder Komponenten wird in (U)CML als weißer Pfeil dargestellt. In der Abbildung 3.146 ist eine zwingende Verbindung zwischen den Komponenten (1) *A* und *B* bzw. (2) den Paketen *C* und *D* in „Black-Box“ Darstellung dargestellt.

Die Flussverbindung (1) zwischen Komponente *A* und Komponente *B* kann nach Definition 3.71 formal

$$A_{Z:St_{Std}:O1} \xrightarrow{F_{Z:S:D}} B_{Z:Bu_{Std}:I1}$$

geschrieben werden. Ist es eindeutig, dass die Verbindung zwischen den beiden Komponenten zwingend ist, so kann verkürzt

$$A_{St_{Std}:O1} \xrightarrow{F_{S:D}} B_{Bu_{Std}:I1}$$

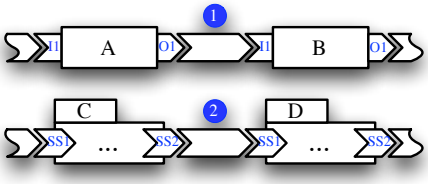


Abbildung 3.146.: Zwingender Flusspfeil zwischen zwei Komponenten (1) bzw. zwei Pakten (2).

geschrieben werden. Die Flussverbindung (2) zwischen den beiden Paketen *C* und *D* kann formal geschrieben werden:

$$C_{Z:SS_{Std}:SS2} \xrightarrow{F_{Z:S:D}} D_{Z:SS_{Std}:SS1}$$

□

Aufbauend auf der Definition der zwingenden (U)CML-Flusspfeile (Def.: 3.71) folgt nun die Definition der formalen Notation der optionalen Flusspfeile (Def.: 3.72). Daran anschließend wird die graphische Darstellung der optionalen Flusspfeile in der Flussansicht eingeführt.

Definition 3.72 ***Optionale Flusspfeile***
Sei o.B.d.A. ***A*** *die Quelle und* ***B*** *die Senke einer Verbindung. Außerdem sei* ***D*** *das zu übertragende Datum. Dann lassen sich* ***optionale Standardflusspfeile*** $F_{OPT:S_{Name}:D}$ *bzw.* ***optionale zusammengefasste Flusspfeile*** $F_{OPT:SZ_{Name}:D}$ *formal schreiben als*

$$A \xrightarrow{F_{OPT:S_{Name}:D}} B \quad mit \quad F_{OPT:S_{Name}:D} = optionaler\ Standardflusspfeil$$
$$A \xrightarrow{F_{OPT:SZ_{Name}:D}} B \quad mit \quad F_{OPT:SZ_{Name}:D} = optionaler\ zusammengefasster\ Standardflusspfeil$$

und ***optionaler Kommunikationsflusspfeile*** $F_{OPT:K_{Name}:D}$ *bzw.* ***optionaler zusammengefasster Kommunikationsflusspfeil*** $F_{OPT:KZ_{Name}:D}$

$$A \xrightleftharpoons{F_{OPT:K_{Name}:D}} B \quad mit \quad F_{OPT:K_{Name}:D} = optionaler\ Kommunikationsflusspfeil$$
$$A \xrightleftharpoons{F_{OPT:KZ_{Name}:D}} B \quad mit \quad F_{OPT:KZ_{Name}:D} = optionaler\ zusammengefasster\ Kommunikationsflusspfeil$$

Definition 3.73 ***Optionale Flusspfeile (graphische Darstellung)***
Optionale Flusspfeile, also optionale Standardflusspfeile sowie Kommunikationsflusspfeile, werden in (U)CML grau gefüllt dargestellt.

Standardflusspfeil

Kommunikationsflusspfeil

Zusammengefasste optionale Flusspfeile haben zusätzlich einen Stern in der Mitte.

Regeln zur Modellierung und Darstellung von optionalen Flusspfeilen
Wie bereits in der Einleitung zu diesem Kapitel erwähnt, gelten für die Modellierung und Darstellung von optionalen Flusspfeilen in (U)CML zusätzliche Regeln, die über die der zwingenden Flusspfeile hinaus gehen. In der Abbildung 3.147 ist ein optionaler Standardflusspfeil dargestellt, der den optionalen Ausgangsstecker *O1* der Komponente *A* mit der optionalen Eingangsbuchse *I1* der Komponente *B* (1) verbindet. Im unteren Teil der Abbildung sind zwei Pakete in Black-Box Darstellung abgebildet (2). Die Paketschnittstelle des Pakets *C SS2* ist mit der Paketschnittstelle *SS1* des Pakets *D* über einen optionalen Standardflusspfeil verbunden.

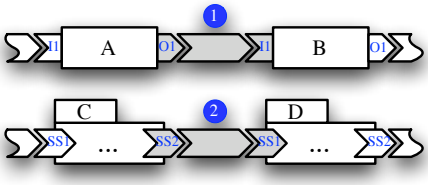


Abbildung 3.147.: Optionaler Flusspfeil zwischen zwei Komponenten (1) bzw. zwei Pakten (2).

Der optionale Flusspfeil (1) zwischen der Komponente *A* und der Komponente *B* kann nach Definition 3.72 formal

$$A_{OPT:St_{Std}:O1} \xrightarrow{F_{OPT:S:D}} B_{OPT:Bu_{Std}:I1}$$

geschrieben werden. Der Flusspfeil zwischen den beiden Paketen *C* und *D* kann auf die selbe Art formal dargestellt werden.
Die (U)CML-Sprachregel 3.2 beschreibt den in der Abbildung 3.147 dargestellten Sachverhalt.

(U)CML Kompatibilitätsregel 3.2 – *(U)CML-Sprachregelwerk*[244]–
*Wird ein **optionaler** Ausgangsstecker mit einer **optionalen** Eingangsbuchse über einen Standardflusspfeil verbunden, so wird die gesamte Verbindung grau, also **optional** dargestellt. Der Flusspfeil erbt diese Eigenschaft der beiden Anschlüsse.*

Die Regel 3.2 gilt insbesondere auch für Kommunikationsstecker/-buchsen sowie für zusammengefasste Verbindungen. In einem (U)CML-Modell kann es auch vorkommen, dass ein zwingender Ausgangsstecker mit einer optionalen Eingangsbuchse verbunden werden soll. Die Regel 3.3 beschreibt, wie in diesem Fall vorgegangen wird.

(U)CML Kompatibilitätsregel 3.3 – *(U)CML-Sprachregelwerk* –
*Wird o.B.d.A. ein **zwingender** Ausgangsstecker mit einer **optionalen** Eingangsbuchse mit einem Standardflusspfeil verbunden, so wird die gesamte Verbindung weiß, also **zwingend** dargestellt. Im Modell bzw. in den entsprechenden Beschreibungsfeldern bleibt die ursprüngliche Information erhalten.*

Die Abbildung 3.148 illustriert die Verbindung von zwei Komponenten *A* und *B*. Die Komponente *A* hat einen zwingenden Ausgangsstecker ($A_{Z:St_{Std}:O1}$) während die Komponente *B* eine optionale Eingangsbuchse ($B_{OPT:Bu_{Std}:I1}$) hat (1). Wird nun der Ausgangsstecker der Komponente *A* mit der optionalen Eingangsbuchse der Komponente *B* über einen Standardflusspfeil verbunden ($A \rightarrow B$), so wechselt die optionale Eingangsbuchse der Komponente *B* ihre Farbe von grau auf weiß (2). Im Beschreibungsfeld der Eingangsbuchse der Komponente *B* bleibt der Eintrag *optional* erhalten, so dass, die Verbindung, wenn sie gelöst werden sollte, wieder auf *optional* zurückgesetzt werden kann.

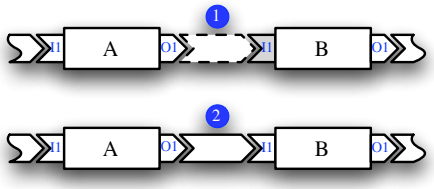


Abbildung 3.148.: Verbindung eines zwingenden Ausgangssteckers mit einer optionalen Eingangsbuchse.

Formal wird die Verbindung zwischen den beiden Komponenten *A* und *B* geschrieben

$$A_{Z:St_{Std}:O1} \xrightarrow{F_{Z:S:D}} B_{OPT:Bu_{Std}:I1}$$

Aus der formalen Darstellung ist stets ersichtlich, dass der Stecker der Komponente *A* zwingend und die Buchse der Komponente *B* optional ist. Der Flusspfeil zwischen den beiden Komponenten ist zwingend, weil einer der beiden Anschlüsse zwingend ist (vgl. (U)CML-Sprachregel 3.3).

Das beschriebene Szenario gilt ebenfalls für die Verbindung von Paketschnittstellen sowie Kommunikationsverbindungen. Sobald ein Stecker oder eine Buchse zwingend modelliert ist und diese mit einem optionalen Stecker oder Buchse verbunden wird, wird die gesamte Verbindung als zwingend eingestuft. Wenn die Verbindung über eine Paketgrenze hinweg über eine Schnittstelle geführt wird, so wird auch diese ebenfalls eingefärbt. In der Abbildung 3.149 auf der nächsten Seite sind sämtliche Kombinationen von Verbindungen über eine Paketschnittstelle hinweg für einen Standardflusspfeil dargestellt.

Alle in der Abbildung 3.149 illustrierten möglichen Verbindungsvarianten, über eine oder mehrere Paketschnittstellen, gelten ebenfalls für alle anderen Verbindungsarten, die die (U)CML zur Verfügung stellt.

Beispiel 72: Austausch einer Komponente mit optionalem Ausgang
Szenario: In einer Baugruppe soll eine alte Komponente gegen eine neuere Komponente mit mehr Ein- und Ausgängen ausgetauscht werden, da die „Funktionalität“ der ursprünglichen Baugruppe nicht mehr den gesteigerten Anforderungen genügt. Dabei sollen die äußeren Schnittstellen der ursprünglichen Baugruppe nicht verändert werden, da diese bereits in dieser Ausbaustufe einen nicht beschalteten Ausgang hat. Dieser soll in der neuen Ausbaustufe benutzt werden, um das neue Leistungsmerkmal nach außen zu reichen.

In der Abbildung 3.150 auf der nächsten Seite ist auf der linken Seite die alte Baugruppe als (U)CML-Modell beschrieben. Das Paket *Baugruppe Vers. A* repräsentiert dabei die komplette Baugruppe mit den vorhandenen Schnittstellen nach außen. Die nicht beschaltete optionale Schnittstelle *SS4* ist im (U)CML-Modell als *optionale* Paketschnittstelle (2) modelliert worden. Komponente *K1* hat ebenfalls einen optionalen Ausgangsstecker *O3* (1), der im aktuellen Bauzustand jedoch nicht benutzt wird.

[244] Das (U)CML-Sprachregelwerk wird im Kapitel „3.6.5.1 *(U)CML-Sprachregelwerk*“ ab Seite 236 beschrieben.

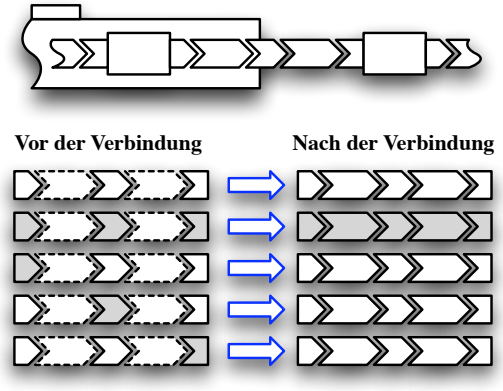


Abbildung 3.149.: Verbindungsarten zwischen Ausgangssteckern und Eingangsbuchsen.

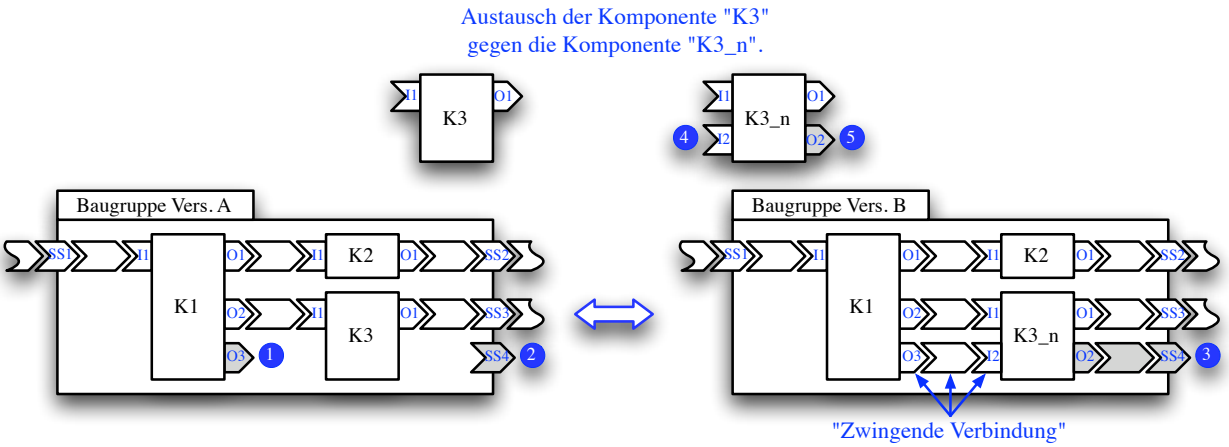


Abbildung 3.150.: Austausch der Komponente *K3* gegen die Komponente *K3_n* mit optionalem Ausgang.

Die nicht mehr zeitgemäße Komponente *K3* des Pakets *Baugruppe Vers. A* soll gegen die neue Komponente *K3_n* ausgetauscht werden um das gewünschte neue Leistungsmerkmal in die Baugruppe integrieren zu können. Die neue Komponente *K3_n* hat gegenüber der alten Komponente *K3* eine weitere zwingende Eingangsbuchse *I2* (4) und einen optionalen Ausgangsstecker *O2* (5). Auf der rechten Seite der Abbildung ist die veränderte Baugruppe *Baugruppe Vers. B* mit der neuen Komponente *K3_n* dargestellt. Aus dem optionalen Ausgangsstecker der Komponente *K1 O3* wurde durch die Verbindung mit der zwingenden Eingangsbuchse *I2* (4) der Komponente *K3_n* eine zwingende Verbindung

$$K1_{OPT:St_{Std}:O3} \xrightarrow{F_{Z:S:D}} K3_{n\ Z:Bu_{Std}:I2}.$$

Im darunter liegenden Modell bleibt die Information, dass die Komponente *K1* einen optionalen Ausgang hat, unverändert erhalten. Nur in der graphischen Repräsentation wurde der Ausgang weiß eingefärbt. Der optionale Ausgang der Komponente *K3_n O2* (5) wurde mit der ebenfalls optionalen Schnittstelle *SS4* (2) bzw. (3) verbunden. Da die Schnittstelle jedoch außerhalb des Pakets *Baugruppe Vers. B* nicht verbunden ist, bleibt die gesamte Verbindung optional. Soll das neue Leistungsmerkmal nun eingesetzt werden, muss die optionale Schnittstelle (3) beschalten werden. Dann wird diese Verbindung ebenfalls als zwingende Verbindung dargestellt.

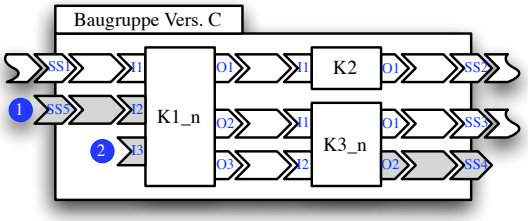


Abbildung 3.151.: Erweiterung des Modells aus Abb. 3.150 um zwei weitere optionale Eingänge.

Durch die Art der Modellierung der Schnittstelle (3) kann die *Baugruppe Vers. B* in jedem alten System verbaut werden, ohne dass es zu einer Inkompatibilität kommt, da die Baugruppe nur intern verändert wurde, die Schnittstelle nach außen jedoch nicht. Durch den Einsatz optionaler Ein- und Ausgänge sowie optionaler Schnittstellen ist es möglich, ein System flexibel zu modellieren, so dass es wiederverwendet werden kann. In der Abbildung 3.151 auf der vorherigen Seite wurde die Baugruppe noch einmal mit zusätzlichen Erweiterungen modelliert.

Die neue optionale Paketschnittstelle *SS5* (1) des Pakets *Baugruppe Vers. C* ist wie der optionale Ausgang aus der Abbildung 3.150 nicht beschalten, jedoch vollständig intern verbunden. Der optionale Eingang *I3* (2) der erweiterten Komponente *K1_n* ist ebenfalls nicht beschalten. Beide Erweiterungen der Baugruppe *Baugruppe Vers. C* sind optional und müssen somit nicht beschalten sein. Aus diesem Grund ist die neue Bauform zu den anderen beiden Versionen (strikt) kompatibel, weil sich das externe statische und dynamische Verhalten der Baugruppe nicht verändert hat (vgl. „3.6.2.3.2 *Verhalten einer (U)CML-Komponente*“ ab Seite 170). □

3.6.3.4. Optionale Systemeingangs- und -ausgangspfeile

In (U)CML ist es auch möglich, externe Systemeingangs- und -ausgangspfeile als optional zu modellieren, so dass sie nicht zwingend vorhanden sein müssen. Theoretisch würde es ausreichen, wenn lediglich die Paketschnittstellen eines Systempakets optional sind, da in diesem Fall kein externer Flusspfeil angeschlossen sein muss. Es hat sich jedoch gezeigt, dass es sehr hilfreich ist, einen optionalen Systemeingang trotzdem explizit zu modellieren. Dies gilt insbesondere unter Einbeziehung von Default-Werten (siehe hierzu: Kapitel „3.6.3.5 *Optionalität und Default-Werte*“ ab Seite 225).

Die folgende Definition 3.74 beschreibt die formale Notation der externen optionalen Systemeingangs- und -ausgangspfeilarten, aufbauend auf der Definition der zwingenden externen Anschlüsse (Def.: 3.50).

Definition 3.74 ***Optionale externe Systemeingangs- und -ausgangspfeile***
Sei S das ***System*** *und U die* ***Umwelt*** *des Systems. Des Weiteren sei D das zu übertragende Datum. Dann gelten folgende Notationen:*
Externer optionaler Systemeingangspfeil:

$$U_{Art:EXT\_Flusspfeilart:Name:(Cond)} \xrightarrow{F_{Art:U_{IN}:D}} S_{Art:EXT\_Steckerart:Name:(Cond)}$$

Externer optionaler Systemausgangspfeil:

$$U_{Art:EXT\_Flusspfeilart:Name:(Cond)} \xleftarrow{F_{Art:U_{OUT}:D}} S_{Art:EXT\_Steckerart:Name:(Cond)}$$

mit

$$Art = OPT\ (optional).$$

Die Definition 3.74 beschreibt die graphische Repräsentation der externen optionalen Systemeingangs- und -ausgangspfeilarten, wie sie in der (U)CML-Flussansicht dargestellt werden.

Definition 3.75 ***Optionale externe Systemeingangs- und -ausgangspfeile (graphische Darstellung)***
Optionale externe Systemeingangs- und -ausgangspfeile *sowie* ***optionale externe Kommunikationspfeile*** *werden in (U)CML grau gefüllt dargestellt.*

optionaler externer Systemeingangs- und -ausgangspfeil *Eingang links:*  *Ausgang rechts:* 

optionaler externer Systemkommunikationspfeil *Eingang links:*  *Ausgang rechts:* 

Optionale externe zusammengefasste Systemeingangs- und -ausgangspfeile sowie optionale externe zusammengefasste Kommunikationspfeile werden mit einem Stern in der Mitte dargestellt.

Auch für die optionalen externen Anschlüsse gilt: Optionale externe Anschlüsse lassen sich zu zusammengefassten optionalen Anschlüssen verbinden, wenn sie vom gleichen Typ sind.

3.6.3.5. Optionalität und Default-Werte

Zusätzlich zur Einführung der Optionalität für Paketschnittstellen und Komponentenanschlüsse wurde in (U)CML das Konzept der Default-Werte eingeführt. Default-Werte sind die optimale Ergänzung des Optionalitätskonzepts. So kann z.B. ein Stecker oder eine Buchse in einer bestimmten Systemvariante optional und nicht angeschlossen sein und trotzdem einen fest definierten Wert haben.

Definition 3.76 ***Vorgegebene Werte der Paketschnittstellen und Komponentenanschlüsse:***
Alle optionalen Paketschnittstellen und Komponentenanschlüsse können in (U)CML einen vorgegebenen (vordefinierten) Wert[245] *haben, der im Beschreibungsfeld eingetragen ist.*

In der formalen Notation der optionalen Paketschnittstellen und der Komponentenanschlüsse wird der vorgegebene Wert in das Feld Cond mit eingetragen (Def_Wert = Wert).

Folgerungen aus der Definition 3.76:
Durch die feste Zuordnung eines Default-Werts zu einem optionalen Stecker, Buchse oder Paketschnittstelle kann sichergestellt werden, dass falls der Anschluss auf einer Seite nicht beschalten ist, an der gegenüberliegenden Seite stets ein definierter Wert vorliegt, der die korrekte Funktionsweise des Anschlusses sicherstellt. Wird kein Default-Wert bei der Modellierung vorgegeben, so kann es zu versteckten Inkompatibilitäten kommen. Aus diesem Grund sollte jede optionale Paketschnittstellenart sowie jeder optionale Komponentenanschlussart stets mit einem Default-Wert versehen werden, weil dadurch die Überprüfung des Systems auf Kompatibilität erheblich verbessert wird. Auch optionale Systemeingangs- und -ausgangspfeile können mit einem Default-Wert versehen werden. Dieser wird wie bei den optionalen Paketschnittstellenarten bzw. den optionalen Komponentenanschlussarten in das Feld *Cond* mit eingetragen, also z.B.

$$U_{OPT:ES:ES1:(Def\_Wert=10)} \xrightarrow{F_{OPT:U_{IN}:D}} S_{OPT:SS_{Std}:SS1}$$

wobei der optionale externe Systemeingangspfeil *ES1* einen Default-Wert von 10 besitzt – im Gegensatz zu der Paketschnittstelle *SS1* des Systems *S*, die keinen vordefinierten Wert besitzt.

Im nachfolgenden Beispiel ist die Modellierung eines einfachen Systems mit einem optionalen Ausgang und einer optionalen Paketschnittstelle dargestellt und erläutert.

Beispiel 73: Variantenbildung durch optionale Ausgänge mit Default-Werten
In der Abbildung 3.152 ist oben links das Modell einer einfachen Baugruppe dargestellt. Diese Baugruppe ist „klassisch“ modelliert, das heißt, ohne die Nutzung der optionalen Ein-/Ausgänge sowie Paketschnittstellen der (U)CML. In der nachfolgenden Aufzählung sind die unterschiedlichen Ausprägungen des links oben modellierten Ausgangssystems (1) dargestellt, in dem eine optionale Paketschnittstelle mit einem Default-Wert verwendet wird, um die spätere Ausbaufähigkeit der Baugruppe zu gewährleisten.

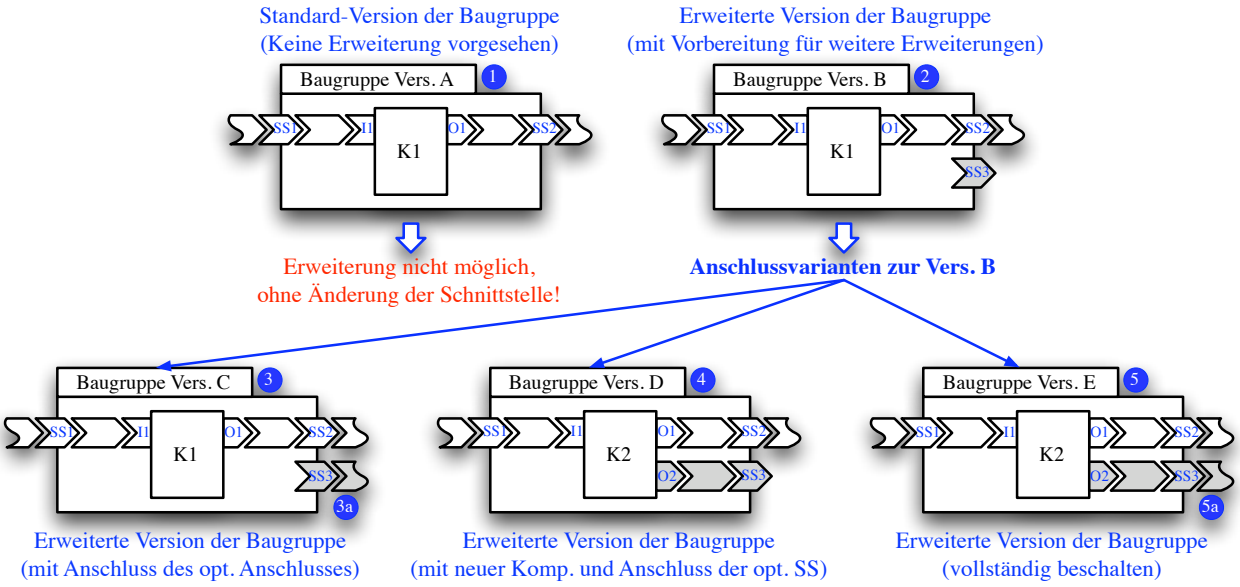


Abbildung 3.152.: Baugruppe mit einer optionalen Paketschnittstelle und Default-Wert.

1. Standardbaugruppe ohne Erweiterungsmöglichkeiten
   Hier ist ein Standardmodell einer in (U)CML modellierten Baugruppe dargestellt. Diese Baugruppe ist klassisch, ohne Erweiterungsmöglichkeiten modelliert worden, so dass eine Erweiterung nur möglich ist, wenn die Schnittstelle des Pakets *Baugruppe Vers. A* nachträglich geändert wird. Dadurch ist die Kompatibilität zur ursprünglichen Version jedoch nicht mehr gewährleistet.

2. Baugruppe mit Erweiterungsmöglichkeiten durch die Nutzung einer optionalen Paketschnittstelle mit Default-Wert
   Im Gegensatz zur *Baugruppe Vers. A* ist bei der Modellierung der *Baugruppe Vers. B* bereits eine Erweiterungsmöglichkeit um einen weiteren Ausgang in Form einer optionalen Paketschnittstelle *SS3* mit Default-Wert vorgesehen. Dadurch lässt sich das

[245] Im englischen wird der „Vorgegebene Wert“ als „Default-Wert“ bezeichnet.

Modell flexibel erweitern, ohne die Schnittstelle des Systems nachträglich ändern oder anpassen zu müssen. Formal kann die optionale Standardpaketschnittstelle *SS3* durch

$$Baugruppe\ Vers.\ B_{OPT:SS_{Std}:SS3:(Def\_Wert=50)}$$

beschrieben werden.

3. Variante C mit angeschlossener optionaler Schnittstelle mit Default-Wert
   An der optionalen Paketschnittstelle *SS3* des Pakets *Baugruppe Vers. C* ist außen der optionale Flusspfeil (3a) angeschlossen, der zu der Buchse einer entfernten Komponente führt (nicht dargestellt). Aufgrund der Tatsache, dass bei der optionalen Paketschnittstelle ein Default-Wert ($Def\_Wert = 50$) angegeben ist, bekommt die angeschlossene Komponente ein Signal zugeschickt, obwohl die Komponente *K1* dieses Signal nicht liefern kann.

4. Variante D mit geändertem inneren Aufbau
   Die Baugruppe in der Ausprägung D enthält die neue Komponente *K2*. Diese Komponente hat einen neuen optionalen Ausgang *O2*, der mit der optionalen Paketschnittstelle *SS3* durch einen Flusspfeils verbunden ist

   $$K2_{OPT:St_{Std}:O2:(Def\_Wert=20)} \xrightarrow{F_{OPT:S:D}} Baugruppe\ Vers.\ D_{OPT:SS_{Std}:SS3:(Def\_Wert=50)}.$$

   Da die Komponente am zweiten Ausgang (*O2*) ebenfalls ein Signal (Default-Wert des optionalen Steckers gleich 20) liefert, wird der Default-Wert der Paketschnittstelle damit überschrieben. Wichtig ist, dass der neu hinzugekommene Ausgang der Komponente *K2* ebenfalls optional ist und einen Default-Wert liefert, solange die Paketschnittstelle *SS3* nach außen nicht verbunden ist.

5. Variante E mit geändertem inneren Aufbau und externem Anschluss
   Das abgebildete Modell stellt eine Erweiterung des in 4 vorgestellten Modells dar. Hier ist an der optionalen Paketschnittstelle *SS3* (5a) wieder eine entfernte Komponente angeschlossen worden. Nun, da eine Flussverbindung von der Komponente *K2* über die optionale Schnittstelle *SS3* (5a) zu der entfernten Komponente besteht, wird der reale Wert, den die Komponente *K2* am zweiten Ausgang ausgibt an die entfernte Komponente übermittelt. Der Default-Wert wird überschrieben.

Ergebnis: Durch optionale Paketschnittstellen mit Default-Wert kann ausgehend von einem Grundmodell eine Vielzahl von unterschiedlichen Varianten modelliert werden. Zusätzlich zur reinen Modellierung kann jede Variante mit Hilfe der (U)CML auf Kompatibilität hin untersucht werden. □

3.6.3.6. Erweiterung des Optionalitätsprinzips auf Pakete und Komponenten

Das Optionalitätsprinzip für Paketschnittstellenarten und Komponentenanschlussarten kann auch auf Pakete und Komponenten erweitert werden. Durch die konsequente Erweiterung des Optionalitätsprinzips auf Pakete und Komponenten ist es möglich, verschiedene Varianten bzw. Konfigurationen eines Grundsystems auf Kompatibilität untersuchen zu können. Dies ist vor allem in der Computer- bzw. der Automobilindustrie von erheblichem Interesse, da es dort ausgehend von einer Grundkonfiguration sehr viele unterschiedliche Varianten geben kann. Im Allgemeinen kommt es bei der Variantenbildung z.B. in der Automobilindustrie, zu mehreren tausenden Konfigurationen, die aufgrund von Kosten- bzw. Komplexitätsgründen nicht alle getestet werden können. In diesem Umfeld ist es möglich, mit Hilfe der (U)CML ein Grundsystem zu beschreiben und dieses auf Kompatibilität zu untersuchen. Im nächsten Schritt können dann verschiedene Konfigurationen, ausgehend vom Grundsystem, untersucht werden. Dazu müssen sämtliche optionalen Baugruppen ebenfalls als (U)CML-Modell vorliegen. Diese können in das Grundsystem integriert und auf Kompatibilität untersucht werden. In der Abbildung 3.153 auf der nächsten Seite ist das hier beschriebene Szenario graphisch dargestellt.

Das Grundsystem ((1) in Abbildung 3.153 auf der nächsten Seite) besteht aus drei Komponenten *K1*, *K2* und *K3*. Die beiden Komponenten *K1* und *K3* haben jeweils einen optionalen Standard Ausgangsstecker ($K1_{OPT:St_{Std}:O3}$ und $K3_{OPT:St_{Std}:O2}$), die im Grundsystem jedoch nicht beschalten sind. Des Weiteren hat das Grundsystem zwei optionale Standardpaketschnittstellen ($Grundsystem_{OPT:SS_{Std}:SS2}$ und $Grundsystem_{OPT:SS_{Std}:SS5}$) zur Umwelt. Diese sind in der Grundversion des Systems ebenfalls nicht beschalten.

In (2) ist die erste Variante des ursprünglichen Grundsystems dargestellt. In dieser Variante wurde die neue Komponente *K4* in das Grundsystem integriert. Durch die Erweiterung des Grundsystems wird die Grundfunktionalität des Systems nicht beeinflusst, d.h. das „neue“ Teilsystem kann nahtlos in ein ansonsten altes System (die Umwelt) eingebaut werden. Wird das Teilsystem (2) jedoch in ein neueres System eingebaut, das die optionalen Paketschnittstellen zur Umwelt belegt, erweitert sich auch das Verhalten des Systems. In (3) ist eine weitere Variante des Grundsystems dargestellt.

Formale Beschreibung der optionalen Standardverbindungen der Variante 2:

- Flusspfeile E2 und F6:

  $$U_{OPT:ES:E2} \xrightarrow{F_{OPT:U_{IN}:D}} Variante\ A_{OPT:SS_{Std}:SS2} \xrightarrow{F_{OPT:S:D}} K4_{OPT:Bu_{Std}:I2}$$

  bzw. in der verkürzten Schreibweise

  $$U \xrightarrow{F_{OPT:U_{IN}:E2:D}} Variante\ A \xrightarrow{F_{OPT:S_{F6}:D}} K4$$

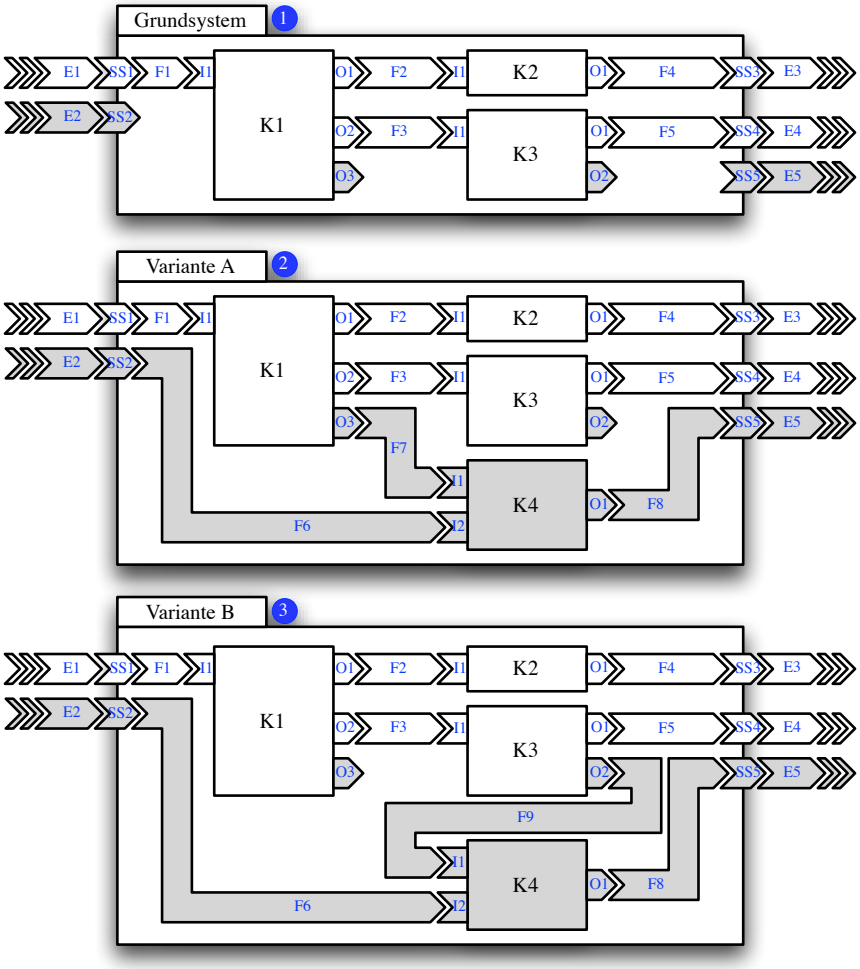


Abbildung 3.153.: Anwendung von optionalen Paketen und Komponenten in einem System.

- Flusspfeil F7:

$$K1_{OPT:St_{Std}:O3} \xrightarrow{F_{OPT:S:D}} K4_{OPT:Bu_{Std}:I1}$$

bzw. in der verkürzten Schreibweise

$$K1 \xrightarrow{F_{OPT:S_{F7}:D}} K4$$

- Flusspfeile F8 und E5:

$$K4_{OPT:St_{Std}:O1} \xrightarrow{F_{OPT:S:D}} Variante\ A_{OPT:SS_{Std}:SS5} \xrightarrow{F_{OPT:U_{OUT}:D}} U_{OPT:ES:E5}$$

bzw. in der verkürzten Schreibweise

$$K4 \xrightarrow{F_{OPT:S_{F8}:D}} Variante\ A \xrightarrow{F_{OPT:U_{OUT:E5}:D}} U$$

Optionale Standardverbindungen der Variante 3 (Verkürzte Schreibweise):

$$K3 \xrightarrow{F_{OPT:S_{F9}:D}} K4$$

Ein weiterer Vorteil, der sich durch die Einführung von optionalen Paketen und Komponenten ergibt, besteht darin, ein System aufbauen zu können, in dem bereits alle Baugruppen enthalten sind jedoch noch nicht freigeschalten wurden. Ein Freischaltesystem für zusätzliche Funktionen, die bereits im Grundsystem enthalten sind, zum gegenwärtigen Zeitpunkt jedoch nicht genutzt werden können, wurde in der Computer- bzw. Automobilindustrie (vgl. Aktivierung/Any Time Upgrade Funktion von Microsoft Windows Vista, Steuergerätefunktionen) bereits erfolgreich eingeführt.

3.6.4. Modellierung von Signalen in (U)CML

Im Kapitel „3.6.2.5 *(U)CML-Flusspfeilarten*“ ab Seite 184 wurden die unterschiedlichen Verbindungsarten, die die (U)CML zur Modellierung von Systemen zur Verfügung stellt, vorgestellt und ihre Grundeigenschaften erläutert, ohne auf die konkrete Verwendung einzugehen. In diesem Kapitel wird nun gezeigt, wie elektrische/elektronische-, mechanische- sowie Softwaresignale in (U)CML modelliert und dargestellt werden können. Begonnen wird der Abschnitt Modellierung von Signalen in (U)CML mit einem kurzen Exkurs zu den Grundlagen der Übertragung von technischen Signalen, die in eingebetteten Systemen vorkommen.

Jede real existierende Verbindung zwischen zwei Baugruppen eines eingebetteten Systems besteht mindestens aus einem *elektrischen/elektronischen* sowie einem *mechanischen* Bestandteil. Beide sind in realen technischen Systemen untrennbar miteinander verbunden, denn für die Übertragung von elektrischen Signalen wird stets ein mechanisches Übertragungsmedium wie z.B. eine Kupferleitung benötigt. Dieser Zusammenhang wird in den meisten Modellierungssprachen vernachlässigt, da sie sich jeweils nur um eine Repräsentation (Sicht), z.B. die elektrischen/elektronischen *oder* die mechanischen Eigenschaften des Systems kümmern und diese modellieren. Zum Beispiel werden elektrische/elektronische Schaltungen mit Hilfe von Stromlaufplänen modelliert und beschrieben, während für die Beschreibung der mechanischen Eigenschaften der selben Baugruppe technische Zeichnungen (CAD) verwendet werden. In modernen eingebetteten softwarelastigen Systemen kommt zu den elektrischen/elektronischen bzw. mechanischen Bestandteilen eines Systems bzw. der Verbindung zwischen den einzelnen Baugruppen des Systems noch die *Softwaretechnik* hinzu. Unter Zuhilfenahme der Softwaretechnik in eingebetteten Systemen können bestimmte Steueraufgaben durch Softwarefunktionen erledigt werden, wodurch die Flexibilität und Anpassbarkeit des eingebetteten Systems gegenüber einer reinen Hardwaresteuerung erheblich zunimmt. Somit sind in einem eingebetteten softwarelastigen System drei unterschiedliche Ansichten einer Verbindung notwendig, um diese vollständig modellieren und beschreiben zu können. In der Abbildung 3.154 wurde der Zusammenhang zwischen der Elektrotechnik, der Mechanik sowie der Softwaretechnik graphisch aufbereitet.

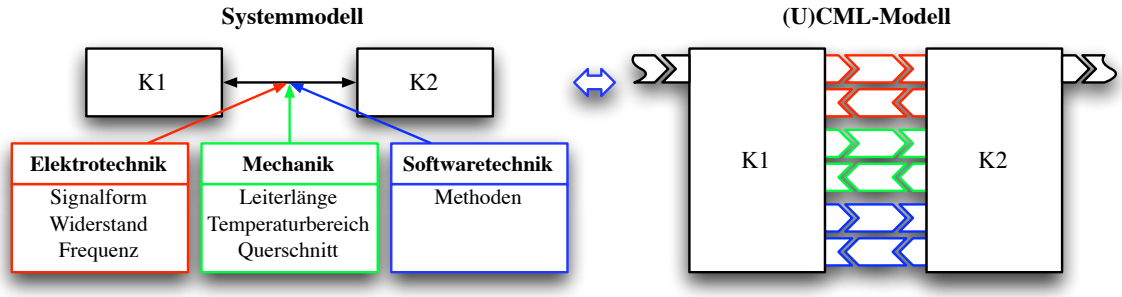


Abbildung 3.154.: Darstellung einer realen Verbindung zwischen zwei Komponenten eines Systems.

Auf der linken Seite der Abbildung sind zwei Baugruppen *K1* und *K2* dargestellt, die durch eine Verbindung miteinander kommunizieren. Die modellierte Verbindung hat im Allgemeinen zwei bzw. drei unterschiedliche Eigenschaften. Zum einen soll über diese Verbindung elektrischer Strom/Spannung übertragen werden und zum anderen hat das Übertragungsmedium (Leitung) verschiedene technische Eigenschaften wie z.B. eine bestimmte Leiterlänge oder einen Querschnitt. In eingebetteten Systemen wird häufig zusätzlich ein Softwaresignal über diese Verbindung übertragen.

Die rechte Seite der Abbildung zeigt das entsprechende (U)CML-Modell des links dargestellten Systemmodells. Im Gegensatz zum Systemmodell auf der linken Seite der Abbildung sind im (U)CML-Modell die drei unterschiedlichen Ausprägungen (Sichten) der Verbindung explizit modelliert und in einem gemeinsamen Modell dargestellt. Im Unterschied zu vielen anderen existierenden Modellierungssprachen für eingebettete softwarelastige Systeme kann in (U)CML die „dreigeteilte Natur“ einer Verbindung explizit in einem Modell modelliert, beschrieben und auf Kompatibilität hin untersucht werden.

Um der zwei- bzw. dreigeteilten Natur von realen technischen Verbindungen gerecht zu werden, wird in den nachfolgenden Unterabschnitten jede Modellierungsart für Signale in (U)CML einzeln angesprochen und anhand von einfachen Beispielen erläutert.

- Modellierung von mechanischen Signalen in (U)CML.
- Modellierung von Softwaretechniksignalen in (U)CML.
- Modellierung von elektrischen/elektronischen Signalen in (U)CML.

Als Grundlage für dieses Kapitel dient das Kapitel „2.7 *Objektorientierte Modellierung kompatibilitätsrelevanter Eigenschaften von eingebetteten Systemen*“ ab Seite 104, in dem die Grundeigenschaften von elektrischen/elektronischen, mechanischen sowie Softwaretechniksignalen vorgestellt und erläutert wurden. Begonnen wird die Beschreibung der unterschiedlichen Modellierungstechniken für Signale in (U)CML mit der Modellierung von mechanischen Signalen.

3.6.4.1. Modellierung von mechanischen Signalen in (U)CML

In modernen Modellierungsumgebungen für technische Systeme wie z.B. CAD Systemen werden mechanische Eigenschaften, die für die Fertigung bzw. die Montage des Systems notwendig sind, wie beispielsweise die Abmessungen oder das verwendete Material eines Bauteils, hinterlegt und gespeichert. In den technischen Zeichnungen sind jedoch nicht alle für die Kompatibilitätsbestimmung notwendigen Bauteileigenschaften bzw. die Wechselwirkungen zwischen den unterschiedlichen

Bestandteilen des Systems explizit modelliert. Für die Modellierung und Bestimmung der mechanischen Kompatibilität zweier Baugruppen sind die mechanischen Eigenschaften der Baugruppen bzw. die mechanische Wechselwirkung der Baugruppen von entscheidender Bedeutung.

In der Abbildung 3.155 ist auf der linken Seite ein einfaches modular aufgebautes System, bestehend aus den beiden Baugruppen *Steckkarte* und *Einschub* dargestellt, wie es heutzutage in fast jedem modernen Rechner vorkommt. Jede der beiden Baugruppen hat spezielle mechanische Eigenschaften, wie z.B. die maximal zulässige Temperatur der Baugruppe. Um die thermische Kompatibilität zwischen den beiden Baugruppen modellieren und verifizieren zu können, werden virtuelle (nur im Modell vorhandene) Verbindungen zwischen den beiden Baugruppen in (U)CML modelliert.

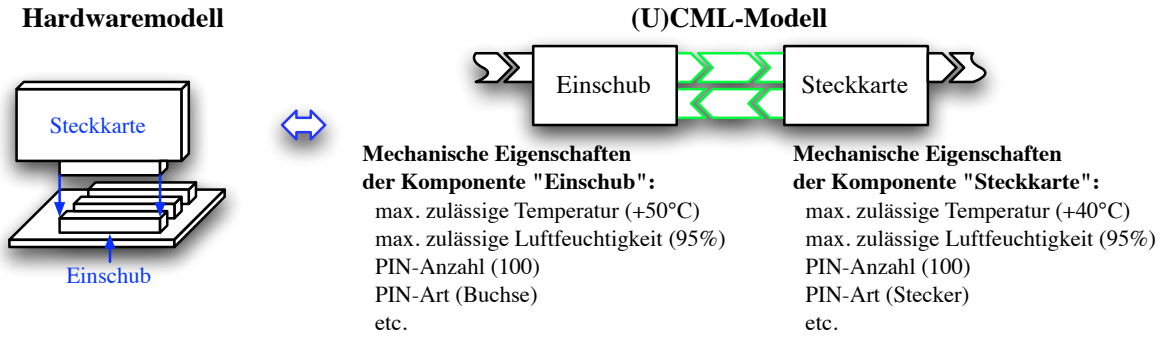


Abbildung 3.155.: Austausch von mechanischen Eigenschaften zwischen zwei Komponenten über einen mechanischen Kommunikationsflusspfeil.

Die Abbildung 3.155 zeigt auf der rechten Seite das selbe System wie links als (U)CML-Modell. Dieses System besteht ebenfalls aus zwei (U)CML-Komponenten *Einschub* und *Steckkarte*, die durch einen zusammengesetzten mechanischen Kommunikationspfeil verbunden sind. Über diese Verbindung tauschen die beiden Komponenten auf Modellebene die kompatibilitätsrelevanten mechanischen Eigenschaften aus. Im Beispiel (Abb.: 3.155) sind dies:

- der maximal zulässige Temperaturbereich,
- die maximal zulässige Luftfeuchtigkeit,
- die PIN-Anzahl sowie die PIN-Art.

Durch den Austausch der Eigenschaften der beiden Komponenten – initiiert durch die Komponente *Einschub* – kann die Kompatibilität der Baugruppe *Steckkarte* zur Baugruppe *Einschub* bereits auf Modellebene sichergestellt werden. Im vorgestellten Modell ist z.B. die maximal zulässige Temperatur der Komponente *Steckkarte* um $10^{\circ}C$ kleiner als die der Komponente *Einschub*. Dies verursacht jedoch keine Inkompatibilität da die maximal zulässige Temperatur des Einschubs nicht erreicht wird und somit die kompatibilitätsrelevanten Eigenschaften des Einschubs („Empfänger“) nicht verletzt werden.

3.6.4.2. Modellierung von Softwaretechniksignalen in (U)CML

Um Software bzw. Softwaresignale in (U)CML zu überführen, müssen sämtliche Softwarestrukturen des Programms in entsprechende (U)CML-Konstrukte überführt werden. In diesem Kapitel wird anhand eines einfachen C++ Programmfragments die Überführung von Software bzw. Softwaresignalen in (U)CML exemplarisch ausgeführt[246]. Begonnen wird mit der Beschreibung des generischen Vorgehens bei der Transformation von Software in ein (U)CML-Modell:

1. *Überführung der Struktur des Programms (Klassen und Hauptprogramm bzw. Methoden zum Programmstart) in (U)CML-Pakete und -Komponenten*
   Zuerst wird die Struktur des Programms, also die Klassen sowie das Hauptprogramm, in (U)CML-Komponenten überführt. Dabei werden die Klassennamen als Komponentennamen verwendet (vgl. Abb.: 3.156). Sollen mehrere Klassen zu einer Struktur zusammengefasst werden, so kann dies mit Hilfe der (U)CML-Pakete erfolgen.

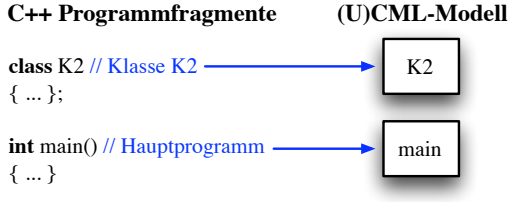


Abbildung 3.156.: Überführung der Struktur eines Programms in ein (U)CML-Modell.

[246] Der hier vorgestellte generische Transformationsprozess für Software kann theoretisch auf alle Programmiersprachen angewendet werden.

2. *Modellierung der Datenstrukturen und Signaturen der Methoden der Klassen als (U)CML-Stecker und -Buchsen*
   In diesem Modellierungsschritt wird die innere Struktur der Klassen, die Signatur der Methoden und Datenstrukturen, in die zuvor erzeugten Komponenten eingetragen und für jede sichtbare Methode bzw. Datenstruktur der ursprünglichen Klasse ein (U)CML-Stecker/Buchse erzeugt (vgl. Abb.: 3.157).

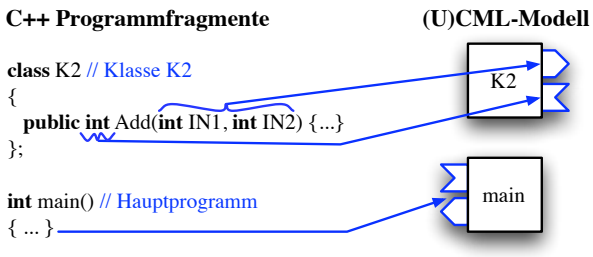


Abbildung 3.157.: Modellierung der Signatur einer Methode in (U)CML.

In Abhängigkeit von der Signatur der zu transferierenden Methode wird entweder ein Kommunikationsstecker/-buchse, ein Stecker oder eine Buchse an der Komponente erzeugt. Kommunikationsstecker/-buchsen werden verwendet, wenn die Methode sowohl Parameter als auch ein Ergebnis hat (Abb.: 3.157). Beim direkten Beschreiben (3) einer Datenstruktur sowie bei „Getter-Methoden“ (1) wird an der Komponente ebenfalls ein Kommunikationsstecker/-buchse erzeugt. Bei „Setter-Methoden“ wird eine Buchse an der Komponente angebracht (2). In der Abbildung 3.158 sind die drei möglichen Modellierungsarten graphisch dargestellt.

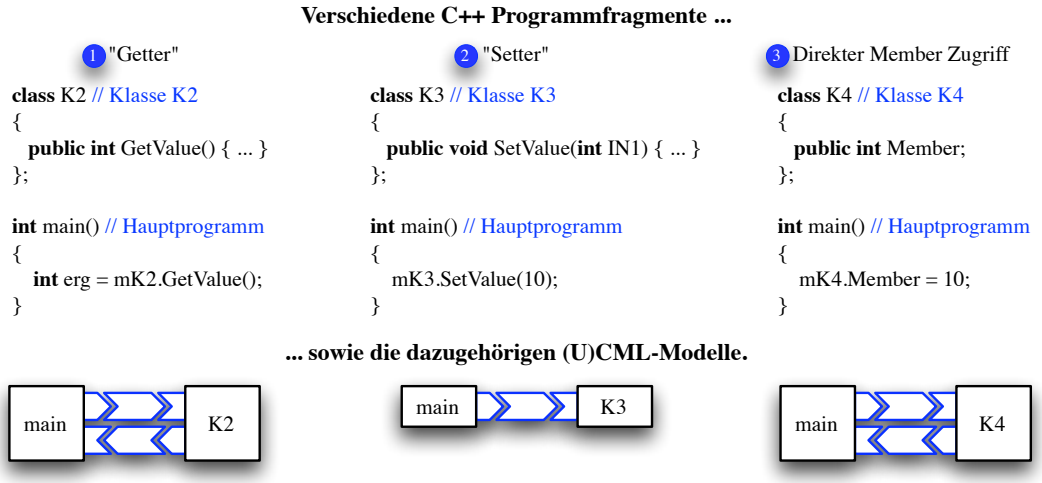


Abbildung 3.158.: Modellierung von Methoden in (U)CML.

3. *Modellierung der Aufrufstruktur der Methoden durch (U)CML-Flusspfeile*
   Nachdem die Struktur der Klassen inklusive der darin enthaltenen Methoden und Datenstrukturen nach (U)CML transferiert worden sind, werden die Methodenaufrufe als Flusspfeile zwischen den einzelnen Anschlüssen modelliert (vgl. Abb.: 3.159).

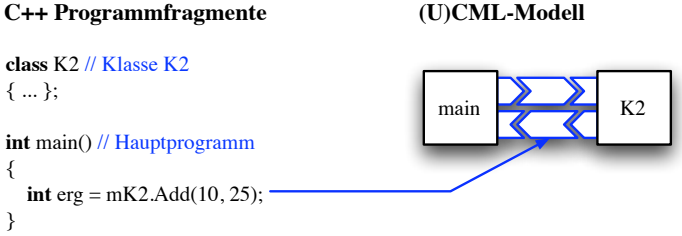


Abbildung 3.159.: Modellierung der Aufrufstrukturen in (U)CML.

4. *Befüllung der (U)CML-Beschreibungsfelder von Steckern und Buchsen*
   Abgeschlossen wird die Transformation der Software bzw. die Modellierung von Signalen durch die Befüllung der (U)CML-Beschreibungsfelder mit den Signaturen der Methoden. Außer den Signaturen der Methoden können noch zusätzliche Werte wie z.B. der Definitionsbereich eines Parameters oder die Einheit in das Beschreibungsfeld eingetragen werden.

In realen Programmen sind nicht alle Werte und Definitionen vorhanden, die in das Beschreibungsfeld eingetragen werden können. Für die Kompatibilitätsbestimmung gilt jedoch der Grundsatz: Je mehr Informationen über eine Methode vorhanden sind, desto genauer kann sie auf Inkompatibilitäten hin untersucht werden.

5. *Modellierung des Verhaltens von Verbindungen des Systems als MSCs*
   Nachdem die statische Struktur der Verbindungen zwischen den beiden Komponenten *main* und *K2* mittels eines Standardkommunikationsflusspfeils modelliert worden ist, muss mit Hilfe der MSCs das Verhalten der Verbindung spezifiziert werden. Die Abbildung 3.160 zeigt das MSC der obigen Verbindung.

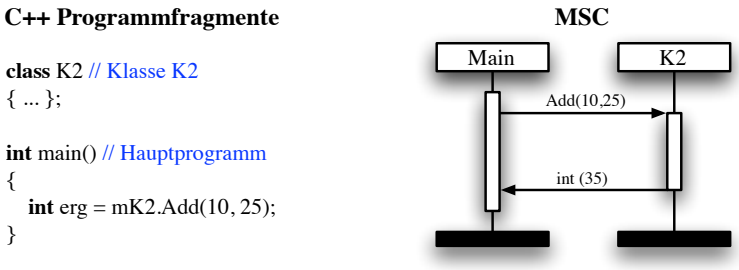


Abbildung 3.160.: MSC eines Softwaresignals.

Bei der expliziten Modellierung der Flussverbindungen zwischen den Steckern und Buchsen der Komponenten kommt es häufig vor, dass ein Stecker-/Buchsenpaar von vielen unterschiedlichen Komponenten aufgerufen wird. Ein (U)CML-Flusspfeil repräsentiert jedoch stets eine Punkt-zu-Punkt Verbindung (vgl. Def.: 3.38); er besitzt keine Abzweigungen und verbindet stets genau einen Stecker mit genau einer Buchse. Aus diesem Grund muss für jeden Aufrufer eine dezidierte Kopie des benötigten Anschlusses erzeugt werden. Die Abbildung 3.161 illustriert den geschilderten Fall anhand eines C++ Methodenaufrufs.

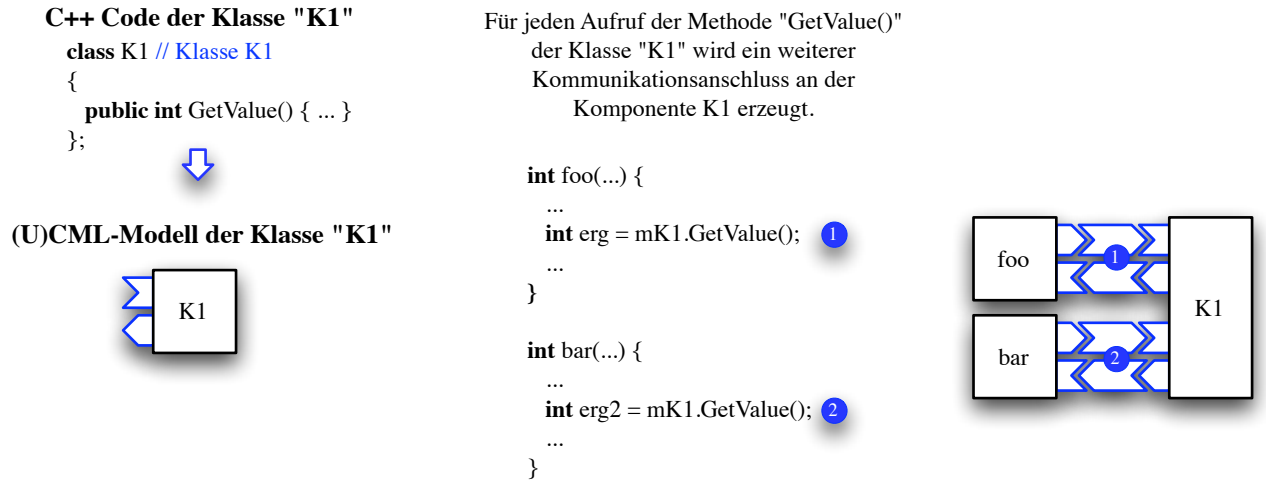


Abbildung 3.161.: Aufruf einer Methode von mehr als einem Aufrufer.

Beispiel 74: Modellierung von Softwaresignalen in (U)CML
Im nachfolgenden Listing ist ein kleiner Ausschnitt aus einem C++ Programm abgebildet. Dieses Listing soll in ein entsprechendes (U)CML-Modell überführt werden. Dabei wird nach dem oben eingeführten Schema vorgegangen.

```
1  // Klasse K2
2  class K2
3  {
4    ...
5    // Methode zur Addition zweier integer-Zahlen
6    // Gültigkeitsbereich: IN1, IN2 [0 ... 50[
7    // Ergebnis: [0 ... 100[
8    public int Add(int IN1, int IN2)
9    {
10     return IN1 + IN2;
11   }
12   ...
13 };
14
```

```
15  // Hauptprogramm
16  int main()
17  {
18    ...
19    mK2 = new K2();
20    ...
21    int erg = mK2.Add(10, 25);
22    ...
23  }
```

Die Abbildung 3.162 zeigt das Ergebnis des abgeschlossenen Transformationsprozesses, der das C++ Listing in ein entsprechendes (U)CML-Modell überführt. Aus der C++ Klasse `K2` mit der öffentlichen Methode `Add` wurde die (U)CML-Komponente *K2* mit einem Kommunikationsstecker/-buchse für die C++ Methode `Add` erzeugt (Abb. 3.162 rechts). Aus dem Hauptprogramm `int main()` wurde die (U)CML-Komponente *main* mit dem entsprechenden Methodenaufruf `int erg = mK2.Add(10,25);` erzeugt. Anschließend wurden die beiden Kommunikationsstecker/-buchsen mit einem Kommunikationsflusspfeil verbunden.

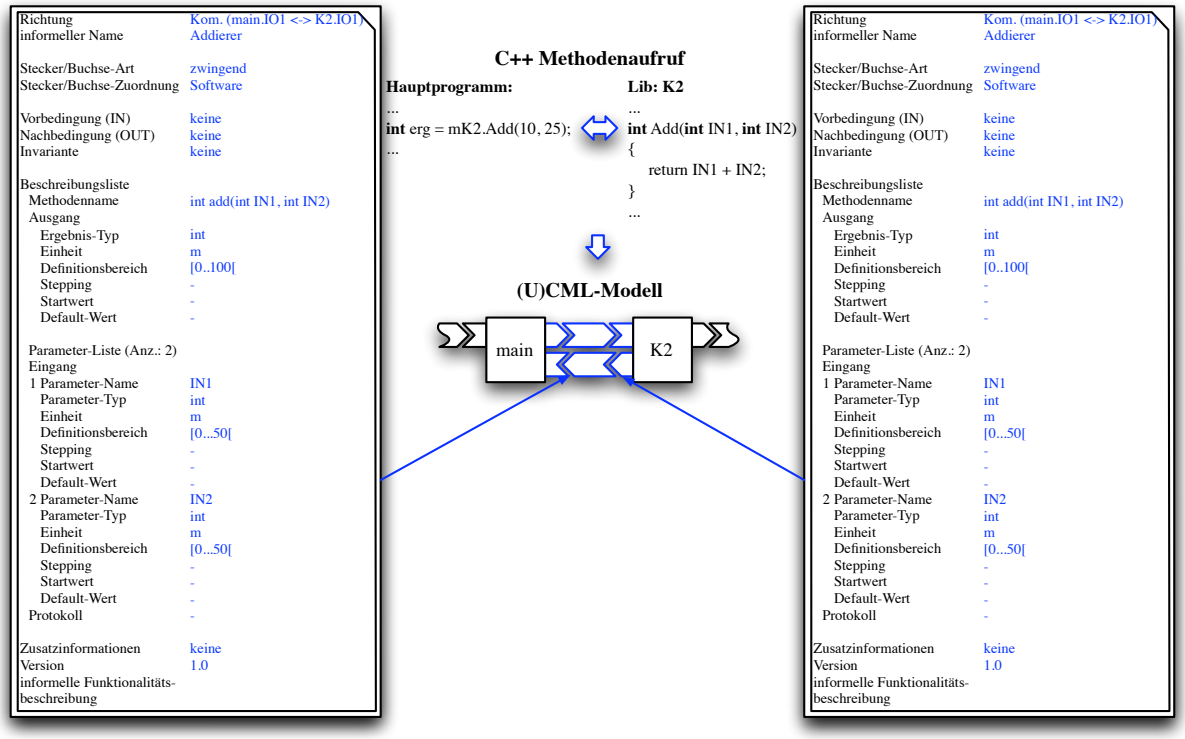


Abbildung 3.162.: Transformation eines Softwaretechniksignals in ein (U)CML-Modell.

Dabei wurde die C++ Methode `Add` als zusammengesetzter Kommunikationspfeil zwischen den beiden Komponenten *main* und *K2* modelliert. Der obere Teil des Kommunikationsflusspfeils (Hinrichtung) überträgt den Methodennamen (`Add`) sowie die Parameter (`IN1, IN2`) von der Komponente *main* zur Komponente *K2*. Diese führt die Berechnung aus und liefert das Ergebnis über den unteren Kommunikationsflusspfeil (Rückrichtung) an die Komponente *main* zurück. Der verbale Programmablauf wird nun als MSC modelliert, in dem die exakte Aufrufreihenfolge zwischen den beiden Komponenten *main* und *K2* genau spezifiziert ist. Die Abbildung 3.163 zeigt das entsprechende MSC.

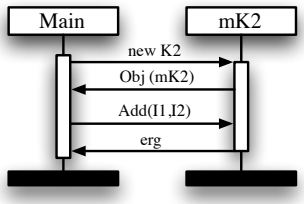


Abbildung 3.163.: MSC der Kommunikationsverbindung zwischen den Komponenten *main* und *K2*.

Zunächst wird die Komponente *K2* durch die Komponente *main* erzeugt. Im Anschluss folgt der Methodenaufruf `int erg = mK2.Add(10,25).`

Anmerkung:
In (U)CML kann die Erzeugung einer Komponente weggelassen werden, da davon ausgegangen wird, dass bereits bei der Modellierung sämtliche Komponenten erzeugt worden sind. Soll jedoch ausdrücklich die Erzeugung einer Komponente zur Laufzeit (Simulationszeit) des Systems betont werden, kann dies auch explizit modelliert werden.

Die Abbildung 3.162 zeigt außerdem die beiden vollständig ausgefüllten Beschreibungsfelder der Kommunikationsstecker/-buchsen. In den Beschreibungsfeldern sind alle Informationen enthalten, die aus dem C++ Listing extrahiert werden konnten, wie z.B. die zulässigen Definitionsbereiche der Methode. In den meisten Programmen sind solche zusätzlichen Informationen nicht enthalten und müssen zur Verbesserung der Kompatibilitätsbestimmung nachträglich modelliert werden. □

3.6.4.3. Modellierung von elektrischen/elektronischen Signalen in (U)CML

Die wohl wichtigste Signalart für die Modellierung von eingebetteten Systemen sind die elektrischen/elektronischen Signale. Mit ihrer Hilfe werden Ströme sowie Datentragende Rechtecksignale von einem Bauteil des Systems zu einem anderen übertragen. Von entscheidender Bedeutung für die Modellierung und Kompatibilitätsprüfung eines eingebetteten Systems sind dabei die Eigenschaften der verschiedenen elektrischen/elektronischen Signale des Systems. Insbesondere ist die kombinierte Betrachtung von Signalen für die Modellierung von Signalen in eingebetteten Systemen von besonderem Interesse, weil am dieser Stelle in der Praxis die meisten Fehler gemacht werden.

Bereits im Kapitel „2.7 *Objektorientierte Modellierung kompatibilitätsrelevanter Eigenschaften von eingebetteten Systemen*“ ab Seite 104 wurden die grundlegenden Eigenschaften von elektrischen/elektronischen Signalen eingeführt[247]. Aufbauend auf den dort vorgestellten Grundlagen wird nun die Modellierung von elektrischen/elektronischen Signalen in (U)CML erläutert. In der nachfolgenden Aufzählung werden drei unterschiedliche Arten der Modellierung von elektrischen/elektronischen Signalen aufgelistet und anhand von einfachen Beispielen illustriert. Begonnen wird mit der grundlegenden Modellierung eines elektrischen/elektronischen Signals in (U)CML.

- **Elektrisches/elektronisches Signal**
  Die Transformation eines elektrischen/elektronischen Signals in ein entsprechendes (U)CML-Signal erfolgt in drei Schritten. Zuerst wird das reale elektrischen/elektronischen Signal analysiert und seine kompatibilitätsrelevanten Eigenschaften erfasst (2). Daran anschließend wird das qualitative (U)CML-Modell erzeugt (3a). In das Beschreibungsfeld des Modells werden dann die in (2) erfassten kompatibilitätsrelevanten Eigenschaften eingetragen (3b). In der Abbildung 3.164 ist der Transformationsprozess graphisch dargestellt.

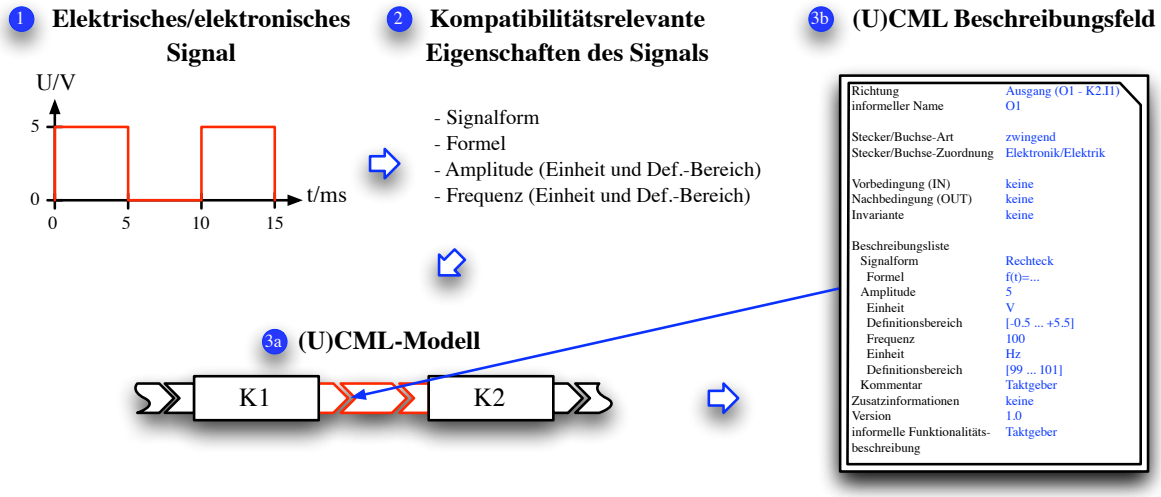


Abbildung 3.164.: Transformation eines realen elektrischen/elektronischen Signals in ein entsprechendes elektrisches/elektronisches (U)CML-Signal.

Jedes elektrische/elektronische Signal lässt sich auf diese generische Weise in ein entsprechendes (U)CML-Signal transformieren.

- **Elektrisches/elektronisches Signal mit überlagertem Softwaretechniksignal**
  Wenn in (U)CML ein elektrisches/elektronisches Signal modelliert werden soll, über das ein oder mehrere Softwarefunktionsaufrufe (Methodenaufrufe) übertragen werden sollen, so muss zuerst das elektrische Signal (Trägersignal) und für jede Funktion bzw. Funktionsaufruf eine weitere Softwareverbindung modelliert werden. Soll der Zusammenhang zwischen dem elektrischen/elektronischen Signal und dem Softwaresignal explizit verdeutlicht werden, wird in allen beteiligten Beschreibungsfeldern der Zusammenhang zwischen den Signalen als Kommentar eingetragen. Dadurch ist es möglich, auf Modellebene zu zeigen, dass ein enger Zusammenhang zwischen dem elektrischen/elektronischen Signal und dem Softwaretechniksignal herrscht.

[247] Weiterführende Informationen zur Modellierung von elektrischen Signalen finden Sie im Anhang unter „F *Modellierung von Signalen*“ ab Seite 341.

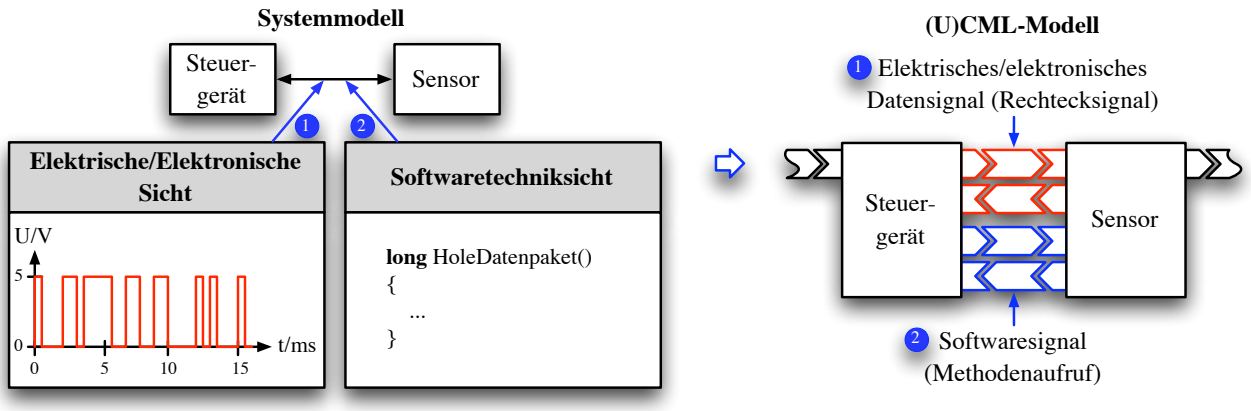


Abbildung 3.165.: Übertragung eines oder mehrerer Softwaretechniksignale über eine dezidierte elektrisches/elektronische Verbindung.

Die Abbildung 3.165 verdeutlicht die Überlagerung eines elektrischen/elektronischen Trägersignals mit einem oder mehreren Softwaretechniksignalen (hier: C++ Methodenaufrufe). Auf der linken Seite der Abbildung ist das abstrakte Systemmodell dargestellt. Das Systemmodell besteht aus zwei unterschiedlichen Ansichten auf das zu modellierende System. Zum einen die *elektrische/elektronische Sicht*, in der das elektrische/elektronische Trägersignal beschrieben wird (1). Zum anderen die *Softwaretechniksicht* auf das System (2). Hier ist z.B. die C++ Methode `long HoleDatenpaket()` dargestellt, mit deren Hilfe die Softwaredaten zwischen der Komponente *Steuergerät* und der Komponente *Sensor* übertragen werden.

Auf der rechten Seite der Abbildung ist das entsprechende (U)CML-Modell des Systems dargestellt. Zwischen den beiden Komponenten *Steuergerät* und *Sensor* sind zwei Kommunikationsflusspfeile modelliert, die die entsprechenden Kommunikationsstecker und -buchsen der beiden Komponenten miteinander verbinden. Der obere Kommunikationsflusspfeil wird zur Übertragung des elektrischen/elektronischen Signals verwendet (1), während der untere Kommunikationsflusspfeil zur Übertragung der Softwaremethode benutzt wird (2). Zusammen bilden sie die Kommunikation zwischen den beiden Komponenten ab. Um zu verdeutlichen, dass die beiden Verbindungen zusammen gehören, wird im Beschreibungsfeld dies als Kommentar eingetragen.

- **Elektrische/elektronische Verbindung mit mechanischen Eigenschaften**
  Soll in einem Modell der duale Charakter – elektrische/elektronische und mechanische Eigenschaften – einer real existierenden Leitung vollständig nachmodelliert werden, können in (U)CML zwei zusammengesetzte Flusspfeile, die eine Referenz zueinander besitzen, eingesetzt werden. Dadurch ist es möglich, sowohl die elektrischen/elektronischen als auch die mechanischen Eigenschaften der Leitung in (U)CML zu modellieren und das Modell der Verbindung auf Kompatibilität zu untersuchen. In der Abbildung 3.166 ist eine elektrische/elektronische Verbindung mit mechanischen Eigenschaften dargestellt.

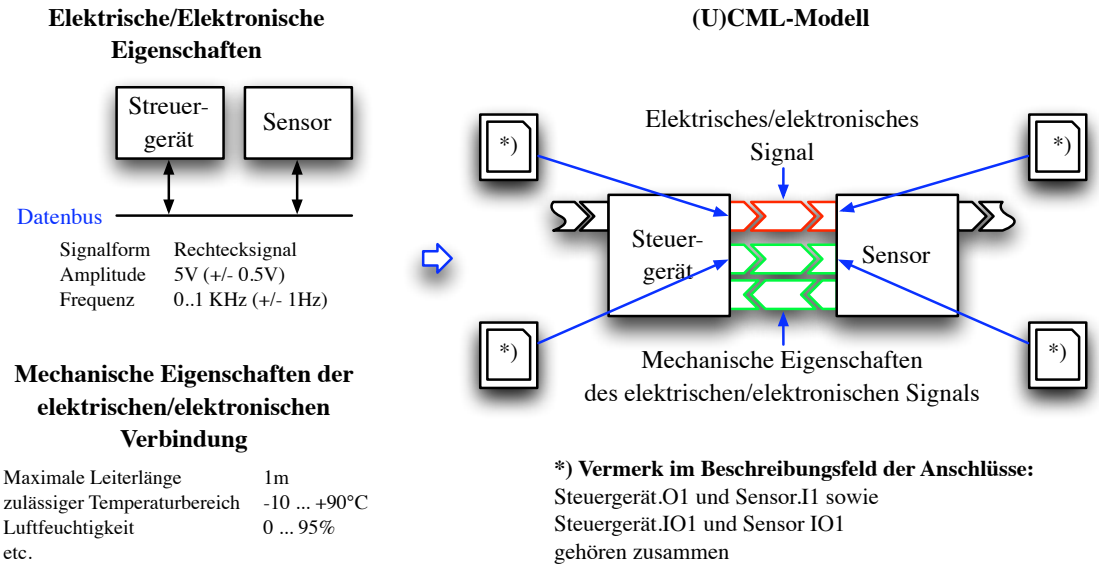


Abbildung 3.166.: Modellierung eines elektrischen/elektronischen Signals mit mechanischen Eigenschaften.

Die Abbildung 3.166 auf der vorherigen Seite zeigt auf der linke Seite (oben) einen Ausschnitt aus einem elektrischen Blockschaltbild, in dem die beiden Baugruppen *Steuergerät* und *Sensor* enthalten sind. Verbunden sind sie über einen Datenbuss, dessen elektrische/elektronische Eigenschaften ebenfalls abgebildet sind. Im unteren linken Bildausschnitt sind die mechanischen Eigenschaften der Verbindung in tabellarischer Form aufgeführt. Zusammen bilden sie die elektromechanische Beschreibung der Verbindung.

Auf der rechten Seite ist das integrierte (U)CML-Modell des links dargestellten Systems modelliert. Im Modell sind sowohl die elektrischen/elektronischen, wie auch die mechanischen Eigenschaften der Verbindung zwischen den Komponenten *Steuergerät* und *Sensor* explizit modelliert und dargestellt. Um zu verdeutlichen, dass die beiden Flussverbindungen zusammengehören und die gleiche reale Leitung zwischen den Komponenten repräsentieren, wird dies im Kommentarfeld der Beschreibungsfelder vermerkt.

In (U)CML ist es insbesondere möglich, alle drei Systemsichten (Elektrik/Elektronik, Mechanik und Softwaretechnik) in einem gemeinsamen Modell zu modellieren und dieses auf Kompatibilität zu untersuchen. Im Kapitel „4 *Modellierung der Fallstudie – Gleitschutzsystem – in (U)CML*“ ab Seite 263 wird dies anhand eines realen Beispiels gezeigt.

Im nachfolgenden Abschnitt wird das (U)CML-Regelwerk eingeführt und anhand von verschiedenen Beispielen ausführlich erläutert.

3.6.5. (U)CML-Sprach- und Kompatibilitätsregelwerk

Nachdem nun sämtliche Sprachelemente sowie das Optionalitätsprinzip der (U)CML eingeführt und erläutert wurden, folgt in diesem Kapitel die Beschreibung des Kompatibilitätsregelwerks der (U)CML. Mit Hilfe des Kompatibilitätsregelwerks kann ein in (U)CML modelliertes System auf Kompatibilität untersucht und bewertet werden. Dabei stellt das Kompatibilitätsregelwerk die zentrale Neuerung, die die (U)CML gegenüber vielen anderen existierenden Modellierungssprachen auszeichnet, dar.

Das (U)CML-Regelwerk unterteilt sich in drei Teilbereiche: das *(U)CML-Sprachregelwerk*, mit dessen Hilfe die Korrektheit der Sprache (U)CML – also die richtige Verwendung der Bestandteile der (U)CML – überprüft wird. Die nächsten beiden Teile des (U)CML-Regelwerks sind das *projektunabhängige Regelwerk*, in dem sämtlich Grunddefinitionen wie z.B. Präfixe, Einheiten und Datentypen definiert sind, und das *projektabhängige Regelwerk*. Im projektabhängigen Regelwerk werden sämtliche Regeln hinterlegt, die nur für das aktuelle Projekt gelten sollen. In der Abbildung 3.167 ist diese Unterteilung des Regelwerks graphisch mit je einem Beispiel dargestellt.

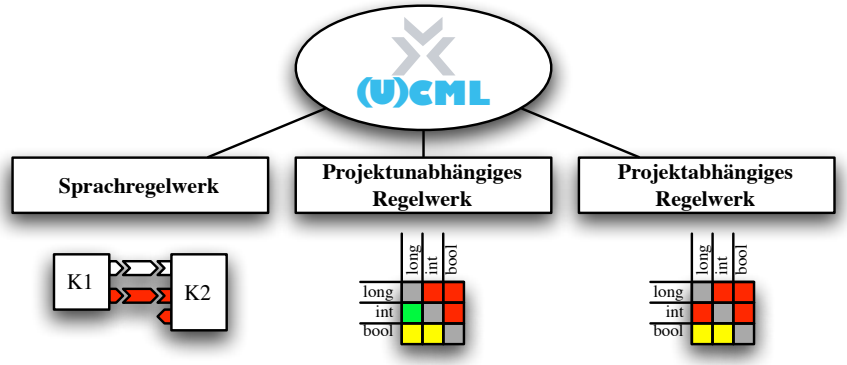


Abbildung 3.167.: Unterteilung des (U)CML-Regelwerks in die drei Bestandteile – (U)CML-Sprachregelwerk, projektunabhängiges und projektabhängiges Regelwerk.

In den nachfolgenden Unterpunkten dieses Kapitels werden die drei Bestandteile des (U)CML-Regelwerks ausführlich erläutert. Begonnen wird mit der Erklärung des (U)CML-Sprachregelwerks.

Anmerkung:
Eine komplette Liste aller Regeln des (U)CML-Sprachregelwerks bzw. des projektunabhängigen Regelwerks finden Sie unter [BK05] bzw. [Zim07].

3.6.5.1. (U)CML-Sprachregelwerk

Das (U)CML Sprachregelwerk[248] beschreibt den korrekten syntaktischen Aufbau und die Struktur der graphischen Kompatibilitätsmodellierungssprache (U)CML. Dabei setzt sich das (U)CML-Sprachregelwerk aus den einzelnen Eigenschaften aller (U)CML-Sprachelemente (genauer: den daraus abgeleiteten Regeln) zusammen. In der Abbildung 3.168 auf der nächsten Seite ist die Unterteilung des (U)CML-Sprachregelwerks in die Teilbereiche – Paket-, Komponenten-, Stecker-, Buchsen-, Schnittstellen-, Flusspfeil- und Beschreibungsfeld-Regeln sowie ihre Ableitung aus den entsprechenden Eigenschaften graphisch dargestellt. Die Eigenschaften der (U)CML-Sprachelemente wurden bereits im Kapitel „Sprachelemente der (U)CML“ eingeführt. Aus diesen Eigenschaften werden nun Regeln abgeleitet, mit deren Hilfe der Aufbau, sowie die Struktur der Sprache (U)CML vollständig beschrieben wird.

[248] Das (U)CML-Sprachregelwerk wird auch als (U)CML-Syntax-Beschreibung bezeichnet.

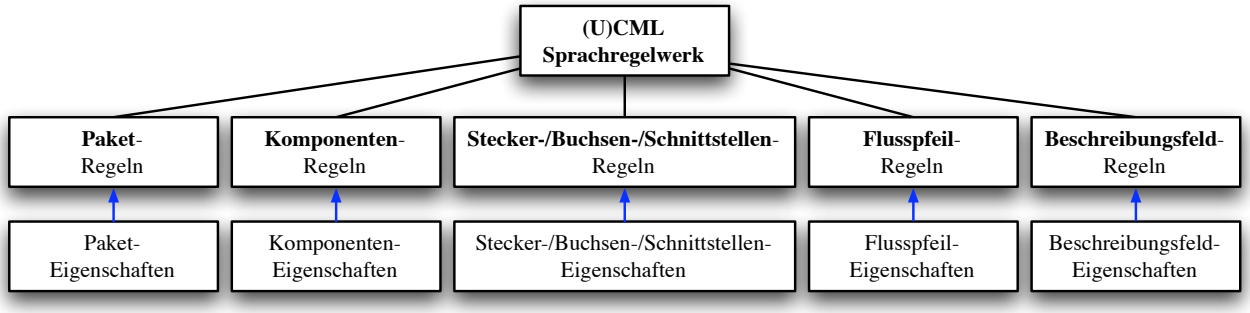


Abbildung 3.168.: Aufbau des (U)CML-Sprachregelwerks.

Definition 3.77 ***Eigenschaften des (U)CML-Sprachregelwerks:***
Das (U)CML-Sprachregelwerk setzt sich aus den abgeleiteten Eigenschaften der Pakete, Komponenten, Stecker, Buchsen, Schnittstellen, Flusspfeile sowie den Eigenschaften für das Beschreibungsfeld zusammen. Außerdem gilt: Alle Regeln des Sprachregelwerks sind stets ***zwingend*** *und können* ***nicht ersetzt*** *oder* ***überschrieben*** *werden. Sämtliche Verletzungen des (U)CML-Sprachregelwerks werden in roter Farbe dargestellt.*

Anmerkung zur Definition 3.77:

- Alle Regeln des (U)CML-Sprachregelwerks gelten stets für alle Elemente der Sprache (U)CML. Diese Regeln können nicht, wie z.B. beim projektunabhängigen Regelwerk durch Regeln des projektabhängigen Regelwerks, ersetzt oder überschrieben werden (vgl. Abschnitt 3.6.5.2.1 bzw. 3.6.5.2.2).
- Bei einer (U)CML-Sprachregelwerksverletzung wird das entsprechende Element (Paket, Stecker, Pfeil), das die Verletzung auslöst, rot markiert und dargestellt (siehe Abb.: 3.170).
- Für die Kompatibilitätsbewertung eines Systems darf das Modell des Systems keine (U)CML-Sprachregelwerksverletzungen enthalten, da sonst der Kompatibilitätstest nicht durchgeführt werden kann[249]. Aus diesem Grund müssen alle (U)CML-Sprachregelwerksverletzungen vollständig beseitigt werden.

Die Abbildung 3.168 illustriert den Zusammenhang zwischen den Eigenschaften einer Komponente und den daraus abgeleiteten Regeln für die Komponente. Die Ableitung der Regeln aus den Eigenschaften der (U)CML-Sprachelemente erfolgt durch die Umsetzung der Definitionen der Sprachelemente in eine Regel. Im nachfolgenden Beispiel wird die Transformation der Eigenschaften eines (U)CML-Standardpakets in die entsprechenden Regeln exemplarisch gezeigt.

Beispiel 75: Umsetzung einer Eigenschaft in eine Regel
Die Definition 3.5 auf Seite 163 definiert einige Eigenschaften eines (U)CML-Standardpakets: „Ein (U)CML-Standardpaket stellt einen Container dar, der beliebig viele weitere Pakete oder andere Systembestandteile wie z.B. Komponenten und Flusspfeile enthalten kann. Des Weiteren gilt: (U)CML-Pakete haben keine eigene Funktionalität.“.

Aus der Definition des Standardpakets lassen sich folgende Regeln ableiten:

- *Ein (U)CML-Paket darf beliebig viele weitere Pakete enthalten.*
- *Ein (U)CML-Paket darf beliebig viele Komponenten und Flusspfeile enthalten.*
- *Ein Paket hat keine Funktionalität.*

□

Nach dem gleichen Schema lassen sich sämtliche Eigenschaften aller (U)CML-Sprachelemente in entsprechende Sprachregeln transformieren.

Auszug aus dem (U)CML-Sprachregelwerk
In der nachfolgenden Aufzählung ist ein kleiner Auszug aus dem (U)CML-Sprachregelwerk aufgelistet. Für jedes (U)CML-Element (z.B. Pakete, Komponenten oder Flusspfeile) gibt es eigene Regeln, die den Aufbau und die Struktur der Sprache festlegen. In der nachfolgenden Aufzählung sind einige (U)CML-Sprachregeln für einzelne Bestandteile der (U)CML exemplarisch aufgelistet.

- **Paket- und Komponentenregeln**
  Regeln, die sowohl für Pakete als auch Komponenten gelten:
  - *Der Name eines Pakets bzw. einer Komponente muss eineindeutig sein* (vgl. Def.: 3.6.2.1).
  - *Jedes Paket bzw. jede Komponente muss mindestens eine (beschaltete) Eingangsbuchse bzw. mindestens einen (beschalteten) Ausgangsstecker haben* (vgl. Def.: 3.5 sowie Def.: 3.9).

[249] Vgl. Abbildung: 3.183 auf Seite 249.

Spezielle Regeln für Pakete und Komponenten:

- Paketregeln
  - *In einem System darf es genau ein Systempaket geben* (vgl. Kapitel 3.4).
  - *Ein (U)CML-Paket darf beliebig viele weitere Pakete enthalten* (vgl. Def.: 3.5 bzw. Def.: 3.7).
  - *Ein Paket muss mindestens eine Komponente oder ein Paket enthalten.*
- Komponentenregeln
  - *Eine (U)CML-Komponente darf keine weiteren Komponenten enthalten* (vgl. Def.: 3.9).
  - *Eine Komponente muss sich stets innerhalb eines Pakets befinden.*

- **Stecker-/Buchsen-/Schnittstellenregeln**
  In der „(U)CML-Ampelkarte“ sind alle in (U)CML möglichen Kombinationen von Steckern und Buchsen eines Pakets bzw. einer Komponente enthalten. In der grünen Sektion sind alle Kombinationen eingetragen, die keinen Fehler verursachen. Die gelbe Sektion zeigt Stecker-/Buchsen-Kombinationen, die einen Fehler verursachen können. In der roten Sektion sind Stecker-/Buchsen-Kombinationen eingezeichnet, die in (U)CML einen (U)CML-Sprachfehler verursachen.

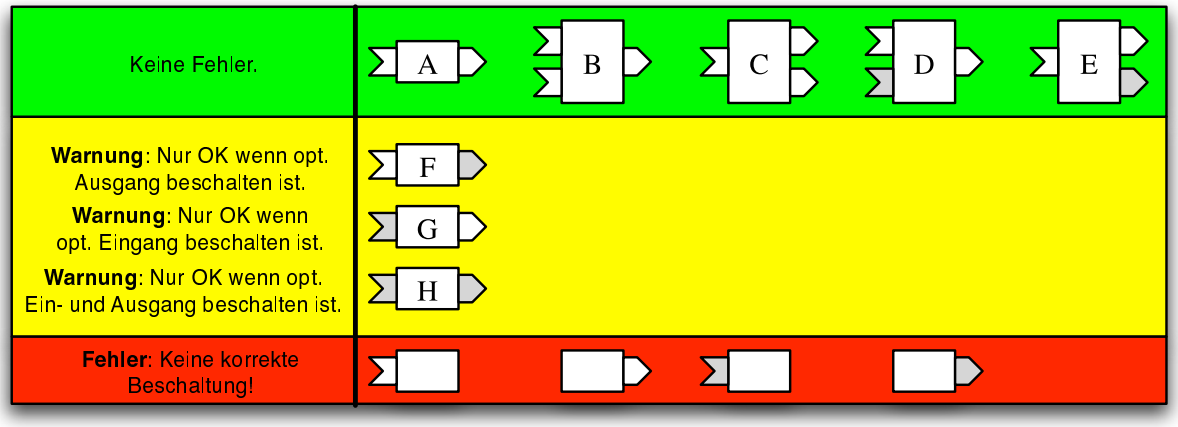


Abbildung 3.169.: „Ampelkarte“ aller möglichen Stecker- und Buchsenkombinationen für Pakete und Komponenten.

Zusätzlich gelten für Stecker/Buchsen und Schnittstellen folgende Regeln:

- *Alle zwingenden Stecker/Buchsen und Schnittstellen müssen angeschlossen sein.*
- *Optionalen Stecker/Buchsen und Schnittstellen müssen nicht angeschlossen sein.*
- *Buchsen (Eingänge) sind stets links, Stecker (Ausgänge) stets rechts an Komponenten angebracht. Ausnahme: Kommunikationsstecker* (vgl. Def.: 3.21).

- **Flusspfeilregeln**
  - *Flusspfeile verbinden stets Stecker mit Buchsen bzw. Stecker/Buchsen mit Schnittstellen* (vgl. Kapitel 3.6.2.5).
  - *Flusspfeile müssen auf beiden Seiten angeschlossen sein.*
  - *Flusspfeile verbinden nur gleiche Arten von Steckern/Buchsen und Schnittstellen miteinander* (vgl. Def.: 3.38).
  - *Quelle und Senke (d.h. die Komponenten/Pakete) einer Verbindung dürfen nicht identisch sein* (vgl. Def.: 3.38).

- **Beschreibungsfeldregeln**
  Alle Beschreibungsfelder eines in (U)CML modellierten Systems müssen mit korrekten Werten befüllt sein, da sonst die Kompatibilitätsprüfung nicht durchgeführt werden kann.

Das nachfolgende Beispiel illustriert die Auswirkungen des (U)CML-Sprachregelwerks auf ein System mit Verletzungen des Sprachregelwerks, das in der Flussansicht modelliert worden ist.

Beispiel 76: Anwendung des (U)CML-Sprachregelwerks
Die Abbildung 3.170 auf der nächsten Seite zeigt ein in (U)CML modelliertes System in der Flussansicht. In diesem Modell sind drei Verletzungen des (U)CML-Sprachregelwerks enthalten.

Sämtliche (U)CML-Sprachregelwerksverletzungen des Systems sind rot markiert und werden in der nachfolgenden Aufzählung einzeln erläutert:

1. Verbindung unterschiedlicher Steckerarten:
   Der Standardflusspfeil verbindet einen Kommunikationsstecker mit einer Standardeingangsbuchse. Diese Art der Verbindung ist nicht erlaubt und verletzt die Definition 3.38.

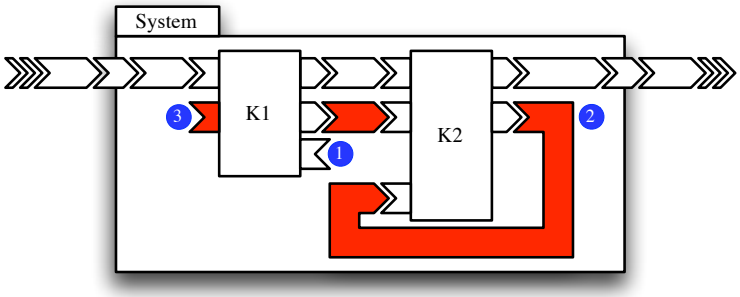


Abbildung 3.170.: Einfaches (U)CML-Beispielsystem mit Verletzungen des (U)CML-Sprachregelwerks.

2. Quelle und Senke sind gleich:
   Direkte rekursive Verbindungen, also Verbindungen, bei denen Quelle und Senke identisch sind, sind in (U)CML nicht erlaubt (vgl. Def.: 3.38). In diesem Beispiel ist zu beachten, dass lediglich der Flusspfeil, der die Quelle mit der Senke verbindet, rot markiert ist, da nur der Flusspfeil, nicht jedoch der Stecker oder die Buchse falsch sind.

3. Zwingende Eingangsbuchse ist nicht angeschlossen:
   Die zwingende Eingangsbuchse (3) muss angeschlossen sein. Optionale Stecker/Buchsen oder Schnittstellen müssen nicht beschalten sein, sofern sie nicht in der roten Einteilung der Ampelkarte (Abb.: 3.169) liegen.

□

3.6.5.2. Projektunabhängiges und projektabhängiges Kompatibilitätsregelwerk

Das *projektunabhängige Regelwerk*, wie auch das *projektabhängige Regelwerk*, ist eine spezielle Neuerung der Kompatibilitätsmodellierungssprache (U)CML. Keine andere Modellierungssprache enthält ein Regelwerk, mit dessen Hilfe z.B. Datentypen, Präfixe, Kompatibilitätsregeln etc. exakt definiert und beschrieben werden können und das modellierte System auf Kompatibilität untersucht werden kann.

Definition 3.78 ***Aufbau des (U)CML-Regelwerks:***
Das Kompatibilitätsregelwerk der (U)CML setzt sich aus dem ***projektunabhängigen*** *sowie dem* ***projektabhängigen*** *Regelwerk zusammen.*

Das projektunabhängige Regelwerk dient zur Definition von projektübergreifenden Definitionen, wie z.B. von Datentypen, Präfixen, Einheiten oder Regeln. Alle im projektunabhängigen Regelwerk definierten Regeln können im gesamten Modell verwendet werden. Zusätzlich zum projektunabhängigen Regelwerk der (U)CML gibt es das projektabhängige Regelwerk. Auch dort können Datentypen, Präfixe, Einheiten etc. definiert werden, die im Gegensatz zum projektunabhängigen Regelwerk nur für das aktuelle Projekt gelten. Dabei können durch das projektabhängige Regelwerk Regeln des projektunabhängigen Regelwerks mit neuer oder geänderter Bedeutung überschrieben bzw. erweitert werden.

3.6.5.2.1. Projektunabhängiges Kompatibilitätsregelwerk

Im projektunabhängigen Regelwerk der (U)CML sind sämtliche Definitionen und Regeln enthalten, die für die Modellierung eines eingebetteten softwarelastigen Systems benötigt werden, jedoch ohne Anspruch auf Vollständigkeit. Dabei unterteilt sich das projektunabhängige Regelwerk in die drei Fachdisziplinen – Elektrotechnik, Mechanik und Softwaretechnik. Zusätzlich zu dem fachspezifischen Regelwerk enthält das projektunabhängige Regelwerk allgemein gültige Definitionen und Regeln, wie zum Beispiel Präfixe oder SI-Einheiten, die keiner Domäne speziell zugeordnet werden können.

Definition 3.79 ***Eigenschaften des projektunabhängigen Regelwerks:***
Im projektunabhängigen (U)CML-Regelwerk sind sämtliche Kompatibilitätsregeln hinterlegt, die für alle (U)CML-Modelle gelten. Das projektunabhängige Regelwerk unterteilt sich in die vier Teilbereiche:

- *Domänenübergreifende Definitionen und Regeln*
- *Kompatibilitätsregeln für die Elektrotechnik*
- *Kompatibilitätsregeln für die Mechanik*
- *Kompatibilitätsregeln für die Softwaretechnik*

Nach Definition 3.79 unterteilt sich das (U)CML-Regelwerk in vier Teilbereiche. Diese werden, in der nachfolgenden Aufzählung einzeln aufgelistet und anhand von einfachen Beispielen erläutert.

- **Domänenübergreifende Definitionen und Regeln**
  Im Abschnitt domänenübergreifende Definitionen und Regeln sind sämtliche Präfixe, SI-Einheiten und Intervalle vordefiniert, die unabhängig von einer speziellen Domäne sind bzw. die sich in keine andere einordnen lassen. Im Kapitel „2.3.2.2.1 *Modellierung von Einheiten, dekadischen und nichtdekadischen Präfixen sowie von Gültigkeitsintervallen in der erweiterten formalen Eigenschaftsnotation für die Modellierung und Bestimmung von Kompatibilität*“ ab Seite 69 wurden bereits die Einheiten, die dekadischen und nicht dekadischen Teiler sowie Intervalle vorgestellt und erläutert. Alle dort eingeführten SI-Einheiten, sowie die dekadischen und nicht-dekadischen Teiler sind im projektunabhängigen Regelwerk der (U)CML enthalten.

  Zusätzlich zu diesen Definitionen wird jeder SI-Einheit eine Menge von Präfixen zugeordnet. Zum Beispiel wird der mechanischen Einheit Meter (m) die Menge von Präfixen m, μ sowie k zugeordnet. Im gesamten (U)CML-Modell können nun die Präfixe m, μ und k als Präfixe für die Einheit Meter (m) verwendet werden. Aus dieser Regeldefinition folgt unmittelbar, dass keine weiteren Präfixe für die Einheit Meter verwendet werden dürfen. Alle anderen Präfixe führen sofort zu einer Regelverletzung.

  Außer den Definitionen für SI-Einheiten, die dazugehörigen Teiler und Zuordnungen sind Regeln für den Umgang mit Intervallen im projektunabhängigen Regelwerk enthalten. Im Kapitel „2.3.2.2.1.3 *Modellierung von Gültigkeitsintervallen in der erweiterten formalen Eigenschaftsdefinition*“ ab Seite 74 (Abbildung 2.45) wurden sämtliche Regeln für die Schachtelung von Intervallen eingeführt. Sämtliche dort aufgeführten Regeln sind vollständig im projektunabhängigen (U)CML-Kompatibilitätsregelwerk enthalten.

- **Kompatibilitätsregeln für die Elektrotechnik**
  Für die Modellierung und Beschreibung von elektrischen/elektronischen Signalen in (U)CML bzw. den entsprechenden Beschreibungsfeldern sind vor allem kompatibilitätsrelevante Eigenschaften, wie z.B. die Signalform, die Einheit sowie das dazugehörige Gültigkeitsintervall des Signals von Interesse. Im Kapitel „3.6.4.3 *Modellierung von elektrischen/elektronischen Signalen in (U)CML*“ ab Seite 234 wurde gezeigt, wie aus einem realen elektrischen/elektronischen Signal ein entsprechendes „elektrisches/elektronisches (U)CML-Signal“ erzeugt, und modelliert werden kann.

  Aus der Art der Modellierung der elektrischen/elektronischen Signale in (U)CML lassen sich Regeln für die Kompatibilität der elektrischen/elektronischen Signale ableiten.

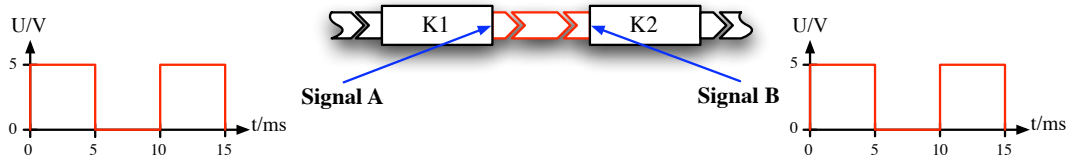
(U)CML Kompatibilitätsregel 3.4 – *Projektunabhängiges Regelwerk* –
Zwei elektrische/elektronische Signale sind in (U)CML kompatibel, wenn gilt:

- *Signalform bzw. Formel*
- *Amplitude und Gültigkeitsbereich (ϵ-Umgebung)*
- *Frequenz und Gültigkeitsbereich (ϵ-Umgebung)*

stimmen überein.

Die (U)CML-Kompatibilitätsregel 3.4 beschreibt die Kompatibilitätsanforderung an zwei elektrische/elektronische Signale. Sind alle drei Anforderungen aus der Definition erfüllt, dann sind die beiden elektrischen/elektronischen Signale in (U)CML kompatibel. Im nachfolgenden Beispiel ist die Anwendung der (U)CML-Kompatibilitätsregel 3.4 für elektrische/elektronische Signale anhand eines einfachen Beispiels dargestellt und erläutert.

Beispiel 77: Kompatibilität von elektrischen/elektronischen Signalen
In der Abbildung 3.171 ist ein Ausschnitt aus einem in (U)CML modellierten System, mit den beiden Komponenten *K1* und *K2* dargestellt, die per Standardflusspfeil verbunden sind. Der Sender sendet das elektrische/elektronische Signal *Signal A*, während die Komponente *K2* das Signal *Signal B* am Eingang erwartet.



	Signal A	Signal B
Signalform (Formel)	Rechteck	Rechteck
Amplitude + Gültigkeitsintervall	5V +/- 0.5V	5V +/- 0.5V
Frequenz + Gültigkeitsintervall	100Hz +/- 1Hz	100Hz +/- 1Hz

Signale sind kompatibel

Abbildung 3.171.: Kompatibilität zweier elektrischer/elektronischer Signale in (U)CML.

Wird die Regel für elektrische/elekronische Signale auf das (U)CML-Modell angewendet, stellt sich heraus, dass die beiden Signale kompatibel sind, da alle drei kompatibilitätsrelevanten Eigenschaften übereinstimmen. □

- **Kompatibilitätsregeln für die Mechanik**
  Für die Modellierung von mechanischen Baugruppen sind vor allem die Toleranzen in den Abmessungen bzw. die Gültigkeitsbereiche von mechanischen Größen von Interesse[250]. Sowohl die Toleranzen (d.h. die Abweichung eines Maßes von der vordefinierten Größe) als auch die Gültigkeit können in (U)CML mit Hilfe der Intervalle modelliert und beschrieben werden.

[250] Vgl. Abschnitt „2.3.2.2.1.3 *Modellierung von Gültigkeitsintervallen in der erweiterten formalen Eigenschaftsdefinition*“ ab Seite 74.

Sämtliche Regeln, die für den Umgang und die Interpretation von Intervallen gelten, wurden bereits im Kapitel „2.3.2.2.1.3 *Modellierung von Gültigkeitsintervallen in der erweiterten formalen Eigenschaftsdefinition*“ ab Seite 74 ausführlich dargestellt und werden an dieser Stelle nicht noch einmal aufgegriffen. Das gleiche gilt für die mechanischen SI-Einheiten. Sie wurden bereits im Kapitel „2.3.2.2.1.1 *Modellierung von Einheiten in der erweiterten formalen Eigenschaftsdefinition*“ ab Seite 69 eingeführt.

Beispiel 78: Kompatibilität von mechanischen Signalen
Die Abbildung 3.172 zeigt einen Ausschnitt eines in (U)CML modellierten mechanischen Teilsystems. Der Sender (die Komponente *K1*) sendet das mechanische Signal *Signal A* an den Empfänger, die Komponente *K2*. Die Komponente *K1* sendet das mechanische Signal *Signal A* mit einem zulässigen Gültigkeitsintervall von $65 \ldots 75°C$, während der Empfänger (*K2*) einen Wertebereich von $67.5 \ldots 77.5°C$ erwartet (vgl. *Signal B*)[251].

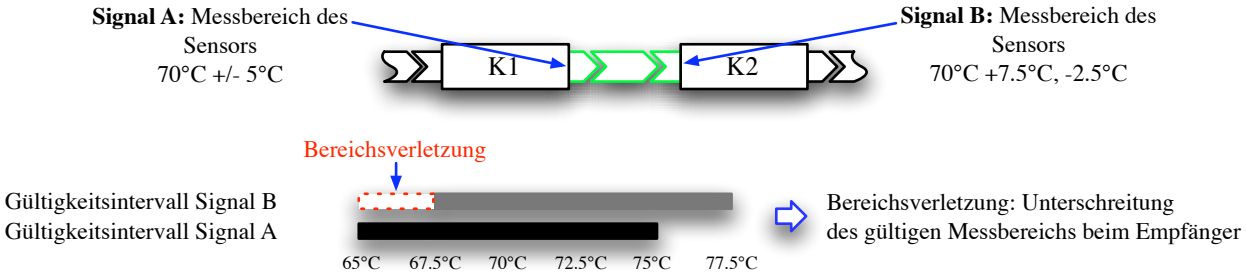


Abbildung 3.172.: Kompatibilität zweier mechanischer Signale in (U)CML.

Ergebnis des Kompatibilitätstests: Die beiden Intervalle sind nicht deckungsgleich. Das Signal A löst beim Empfänger (*K2*) eine Bereichsverletzung aus. Die Komponente *K2* kann die gesendeten Werte der Komponente *K1* nicht vollständig aufnehmen bzw. interpretieren, da der Bereich von $65 \ldots 67.5°C$ nicht abgedeckt ist (Datenverlust). Es liegt ein mechanischer Kompatibilitätsfehler vor. □

- **Kompatibilitätsregeln für die Softwaretechnik**
  Das projektunabhängige Softwaretechnikkompatibilitätsregelwerk der (U)CML unterteilt sich in drei Bereiche:

  – Präfixe und Einheiten für die Softwaretechnik
  Zusätzlich zu den im Abschnitt „Domänenübergreifende Definitionen und Regeln“ dieser Aufzählung eingeführten allgemein gültigen Präfixe und Einheiten wurden speziell für die Softwaretechnik angepasste Einheiten und Präfixe im Kapitel „2.3.2.2.1.2 *Modellierung von dekadischen und nicht dekadischen Präfixen mit Hilfe der erweiterten formalen Eigenschaftsdefinition*“ ab Seite 72 (Tabelle 2.5) eingeführt und erläutert. Sämtliche dort eingeführten Präfixe und Einheiten sind im projektunabhängigen (U)CML-Kompatibilitätsregelwerk enthalten und können in der erweiterten (U)CML-Methodensignatur verwendet werden[252].

  – Kompatibilitätsregeln für Softwaredatentypen
  In der Softwaretechnik werden viele unterschiedliche Datentypen verwendet. Die Kompatibilität der Datentypen in (U)CML wird mit Hilfe der *Datentypmatrix*[253] festgelegt. Die Kompatibilitätsregel 3.5 beschreibt, aufbauend auf der Datentypmatix, wann zwei Datentypen in (U)CML kompatibel sind.

  (U)CML Kompatibilitätsregel 3.5 – *Projektunabhängiges Regelwerk* –
  Zwei Datentypen sind in (U)CML kompatibel, wenn:
  * *Die Datentypausrichtung*[254]*beider Datentypen identisch ist*
  * *Es zu keinem Datenverlust kommt*

  – Kompatibilitätsregeln für Methoden
  Um sicherzustellen, dass zwei Methoden in (U)CML kompatibel sind, müssen ihre Signaturen (d.h. der Methodenname, die Anzahl und Art der Parameter sowie der Ergebniswert) übereinstimmen. In der Kompatibilitätsregel 3.6 ist dies unabhängig für alle Programmiersprachen definiert.

  (U)CML Kompatibilitätsregel 3.6 – *Projektunabhängiges Regelwerk* –
  Zwei Methoden sind in (U)CML genau dann kompatibel, wenn die Signatur beider Methoden übereinstimmt.

Beispiel 79: Kompatibilität von Softwaretechniksignalen
Die Abbildung 3.173 auf der nächsten Seite zeigt in der Bildmitte ein in (U)CML modelliertes Teilsystem, welches zwei Komponenten *K1* und *K2* umfasst, die durch einen Softwarekommunikationsflusspfeil verbunden sind. Die Komponente *K1* ruft die Methode `int add(int IN1, int IN2)` der Komponente *K2* auf.

[251] Vgl. Abschnitt „3.6.4.1 *Modellierung von mechanischen Signalen in (U)CML*“ ab Seite 229.
[252] Vgl. Abschnitt „3.6.4.2 *Modellierung von Softwaretechniksignalen in (U)CML*“ ab Seite 230.
[253] Vgl. Abschnitt „2.4.2.1 *Statisches projektspezifisches Regelwerk*“ ab Seite 84.
[254] Vgl. Abschnitt „2.4.2.1 *Statisches projektspezifisches Regelwerk*“ ab Seite 84.

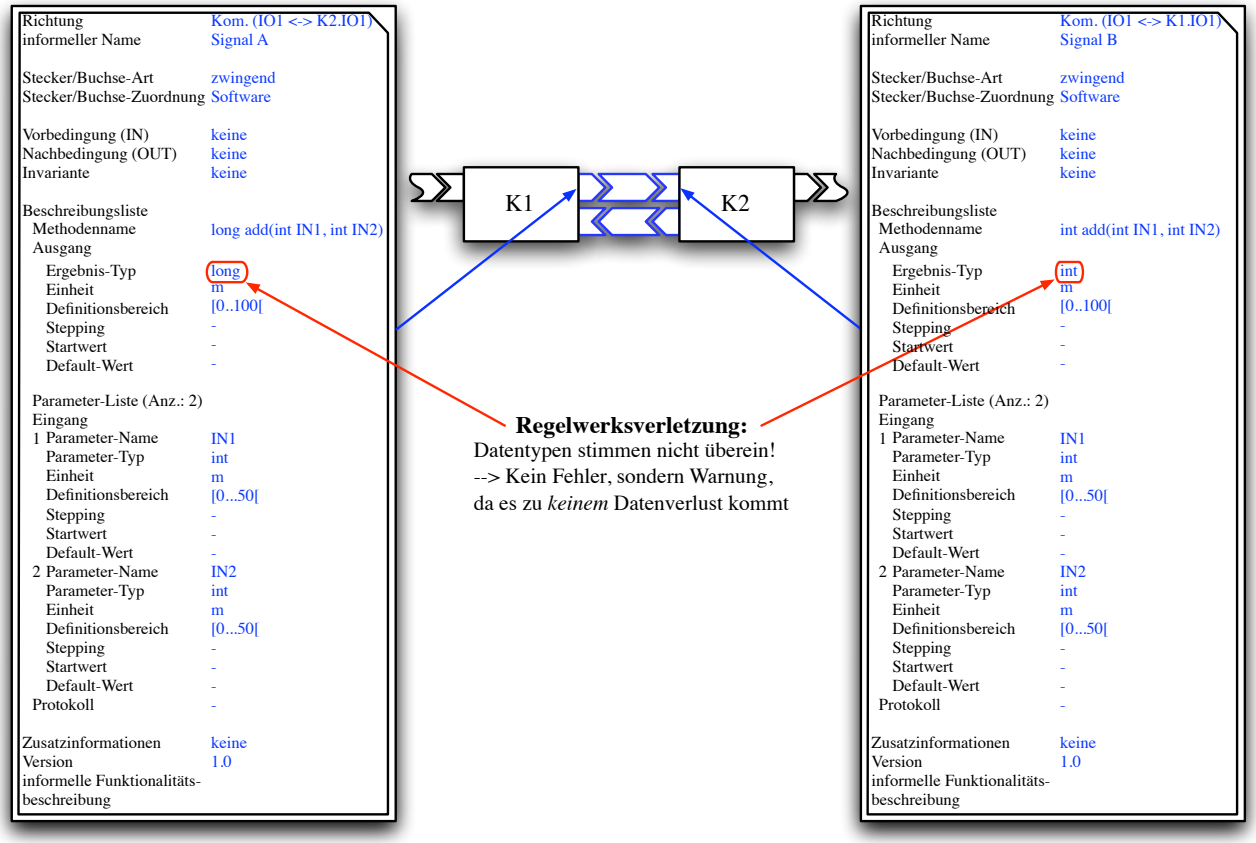


Abbildung 3.173.: Kompatibilität zweier Softwaretechniksignale in (U)CML.

Bei diesem Methodenaufruf kommt es zu einer Datentypinkompatibilität zwischen der Komponente *K1* und der Komponente *K2*. Die Komponente *K1* erwartet den Ergebnisdatentyp **`long`**, während die Komponente *K2* den Ergebnisdatentyp **`int`** liefert. Dadurch wird eine Datentypinkompatibilität ausgelöst (vgl. Def.: 3.5). Durch diese Datentypinkompatibilität entsteht jedoch kein Fehler, da es bei der Konvertierung der Daten zu keinem Datenverlust kommt (vgl. Datentypmatrix 2.6). □

Anmerkung:
Eine ausführliche Beschreibung der Kompatibilitätsregeln für eingebettete Systems finden Sie unter [Zim07].

Mit dem allgemein gültigen projektunabhängigen Regelwerk lassen sich die meisten eingebetteten softwarelastigen Systeme vollständig beschreiben und auf Kompatibilität untersuchen. Wenn in einem Modell eines Systems zusätzliche Einheiten oder Präfixe für die Modellierung benötigt werden, können diese im projektabhängigen Regelwerk definiert werden.

3.6.5.2.2. Projektabhängiges Kompatibilitätsregelwerk

Im projektabhängigen Kompatibilitätsregelwerk sind sämtliche Definitionen und Regeln enthalten, die speziell auf die Bedürfnisse eines Projekts angepasst wurden. Dabei kann mit Hilfe des projektabhängigen Regelwerks das projektunabhängige Regelwerk um z.B. neue Einheiten oder Regeln erweitert, oder dort definierte ersetzt (überschrieben) werden. Dadurch ist es möglich, das allgemein gültige projektunabhängige Regelwerk unverändert für alle Projekte zu verwenden und lediglich spezielle Regeln an die Bedürfnisse des aktuellen Projekts anzupassen.

Definition 3.80 ***Eigenschaften des projektabhängigen Regelwerks:***
Wenn in einem Projekt ein projektabhängiges Regelwerk vorhanden ist, so überschreibt bzw. ersetzt es die Definitionen und Regeln, des projketunabhängigen Regelwerken die neu definiert worden sind. Alle anderen Regeln des projektunabhängigen Regelwerks gelten unverändert weiter.

Bei der Anpassung des projektunabhängigen Regelwerks an die speziellen Bedürfnisse des aktuellen Projekts ist mit besonderer Vorsicht vorzugehen, da jede Änderung am Regelwerk stets **globale Auswirkungen** auf das gesamte aktuelle Projekt hat. Das heißt: Eine Änderung der Kompatibilität z.B. eines Datentyps wirkt sich auf alle Datentypen im gesamten System aus. Aus diesem Grund sollten Änderungen am Regelwerk nur vorgenommen werden, wenn es zwingend notwendig ist. Im nachfolgenden Beispiel sind zwei Anpassungen des projektunabhängigen Regelwerks in das projektabhängige Regelwerk aufgenommen worden.

Beispiel 80: Projektabhängiges Regelwerk
In diesem Beispiel werden zwei Anpassungen des projektunabhängigen Regelwerks im projektabhängigen Regelwerk aufgenommen.

- **Softwareregelanpassung**
  Szenario: In einem in (U)CML modellierten System wird die aufgerufene C++ Methode durch eine modifizierte C++ Methode mit leicht veränderter Signatur ersetzt, während die aufrufende Methode unverändert bleibt (Abb.: 3.174). Die ursprüngliche C++ Methode `void setzeWert(long wert);` besitzt den Parameter-Datentyp **`long`**. Dieser wird in der modifizierten Methode `void setzeWert(int wert);` durch den Parameter-Datentyp **`int`** ersetzt. Durch die Ersetzung kommt es zu einem Datenverlust, da der ursprüngliche Datentyp **`long`** Werte von $-2.147.483.648$ bis $2.147.483.647$ speichern konnte, der Datentyp **`int`** jedoch nur Werte von -32.768 bis $+32.767$ (bei **`long`** gleich 32-Bit und **`int`** gleich 16-Bit Datenwortlänge).

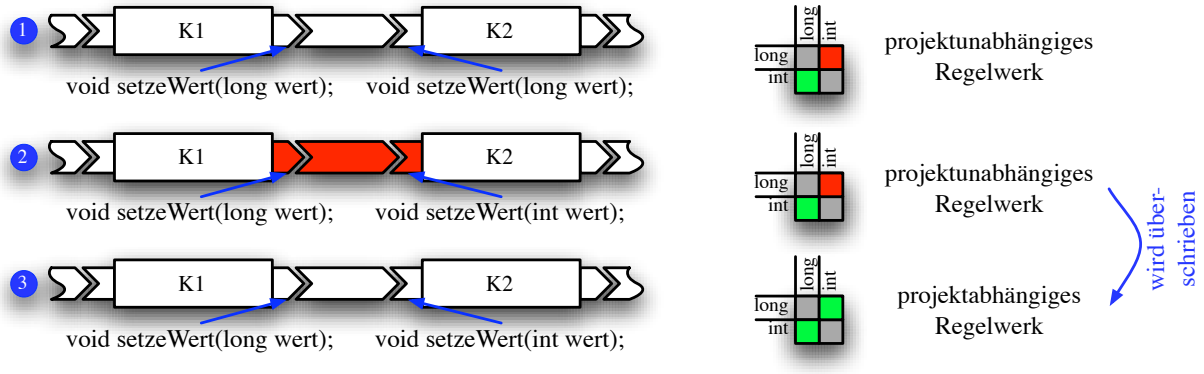


Abbildung 3.174.: (U)CML-Modell: Anpassung des projektunabhängigen Regelwerks an die speziellen Bedürfnisse des Systems.

Die Abbildung 3.174 zeigt die beschriebene Veränderung am Modell:

1. Ausgangssystem
   Datentypkompatibilität – die Datentypen der Quelle stimmen mit der Senke überein.
2. Anpassung des Parameter-Datentyps der C++ Methode
   Ergebnis: Das projektunabhängige Regelwerk meldet einen Datentypkompatibilitätsfehler im Modell und markiert die entsprechende Verbindung im Modell rot.
3. Projektabhängiges Regelwerk – Datentypanpassung im Regelwerk
   Ergebnis: Nach der Datentypanpassung im projektabhängigen Regelwerk ist die Verbindung kompatibel. Es wird kein Fehler angezeigt.

Die Veränderung der C++ Methode bewirkt, dass das projektunabhängige Regelwerk einen Fehler (Datenverlust) meldet (1). Wenn in dem gesamten modellierten System jedoch stets **`long`** Werte kleiner $]-32.768\ldots+32.767[$ vorkommen, kann die Kompatibilität des Datentyps **`long`** mit dem Datentyp **`int`** im projektabhängigen Regelwerk als „kompatibel“ definiert werden (3). Nachdem die Änderung am Regelwerk vorgenommen wurde, wird kein Datentypfehler mehr gemeldet, da das projektabhängige Regelwerk das projektunabhängige Regelwerk ersetzt bzw. modifiziert.

- **Zuordnung Einheiten und Präfix**
  In (U)CML kann jeder Einheit ein oder mehrere Präfixe zugeordnet werden. In der nachfolgenden Tabelle werden der mechanischen Einheit Meter (m) die Präfixe $\mathrm{m}, \mu, \mathrm{n}$, sowie der elektrischen Einheit Volt (V) die Präfixe m, k zugeordnet. Für das Beispiel sind diese Zuordnungen im projektunabhängigen Regelwerk hinterlegt.

Einheit	Abkürzung	zugelassene Präfixe
Meter	m	$\mathrm{m}, \mu, \mathrm{n}$
Volt	V	m, k

  Szenario 1: Erweiterung der gültigen Präfixe für die Einheit Meter um k. Diese Änderung bewirkt, dass im gesamten aktuellen Projekt zusätzlich zu den in der obigen Tabelle definierten Präfixen das Präfix k als gültig spezifiziert wird.

  Szenario 2: Mit Hilfe des projektabhängigen Regelwerks können Präfixe als ungültig erklärt werden. In der obigen Tabelle wird für die elektrische Einheit Volt das Präfix k gelöscht. Diese Änderung am Regelwerk hat zur Folge, dass im gesamten Modell das Präfix k im Zusammenhang mit der Einheit Volt nicht mehr verwendet werden darf. Wird es dennoch verwendet, wird eine Fehlermeldung ausgegeben.

□

3.6.6. Der (U)CML-Modellbildungsprozess

Im Kapitel „Modellbildung“ wurde der generische Modellbildungsprozess nach Dr. Negele vorgestellt, mit dessen Hilfe es möglich ist, sowohl technische als auch nichttechnische Systeme zu modellieren und anschließend das Modell zu simulieren. Für die Kompatibilitätsbestimmung mit Hilfe der Modellierungssprache (U)CML wird der von Dr. Negele vorgestellte generische

Modellbildungsprozess noch einmal aufgegriffen und um weitere Aktivitäten erweitert bzw. die existierenden Aktivitäten ergänzt und an die speziellen Anforderungen der (U)CML angepasst.

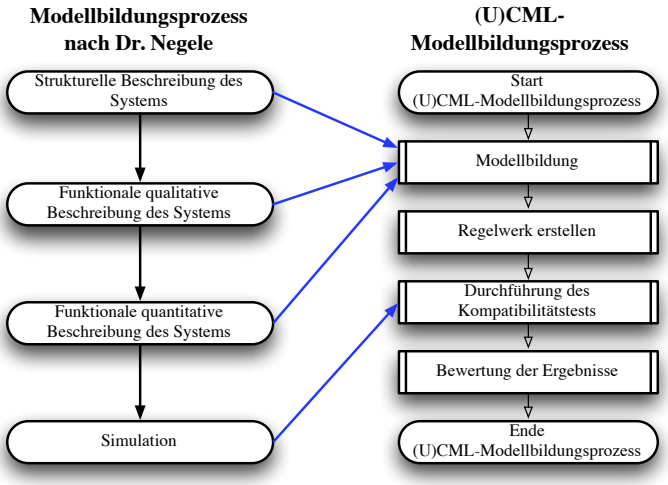


Abbildung 3.175.: Links: Der vereinfachte generische Modellbildungsprozess nach Dr. Negele. Rechts: Der (U)CML-Modellbildungsprozess.

Die Abbildung 3.175 zeigt auf der linken Seite den generischen Modellbildungsprozess nach Dr. Negele[255]. Auf der rechten Seite ist der an den allgemein gültigen Modellbildungsprozess angelehnte (U)CML-Modellbildungsprozess abgebildet. Der (U)CML-Modellbildungsprozess lässt sich in vier Hauptprozessschritte unterteilen:

- Modellbildung,
- Regelwerk,
- Durchführung der Kompatibilitätsprüfung,
- Bewertung der Ergebnisse.

Die vier Prozessschritte des allgemein gültigen Modellbildungsprozesses lassen sich auf den (U)CML-Modellbildungsprozess folgendermaßen abbilden: Die drei Phasen – *Strukturelle Beschreibung des Systems*, *Funktionale qualitative Beschreibung des Systems* sowie *Funktionale quantitative Beschreibung des Systems* – lassen sich auf die Phase *Modellbildung* des (U)CML-Modellbildungsprozesses abbilden. Der Prozessschritt *Simulation* wird auf die (U)CML-Phase *Durchführung des Kompatibilitätstests* abgebildet (vgl. Abb.: 3.175 blaue Pfeile). Im Unterschied zur Simulation im allgemeinen Modellbildungsprozess wird bei der Durchführung des Kompatibilitätstests das Modell nicht simuliert sondern lediglich auf Kompatibilität untersucht. Sobald das (U)CML-Modell (die Komponenten des Systems) ein modelliertes Verhalten[256] besitzen, kann das (U)CML-Modell ebenfalls simuliert werden.

Alle vier Prozessschritte des (U)CML-Modellbildungsprozesses werden, in den nachfolgenden Kapiteln ausführlich beschrieben. Begonnen wird mit dem Abschnitt Modellbildung.

Anmerkung:
Alle Phasen des (U)CML-Modellbildungsprozesses werden im Abschnitt „II *Fallstudie – Gleitschutzsystem*“ ab Seite 263 anhand eines ausführlichen Beispiels erläutert.

3.6.6.1. Modellbildung

Der wichtigste Schritt im (U)CML-Modellbildungsprozess ist die Modellierung des Systems in (U)CML. Die Modellbildung unterteilt sich dabei in die in der Abbildung 3.176 auf der nächsten Seite dargestellten drei aufeinander aufbauenden Teilbereiche – *Identifikation und Aufbereitung der Daten*, *Modellierung des Systems in (U)CML* sowie *Befüllung der Beschreibungsfelder*.

Begonnen wird die Modellbildung mit der Identifikation und Aufbereitung der vorhanden Daten und Informationen über das zu modellierende System. Anschließend folgt die eigentliche Modellierung des Systems in der Kompatibilitätsmodellierungssprache (U)CML. Nachdem das qualitative Modell des Systems vollständig in (U)CML modelliert wurde, folgt im letzten Schritt der Modellbildung die Befüllung der Beschreibungsfelder der Pakete, Komponenten sowie der Buchsen und Stecker.

Alle vorgestellten Modellbildungsschritte werden in den nachfolgenden Abschnitten ausführlich beschrieben und anhand eines einfachen Beispiels erläutert.

[255] Siehe hierzu die beiden Abbildungen 2.3 auf Seite 33 bzw. 2.2 auf Seite 33.
[256] Vgl. Abschnitt „3.6.2.3.2 *Verhalten einer (U)CML-Komponente*“ ab Seite 170.

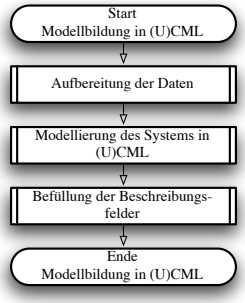


Abbildung 3.176.: Ablauf des Teilprozesses Modellbildung in (U)CML.

3.6.6.1.1. Aufbereitung der vorhandenen Daten aus unterschiedlichen Quellen

Der erste Prozessschritt bei der Modellierung eines real existierenden oder gedanklichen Systems ist die strukturierte Aufbereitung der vorhandenen Daten und Informationen über das zu modellierende System. In der Abbildung 3.177 ist die Identifikation und die strukturierte Aufbereitung der kompatibilitätsrelevanten Ausgangsdaten eines Systems dargestellt. In der Mitte der Abbildung sind unterschiedliche Ausgangsdaten (Quellen), mit deren Hilfe ein technisches System beschrieben werden kann, dargestellt. Im Fall eines eingebetteten softwarelastigen Systems sind dies z.B. Quellcodes, technische Zeichnungen, Stromlaufpläne, Bauteillisten, PDF-Dokumente[257] etc..

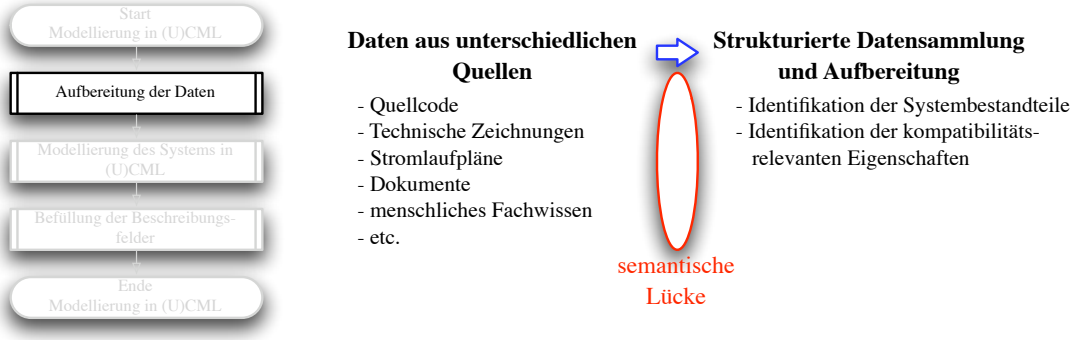


Abbildung 3.177.: Aufbereitung der vorhandenen Daten und Informationen.

Auf der rechten Seite der Abbildung ist das Ziel der Datenaufbereitung dargestellt (die Identifikation der Systembestandteile) sowie die Identifikation der kompatibilitätsrelevanten Eigenschaften des zu modellierenden Systems. Die Identifikation bzw. die Transformation der unterschiedlichen Ausgangsdaten in eine strukturierte Datensammlung kann auch als Überwindung der „semantischen Lücke“ gesehen werden. Beim Schließen der semantischen Lücke ist vor allem domänenspezifisches Fachwissen und Erfahrung bei der Identifikation der relevanten Eigenschaften eines Systems notwendig, um unnötige Bestandteile bzw. Eigenschaften, die für die Kompatibilitätsbestimmung unwichtig sind, von den relevanten Eigenschaften zu trennen, um das Modell des Systems nicht mit irrelevanten Eigenschaften zu überfrachten.

In der nachfolgenden Aufzählung werden die beiden angesprochenen Identifikations-Schritte – *Identifikation der Systembestandteile* sowie die *Identifikation der kompatibilitätsrelevanten Eigenschaften des Systems* – erläutert[258].

- **Identifikation der Systembestandteile**
  Im ersten Schritt werden sämtliche Ausgangsdokumente, die das zu modellierende System beschreiben, gesichtet. Dabei wird besonderes Augenmerk auf die Identifikation der Systembestandteile (der elektrischen/elektronischen, der mechanischen Baugruppen und Softwaremodule) gelegt. Im zweiten Schritt werden sämtliche identifizierten Baugruppen in entsprechende (U)CML-Pakete und -Komponenten transformiert. Dabei werden die Baugruppen als Pakete modelliert, während die (atomaren) Bestandteile einer Baugruppe als Komponenten modelliert werden.

  Die Abbildung 3.178 auf der nächsten Seite zeigt die Transformation einer real existierenden elektrischen Baugruppe in ein entsprechendes (U)CML-Modell.

  Auf der linken Seite der Abbildung ist ein Blockschaltbild einer real existierenden elektrischen Baugruppe, die aus zwei ICs besteht dargestellt, die sich auf einer Platine befinden. Auf der rechten Seite der Abbildung ist die identifizierte Baugruppe als (U)CML-Paket *Baugruppe*, die „innere Struktur“ der elektrischen Platine, also die beiden ICs, als Komponenten *IC1* und *IC2* innerhalb des Pakets *Baugruppe* dargestellt.

[257] Das Akronym *PDF* steht für **P**ortable **D**ocument **F**ormat [Inc07b]. Das Dokumenten-Format wurde von der Firma Adobe eingeführt [Ado07].

[258] Die Identifikation der kompatibilitätsrelevanten Eigenschaften eines Systems zur Überwindung der semantischen Lücke wurde bereits im Kapitel „2.3.2.1 *Identifikation der kompatibilitätsrelevanten Systemeigenschaften*“ ab Seite 59 angesprochen.

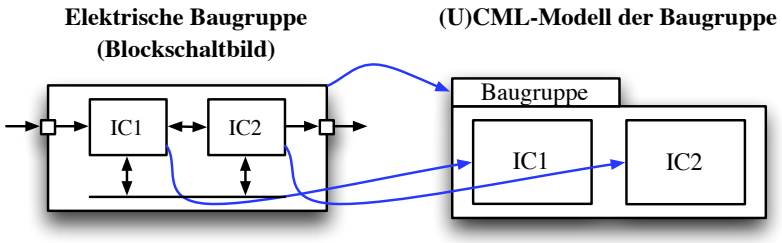


Abbildung 3.178.: Transformation einer real existierenden elektrischen Baugruppe in ein (U)CML-Modell.

- **Identifikation der kompatibilitätsrelevanten Eigenschaften des Systems**
  Nachdem im ersten Schritt die Systembestandteile des zu modellierenden Systems identifiziert wurden, folgt nun die Identifikation der kompatibilitätsrelevanten Eigenschaften der identifizierten Systembestandteile. Dazu werden, ausgehend von den Ausgangsdokumenten, sämtliche kompatibilitätsrelevanten Eigenschaften eines Systems identifiziert und den entsprechenden Systembestandteilen (Baugruppen) zugeordnet.

  Im obigen Beispiel (Abb.: 3.178) besitzt die elektrische Baugruppe z.B. folgende kompatibilitätsrelevante Eigenschaften:

  - Strom und Spannung der beiden ICs (IC1 und IC2)
  - Größe der Bauteile
  - Verlustleistung und Wärmeabgabe
  - Beschaltung der Anschlusspins der beiden ICs
  - etc.

  Die identifizierten kompatibilitätsrelevanten Eigenschaften werden für die Modellierung des Systems in (U)CML bzw. für die Befüllung der Beschreibungsfelder benötigt (quantitatives Modell).

Das **Ergebnis** der Aufbereitung der vorhandenen Daten ist die Identifikation der Pakete, Komponenten sowie der kompatibilitätsrelevanten Eigenschaften des zu modellierenden Systems. Diese Daten bilden die Ausgangsbasis für die weitere Modellierung des Systems.

Anmerkung:
Die Identifikation der Baugruppen eines Systems sowie ihre Eigenschaften wurde zum Teil bereits im Kapitel „2.2 *Modellbildung*“ ab Seite 31 erläutert.

3.6.6.1.2. Modellierung des Systems in (U)CML

Aufbauend auf den im letzen Abschnitt identifizierten Bestandteilen, also den Paketen und Komponenten, sowie ihrer kompatibilitätsrelevanten Eigenschaften folgt nun die Transformation der identifizierten Daten in ein qualitatives (U)CML-Modell des Systems. In der Abbildung 3.179 ist auf der linken Seite, der Prozessschritt *Modellierung des Systems in (U)CML* dargestellt, während auf der rechten Seite das Ergebnis des (U)CML-Modellierungsschritts, aufbauend auf den gesammelten Daten aus dem obigen Beispiel (Abb.: 3.178), abgebildet ist.

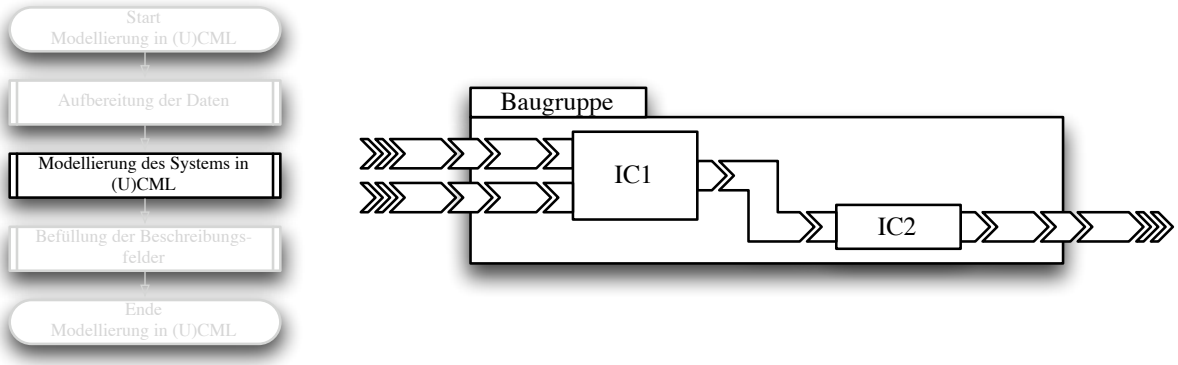


Abbildung 3.179.: Ablauf des Teilprozesses *Modellierung in (U)CML* .

Der Prozessschritt *Modellierung des Systems in (U)CML* teilt sich in zwei aufeinander aufbauende Teilaktivitäten auf. Zum einen ist dies die Modellierung der identifizierten Strukturelemente (Pakete/Komponenten) des Systems und zum anderen die Modellierung der Anschlüsse (Stecker/Buchsen/Schnittstellen) und Verbindungen (Flusspfeile) des Systems. Diese beiden Aktivitäten werden in der nachfolgenden Aufzählung dargestellt:

- **Identifikation der Struktur des Systems und Transformation der Strukturelemente in ein (U)CML-Modell**
  Der erste Schritt bei der Modellierung des Systems in (U)CML ist die Identifikation der Struktur des Systems und deren Umsetzung in (U)CML. Dazu werden die einzelnen identifizierten Baugruppen des Systems als *Pakete* und ihre inneren (atomaren) Bestandteile als *Komponenten* modelliert. Alle für die Modellierung der Struktur des Systems notwendigen Daten und Informationen wurden während des vorangegangenen Prozessschritts identifiziert.

  Das in der Abbildung 3.179 (rechts) modellierte System besteht aus den drei identifizierten Strukturelementen Systempaket *Baugruppe* und den beiden Komponenten *IC1* und *IC2*.

- **Identifikation und Modellierung der Anschlüsse (Stecker/Buchsen/Schnittstellen) und Verbindungen (Pfeile) von/zwischen Komponenten des Systems**
  Nachdem sämtliche Strukturelemente des Systems in (U)CML modelliert wurden, folgt die Identifikation und Modellierung der Stecker, Buchsen und Schnittstellen sowie die Modellierung der Verbindungen zwischen den Steckern/Buchsen und Schnittstellen des Systems. Sämtliche dafür notwendigen Informationen wurden bereits während des letzten Prozessschritts (*Aufbereitung der Daten*) erarbeitet.

  Das Modell in der Abbildung 3.179 besitzt drei Paketschnittstellen zur Systemumwelt sowie vier Flusspfeile innerhalb des Systempakets. Die vier Flusspfeile verbinden die Komponenten, genauer die Stecker und Buchsen der Komponenten, sowie die Paketschnittstellen des Systems untereinander. Des Weiteren besitzt das Systempaket *Baugruppe* zwei externe Systemeingangspfeile und einen externen Systemausgangspfeil über die das System mit der Umwelt kommuniziert.

Das **Ergebnis** der Modellierung ist ein vollständiges qualitatives (U)CML-Modell des Systems. Das qualitative (U)CML-Modell wird anschließend, im nächsten Prozessschritt mit konkreten Werten befüllt. Genauer: Die Beschreibungsfelder der Pakete, Komponenten, Stecker und Buchsen werden mit den im ersten Prozessschritt identifizierten kompatibilitätsrelevanten Eigenschaften befüllt.

3.6.6.1.3. Befüllung der Beschreibungsfelder mit den Eigenschaften des Systems

Nachdem im letzten Modellierungsschritt das qualitative Modell des Systems entstanden ist, werden nun die Beschreibungsfelder der Pakete/Komponente sowie der Stecker/Buchsen und Schnittstellen mit konkreten Werten befüllt. Sämtliche Werte wurden während der ersten Phase des (U)CML-Modellbildungsprozesses identifiziert[259] und beschrieben.

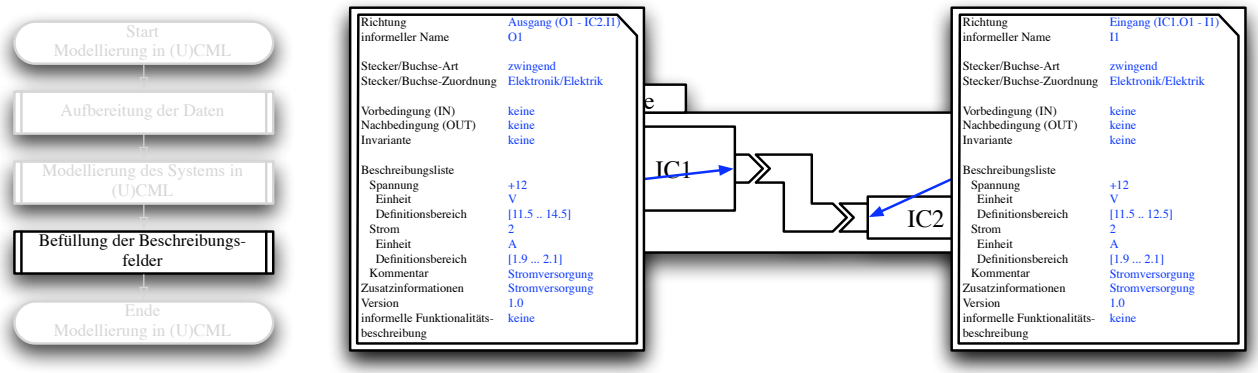


Abbildung 3.180.: Prozessschritt: *Befüllung der Beschreibungsfelder*.

Die Abbildung 3.180 zeigt exemplarisch für einen Stecker und eine Buchse die Befüllung der entsprechenden Beschreibungsfelder mit quantitativen Werten. Die in den Beschreibungsfeldern eingetragenen Werte wurden während der ersten Phase des (U)CML-Modellbildungsprozesses als kompatibilitätsrelevante Eigenschaften identifiziert und während der Modellierungsphase als Stecker/Buchsen bzw. Verbindungen modelliert.

Das **Ergebnis** der Befüllung der Beschreibungsfelder ist ein vollständiges, mit allen kompatibilitätsrelevanten Eigenschaften befülltes, quantitatives Modell des Systems. Dieses Modell bildet die Ausgangsbasis für den anschließenden Kompatibilitätstest. Jedoch bevor mit dem Kompatibilitätstest begonnen werden kann, müssen Regeln für die Bestimmung der Kompatibilität festgelegt werden.

3.6.6.2. Regelwerk

In diesem Schritt des (U)CML-Modellbildungsprozesses werden die Kompatibilitätsregeln, also das projektunabhängige bzw. das projektabhängige Regelwerk festgelegt, aufgrund dessen das quantitative Modell des Systems auf Kompatibilität untersucht werden soll. Die Abbildung 3.181 auf der nächsten Seite zeigt auf der linken Seite den Prozessschritt *Regelwerk erstellen*. Auf der rechten Seite der Abbildung ist die Struktur des Kompatibilitätsregelwerks[260] dargestellt.

[259] Siehe hierzu: Kapitel „3.6.6.1.1 *Aufbereitung der vorhandenen Daten aus unterschiedlichen Quellen*“ ab Seite 245.
[260] Siehe hierzu: Kapitel „3.6.5.2 *Projektunabhängiges und projektabhängiges Kompatibilitätsregelwerk*“ ab Seite 239.

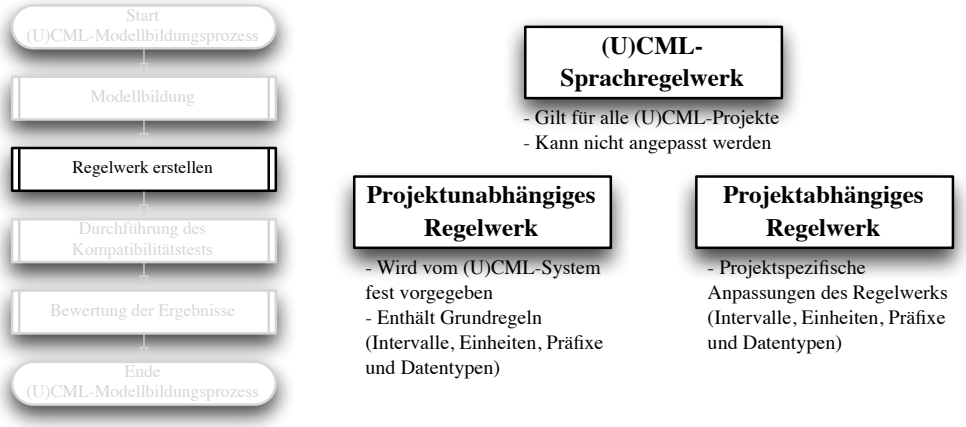


Abbildung 3.181.: Prozessschritt: *(U)CML-Kompatibilitätsregelwerk* erstellen.

Für die anschließende Kompatibilitätsbestimmung des Beispielsystems aus den vorangegangenen Abschnitten des (U)CML-Modellbildungsprozesses ist keine Anpassung des projektspezifischen Regelwerks notwendig, da alle Regeln bereits im projektunabhängigen Regelwerk enthalten sind.

Das **Ergebnis** der Festlegung des Kompatibilitätsregelwerks ist ein an die Bedürfnisse des Systems angepasstes Regelwerk zur Überprüfung des Systems auf Inkompatibilitäten. Nachdem nun sowohl das System als auch das Kompatibilitätsregelwerk vorliegen, kann mit der Kompatibilitätsprüfung begonnen werden.

3.6.6.3. Durchführung der Kompatibilitätsprüfung

In den vorangegangenen Abschnitten des (U)CML-Modellbildungsprozesses wurde das quantitative (U)CML-Modell des Systems schrittweise aufgebaut und im (U)CML-Flussdiagramm modelliert sowie das Kompatibilitätsregelwerk definiert, mit dessen Hilfe das Modell auf Kompatibilitätsfehler untersucht werden kann. Im Prozessschritt *Durchführung des Kompatibilitätstests* wird das zuvor erstellte quantitative (U)CML-Modell des Systems anhand des Kompatibilitätsregelwerks auf Inkompatibilitäten untersucht.

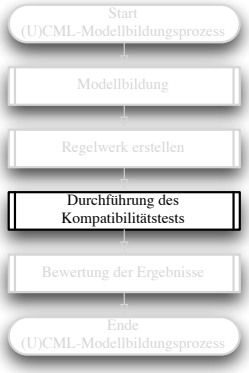


Abbildung 3.182.: Prozessschritt: *Durchführung des Kompatibilitätstests*.

Die Kompatibilitätsbestimmung unterteilt sich in drei aufeinander aufbauende Teilschritte. Zunächst wird das (U)CML-Modell auf (U)CML-Sprachregelfehler[261] untersucht. Wenn das Modell keine Sprachregelfehler enthält, folgt anschließend die Überprüfung des Modells anhand des projektunabhängigen sowie des projektabhängigen Regelwerks[262]. Die Abbildung 3.183 auf der nächsten Seite (links) zeigt den Ablaufplan des Kompatibilitätstests als Flussdiagramm. Auf der rechten Seite ist der entscheidende Ausschnitt aus dem Modell (Abb.: 3.180) dargestellt, wo es zu einer Inkompatibilität zwischen dem Sender *IC1* (Stecker) und dem Empfänger *IC2* (Buchse) kommt.

Der dargestellte Kompatibilitätsfehler wird durch eine Intervallverletzung des Senders *IC1* (Stecker 1) ausgelöst. Er überträgt eine Versorgungsspannung von $+12\text{V}$ mit einem Gültigkeitsintervall von $[11.5 \ldots 14.5]\text{V}$, während der Empfänger *IC2* (Buchse 1) lediglich eine Versorgungsspannung von $+12\text{V}$ mit einem Gültigkeitsintervall von $[11.5 \ldots 12.5]\text{V}$ erwartet. Diese Inkompatibilität würde bei einem realen System zum „Durchbrennen“ des Eingangs des *IC2* führen.

[261] Siehe hierzu: Kapitel „3.6.5.1 *(U)CML-Sprachregelwerk*“ ab Seite 236.
[262] Siehe hierzu: Kapitel „3.6.5.2.1 *Projektunabhängiges Kompatibilitätsregelwerk*“ ab Seite 239. sowie 3.6.5.2.2 *Projektabhängiges Kompatibilitätsregelwerk* ab Seite 242.

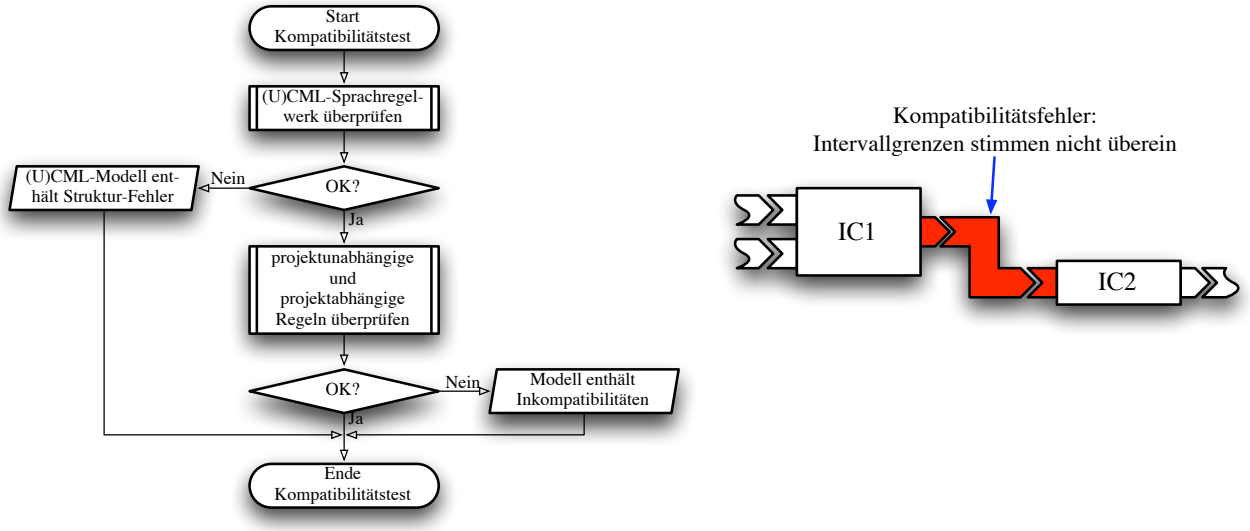


Abbildung 3.183.: Ablauf des (U)CML-Kompatibilitätstest.

Ergebnis des Kompatibilitätstests: Ein (U)CML-Modell eines Systems, das auf Kompatibilitätsfehler untersucht worden ist. Sämtliche Inkompatibilitäten werden im Modell rot markiert und angezeigt.

3.6.6.4. Bewertung der Ergebnisse

Abgeschlossen wird der (U)CML-Modellbildungsprozess mit der Bewertung der Ergebnisse des zuvor durchgeführten Kompatibilitätstests. Zur Bewertung der Ergebnisse gehört auch die Identifikation geeigneter Gegenmaßnahmen, um Inkompatibilitäten im Modell zu beseitigen.

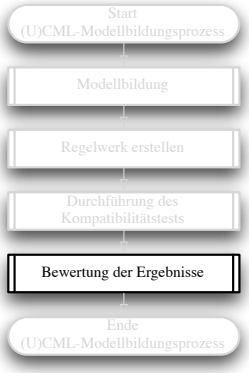


Abbildung 3.184.: Prozessschritt: *Bewertung der Ergebnisse* des Kompatibilitätstests.

Das Beispielsystem aus Abbildung 3.179 enthält eine Intervallverletzung. Dieser Kompatibilitätsfehler würde, wie bereits erwähnt, zu einer Zerstörung des Eingangs von *IC2* führen und muss aus diesem Grund beseitigt werden. Um den Kompatibilitätsfehler zu beseitigen, kann entweder der Sender (*IC1*) oder der Empfänger (*IC2*) angepasst werden.

Ergebnis: Protokoll aller notwendigen Änderungen bzw. Anpassungen am (U)CML-Modell des Systems, so dass das System in einen kompatiblen Zustand überführt werden kann.

Anmerkung:
Nachdem die Inkompatibilitäten aus einem Modell entfernt worden sind, muss das gesamte Modell noch einmal auf Kompatibilität untersucht werden. Erst wenn auch bei dieser Überprüfung keine Inkompatibilitäten lokalisiert werden, kann davon ausgegangen werden, dass das Modell im Rahmen seiner Realitätsnähe keine Kompatibilitätsfehler enthält.

3.6.7. Modellierung des Beispielsystems „KOMPTEST“ in (U)CML

Nachdem nun sämtliche Sprachelemente der (U)CML eingeführt und kurz erläutert wurden, folgt in diesem Kapitel die Modellierung des Beispielsystems „KOMPTEST“, wie es im Kapitel 3.3 eingeführt worden ist in (U)CML. Dazu wird zunächst die Struktur

des Beispielsystems und im Anschluss das Verhalten des Systems modelliert (Szenario 1). Nachdem das Grundmodell erstellt und auf Kompatibilität untersucht worden ist, wird, wie im Szenario 2 beschrieben, eine Komponente des Systems gegen eine andere ersetzt und das neue System ebenfalls auf Kompatibilität untersucht sowie die Ergebnisse bewertet.

Szenario 1: Modellierung der Struktur des Beispielsystems „KOMPTEST“[263]
Im Kapitel 3.6.2.6.4 (vgl. Abbildung: 3.133 auf Seite 210) wurde bereits die Struktur des Beispielsystems „KOMPTEST“ inklusive aller notwendigen Beschreibungsfelder vollständig in (U)CML modelliert und als Flussdiagramm dargestellt.

Ergebnis: Die Struktur des Beispielsystems „KOMPTEST“ konnte vollständig in (U)CML modelliert werden. Zusätzlich zur Strukturbeschreibung wurden sämtliche Eigenschaften (Attribute) der Komponenten mit Hilfe der Beschreibungsfelder erfasst und beschrieben. Aufbauend auf dem Modell und dem generischen projektunabhängigen Kompatibilitätsregelwerk konnte das Modell auf Kompatibilität untersucht werden. Dabei traten keine Inkompatibilitäten auf.

Szenario 1: Modellierung des Verhaltens des Beispielsystems „KOMPTEST“
Aufgrund der Tatsache, dass die Verhaltensbeschreibung des gesamten Beispielsystems zu umfangreich wäre, wird hier beispielhaft das Kommunikationsverhalten der beiden Komponenten *Element1* und *Element2* untersucht und das Ergebnis bewertet. Die Abbildung 3.185 zeigt das Kommunikations-MSC der beiden Komponenten *Element1* und *Element2*.

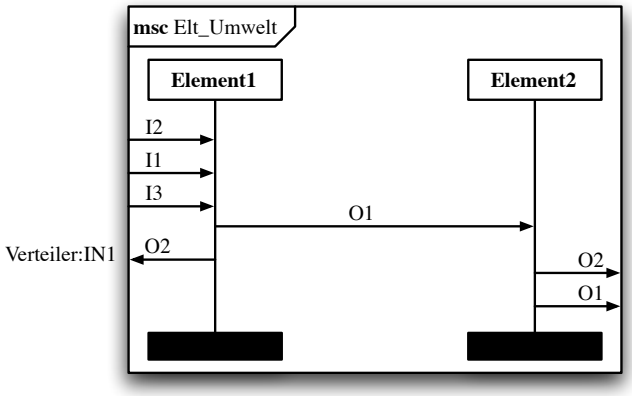


Abbildung 3.185.: MSC der beiden Komponenten *Element1* und *Element2* des Beispielsystems.

Ergebnis: Mit Hilfe der MSCs kann das (Kommunikations-) Verhalten von (U)CML-Komponenten beschrieben und auf Kompatibilität untersucht werden. Das abgebildete MSC enthält keine Inkompatibilität.

Szenario 2: Austausch der Komponente *Element2* gegen die neue Komponente *ElementX*[264]
Aufbauend auf dem Modell aus Szenario 1 folgt die Beschreibung des zweiten Szenarios aus der Einleitung, nämlich der Austausch einer Komponente des Beispielsystems. Die Abbildung 3.186 zeigt den relevanten Ausschnitt aus dem statischen (U)CML-Modell des Beispielsystems „KOMPTEST“ sowie die auszutauschende Komponente *Element2* (links).

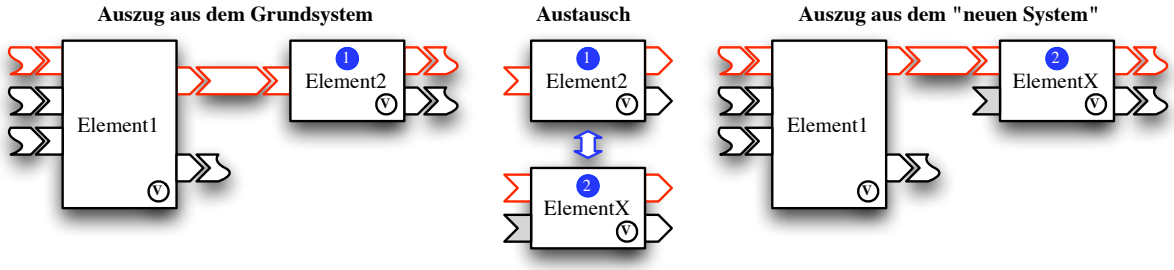


Abbildung 3.186.: (U)CML-Beispielsystem – Austausch der Komponente *Element2* durch die neue Komponente *ElementX*.

In der Bildmitte ist der Austausch der alten Komponente *Element2* (1) gegen die neue Komponente *ElementX* (2) dargestellt. Im rechten Segment der Abbildung 3.186 ist das neu entstandene Beispielsystem abgebildet, in dem die Komponente *Element2* gegen die Komponente *ElementX* ersetzt wurde. Die neue Komponente *ElementX* besitzt im Gegensatz zur alten Komponente einen optionalen Eingang mehr. Dieser Eingang ist jedoch innerhalb des Systems nicht angeschlossen. Dies verursacht keine Kompatibilitätsprobleme, da der Eingang optional ist, und somit nicht angeschlossen werden muss. Auch ohne die Beschaltung des optionalen Eingangs der Komponente *ElementX* muss diese einwandfrei funktionieren und darf auch das restliche System nicht negativ beeinflussen. In diesem Beispiel verursacht der Austausch der Komponente keine Inkompatibilität.

[263] Siehe hierzu insbesondere das Kapitel „3.3.1 *Szenario 1: Modellierung des Beispielsystems „KOMPTEST““* ab Seite 110.
[264] Siehe hierzu insbesondere das Kapitel „3.3.2 *Szenario 2: Austausch einer Komponente des Beispielsystems „KOMPTEST““* ab Seite 111.

Ergebnis: Mit Hilfe der (U)CML ist es möglich, die Anforderung – Modellierung von Austauschkompatibilität – aus der Einleitung vollständig zu erfüllen. Das heißt: Sowohl die Struktur als auch das Verhalten kann überprüft und das Ergebnis der Prüfung graphisch dargestellt werden.

Anmerkung:
Wenn der neu hinzugekommene Eingang der Komponente *ElementX* zwingend wäre, würde das neue Beispielsystem sowohl einen (U)CML-Sprachregelwerksfehler als auch eine Kompatibilitätsregelwerksverletzung verursachen. Diese Verletzung der Kompatibilität würde rot markiert und im Flussdiagramm dargestellt werden. Die Abbildung 3.187 zeigt die Kompatibilitätsverletzung, die entstehen würde, wenn die Komponente *Element2* (1) gegen die Komponente *ElementY* (2) mit zwingendem zweiten Eingang ersetzt werden würde.

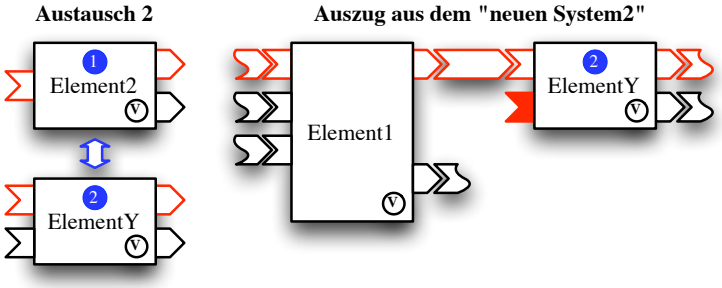


Abbildung 3.187.: (U)CML-Beispielsystem – Austausch der Komponente *Element2* durch die neue Komponente *ElementY*.

3.6.8. Anpassung und Erweiterung der (U)CML auf andere Fachdisziplinen

Mit Hilfe der (U)CML können eingebettete Systeme, bestehend aus Mechanik, Elektronik und Software, graphisch modelliert und auf Kompatibilität untersucht und bewertet werden. Aufgrund des modularen Aufbaus sowie der flexiblen Struktur der (U)CML ist es möglich, sie ohne großen Aufwand auf andere Domänen anzupassen und zu erweitern. In der nachfolgenden Aufzählung sind die wesentlichen drei Schritte aufgezeigt, mit deren Hilfe die (U)CML für die Modellierung anderer Fachbereiche vorbereitet werden kann.

Generisches Vorgehen bei der Erweiterung der (U)CML auf andere Domänen:

1. **Einführung neuer Beschreibungsfeldinhalte für Komponenten und Anschlüsse**
   Als erster Schritt bei der Anpassung der (U)CML auf andere Fachdisziplinen ist die domänenspezifische Identifikation und Beschreibung der Eigenschaften der jeweiligen Domäne. Die identifizierten Eigenschaften (Attribute) werden als entsprechende Eigenschaftsfelder in die (U)CML-Beschreibungsfelder eingetragen. Anhand der Eigenschaftsfelder können, im dritten Schritt, die Regeln für die Bestimmung der „Kompatibilität“ beschrieben und festgelegt werden.

2. **Anpassung des Farbschemas**
   Als nächster Schritt muss das bereits existierende Farbschema der (U)CML erweitert und an die speziellen Bedürfnisse der zu modellierenden Domäne angepasst werden. Dazu wird der neuen Domäne eine noch nicht vergebene Farbe zugewiesen, die für die Darstellung der (U)CML-Elemente im Flussdiagramm verwendet wird.

3. **Einführung domänenspezifischer Regeln**
   Der wichtigste Punkt bei der Erweiterung bzw. Anpassung der (U)CML auf andere Anwendungsfelder ist die Erweiterung des Kompatibilitätsregelwerks. In diesem Schritt müssen Regeln definiert werden, mit deren Hilfe die in Schritt 1 definierten Eigenschaften verknüpft und bewertet werden können. Anhand des so erweiterten Regelwerks können die Eigenschaften der neuen Domäne ebenfalls auf Kompatibilität überprüft und bewertet werden.

   Anmerkung:
   Im Regelwerk ist es auch möglich, Regeln zu definieren, die einzelne Disziplinen miteinander verknüpfen.

Im nächsten Abschnitt dieses Kapitels wird die Erweiterung der (U)CML auf eine andere Domäne exemplarisch durchgeführt und anhand eines einfachen Beispiels erläutert.

3.6.8.1. Modellierung von Anforderungen mit Hilfe der (U)CML

Eine mögliche Erweiterung der (U)CML zu einer universell einsetzbaren Modellierungssprache, ist die Ausdehnung des (U)CML-Sprachschatzes auf andere Domänen wie z.B. auf die Modellierung von Anforderungen (Requirements). Um die einfache und effiziente Anpassung der (U)CML auf die Modellierung von Anforderungen zu demonstrieren, wird nach dem im vorangegangenen Abschnitt eingeführten Verfahren vorgegangen.

1. **Einführung neuer Beschreibungsfeldinhalte für Komponenten und Anschlüsse**
   Für die Modellierung von Anforderungen ist es notwendig, diese so zu beschreiben, dass sie mit „erfüllt“ oder „nicht erfüllt“ bewertet werden können. Im Beschreibungsfeld z.B. eines Kommunikationssteckers/-buchse werden auf der linken Seite die beiden Eigenschaftsfelder *Anforderung* und *Ergebnis* hinzugefügt. Auf der rechten Seite des Beschreibungsfelds wird

dann der *Anforderungstext* sowie das erwartete *Ergebnis* in die beiden Spezifikationsfelder eingetragen. Die Eigenschaftsfelder *Richtung*, *informeller Name*, *Stecker-/Buchse-Art*, *Stecker-/Buchse-Zuordnung* sowie das Eigenschaftsfeld *Zusatzinformation* des Kommunikationssteckers/-buchsebeschreibungsfelds werden unverändert übernommen.

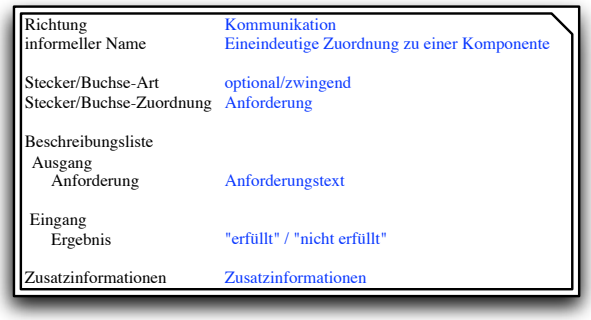


Abbildung 3.188.: Beschreibungsfeld für Anforderungen.

Die Abbildung 3.188 zeigt das an die Domäne Anforderungen angepasste Kommunikationsstecker/-buchsebeschreibungsfeld. Auf die gleiche Weise werden auch die restlichen Beschreibungsfelder an die neue Domäne angepasst.

2. **Anpassung des Farbschemas**
   Für die Modellierung von Anforderungen in (U)CML soll die Farbe violett verwendet werden. Das heißt: Alle Komponenten, Stecker-, Buchsen- und Pfeilarten, die für die Modellierung von Anforderungen verwendet werden sollen, werden im Flussdiagramm in der Farbe violett dargestellt.

3. **Einführung domänenspezifischer Regeln**
   Eine Anforderung an eine Komponente, einen Stecker oder eine Buchse kann entweder erfüllt oder nicht erfüllt sein.

   (U)CML Kompatibilitätsregel 3.7 – *(U)CML-Sprachregelwerk* –
   *Eine Anforderung kann entweder **erfüllt** oder **nicht erfüllt** sein. Ist sie nicht erfüllt, so wird der entsprechende zusammengesetzte Pfeil (bzw. die einzelnen Bestandteile) rot markiert.*

   Ist eine Anforderung nicht erfüllt, wird der entsprechende zusammengesetzte Pfeil rot eingefärbt. In diesem Fall muss der Fehler manuell durch den Benutzer beseitigt werden, so wie es im Kapitel „3.6.6.4 *Bewertung der Ergebnisse*“ ab Seite 249 gezeigt wurde.

Im nachfolgenden Beispiel ist eine Anforderung an eine Komponente in (U)CML modelliert und dargestellt.

Beispiel 81: Modellierung von Anforderungen in (U)CML
Die Anforderung: „Die Komponente *NAND* muss zwei NAND-Gatter mit je zwei Eingängen und einem Ausgang haben“ wird in (U)CML folgendermaßen modelliert und dargestellt.

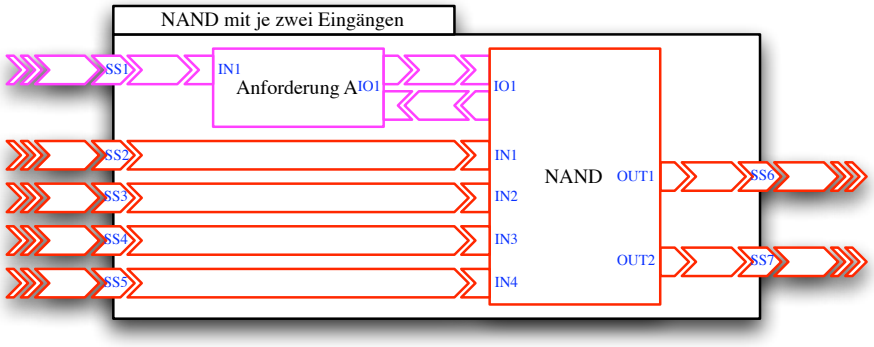


Abbildung 3.189.: Anforderungen an eine Komponente.

Die Abbildung 3.189 zeigt ein System, in dem sich die Komponente *NAND* befindet. Die Komponente hat, wie in der Anforderung gefordert, zwei getrennte NAND-Gatter mit je zwei Eingängen und einem Ausgang.

Gatter Bezeichnung	**Eingänge**	**Ausgänge**
NAND-Gatter 1	IN1, IN2	OUT1
NAND-Gatter 2	IN3, IN4	OUT2

Die Anforderung an die Komponente *NAND* wird mit Hilfe der Komponente *Anforderung A* modelliert. Sie bekommt die Anforderung an die Komponente *NAND* von der Systemumwelt übertragen (*IN1*) und verifiziert diese durch die Kommunikationsverbindung zwischen den beiden Komponenten

$$Anforderung\ A_{Z:St_{Kom}:IO1} \overset{F_{Z:K:D}}{\rightleftharpoons} NAND_{Z:Bu_{Kom}:IO1}.$$

Ergebnis: Die Anforderung an das System bzw. an die Komponente *NAND* ist vollständig erfüllt. Das System ist fehlerfrei. □

3.6.8.2. Erweiterung der (U)CML auf weitere Domänen

Theoretisch ist es möglich, den (U)CML-Sprachumfang auf nahezu beliebig viele verschiedene Anwendungsgebiete anzupassen und zu erweitern. Durch die Erweiterung der (U)CML auf weitere Fachdisziplinen kann der Grundgedanke der (U)CML, eine einfach zu erlernende domänenübergreifende Modellierungssprache für verschiedene Anwendungsgebiete bereitzustellen, vorangetrieben werden. In der Abbildung 3.190 sind einige mögliche Erweiterungen der (U)CML auf andere Domänen beispielhaft dargestellt.

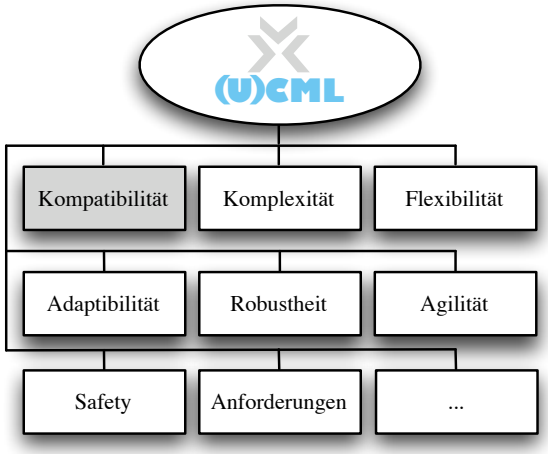


Abbildung 3.190.: Erweiterung der (U)CML auf weitere Domänen.

Sämtliche in der Abbildung 3.190 dargestellten Erweiterungen können nach dem im Kapitel „3.6.8 *Anpassung und Erweiterung der (U)CML auf andere Fachdisziplinen*“ ab Seite 251 eingeführten generischen Erweiterungsschema der (U)CML vorgenommen werden.

3.6.9. Werkzeugunterstützung

Die (U)CML ist wie die meisten anderen textuellen oder graphischen Modellierungssprachen (VHDL [Wik07y][Mäd07][265], SE Element-Konzept, UML etc.) ohne Werkzeugunterstützung nur bedingt einsetzbar. Zum einen ist es sehr mühsam alle Diagramme per Hand in einem Zeichenprogramm zu zeichnen und insbesondere konsistent zu halten und zum anderen ist es extrem aufwendig, anschließend alle Beschreibungsfelder zu überprüfen um die Kompatibilität feststellen zu können.

Aus diesem Grund wurde ein spezieller plattformunabhängiger prototypischer Editor für die (U)CML, der – *(U)CML-ed* – entwickelt, mit dem alle Diagramme gezeichnet und die Kompatibilitätsregeln beschrieben werden können. Des Weiteren kann mit dem Editor der eigentliche Kompatibilitätstest automatisch durchgeführt werden. Eine genauere Beschreibung der Funktionalität des *(U)CML-ed* finden Sie im Kapitel „3.8 *(U)CML-ed, ein graphischer Editor für die Kompatibilitätsmodellierungssprache (U)CML*“ ab Seite 254.

3.7. Vergleich und Bewertung der vorgestellten Modellierungssprachen

Nachdem in den vorangegangenen Kapiteln verschiedene Modellierungssprachen und -konzepte für komplexe technische Systeme vorgestellt wurden, folgt in diesem Abschnitt eine abschließende Bewertung der unterschiedlichen Sprachen anhand der im Kapitel „*Anforderungen an eine Kompatibilitätsmodellierungssprache*“ beschriebenen Anforderungen an eine domänenübergreifende, einfach zu erlernende Kompatibilitätsmodellierungssprache. Bei dieser Bewertung wird vor allem auf die Modellierung sowie die Bestimmung und Darstellung von Kompatibilität Wert gelegt.

Die folgende Abbildung 3.191 auf der nächsten Seite zeigt, anschaulich die Überlappung der vorgestellten Modellierungssprachen und -konzepte.

[265] VHDL steht für **V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage.

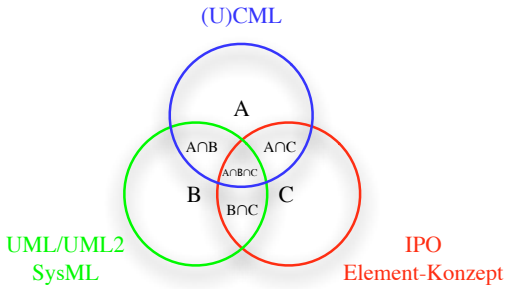


Abbildung 3.191.: Schnittmenge der unterschiedlichen Modellierungstechniken für komplexe technische Systeme.

Dabei zeigt sich, dass es Konzepte und Techniken gibt, die in allen fünf Modellierungssprachen und -methoden gleichermaßen vorhanden sind (der „Kern“: $A \cup B \cup C$). So wurde gezeigt, dass die für die Kompatibilitätsmodellierung und -bestimmung grundlegenden Techniken in allen vorgestellten Modellierungssprachen und -konzepten prinzipiell vorhanden sind, jedoch ist bis auf die (U)CML keine Modellierungssprache in der Lage, alle Anforderungen an eine Kompatibilitätsmodellierungssprache vollständig zu erfüllen. So fehlen den anderen Modellierungssprachen beispielsweise das für die Kompatibilitätsbestimmung zwingend notwendige Regelwerk (Bereich A), in dem genau festgelegt werden kann, unter welchen Umständen zwei Eigenschaften als kompatibel zueinander bewertet werden können. Im Gegenzug dazu kann beispielsweise die (U)CML noch nicht so flexibel in der Entwicklung eingesetzt werden wie beispielsweise die UML/UML2 oder die SysML (Bereich B). Den Bereich C bilden im Diagramm die eher „beschreibenden“ Modellierungssprachen, zu denen beispielsweise das Systems Engineering Element-Konzept zählt. Dafür kann mit ihrer Hilfe sehr einfach ein komplexes System ohne viel Formalismus beschrieben werden. Jedoch gerade für die Kompatibilitätsmodellierung und -bestimmung ist eine exakte formale Beschreibung, wie beispielsweise durch die Axiome der Mathematik, zwingend notwendig.

Eine andere Art der Bewertung liefert die folgende Tabelle. Grundlage für die in der Tabelle 3.12 eingetragen Anforderungen an eine domänenunabhängige graphische Kompatibilitätsmodellierungssprache liefern die Anforderungen (vgl. „*Anforderungen an eine Kompatibilitätsmodellierungssprache*“), die während des Forschungsvorhabens MOKOMA gesammelt worden sind.

Legende zur Tabelle 3.12:
Die mit (*) markierten Einträge bedeuten, dass die Eigenschaft nicht direkt in der Sprache modelliert, jedoch durch spezielle Sprachkonstrukte teilweise nachgebildet werden kann.

Ergebnis: Jede der in der Tabelle 3.12 bewerteten Modellierungssprachen und -methoden hat spezielle Stärken und Schwächen die sie für ein spezielles Einsatzgebiet auszeichnet. Das Systems Engineering Element-Konzept zum Beispiel kann zur Beschreibung sehr unterschiedlicher technischer Systeme verwendet werden, ist jedoch nicht für die Modellierung komplexer Systeme geeignet, da eine entsprechende Werkzeugunterstützung fehlt. Für die beiden Modellierungssprachen UML/UML2 und SysML gilt: Die UML/UML2 ist speziell für die graphische Modellierung von Software entwickelt worden und eignet sich nur bedingt für die Modellierung von Hardware. Demgegenüber ist die SysML eine Sprache, mit der sowohl Hard- als auch Software (u.a. durch die Unterstützung der UML) graphisch modelliert werden kann. Beide Modellierungssprachen haben jedoch keine Unterstützung für die Modellierung und Bestimmung von Kompatibilität, da sie, aufgrund ihres breiten Einsatzgebietes nicht dafür ausgelegt sind.

Die (U)CML hingegen wurde speziell für die interdisziplinäre Modellierung von Hard- und Software in einem gemeinsamen Modell sowie die Verifikation dieses Modells auf Kompatibilität entwickelt. Aus diesem Grund kann die (U)CML die meisten der in der Tabelle 3.12 aufgelisteten Anforderungen erfüllen. Die (U)CML ist jedoch im Gegensatz zu den beiden weit verbreiteten Modellierungssprachen UML/UML2 und SysML noch nicht außerhalb eines eng beschränkten Einsatzgebietes erprobt worden, so dass noch keine abschließenden Aussagen über die Mächtigkeit der (U)CML zum gegenwärtigen Zeitpunkt gemacht werden können. Ein weiters Problem der (U)CML besteht darin, dass der dazugehörige Editor – *(U)CML-ed* – das Prototypenstadium noch nicht verlassen hat und noch nicht alle Sprachelemente der (U)CML mit Hilfe des Editors modelliert werden können. Dieser Nachteil verliert jedoch an Gewicht, wenn man die Tatsache, dass sich der Editor momentan noch in der Entwicklung befindet, in Betracht zieht.

3.8. *(U)CML-ed*, ein graphischer Editor für die Kompatibilitätsmodellierungssprache (U)CML

Im Kapitel „3.6 *Einführung in die Kompatibilitätsmodellierungssprache (U)CML*“ ab Seite 153 wurden sämtliche theoretischen Grundlagen der graphischen Kompatibilitätsmodellierungssprache (U)CML ausführlich vorgestellt und anhand von einfachen Beispielen erläutert. Aufbauend auf den dort aufgezeigten theoretischen Grundlagen der (U)CML wird in diesem Kapitel der Prototyp des graphischen plattformunabhängigen CASE-Werkzeugs *(U)CML-ed* vorgestellt[266], mit dem technische Systeme und insbesondere softwarelastige eingebettete Systeme, in der graphischen Kompatibilitätsmodellierungssprache (U)CML modelliert und anhand des Kompatibilitätsregelwerks automatisch auf Kompatibilität untersucht werden können.

[266] Genauere Informationen sowohl zur graphischen Kompatibilitätsmodellierungssprache (U)CML, als auch zum CASE-Werkzeug *(U)CML-ed* finden Sie unter [KB07], [Eck06], [Gaj06], [Zim07] sowie [Tea08].

	IPO	SE Element	SysML	UML/UML2	(U)CML (-ed)
Hauptanforderungen					
Aufbauend auf bereits existierenden Modellierungssprachen und -techniken	Ja	Ja	Ja	Ja	Ja
Domänenunabhängigkeit (gleichermaßen einsetzbar in der Informatik, dem Maschinenbau und der Elektrotechnik)	Teils	Teils	Teils	Teils	Ja
Modellierung von Hard- und Software in einem gemeinsamen Modell	Teils	Teils	Teils	Nein (*)	Ja
Sowohl graphische als auch textuelle Notation aller Sprachelemente	Nein	Nein	Nein (*)	Nein (*)	Ja
Definition von Kompatibilitätsregeln für Hard- und Software	Nein	Nein	Nein	Nein	Ja
Modellierung von neuen bzw. erweiterten Datentypen	Nein	Nein (*)	Nein (*)	Nein (*)	Ja
Beschreibung von Intervallen	Nein	Nein (*)	Nein (*)	Nein (*)	Ja
Verbindung von Datentyp und Einheiten	Nein	Nein (*)	Nein (*)	Nein (*)	Ja
Graphische und textuelle Notation der Kompatibilitätsregeln	Nein	Nein	Nein	Nein	Ja
Graphische Bewertung, Auswertung und Darstellung von Kompatibilität bzw. Inkompatibilität	Nein	Nein	Nein	Nein	Ja
Interaktive graphische Modellierung und Darstellung von Systemen	Nein	Ja	Ja	Ja	Ja
Nebenanforderungen					
Einfache Erweiterung auf andere Fachgebiete (z.B. Automobilindustrie, Raumfahrt, etc.)	Ja	Ja	Ja	Ja	Ja
Sowohl von Informatikern als auch von Ingenieuren einfach zu erlernen	Ja	Ja	Ja	Ja	Ja
Einbettung der Modellierungssprache in den generischen Kompatibilitätsprozess	Teils	Teils	Teils	Teils	Ja
Entwicklung bzw. Anpassung eines Werkzeugs	Nein	Ja	Nein	Nein	Ja
Beachtung der SIL-Level	Nein	Nein	Nein	Nein	Ja

Tabelle 3.12.: Vergleich von IPO, SE Element, SysML , UML/UML2 und (U)CML / *(U)CML-ed*

Abbildung 3.192.: (U)CML in Theorie und Anwendung.

Von der graphischen Kompatibilitätsmodellierungssprache (U)CML zum graphischen Editor *(U)CML-ed*
Um die graphische Kompatibilitätsmodellierungssprache (U)CML effektiv im „Alltagsgeschäft“ einsetzen zu können, ist es zwingend notwendig, von der reinen „Zeichenblattdarstellung“, so wie sie im Kapitel 3.6 vorgestellt wurde, hin zu einer Werkzeugunterstützung zu gelangen, mit deren Hilfe die graphischen Elemente der (U)CML schnell und einfach gezeichnet und dargestellt werden können. Zusätzlich zur graphischen Darstellung der (U)CML-Sprachelemente kann das Kompatibilitätsregelwerk im Editor angelegt und manipuliert werden, mit dem das erstellte (U)CML-Modell auf Kompatibilität untersucht und bewertet werden kann. Außerdem enthält der Editor sämtliche für die Kompatibilitätsüberprüfung und -bewertung notwendigen Testroutinen sowie eine textuelle und graphische Darstellung der Ergebnisse des Kompatibilitätstests.

Um von der reinen Papierform hin zu einem einsetzbaren Werkzeug zu gelangen, wurde und wird das graphische plattformunabhängige Werkzeug *(U)CML-ed* als Studentenprojekt sowohl am Lehrstuhl für Raumfahrttechnik (LRT) – Systems Engineering Gruppe – als auch am Lehrstuhl für Software- und Systems Engineering (S&SE) der Technischen Universität München, entwickelt. Der Editor ist dabei als prototypischer Demonstrator gedacht, um zum einen Studenten die unterschiedlichen Konzepte der Kompatibilitätsmodellierung und -bewertung mit (U)CML nahe zu bringen, und zum anderen, um Studentengruppen das Vorgehen und die Entwicklung von Software anhand des umfangreichen Softwareprojekts *(U)CML-ed*, zu demonstrieren. Dabei arbeiten die Studenten größtenteils in kleinen Entwicklerteams an Teilstücken des Editors unter Anleitung von Mitarbeitern der beiden Lehrstühle. Die Abbildung 3.193 auf der nächsten Seite zeigt das Hauptfenster des plattformunabhängigen graphischen Editors *(U)CML-ed*, ausgeführt unter den drei Betriebssystemen: Suse Linux, Microsoft Windows Vista und Apple Mac OS-X.

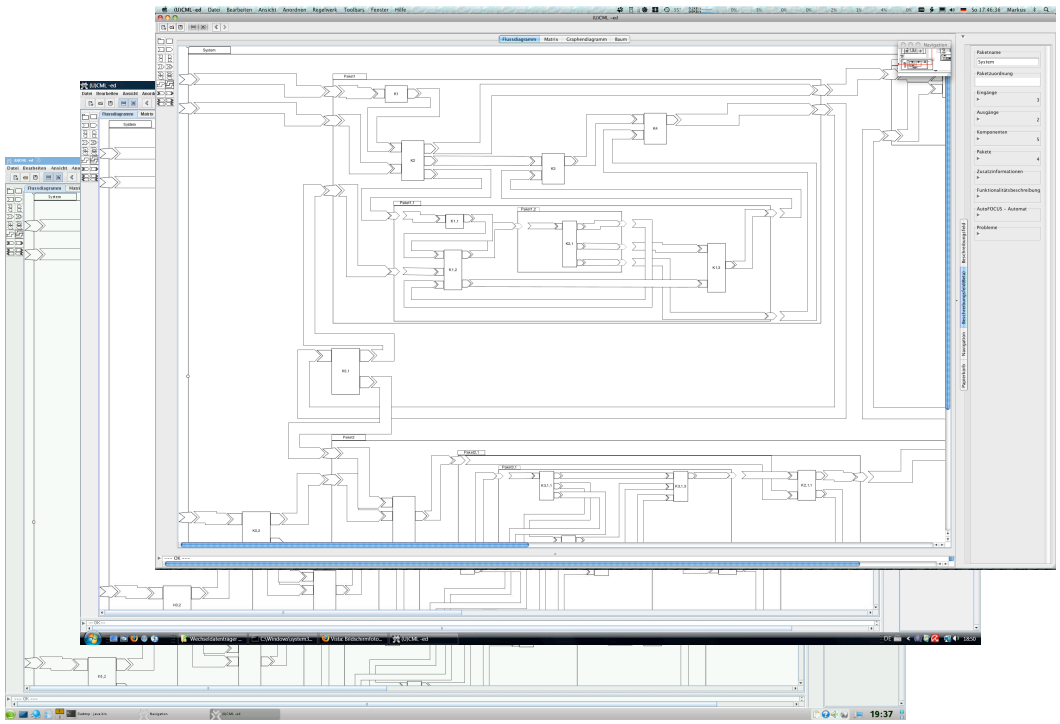


Abbildung 3.193.: Bildschirmphoto des Hauptfensters des plattformunabhängigen graphischen Kompatibilitätsmodellierungswerkzeug s*(U)CML-ed*.

Die Abbildung 3.193 zeigt, dass der Editor an die graphische Benutzeroberfläche des jeweils laufenden Betriebssystems anpasst ist. Alle (U)CML-spezifischen Elemente, wie beispielsweise die Pakete, Komponenten oder die Beschreibungsfelder, sind auf allen drei Plattformen identisch. Dadurch ist eine plattformübergreifende Entwicklung eines Systemmodells möglich, ohne dass die Ingenieur-Teams auf eine bestimmte Systemplattform wie beispielsweise MS Windows, angewiesen sind.

Im nächsten Abschnitt werden einige der grundlegenden Eigenschaften und Konzepte, die hinter dem graphischen Werkzeug *(U)CML-ed* stehen, vorgestellt.

Eigenschaften des graphischen Modellierungswerkzeugs *(U)CML-ed*
Zu den wichtigsten Eigenschaften des Werkzeugs *(U)CML-ed* gehört die plattformunabhängige Einsetzbarkeit. Die einzige Voraussetzung dafür ist, dass auf dem Zielsystem mindestens die JAVA Runtime Umgebung in der Version 1.5[267] installiert ist. Um die Plattformunabhängigkeit zu unterstreichen, wurde als Dateiformat die Extensible Markup Language (XML)[268] für sämtliche Dateien des Editors verwendet. Außerdem sind sämtliche textuellen Ausgaben des Editors sowohl an die Deutsche als auch die Englische Sprache angepasst[269].

In der folgenden Aufzählung werden einige der grundlegenden Eigenschaften und Konzepte des Editors in Stichpunkten vorgestellt. Nähere Informationen zum aktuellen Stand[270] der Entwicklung bzw. zu den unterschiedlichen Eigenschaften finden Sie auf der Projektwebseite unter [Tea08].

- Plattformunabhängigkeit (ab JAVA Runtime Umgebung Version 1.5 einsetzbar)
- Multilingualität (Deutsch, Englisch)
- Benutzeridentifikation gegenüber dem Werkzeug
- Protokollierung aller Benutzeraktionen am Modell
- XML-Datenschnittstelle
- Interaktiver automatischer Modelltest und Bewertung der Ergebnisse
- Kontextsensitiver Zooming-Mechanismus

Im gegenwärtigen Entwicklungstand des Editors sind jedoch bei weitem noch nicht alle Konzepte der (U)CML enthalten. So fehlt beispielsweise die Modellierung von zusammengesetzten Verbindungen oder die Modellierung von BUS-Systemen. Des Weiteren kann im Moment das Verhalten eines Systems noch nicht simuliert werden. Die noch fehlenden Konzepte der (U)CML werden im Lauf der Zeit in den Editor integriert, so dass er in einem weiteren Umfeld eingesetzt werden kann.

[267] Nähere Informationen zur Programmiersprache JAVA finden Sie unter [Mic08], [Wik08k] sowie [Ull07].
[268] Nähere Informationen zur XML finden Sie unter [Wik08i], [W3C08] sowie [O'R08].
[269] Anmerkung:
Weitere Sprachenanpassungen sind vorgesehen, jedoch noch nicht im Editor implementiert.
[270] Stand des Editors *(U)CML-ed*: 20.08.2008.

3.8.1. Überblick über das Werkzeug *(U)CML-ed*

Nach der Anmeldung am *(U)CML-ed* erscheint die *Menüzeile* sowie das Hauptfenster des Werkzeugs. Das Hauptfenster unterteilt sich in sechs große Kernbereiche. Im oberen Fenterbereich ist die *Hilfspalette* eingeblendet, über die die wichtigsten Aktionen, wie beispielsweise das Laden oder Speichern eines Modells ausgelöst werden kann. Der linke obere Fensterbereich enthält die *Werkzeugpalette*, in der sämtliche (U)CML-Sprachkonstrukte in graphischer Form (Minisymbole), enthalten sind. Am rechten Fensterrand ist das Tab *Beschreibungsfeld* aufgeklappt dargestellt, in dem beispielsweise die Eigenschaften eines Pakets visualisiert und beeinflusst werden können. Weitere Tabs, die anstatt eines Beschreibungsfelds eingeblendet werden können, sind: das *Navigationstab*, mit dessen Hilfe innerhalb eines Modells navigiert werden kann (Stichwort: kontextsensitiver Zoom) sowie das *Papierkorb-Tab*, das sämtliche gelöschten Elemente enthält. Am unteren Rand des Hauptfensters befindet sich das Fenster *Systemmeldungen*, in dem die wichtigsten Meldungen des Editors ausgegeben werden. In der Fenstermitte befindet sich die eigentliche *Zeichenfläche*. Diese zeigt, in Abhängigkeit des ausgewählten Tabs – Fluss-, Matrix-, Graphen- und Baumdiagramm – eine unterschiedliche Repräsentation des Modells an (im Bild wird das Flussdiagramm angezeigt). Die Abbildung 3.194 zeigt das soeben vorgestellte Hauptfenster des Editors.

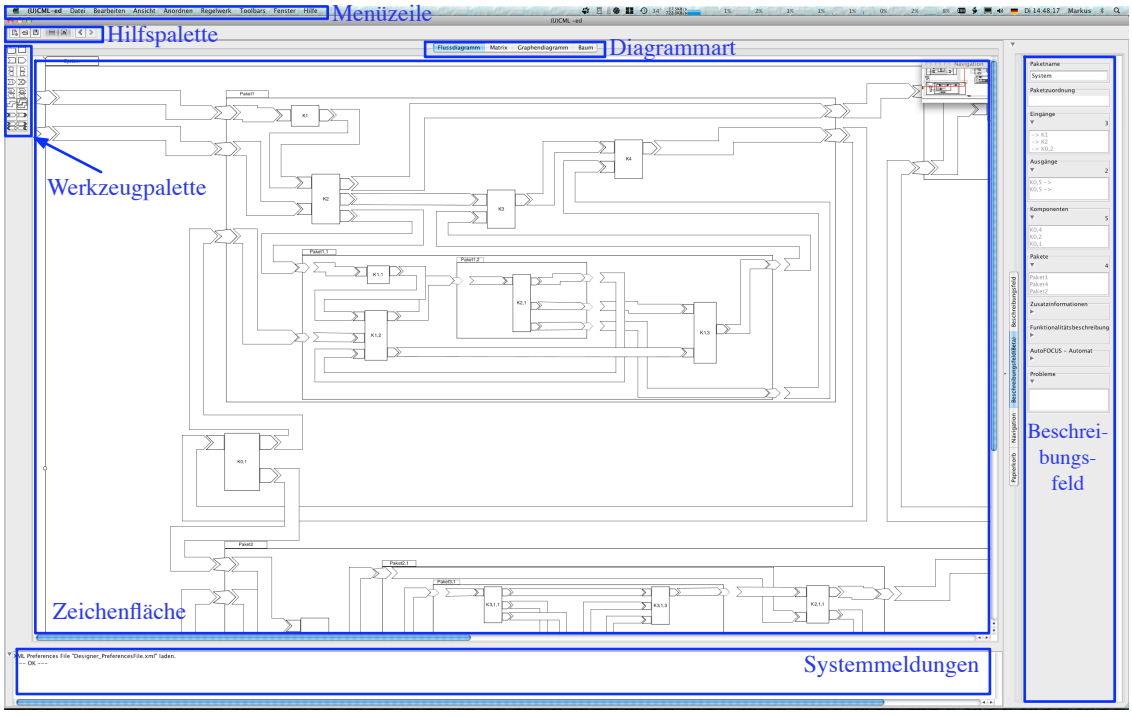


Abbildung 3.194.: Bildschirmphoto des Kompatibilitätsmodellierungswerkzeugs *(U)CML-ed*.

Um das Werkzeug an die individuellen Wünsche des Benutzers anpassen zu können, können die beiden Werkzeugpaletten aus dem Hauptfenster abgedockt und individuell als eigenständige so genannte Hover-Fenster[271] auf der Zeichenfläche platziert werden. Weitere benutzerspezifische Einstellungsmöglichkeiten sind im Editor (noch) nicht integriert, jedoch in den Programmmodulen vorgesehen, so dass sie zu einem späteren Zeitpunkt umgesetzt werden können.

Regelwerkeditor
Die Abbildung 3.195 auf der nächsten Seite zeigt den Regelwerkeditor des *(U)CML-ed*. Auf der linken Seite der Abbildung ist das Tab *Attribute* des Datentypeditors für die Bearbeitung von Softwaredatentypen dargestellt. Rechts daneben ist das Tab *Einheiten – Präfixe* des Regelwerkeditors zur Festlegung der kompatiblen Präfixe und Einheiten dargestellt.

Durch die beiden Ausprägungen des Regelwerkeditors kann das Verhalten des Regelwerks bei der Kompatibilitätsprüfung beeinflusst werden. So kann beispielsweise mit Hilfe des Datentypeditors festgelegt werden, dass der Datentyp `integer` die Byte-Reihenfolge „big-endian“, eine Datengröße von 128-Bit und ein Vorzeichen hat, sowie eine genau spezifizierte Ober- und Untergrenze. Rechts hingegen wurden der Einheit *Gramm* über den Einheiteneditors die Präfixe y, p, n, μ, m und k zugeordnet.

CCL Konsole
Das Akronym *CCL* steht für ***C**ompatibility **C**onstraint **L**anguage* – eine speziell für den *(U)CML-ed* entwickelten formalen funktionalen Sprache. Mit Hilfe der CCL können sowohl die Vor- als auch die Nachbedingungen und Invarianten formuliert sowie kleine Programme innerhalb des *(U)CML-ed* ausgeführt werden. So kann beispielsweise mittels CCL festgelegt werden, dass eine Komponente mindestens einen Ein- und einen Ausgang haben muss, oder dass der Stromeingang der Komponente *IC* an Pin 1 (Eingangsbuchse) mit der Komponente Stromversorgung Pin 1 (Ausgangsstecker) verbunden sein muss. Die Abbildung 3.196 auf der nächsten Seite zeigt die CCL Konsole des Editors.

[271] Als *Hover-* bzw. *floating-Fenster* werden Fenster bezeichnet, die über allen anderen Fenster schweben, und somit ständig sichtbar sind.

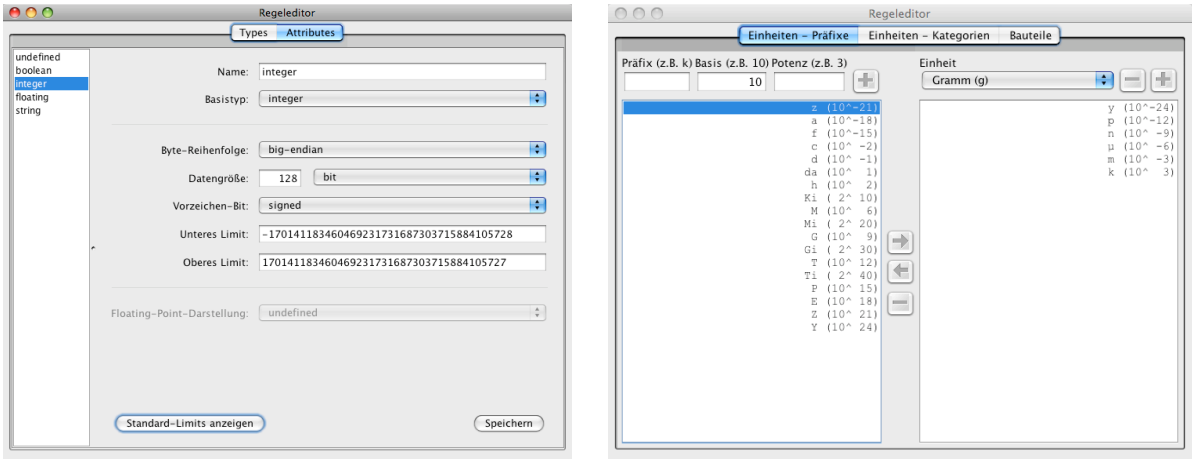


Abbildung 3.195.: Bildschirmphoto des Regelwerkeditors: links Datentypeditor, rechts Präfixdeklaration.

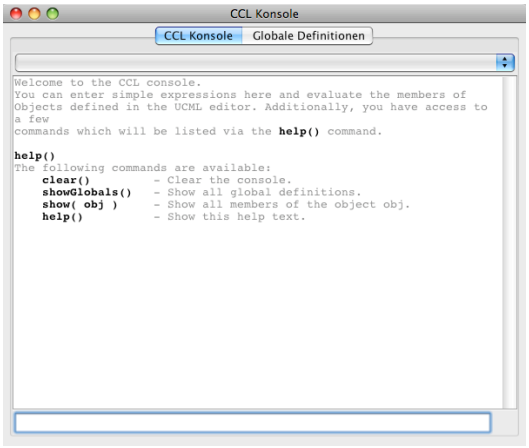


Abbildung 3.196.: Bildschirmphoto der CCL Konsole.

3.8.2. Kontextsensitives Zooming und Black-Box Ansicht des Werkzeugs *(U)CML-ed*

Bereits mehrfach wurde der Begriff des „kontextsensitiven Zooms“ im Zusammenhang mit der Beherrschung der Komplexität eines graphischen Systemmodells erwähnt. Wird beispielsweise ein Modell eines komplexen Systems mit Hilfe der (U)CML entwickelt, so kann es sehr schnell vorkommen, dass die Übersicht über das zu entwickelnde System „verloren geht“ da das Modell sehr groß und dabei unübersichtlich wird. Die Abbildung 3.193 auf Seite 256 zeigt beispielsweise einen „relativ kleinen“ Ausschnitt aus dem Modell eines komplexen Systems. Um die Übersicht über die im Modell vorhandenen Pakete, Komponenten sowie die Verbindungen zwischen den Steckern und Buchsen nicht zu verlieren, kann das Modell des Systems mit Hilfe des Werkzeugs *(U)CML-ed* so gezoomt werden, dass entweder nur kleine Bestandteile des Systemmodells oder das gesamte Modell gleichzeitig dargestellt wird. Durch den kontextsensitiven Zoom kann somit die Verständlichkeit des Systemmodells erheblich verbessert werden. Die Abbildung 3.197 auf der nächsten Seite zeigt zwei weitere Ansichten auf das Modell, das bereits in Ausschnitten in der Abbildung 3.193 dargestellt worden ist. Dabei wurde insbesondere die Komponente *K3.1* herausgezoomt.

Neben dem reinen Zoomen von Komponenten oder Paketen, kann durch den Mechanismus des kontextsensitiven Zooming, auch ein bestimmter Kontext eines Systemmodells hervorgehoben werden. So können beispielsweise mittels des kontextsensitiven Zoomings sämtliche Stecker, Buchsen und Komponenten, die einem bestimmten Typ, also beispielsweise Mechanik, angehören dargestellt werden, während andere Bestandteile ausgeblendet werden. Diese Art des Zoomings ist eng mit dem Sichtenkonzept verwandt, wo ebenfalls nur diejenigen Bestandteile eines Systems angezeigt werden, die eine bestimmte Eigenschaft aufweisen. Der kontextsensitive Zoomingansatz des *(U)CML-ed* geht jedoch bei weitem über das Sichtenkonzept hinaus, aufgrund der Tatsache, dass es im *(U)CML-ed* möglich ist, den Inhalt der Komponenten sichtbar zu machen, sobald sie vollständig gezoomt dargestellt sind. Diese besondere Art des Zoomings ist jedoch noch nicht in dem Editor integriert[272].

Eine weitere Möglichkeit, um das Komplexitätsproblem bei der graphischen Darstellung von Systemen zu beherrschen, liegt in der Verwendung des Black-Box Sichtenkonzepts des *(U)CML-ed*. Die Abbildung 3.198 auf der nächsten Seite zeigt das Paket *Paket1.2* in der Black-Box Darstellung. Durch das einfache Ausblenden von Implementierungsdetails wird die Komplexität des Systemmodells erheblich reduziert.

[272] Nähere Informationen zum kontextsensitiven Zooming finden Sie ebenfalls unter [Lab08].

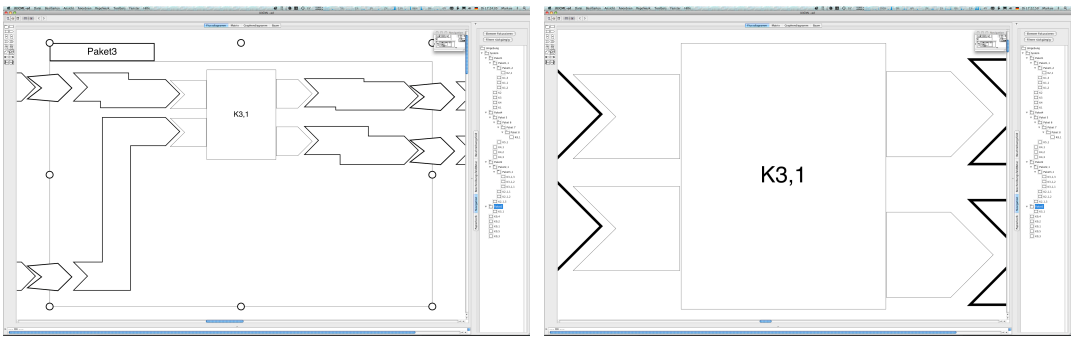


Abbildung 3.197.: Bildschirmphoto: Verschiedene Zoomstufen der Komponente *K3.1*.

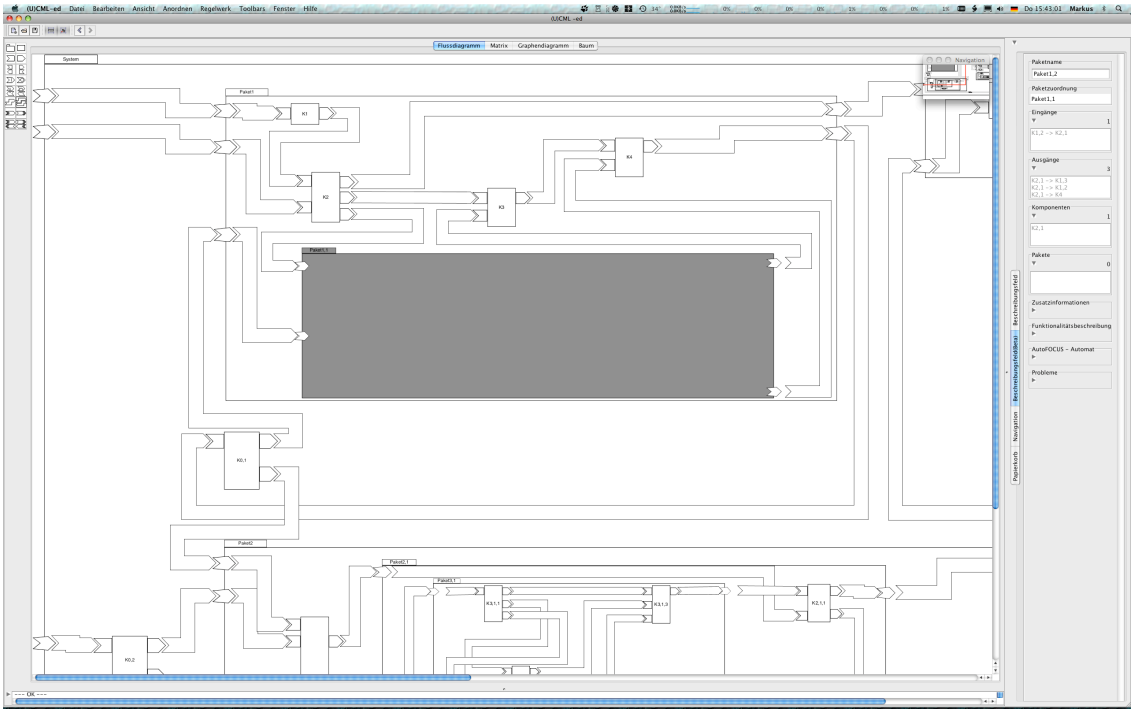


Abbildung 3.198.: Bildschirmphoto: Black-Box Sichtenkonzept zur Verringerung der Komplexität des Systemmodells.

Des Weiteren kann das Black-Box Sichtenkonzept ebenfalls zum „Verstecken“ von zu schützenden Modellierungsdetails verwendet werden. Wird beispielsweise das Paket *Paket1.2* von einem externen Dienstleister (beispielsweise einem Zulieferer) geliefert, so kann dieser sämtliche Implementierungsdetails der Baugruppe verstecken und über Zugriffsrechte so schützen, dass lediglich die öffentliche Schnittstelle der Baugruppe sichtbar ist. Nicht autorisierte Benutzer können das versteckte und geschützte Paket nicht einsehen oder gar manipulieren[273]. Dadurch ist es möglich, komplette Systemmodelle eines Systems anzufertigen, in denen sämtliche Details des Systems enthalten sind. Zusätzlich wird beim Kompatibilitätstest eine erheblich bessere Aussagekraft erreicht, wenn das gesamte Systemmodell auf Kompatibilität untersucht werden kann, als wenn wesentliche Bestandteile nicht vollständig modelliert vorliegen. Dies gilt insbesondere für die dynamische Simulation von Systemmodellen.

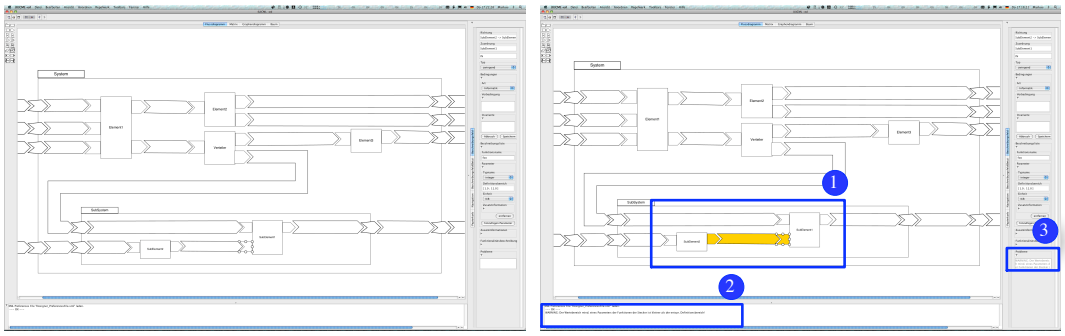
3.8.3. Kompatibilitätsbestimmung

Abgeschlossen wird die Vorstellung des Werkzeugs *(U)CML-ed* mit der konkreten Durchführung des Kompatibilitätstests eines einfachen Systemmodells. Zunächst wird mit Hilfe des Werkzeugs das statische Modell des Systems modelliert sowie die Beschreibungsfelder der Stecker und Buchsen der Komponenten befüllt. Im Anschluss folgt die Durchführung des eigentlichen Kompatibilitätstests[274]. Wird der Kompatibilitätstest mit dem Standardregelwerk durchgeführt, so muss lediglich der Untermenüpunkt *Modell testen* des Menüs *Regelwerk* ausgewählt werden. Sobald der Menüpunkt ausgewählt ist, wird das aktuelle

[273] Anmerkung:
Um geschützte Modellteile (i.A. Baugruppen) vor nicht berechtigter Manipulation durch Dritte zu schützen, werden diese durch ein public-key-Verschlüsselungsalgorithmus bereits auf XML Dateiebene verschlüsselt. Somit ist gewährleistet, dass geschützte Modellteile auch auf Dateiebe und nicht nur im Editor geschützt sind.

[274] Dies entspricht dem Vorgehen beim Szenario 1: der Modellierung eines Systems anhand der Spezifikationen der Baugruppen.

Systemmodell anhand des Regelwerks auf Kompatibilität untersucht und das Ergebnis sowohl graphisch als auch textuell dargestellt. Die Abbildung 3.199 zeigt links das Systemmodell vor der Durchführung des Kompatibilitätstests. Rechts daneben ist das Modell dargestellt, nachdem das Systemmodell mit dem aktuell gültigen Kompatibilitätsregelwerk auf Kompatibilität untersucht und bewertet worden ist.



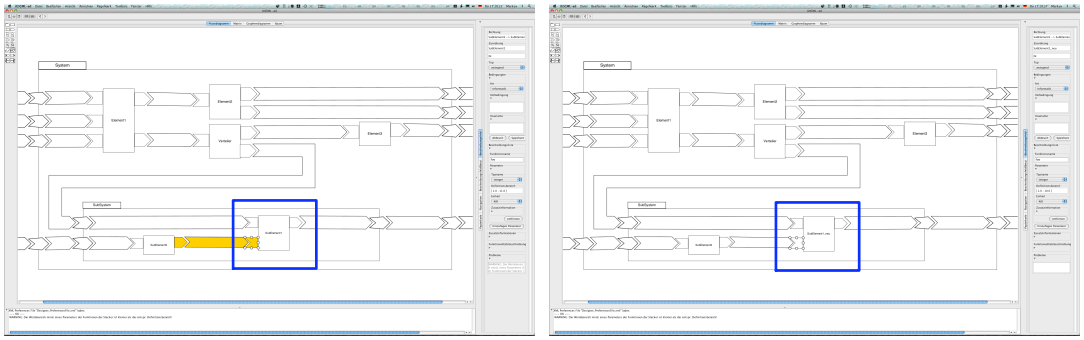
Modell **vor** dem Kompatibilitätstest

Modell **nach** dem Kompatibilitätstest

Abbildung 3.199.: Bildschirmphoto: Durchführung des Kompatibilitätstests mit dem Werkzeug *(U)CML-ed*.

Die zweite Eingangsbuchse der Komponente *SubElement01* verursacht eine Kompatibilitätswarnung (1). Dies wird dadurch symbolisiert, dass die gesamte Verbindung zwischen den beiden beteiligten Komponenten, genauer: zwischen dem Ausgangsstecker der Komponente *SubElement02* und der Eingangsbuchse der Komponente *SubElement01*, gelb eingefärbt wird. Zusätzlich wird im Systemmeldungsbereich (2) die Warnung – *WARNING: Der Wertebereich mind. eines Parameters der Funktionen der Stecker ist kleiner als der entspr. Definitionsbereich!* – ausgegeben und ebenfalls im Feld „Problem“ des Beschreibungsfelds der Buchse der Komponente *SubElement01* angezeigt (3).

Um das zweite Szenario, also den Austausch einer Komponente gegen eine neue zu simulieren, wird nun im Modell die Komponente *SubElement01*, die die Warnung verursacht, gegen die neue Komponente *SubElement01_neu* ersetzt und anschließend das Systemmodell erneut auf Kompatibilität untersucht. Die Abbildung 3.200 zeigt das Systemmodell vor- und nachdem die Komponente *SubElement01* gegen die Komponente *SubElement01_neu* ersetzt worden ist.



Modell **vor** dem Austausch der Komponente SubElement01 mit Kompatibilitätswarnung

Fehlerfreies Modell **nach** dem Austausch der Komponente SubElement01 gegen die Komponente SubElement01_neu

Abbildung 3.200.: Bildschirmphoto: Austausch der Komponente *SubElement01* gegen die Komponente *SubElement01_neu*.

Das neue Systemmodell (rechts) enthält keine Kompatibilitätsfehler oder Warnungen mehr; die Komponente *SubElement01_neu* ist demnach vollständig kompatibel zum ursprünglichen Systemmodell.

Teil II.

Fallstudie – Gleitschutzsystem

In diesem Abschnitt wird zunächst die Fallstudie Gleitschutzsystem des MOKOMA-Projekts in der Modellierungssprache (U)CML modelliert, das Modell anschließend auf Kompatibilität untersucht sowie das Ergebnis bewertet.

Kapitel 4.

Modellierung der Fallstudie – Gleitschutzsystem – in (U)CML

> Es liegt in der Natur des Menschen, zu gehen, zu sehen, zu verstehen. Zu forschen ist keine Möglichkeit, sondern ein Muss.
>
> *(Michael Collins)*

Nachdem nun die Beschreibung der Grundlagen der Modellierung und vor allem die Einführung der Kompatibilitätsmodellierungssprache (U)CML abgeschlossen worden ist, folgt in diesem Kapitel der „praktische Einsatz" der Modellierungssprache (U)CML anhand der Fallstudie Gleitschutzsystem.

Die Fallstudie wurde während des vom BMBF geförderten Forschungsvorhabens MOKOMA vom Industriepartner Knorr-Bremse AG zur Verfügung gestellt und dient hier, in stark vereinfachter Form, als Grundlage für die Modellierung der Fallstudie.

Problembeschreibung
Schienenfahrzeuge haben eine sehr lange Lebensdauer von zum Teil mehr als 30 Jahren. Innerhalb dieses Zeitraums ist es notwendig, stets Ersatzteile für alle ausgelieferten Varianten des Systems bereitzustellen. Dies ist generell jedoch nicht möglich, da viele unterschiedliche Varianten einer Baugruppe gelagert werden müssten. Im Schienenfahrzeugbau gibt es im Allgemeinen keine Massenfertigung sondern für jeden Kunden wird eine speziell an seine Bedürfnisse angepasste Version des Systems entwickelt bzw. aus bereits existierenden Versionen abgeleitet. Dadurch vervielfältigt sich die Anzahl der möglichen Varianten des Systems enorm, wodurch auch der Lagerungsaufwand und die damit verbundenen Kosten stark ansteigen würden.

Um diesem Dilemma zu entkommen, bieten sich zwei unterschiedliche Vorgehensweisen an. Zum einen die Entwicklung von langlebigen Standardkomponenten und zum anderen die Modularisierung des Systems bzw. der verwendeten Baugruppen. Auch Kombinationen aus beiden Ansätzen sind möglich und werden in der Industrie eingesetzt. Durch diese beiden Maßnahmen lässt sich ein Teil des Problems elegant umgehen. Es bleibt jedoch noch ein kleiner Teil, der so nicht ausreichend gelöst werden kann. Zum Beispiel wie reagiert man auf die Abkündigung einer Baugruppe oder eines bestimmten Bauteils?

- **Lösung A:** *Lagerung der benötigten Baugruppen und Bauteile über den gesamten Produktlebenszyklus hinweg*
  Diese Maßnahme hätte wiederum zur Folge, dass viele unterschiedliche Versionen von Baugruppen und Bauteilen über einen sehr langen Zeitraum eingelagert werden müssten.
- **Lösung B:** *Neuentwicklung kompatibler Baugruppen und Bauteile*
  Die Neuentwicklung kompatibler Baugruppen oder Bauteile ist mit sehr hohen Kosten verbunden. Außerdem setzt sie die Kenntnis der inneren Struktur der Baugruppe bzw. des Bauteils voraus. Somit ist eine Neuentwicklung kompatibler Baugruppen nur für Eigenentwicklungen möglich. Hinzugekaufte Bauteile und Baugruppen müssten dann aufwendig „reengineered" und an den aktuellen Bauzustand angepasst werden.
- **Lösung C:** *Kompatible Weiterentwicklung und Anpassungen existierender Baugruppen und Bauteile an neue und alte Anforderungen*
  Dies ist eine Mischform aus den beiden benannten Lösungsansätzen. Zum einen werden Baugruppen und Bauteile eingelagert, die sehr alt sind und bei denen sich eine kompatible Weiterentwicklung nicht rechnen würde. Zum anderen wird eine Produktlinie stets so weiterentwickelt, dass sie kompatibel zur vorherigen Produktversion ist. Dadurch kann eine neue Komponente stets in einem alten System verwendet werden, ohne dass dafür spezielle Anpassungen notwendig wären.

Um den Lösungsansatz C zu unterstützen, bietet sich die (U)CML zur Modellierung des gesamten Systems an. In der Fallstudie – Gleitschutzsystem – der Firma Knorr-Bremse AG wird zunächst das gesamte real existierende Gleitschutzsystem in (U)CML nachmodelliert und auf „versteckte" Inkompatibilitäten untersucht. Nachdem das vollständige System modelliert und getestet worden ist, wird eine alte abgekündigte Komponente des Systems durch eine andere ersetzt, was dem Lösungsansatz C entspricht, bzw. durch eine andere Bauform der Komponente ersetzt.

Jedoch bevor mit der Modellierung der Fallstudie begonnen wird, folgt zuerst eine kurze Erläuterung der Fallstudie Gleitschutzsystem.

Erläuterung der Fallstudie – Gleitschutzsystem
Alle modernen Schienenfahrzeuge – angefangen bei einer Straßenbahn bis hin zu einem ICE – besitzen ein so genanntes *Gleitschutzsystem*, das verhindern soll, dass bei einer starken Verzögerung (Bremsvorgang) ein Rad bzw. ein Räderpaar eines Drehgestells blockiert[275]. Das Blockieren eines Radsatzes hat zur Folge, dass sich das Rad an dieser Stelle besonders stark abnutzt und es somit zur Bildung einer „Flachstelle" kommt. Diese muss anschließend bei der nächsten Inspektion ausgebessert werden, da es ansonsten zu einer starken Beeinflussung der Rolleigenschaften des Radsatzes kommt (z.B. „Holpergeräusche"). Da das Überholen eines Radsatzes jedoch sehr zeit- und vor allem kostenintensiv ist, muss das Blockieren vermieden werden. In der Abbildung 4.1 ist ein Laufrad eines Drehgestells mit Gleitschutzsensor abgebildet.

[275] Beim Blockieren eines Radsatzes reißt der Kraftschluss zwischen Rad und Schiene plötzlich ab. Der Radsatz hat keine Haftung mehr und es kommt zum Blockieren des Radsatzes. Dies ist vergleichbar mit dem Blockieren eines Rades beim PKW (vgl. ABS).

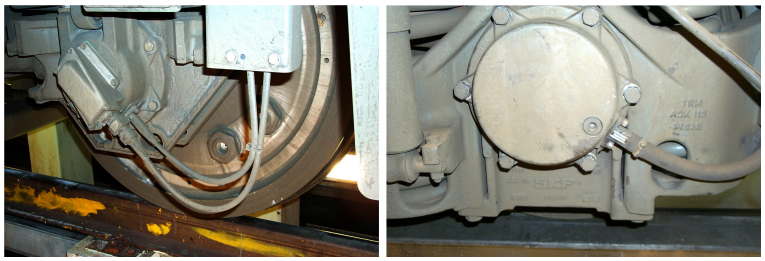


Abbildung 4.1.: Abbildung des Knorr-Bremse Gleitschutzsystems – Gleitschutzsensor [DC04].

In der Schienenfahrzeugindustrie gibt es zwei Ansätze ein Gleitschutzsystem zu realisieren. Zum einen ein rein mechanisches System ohne den Einsatz von spezieller Steuerelektronik bzw. Software und zum anderen ein System bestehend aus mechanischen, elektrischen/elektronischen und Softwarekomponenten – ein eingebettetes System. Die einfachen mechanischen Gleitschutzsysteme werden vor allem im Güterverkehr eingesetzt, aufgrund der Tatsache, dass im Güterverkehr im Allgemeinen geringere Geschwindigkeiten als im Personenverkehr gefahren werden und das „Holpergeräusch“ eines Drehgestells nicht so störend ist. Die elektronischen bzw. softwaregesteuerten Geitschutzsysteme werden vor allem im Personenverkehr eingesetzt. Das untersuchte Gleitschutzsystem der Firma Knorr-Bremse AG gehört der zweiten Art an, ist also softwaregesteuert. Im nächsten Abschnitt dieses Kapitels wird der grundsätzliche Aufbau und die Struktur eines softwaregesteuerten elektromechanischen Gleitschutzsystems erläutert.

Abbildung 4.2.: Abbildung des Knorr-Bremse Gleitschutzsystems – Steuereinheit [DC04].

Nähere Informationen zum Thema Gleitschutz bzw. Drehgestell von Schienenfahrzeugen finden sie unter [SV06][Spi][AG] bzw. [Wik07a].

Aufbau und Struktur des Gleitschutzsystems
Das softwaregesteuerte elektromechanische Gleitschutzsystem der Firma Knorr-Bremse AG besteht aus zwei Teilen. Zum einen aus der *Steuereinheit* (Abb.: 4.2), mit deren Hilfe die Ansteuerung des Bremssystems und die Kommunikation mit den unterschiedlichen Fahrzeugbussystemen erfolgt und zum anderen aus den *Gleitschutzsensoren* (Abb.: 4.1) sowie den mechanischen Ventilen zur Ansteuerung der Bremsanlage. In der Abbildung 4.3 ist der schematische Aufbau eines Gleitschutzsystems der Firma Knorr-Bremse AG dargestellt.

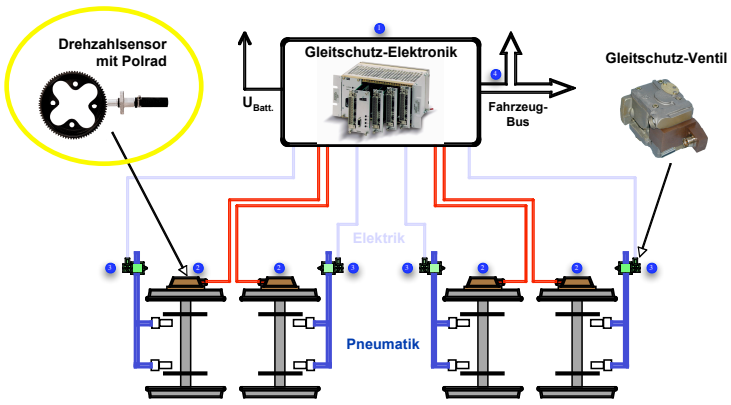


Abbildung 4.3.: Schematischer Aufbau des Knorr-Bremse Gleitschutzsystems [DC04].

In der oberen Bildmitte (1) wird die Steuereinheit des Gleitschutzsystems, die Gleitschutz-Elektronik, dargestellt. Die Gleitschutz-Elektronik ist in einem 19-Zoll Rack im Fahrzeug eingebaut (hier nicht dargestellt). Über die Gleitschutz-Elektronik wird die Verbindung zum Fahrzeug bzw. den unterschiedlichen BUS-Systemen des Fahrzeugherstellers hergestellt (4). An der Gleitschutz-Elektronik sind sowohl die Gleitschutzsensoren (2) als auch die Gleitschutz-Ventile (3) angeschlossen. Das gesamte Bremssystem,

also die Steuerung und die Sensorik/Aktuatorik wird als ESRA-System[276] bezeichnet, wobei die Steuerung des Systems, die Gleitschutz-Elektronik, das Herzstück der Anlage bildet. In der Abbildung 4.4 ist die Gleitschutz-Elektronik mit ihren unterschiedlichen Aufgabenbereichen beispielhaft dargestellt.

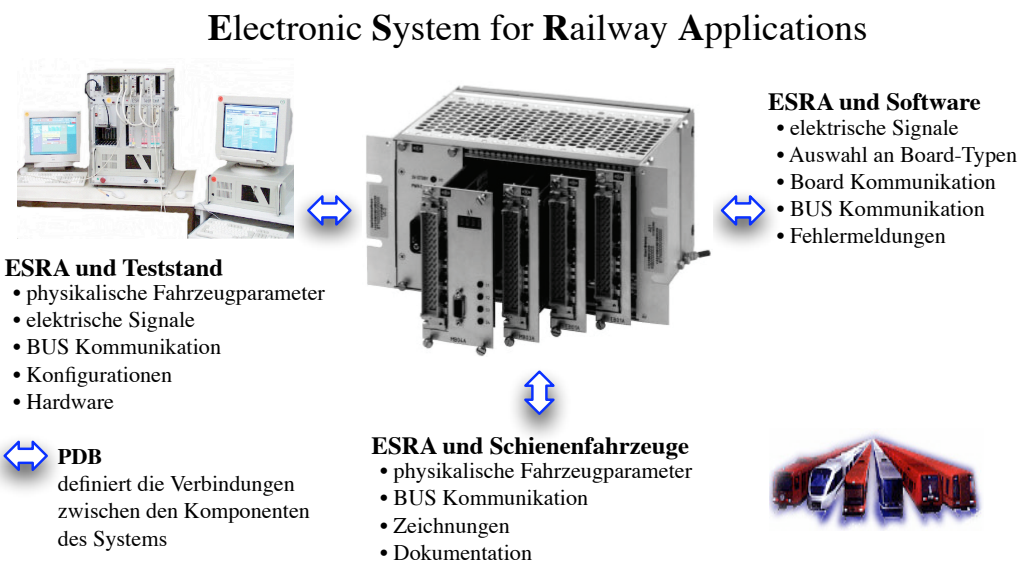


Abbildung 4.4.: Das Knorr-Bremse ESRA-System [DC04].

Das ESRA-System der Firma Knorr-Bremse AG wird in den nachfolgenden Kapiteln in stark vereinfachter Form in (U)CML modelliert und auf Kompatibilität untersucht. Dabei wird – in Anlehnung an die beiden Lösungsansätze B und C (Seite: 263) – zwischen zwei unterschiedlichen Szenarien unterschieden:

- **Szenario 1:** Modellierung eines neuen Systems bzw. einer neuen Baugruppe in (U)CML
  Ein neues Gleitschutzsystem bzw. eine Baugruppe des Systems wird komplett neu entwickelt. Dieses Vorgehen wird im ersten Abschnitt des folgenden Kapitels beispielhaft untersucht. Zu diesem Zweck wird ein stark vereinfachtes Modell des ESRA-Systems in (U)CML modelliert und auf Kompatibilität untersucht. Dieses Modell dient im zweiten Szenario als *Referenzmodell*.

- **Szenario 2:** Austausch einer Komponente aus einem in (U)CML modellierten System
  Nachdem im ersten Szenario das Modell des Gleitschutzsystems bzw. einer Baugruppe in (U)CML modelliert worden ist, wird nun eine Komponente des Gleitschutzsystem gegen eine andere ausgetauscht. Daran anschließend wird das neue Modell wieder auf Kompatibilität untersucht (vgl. Definition 1.14 Austauschkompatibilität auf Seite 20).

  Der Austausch einer defekten Komponente eines alten Gleitschutzsystem gegen eine neue und der anschließende Kompatibilitätstest ist das wichtigste Szenario, um das Lagerhaltungsproblem für verschiedene Versionen bzw. Ausbaustufen der unterschiedlichen Baugruppen zu umgehen (vgl. Lösung C auf Seite: 263).

Anmerkung:
In der „realen Welt“ wird das geschilderte Szenario 1 nur selten vollständig zum Einsatz kommen, weil in den meisten Fällen bereits ein reales System existiert, das weiterentwickelt werden soll. Um die Weiterentwicklung von der Kompatibilitätsseite her zu unterstützen, ist es jedoch zwingend notwendig, dass das „alte“ System vollständig in (U)CML modelliert vorliegt, da sonst keine solide Aussage über die Kompatibilität des neuen Systems bzw. die Änderungen am alten System gemacht werden können.

4.1. Anwendung des (U)CML-Modellbildungsprozesses auf die Fallstudie

In diesem Kapitel wird zunächst das stark vereinfachte, real existierende, Gleitschutzsystem der Firma Knorr-Bremse AG schrittweise in ein (U)CML-Modell überführt und anschließend auf „versteckte Inkompatibilitäten“ untersucht (Szenario 1). Nachdem das Modell des Gleitschutzsystems vollständig aufgebaut und verifiziert wurde (Referenzsystem), wird eine Komponente des Modells durch eine andere ersetzt (vgl. Definition 1.14 Austauschkompatibilität). Auch das neue Modell wird nun auf Kompatibilität innerhalb des Modells und auf Kompatibilität zum ursprünglichen Modell untersucht (Szenario 2)[277].

Die nachfolgenden Abschnitte sind den unterschiedlichen Prozessschritten zugeordnet, mit deren Hilfe ein neues System bzw. eine Komponente eines real existierenden Systems in (U)CML modelliert werden kann. Zu jedem hier aufgeführten Punkt folgt eine ausführliche Erläuterung in den nachfolgenden Unterabschnitten dieses Kapitels.

[276] Das Akronym ESRA steht für **E**lectronic **S**ystem for **R**ailway **A**pplications.
[277] Der hier zugrunde liegende (U)CML-Modellbildungsprozess wurde bereits im Kapitel „3.6.6 *Der (U)CML-Modellbildungsprozess*“ ab Seite 243 ausführlich vorgestellt.

- **Modellbildung**
  Die Modellbildung bzw. der (U)CML-Modellbildungsprozess unterteilt sich in drei aufeinander aufbauende Bereiche:

  1. **Aufbereitung der vorhandenen Daten aus unterschiedlichen Quellen**
     Um die semantische Lücke, die zwischen den verschiedenen Datenquellen und dem Modell existiert, zu schließen, müssen die unterschiedlichen Informationsquellen systematisch für die nachfolgenden Schritte der Modellbildung aufbereitet werden. Für die Fallstudie Gleitschutz wird dieser Prozess anhand des *Mainboards* sowie des *Sensors GI5* exemplarisch gezeigt. Dieser Prozess muss später für alle anderen Komponenten des Systems wiederholt werden.

     Das **Ergebnis** der Aufbereitung der unterschiedlichen Ausgangsdaten ist die Identifikation der Pakete und Komponenten sowie ihrer (kompatibilitätsrelevanten) Eigenschaften.

  2. **Modellierung des Gleitschutzsystems in (U)CML**
     Nachdem alle Pakete, Komponenten sowie ihre Eigenschaften im Schritt 1 identifiziert worden sind, folgt nun die eigentliche Modellierung des Gleitschutzsystems in (U)CML. Die Modellierung unterteilt sich dabei in zwei Teilbereiche:

     – Identifikation der Struktur des Gleitschutzsystems sowie die Transformation der Strukturelemente in ein (U)CML-Modell.

     – Identifikation und Modellierung der Anschlüsse (Stecker/Buchsen/Schnittstellen) und Verbindungen (Pfeile) der Komponenten des Systems.

     Das **Ergebnis** der Modellierung ist das vollständige qualitative (U)CML-Modell des Gleitschutzsystems mit allen Paketen, Komponenten sowie den Verbindungen zwischen den Komponenten.

  3. **Befüllung der Beschreibungsfelder mit den Eigenschaften des Systems**
     Überführung des qualitativen (U)CML-Gleitschutzsystemmodells in ein quantitatives Modell durch die Befüllung der Beschreibungsfelder der Komponenten, Stecker und Buchsen mit den im Schritt 1 identifizierten Eigenschaften des Gleitschutzsystems. Besonderes Augenmerk wird hier auf die kompatibilitätsrelevanten Eigenschaften des Systems gelegt, weil diese die Grundlage für den anschließenden Kompatibilitätstest bilden.

     Ergebnis: Vollständiges quantitatives (U)CML-Modell befüllt mit allen Eigenschaften, die für die Kompatibilitätsprüfung notwendig sind.

- **Regelwerk erstellen**
  Für die Bestimmung der Kompatibilität des Systems muss jetzt das projektabhängige Regelwerk der (U)CML an die speziellen Bedürfnisse des Gleitschutzsystems angepasst werden. Dabei kann das Regelwerk in den drei Domänen – Elektrik/Elektronik, Mechanik und Software – angepasst werden.

  Ergebnis: Ein speziell an die Bedürfnisse des Projekts angepasstes Regelwerk für die anschließende Bestimmung der Kompatibilität des Gleitschutzsystems.

- **Durchführung des Kompatibilitätstests**
  Nachdem das (U)CML-Modell des Gleitschutzsystems sowie das angepasste Kompatibilitätsregelwerk vorliegt, kann mit der Durchführung des Kompatibilitätstests auf der Grundlage des Regelwerks begonnen werden.

  Ergebnis: Graphische Darstellung sowie Liste aller Fehler und Warnungen des auf Kompatibilität hin untersuchten Gleitschutzsystems.

- **Bewertung der Ergebnisse**
  Bewertung der Ergebnisse des Kompatibilitätstests und Rückführung der Ergebnisse in das Modell.

  Ergebnis: Protokoll aller notwendigen Änderungen am Modell.

Anmerkung:
Es ist bereits während der Modellierungsphase des Systems möglich, dieses mit dem projektunabhängigen Regelwerk auf Kompatibilität zu untersuchen. Dies ist besonders wichtig, weil so bereits während der Modellierung Fehler im Design des Systems gefunden werden können (Siehe hierzu: Kapitel „3.8 *(U)CML-ed, ein graphischer Editor für die Kompatibilitätsmodellierungssprache (U)CML*“ ab Seite 254.).

4.1.1. Szenario 1: Modellierung eines neuen Systems bzw. einer Baugruppe in (U)CML

Für die Kompatibilitätsbestimmung mit Hilfe der (U)CML ist ein vollständiges quantitatives Modell des Systems zwingend notwendig. Da jedoch nicht von jedem System ein vollständiges (U)CML-Modell vorliegt, muss dieses in den meisten Fällen erst aus den vorhanden Dokumenten generiert oder aus einem realen existierenden System „reengineered“ werden. In den folgenden Unterabschnitten dieses Kapitels wird die Neuentwicklung einzelner Bestandteile des Gleitschutzsystems, anhand der vorhandenen Dokumentation der Fallstudie exemplarisch modelliert und dargestellt. Dazu wird im ersten Schritt mit der strukturierten Datensammlung und der Modellierung des Systems in (U)CML begonnen. Anschließend folgt die Definition der Kompatibilitätsregeln sowie die Durchführung des eigentlichen Kompatibilitätstests. Abgeschlossen wird der Modellbildungs- und Kompatibilitätsbestimmungsprozess mit der Bewertung und Rückführung der Ergebnisse des Kompatibilitätstests in das Modell.

Das entwickelte und auf Kompatibilität hin untersuchte (U)CML-Modell des Gleitschutzsystems dient, im nächsten Kapitel „4.1.2 *Szenario 2: Austausch einer Komponente aus einem in (U)CML modellierten System*“ ab Seite 299 als Referenzmodell. Dort wird eine Komponente des Systems gegen eine andere ausgetauscht und das neue System wieder auf Kompatibilität untersucht.

4.1.1.1. Modellbildung

Die Modellbildung unterteilt sich in drei eng gekoppelte Teilbereiche. Zuerst müssen alle vorhanden Daten und Informationen über das zu modellierende System gesichtet und ausgewertet werden. Besonderes Augenmerk wird dabei auf die Identifikation der einzelnen Bestandteile des Systems und ihrer Eigenschaften gelegt. Daran anschließend folgt die eigentliche Modellierung des Systems in (U)CML, also die Erzeugung des qualitativen Modells. Nachdem die Modellierungsphase abgeschlossen wurde, folgt im letzten Schritt der Modellbildung die Befüllung der (U)CML-Beschreibungsfelder mit den zuvor, im Schritt 1 identifizierten, kompatibilitätsrelevanten Eigenschaften des Systems.

In den nächsten drei Unterpunkten wird jeder Schritt des Modellbildungsprozesses anhand zweier Bestandteile des Gleitschutzsystems einzeln ausgeführt.

4.1.1.1.1. Aufbereitung der vorhandenen Daten aus unterschiedlichen Quellen

Der wichtigste Schritt bei der Modellbildung ist die Sichtung und Identifikation sowie die strukturierte Aufbereitung der vorhandenen Daten und Informationen. Die strukturierte Aufbereitung der vorhandenen Daten und Informationen dient der Überwindung der *semantischen Lücke*, die zwischen den unterschiedlichen Ausgangsdaten und dem Modell liegt. Um diese Lücke zu überwinden, ist es notwendig, die unterschiedlichen Daten und Informationen aus den vorhandenen Dokumenten (Textdateien, Schaltplänen, Quellcodes etc.) so aufzubereiten, dass sie anschließend leicht in das (U)CML-Modell übernommen werden können. Bei der Identifikation der Daten ist vor allem auf die Relevanz für die Kompatibilitätsbestimmung zu achten. Unwichtige Daten die für die Kompatibilitätsbestimmung bzw. die Modellierung nicht benötigt werden, sollten nicht in das Modell aufgenommen werden, da sie das Modell „überfrachten“ würden, worunter vor allem die Übersichtlichkeit leiden und die Komplexität des Systemmodells unnötig erhöht werden würde[278]. In der Abbildung 4.5 ist die semantische Lücke die bei der Modellierung überwunden werden muss exemplarisch für einige Datenformate dargestellt.

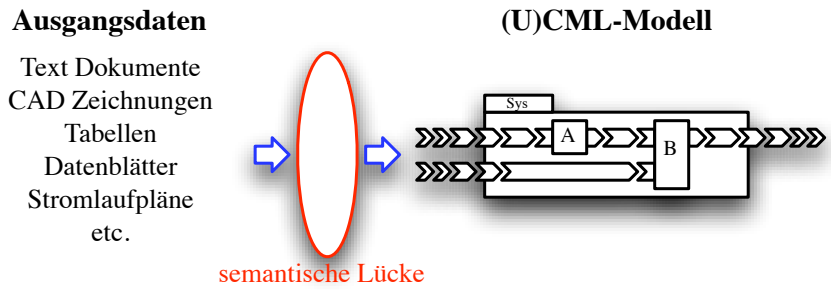


Abbildung 4.5.: Semantische Lücke zwischen den Ausgangsdaten und dem (U)CML-Modell.

Die semantische Lücke besteht vor allem in der Vielfalt und der teilweisen Inkonsistenz der unterschiedlichen Daten und Informationen, die für die Erstellung des Modells notwendig sind. Die Überwindung der semantischen Lücke stellt einen Filterprozess dar, der sicherstellen muss, das alle relevanten Daten aus den unterschiedlichen Ausgangsdaten in das (U)CML-Modell einfließen, jedoch irrelevante Daten nicht. Für die Modellierung in (U)CML sind vor allem die kompatibilitätsrelevanten Eigenschaften des (eingebetteten) Systems[279] interessant. Für das Modell sind jedoch nicht nur die kompatibilitätsrelevanten Eigenschaften des Systems von Interesse, sondern auch der Aufbau und die Struktur des Systems. Dazu zählen vor allem die Verbindungen zwischen den einzelnen Baugruppen des Systems. Die Verbindungen zwischen den einzelnen Systembestandteilen sind für die Kompatibilitätsbewertung essentiell; ohne die explizit modellierten Verbindungen können die Eigenschaften und somit die Kompatibilität des Modells nicht überprüft werden.

Die Aufbereitung der unterschiedlichen Daten und Informationen der Fallstudie Gleitschutzsystem wird in die drei Domänen der (U)CML – Elektrotechnik, Mechanik und Software – unterteilt dargestellt. In der nachfolgenden Aufzählung wird die Aufbereitung der Daten anhand der beiden Komponenten *Mainboard* sowie *Sensor GI5* exemplarisch dargestellt. Für alle anderen Komponenten des Gleitschutzsystems kann auf die gleiche Art und Weise vorgegangen werden.

Anmerkung:
In der Praxis sollte die Datensammlung und deren anschließende Aufbereitung durch die jeweiligen Expertengruppen bzw. die Domänenverantwortlichen erfolgen, weil nur sie eine ausreichend genaue Einschätzung der kompatibilitätsrelevanten Eigenschaften des Systems machen können. Werden bei der Identifikation der Eigenschaften des Systems essentielle Eigenschaften „übersehen“, so kann es zu Inkonsistenzen im Modell kommen, die zum Teil nicht gefunden werden können. Außerdem gilt der allgemeine Grundsatz bei der Modellierung: *„Das Modell kann nur Fragen beantworten, die modelliert sind.“* (Prof. Dr.-Ing. E. Igenbergs).

Identifikation der Eigenschaften sowie der Struktur des Gleitschutzsystems

Die Fallstudie Gleitschutzsystem ist, wie bereits in der Einleitung erwähnt, ein eingebettetes softwarelastiges System, das aus unterschiedlichen elektrischen und mechanischen Baugruppen sowie zahlreichen Softwaremodulen besteht. Um aus den unterschiedlichen Ausgangsdokumenten ein Modell des Systems zu generieren, müssen zunächst die Strukturelemente, also die Baugruppen und Softwaremodule identifiziert werden. Die Hardwarestruktur, also die Elemente des Gleitschutzsystems, können z.B. aus der Abbildung 4.3 auf Seite 264 entnommen werden.

[278] Siehe hierzu insbesondere das Kapitel „2.3.2.1 *Identifikation der kompatibilitätsrelevanten Systemeigenschaften*“ ab Seite 59.
[279] Siehe hierzu: Kapitel „2.7 *Objektorientierte Modellierung kompatibilitätsrelevanter Eigenschaften von eingebetteten Systemen*“ ab Seite 104.

- **Identifikation der HW/SW-Elemente des Gleitschutzsystems**
  Der erste Schritt besteht in der Identifikation der Hardwareelemente sowie der Softwaremodule des ESRA-Gleitschutzsystems. Aus den unterschiedlichen Dokumentationen wie z.B. Baugruppenbeschreibungen, CAD Zeichnungen, Stomlaufplänen, Softwaremodelbeschreibungen, C Headerfiles etc., können folgende Hardware- und Softwareelemente identifiziert werden:

  – *Hardware*
    Die Hardwarebestandteile des Systems Gleitschutz lassen sich in zwei Bereiche unterteilen: zum einen in die Elemente des Steuersystems, die so genannte ESRA-*Gleitschutz-Elektronik* und zum anderen in den *Einbauort* des Sensors sowie den *Sensor* selbst (vgl. Abbildung 4.3 auf Seite 264).

    * 19-Zoll Einbau-Rack
      Im Rack sind alle Baugruppen (19-Zoll Einschubkarten) des Gleitschutzsystems eingebaut und über ein rückwärtiges BUS-Systems miteinander verbunden. Das einfachste Gleitschutzsystem besteht aus einem Netzteil und einem Mainboard an dem die Gleitschutzsensoren angeschlossen werden. Komplexere Gleitschutzsysteme haben zusätzlich ein *Kommunikationsboard* und ein *Erweiterungsboard*, durch die die Funktionalität des Systems erweitert werden kann.

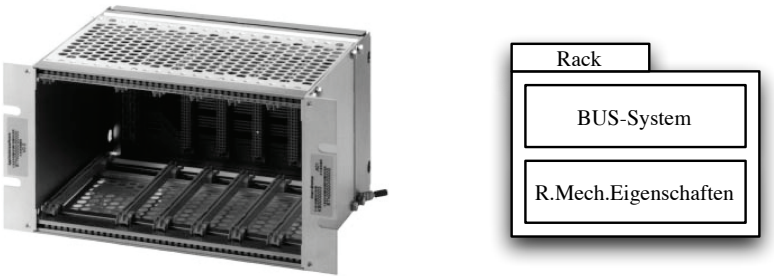


Abbildung 4.6.: 19-Zoll Rack [AG06].

    · Netzanschluss
      Das Netzteil des ESRA-Gleitschutzsystems versorgt alle 19-Zoll Einschubkarten des Racks mit den notwendigen Versorgungsspannungen. Diese werden innerhalb des Netzteils aus der externen Versorgungsspannung erzeugt und über das BUS-System des Racks verteilt.

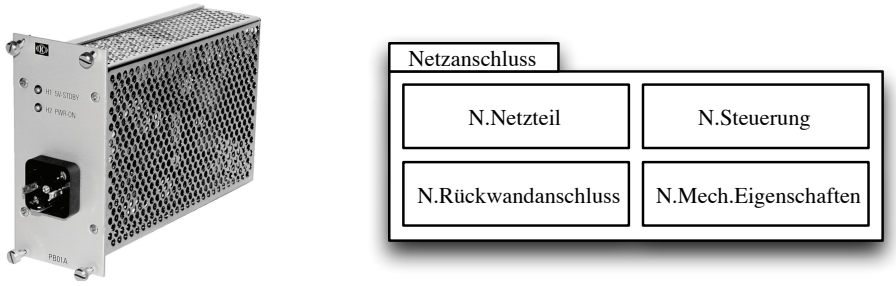


Abbildung 4.7.: 19-Zoll Einschubkarte – Netzanschluss [AG06].

    · Mainboard
      Das Mainboard dient zum Anschluss der Gleitschutzsensoren. Außerdem werden im Mainboard sämtliche eingehenden Daten des Sensors verarbeitet und ausgewertet.

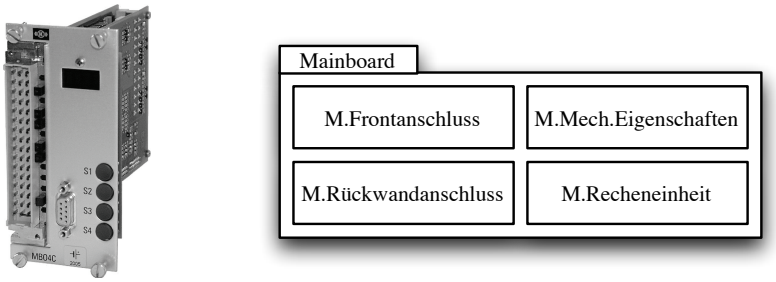


Abbildung 4.8.: 19-Zoll Einschubkarte – Mainboard [AG06].

      Alle weiteren Funktionen des Mainboards werden für die Fallstudie nicht benötigt und aus diesem Grund auch nicht weiter ausgeführt.

· Restliche Boards des Gleitschutzsystems
Das *Kommunikationsboard* sowie das *Erweiterungsboard* werden in dieser Fallstudie nicht berücksichtigt und auch nicht explizit in das (U)CML-Modell aufgenommen.

* Einbauort des Sensors
Der Einbauort des Gleitschutzsensors befindet sich direkt am Drehgestell des Wagens. Der Einbauort ist vor allem für die mechanischen Belastungen und die Umwelteinflüsse auf den Sensor wie z.B. Temperatur und Luftfeuchtigkeit, von erheblichem Interesse, wenn ein Sensor gegen einen anderen ausgetauscht werden soll.

* Sensor GI5
Der Gleitschutzsensor GI5 dient zur Bestimmung der aktuellen Radgeschwindigkeit. Diese Information wird durch ein elektrisches bzw. Softwaresignal an das Mainboard übermittelt.

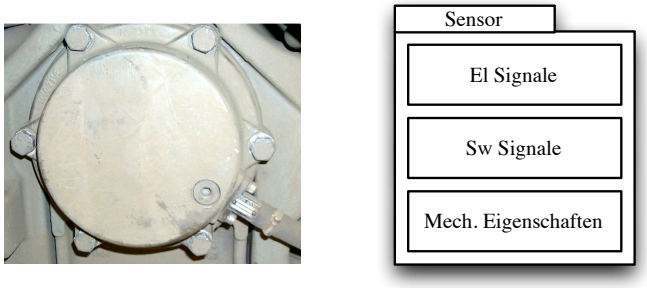


Abbildung 4.9.: Gleitschutzsensor [AG06].

– *Softwaremodule*
Das ESRA-Gleitschutzsystem beinhaltet eine große Anzahl an unterschiedlichen Softwaremodulen, die zusammen die Funktionalität des Systems bilden. In der stark vereinfachten Fallstudie wird nur eine Funktion des Softwarepakets („Lese Wert“ des Gleitschutzsensors) exemplarisch herausgegriffen und in (U)CML modelliert.

* Betriebssystem und Steuersoftware
Das Betriebssystem bildet die Grundlage aller anderen Softwarepakete des Systems. Es steuert alle internen Funktionen des Mainboards. Des Weiteren bildet es die Grundlage für die Kommunikation mit allen anderen Bestandteilen des Gleitschutzsystems, vor allem zur Auswertung des Gleitschutzsensors.

* Kommunikationssoftware
Die Kommunikationssoftware stellt die Verbindung zwischen den unterschiedlichen Bestandteilen der ESRA-Steuerelektronik her. Außerdem wird durch die Kommunikationssoftware die Verbindung zu den restlichen Zugsystemen hergestellt. In der Fallstudie ist vor allem eine C-Funktion der Kommunikationssoftware von Interesse, die die Kommunikation des Mainboards mit dem angeschlossenen Gleitschutzsensor realisiert.

- **Identifikation der Eigenschaften der HW/SW-Elemente des Gleitschutzsystems**
Nachdem die Identifikation der Hard- und Softwarestrukturelemente des Gleitschutzsystems abgeschlossen ist, folgt nun die Identifikation der wichtigsten Eigenschaften der gefundenen Komponenten. Um das (U)CML-Modell bzw. den Rahmen dieser Arbeit nicht zu sprengen, wird lediglich die Komponente *Mainboard* sowie der *Gleitschutzsensor* exemplarisch für alle anderen Komponenten des Systems herausgegriffen und ihre (kompatibilitätsrelevanten) Eigenschaften aufgelistet.

  Begonnen wird mit der Identifikation und Auflistung der kompatibilitätsrelevanten Eigenschaften der Komponente *Mainboard*.

  Baugruppe Mainboard
  Das Mainboard ist die komplexeste Baugruppe des gesamten ESRA-Gleitschutzsystems. Sie dient zur Ansteuerung und Auswertung der Gleitschutzsensoren, steuert alle anderen 19-Zoll Einschübe des Racks und stellt die Kommunikation mit den restlichen Zugsystemen her. Außerdem besitzt sie eine einfache Mensch-Maschine-Schnittstelle, über die der Zugführer den Status des gesamten ESRA-Systems abfragen kann. Für das (U)CML-Modell der Fallstudie sind jedoch nur drei Baugruppen des Mainboards von Interesse. Diese Baugruppen sind in der nachfolgenden Aufzählung aufgeführt sowie ihre Funktion kurz angerissen:

  – *Frontanschluss*
  Der Frontanschluss dient zum Anschluss sämtlicher Gleitschutzsensoren an das Mainboard.
  – *Rückwandanschluss*
  Über den Rückwandanschluss (BUS-Stecker) ist das Mainboard mit dem BUS-System des 19-Zoll Racks verbunden. Über dieses BUS-System wird das Mainboard mit sämtlichen Versorgungsspannungen, die für die internen Baugruppen sowie die Gleitschutzsensoren benötigt werden, versorgt.
  – *Recheneinheit*
  Mit Hilfe der Recheneinheit werden die ankommenden elektrischen bzw. Softwaresignale der Gleitschutzsensoren ausgewertet und die Ergebnisse an das BUS-System bzw. die Steuereinheit der Gleitschutzventile übergeben. Außerdem sind in der Recheneinheit alle Softwarekomponenten gespeichert, die für den Betrieb des Gleitschutzsystems notwendig sind. Im Beispiel sind dies insbesondere die Kommunikations- und Auswertefunktionen der Gleitschutzsensoren.

Die „restlichen“ HW/SW-Baugruppen des Mainboards sind für das vereinfachte Gleitschutzsystem nicht von Bedeutung und werden aus diesem Grund auch nicht modelliert. Die kompatibilitätsrelevanten Eigenschaften der aufgeführten Baugruppen des Mainboards – Front- bzw. Rückwandanschluss und Recheneinheit – werden nun einzeln untersucht und entsprechend ihrer Domänenzugehörigkeit aufgelistet.

– **Elektrotechnik**
In diesem Abschnitt der Aufzählung werden sämtliche kompatibilitätsrelevanten elektrischen und elektronischen Eigenschaften des Pakets *Mainboard* bzw. der drei Hauptkomponenten des Pakets *Mainboard* – M.Frontanschluss, M.Rückwandanschluss und M.Recheneinheit – einzeln aufgelistet und erläutert. Begonnen wird mit den elektrischen/elektronischen Eigenschaften der Komponente *M.Frontanschluss* des Mainboards.

* *Frontanschluss*
In der Tabelle 4.1 sind alle kompatibilitätsrelevanten elektrischen/elektronischen Eigenschaften der Komponente *M.Frontanschluss* – der logische Signalname, die Richtung des Signals (IN, OUT, IN/OUT), der elektrische Signalpegel inklusive möglicher Abweichungen (Toleranzen) und ein optionales Kommentarfeld – aufgelistet. Zusätzlich zu den elektrischen/elektronischen Signalen sind die benötigten Softwarefunktionen (C++ Funktionen) aufgelistet, weil diese in einem eingebetteten System sehr eng an die entsprechenden elektrischen Signalleitungen gekoppelt sind (siehe unten).

  Sämtliche in der Tabelle enthaltenen Daten und Informationen wurden aus unterschiedlichen Datenblättern der entsprechenden Bauteile bzw. gegebener Stromlaufpläne wie z.B. der Abbildung 4.10, entnommen und in der Tabelle 4.1 kompakt zusammengefasst.

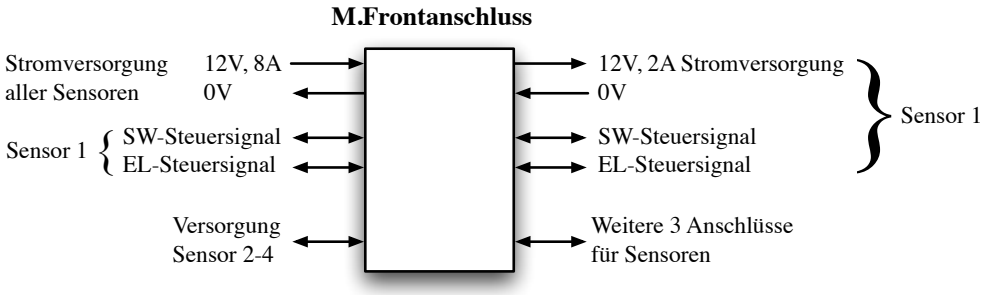


Abbildung 4.10.: Logischer Schaltplan des *M.Frontanschluss*.

Sig.Name	Richtung	Signalpegel		Kommentar
I2	IN	$+12V, \pm 0.5V; 8A, \pm 0.1A$		Gemeinsame Stromversorgung aller Sensoren
O2	OUT	$0V$		Gemeinsame Stromversorgung aller Sensoren
I1	IN	$0V$		Stromversorgung Sensor 1
O1	OUT	$+12V, \pm 0.5V; 2A, \pm 0.1A$		Stromversorgung Sensor 1
IO1	IN/OUT	C++ Methode		Daten Sensor 1
IO2	IN/OUT	Rechtecksignal	$Low: 0 \ldots 2V, \pm 0.2V$ $High: 3 \ldots 5V, \pm 0.2V$ $f_{min} = 10Hz$ $f_{max} = 10Khz$	El-Signal des Sensors 1 (IO1)
IO3	IN/OUT	C++ Methode		Datenverkehr Sensor 1
IO4	IN/OUT	Rechtecksignal	$Low: 0 \ldots 2V, \pm 0.2V$ $High: 3 \ldots 5V, \pm 0.2V$ $f_{min} = 10Hz$ $f_{max} = 10Khz$	El-Signal des Sensors 1(IO3)

Tabelle 4.1.: Signalbeschreibung Mainboard – Komponente *M.Frontanschluss*.

Erläuterungen zur Tabelle 4.1:

· Die beiden logischen Anschlüsse *IO2* und *IO4* des Frontanschlusses sind als Ein- und Ausgang (IO) modelliert, obwohl ein elektrisches/elektronisches Signal nur in eine Richtung (im Allgemeinen von *plus* nach *minus*) „fließen“ kann[280]. In diesem Beispiel kann der Anschluss *IO2* bzw. *IO4* jedoch sowohl als Eingang als auch als Ausgang durch die Komponente *M.Rechneneinheit* beschalten bzw. konfiguriert werden. Aus diesem Grund wurden die beiden Anschlüsse als Kommunikationsanschluss bzw. Kommunikationsstecker/-buchse in (U)CML definiert.

[280] In Stromlaufplänen und Bauteilbeschreibungen wird stets die *technische Stromrichtung* von plus nach minus verwendet.

· Soll in einem kombinierten Hardware/Software (U)CML-Modell ein Softwaresignal von einer Komponente zu einer anderen übertragen werden, so muss eine entsprechende elektrische Verbindung zwischen diesen beiden Komponenten vorhanden sein, wenn der Zusammenhang zwischen dem elektronischen/elektrischen Signal und dem Softwaresignal explizit modelliert werden soll. Im Allgemeinen können über eine elektrische/elektronische Verbindung beliebig viele Softwarefunktionen (sequentiell) ausgeführt bzw. übertragen werden. Die Koppelung der elektrischen und Softwareeigenschaften einer Verbindung kann als Kommentar im Beschreibungsfeld angegeben werden[281]. In der Tabelle sind die beiden Ein- und Ausgänge *IO1*, *IO2* sowie *IO3* und *IO4* gekoppelte elektrische/elektronische Verbindungen, über die beispielsweise eine Softwarefunktion aufgerufen werden kann.

  In (U)CML ist es auch möglich, eine Softwarefunktion ohne eine explizit modellierte elektrische/elektronische Verbindung zu realisieren. Diese Modellierungsart sollte jedoch nur verwendet werden, wenn die Kopplung zwischen elektronischer Verbindung und Softwarefunktionsaufruf vernachlässigt werden kann, wie z.B. in einem „reinen" Softwaresystem, bzw. dem Aufruf einer Softwarefunktion aus einer Softwarebibliothek.

Alle Angaben aus der Tabelle können im Modellierungsschritt „Befüllung der Beschreibungsfelder mit den Eigenschaften des Systems" in die entsprechenden Beschreibungsfelder der Komponenten bzw. der Stecker und Buchsen der Komponente *M.Frontanschluss* eingetragen werden.

* *Rückwandanschluss*
Der *M.Rückwandanschluss* des Pakets *Mainboard* dient als Verbindung zwischen dem BUS-System des Racks und den internen Komponenten des Pakets *Mainboard*. In der Abbildung 4.11 ist der logische Schaltplan der Komponente *M.Rückwandanschluss* dargestellt und in der darauf folgenden Tabelle 4.2 sind sämtliche kompatibilitätsrelevanten elektrischen/elektronischen Eigenschaften des Rückwandanschlusses, unterteilt in Komponenten- und BUS-System Seite, aufgelistet.

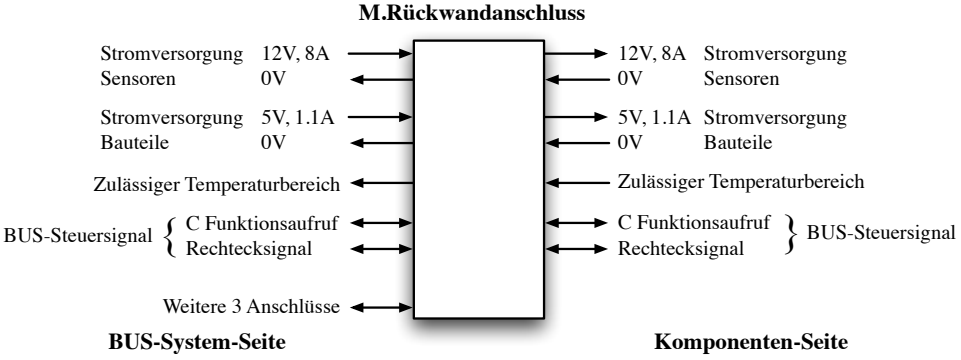


Abbildung 4.11.: Logischer Schaltplan der Komponente *M.Rückwandanschluss* des Mainboards.

Sig. Name	Richtung	Signalpegel		Kommentar
		Komponenten-Seite		
O1	OUT	$+12$V, ± 0.5V; 8A, ± 0.1A		Stromversorgung Sensoren
I3	IN	0V		Stromversorgung Sensoren
O2	OUT	$+5$V, ± 0.5V; 1.1A, ± 0.1A		Stromversorgung Bauteile
I2	IN	0V		Stromversorgung Bauteile
I1	IN	$-40^\circ \ldots +120^\circ$C		Zulässiger Temperaturbereich der Komponente *M.Recheneinheit* („mechanisches Signal")
IO1	IN/OUT	C++ Methode		Softwarefunktion des Signals IO2
IO2	IN/OUT	Rechtecksignal	$Low: 0 \ldots 2$V, ± 0.2V $High: 3 \ldots 5$V, ± 0.2V $f_{min} = 10$Hz $f_{max} = 10$Khz	Rechtecksteuersignal für angeschlossene ESRA-Baugruppen
		BUS-System-Seite		
		Alle Signale sind gleich, nur die Richtung ändert sich.		

Tabelle 4.2.: Signalbeschreibung Mainboard – Rückwandanschluss.

Alle Signale, die an der Komponente *M.Rückwandanschluss* angeschlossen sind (*I1-I3*, *O1* und *O2* sowie *IO1* und *IO2*), werden über den BUS-System-Anschluss an das BUS-System übergeben (BUS-System-Seite). Dabei ändert sich lediglich die *Richtung* des Signals (siehe Abbildung 4.11). Aus einem Eingangssignal wird ein Ausgangssignal und umgekehrt.

[281] Siehe hierzu: Kapitel „3.6.4.3 *Modellierung von elektrischen/elektronischen Signalen in (U)CML*" ab Seite 234.

Jede am BUS-System angeschlossene Komponente kann alle verfügbaren Signale am BUS-System abgreifen und intern weiter nutzen. Die Steuerung (also die Arbitrierung des BUS-Systems) wird durch die Steuerlogik der Komponente *M.Recheneinheit* wahrgenommen. Somit ist die Komponente *M.Recheneinheit* die einzige aktive Komponente (Master-Komponente) am BUS-System. Alle anderen angeschlossenen Baugruppen werden vom Mainboard gesteuert (Slave-Komponenten).

* *Recheneinheit*
Die Komponente *M.Recheneinheit* des Pakets *Mainboard* ist die zentrale Steuereinheit des Gleitschutzsystems. In der Tabelle 4.3 sind alle kompatibilitätsrelevanten elektrischen/elektronischen Eigenschaften eingetragen.

Sig. Name	Richtung	Signalpegel		Kommentar
I1	IN	$+5$V, ± 0.5V; 1.1A, ± 0.1A		Stromversorgung Recheneinheit
O1	OUT	0V		Stromversorgung Recheneinheit
IO1	IN/OUT	C++ Methode		Datenverkehr Sensor 1 (IO2)
IO2	IN/OUT	Rechtecksignal	$Low : 0 \ldots 2\text{V}, \pm 0.2\text{V}$ $High : 3 \ldots 5\text{V}, \pm 0.2\text{V}$ $f_{min} = 10\text{Hz}$ $f_{max} = 10\text{Khz}$	El-Signal des Sensors 1
IO3	IN/OUT	C++ Methode		Kommunikationssignal (IO4)
IO4	IN/OUT	Rechtecksignal	$Low : 0 \ldots 2\text{V}, \pm 0.2\text{V}$ $High : 3 \ldots 5\text{V}, \pm 0.2\text{V}$ $f_{min} = 10\text{Hz}$ $f_{max} = 10\text{Khz}$	El-Kommunikationssignal
O2	OUT	$-40° \ldots +120°\text{C}$		Zulässiger Temperaturbereich der Recheneinheit („mechanisches Signal“)

Tabelle 4.3.: Signalbeschreibung Mainboard – Recheneinheit.

Anmerkung zur Tabelle 4.3:
In der Tabelle ist die mechanische Eigenschaft *zulässiger Temperaturbereich* (Zeile *O2*) der Komponente *M.Recheneinheit* modelliert. Diese Eigenschaft dient der Überwachung des maximal zulässigen Betriebstemperaturbereichs der Komponente *M.Recheneinheit*. Die mechanische Funktion *O2* existiert dabei nur auf der Modellebene. Aus diesem Grund muss hier keine elektrische „Trägerleitung“ wie bei den Softwarefunktionen modelliert werden. Wenn die Temperatur des Mainboards nicht nur auf der Modellebene, sondern im realen System von einer Komponente zu einer anderen übertragen werden soll, muss sie ebenfalls als elektrische Verbindung im Modell vorhanden sein (vgl. Erläuterung zwischen EL-Signal und SW-Signal in der Tabelle 4.1 auf Seite 271 bzw. das Kapitel 3.6.4.1 *Modellierung von mechanischen Signalen in (U)CML* ab Seite 229.).

– **Mechanik**
Nachdem nun alle für das Gleitschutzsystem relevanten elektrischen/elektronischen Eigenschaften und Signale des Pakets *Mainboard* identifiziert und in tabellarische Form gebracht worden sind, folgt nun die Beschreibung der mechanischen Eigenschaften der Baugruppe *Mainboard*. Dabei werden die beiden Komponenten des Pakets *Mainboard* – Frontanschluss und Rückwandanschluss – einzeln untersucht. Die Komponente *M.Recheneinheit* wird im Rahmen der Fallstudie Gleitschutzsystem nicht weiter analysiert. Stellvertretend für ihre Eigenschaften wird hier nur der maximal zulässige Temperaturbereich der Komponente untersucht. Des Weiteren werden die mechanischen Eigenschaften der Baugruppe *Mainboard* (genauer: des (U)CML-Pakets *Mainboard*) aufgeführt.

* *Frontanschluss*
An der Komponente *M.Frontanschluss* bzw. an der physikalischen Repräsentation des Frontanschlusses des Mainboards, also der Buchse DIN 41612F (Messerleiste), wird der entsprechende Stecker (Federleiste), an dem die einzelnen Verbindungsleitungen zu den Gleitschutzsensoren angebracht sind, angeschlossen. In der Abbildung 4.12 auf der nächsten Seite ist die CAD Zeichnung der 48-poligen Anschlussbuchse DIN 41612F und in der Abbildung 4.13 der reale Stecker bzw. die dazugehörige Buchse, dargestellt. In der Fallstudie wird genau ein Sensor GI5 an die Frontbuchse des Mainboards angeschlossen.

  In der nachfolgenden Tabelle ist die Pin-Belegung (die Zuordnung zwischen den *logischen Signalen* aus der Tabelle 4.1 auf Seite 270 und den *physikalischen Pins*) der Buchse *M.Frontanschluss* einzeln aufgelistet. Die Belegung des Steckers ergibt sich aus der Pin-Belegung der Buchse und wird nicht einzeln aufgeführt.

* *Rückwandanschluss*
In der Abbildung 4.14 auf der nächsten Seite ist die CAD Zeichnung der Komponente *M.Rückwandstecker* des Pakets *Mainboard* dargestellt. Der Rückwandstecker, der das Mainboard mit der BUS-Systembuchse des Racks verbindet, hat 96-Pins, die in drei Reihen (A-B-C) zu je 32-Pins angeordnet sind.

  In der Tabelle 4.5 auf der nächsten Seite ist die Zuordnung der physikalischen Pins zu den logischen Signalen aus der Tabelle 4.2 auf der vorherigen Seite aufgelistet. Dabei ist zu beachten, dass die in der Tabelle 4.2 aufgelisteten logischen Signalnamen sich auf die interne Verbindung, also die Verbindung der Komponente *M.Rückwandanschluss* mit der Komponente *M.Recheneinheit* (Komponenten-Seite) bezieht und nicht auf die BUS-System-Seite.

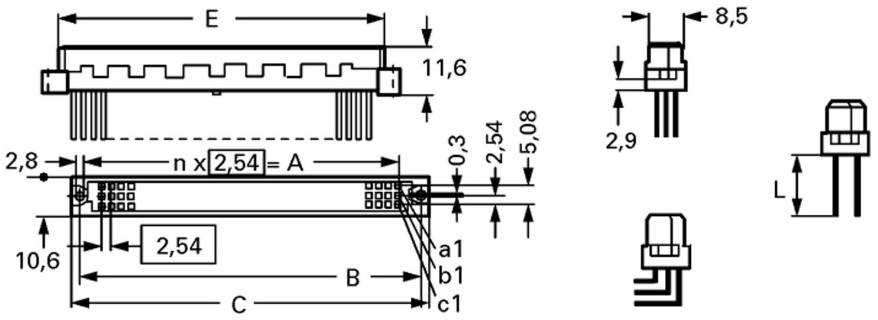


Abbildung 4.12.: CAD Zeichnung der Messerleiste DIN 41612F [OHG07].

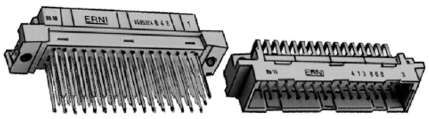


Abbildung 4.13.: Schematische Darstellung der Messer- sowie Federleiste DIN 41612F[OHG07].

Pin	Signalname	Signalbeschreibung
1	IN1	Stromversorgung Sensor 1
2	O1	Stromversorgung Sensor 1
3	IO2	El-Signal des Sensors 1
4-48		Restliche Anschlüsse

Tabelle 4.4.: Pin-Belegung der Buchse (Messerleiste) DIN 41612F des Mainboards.

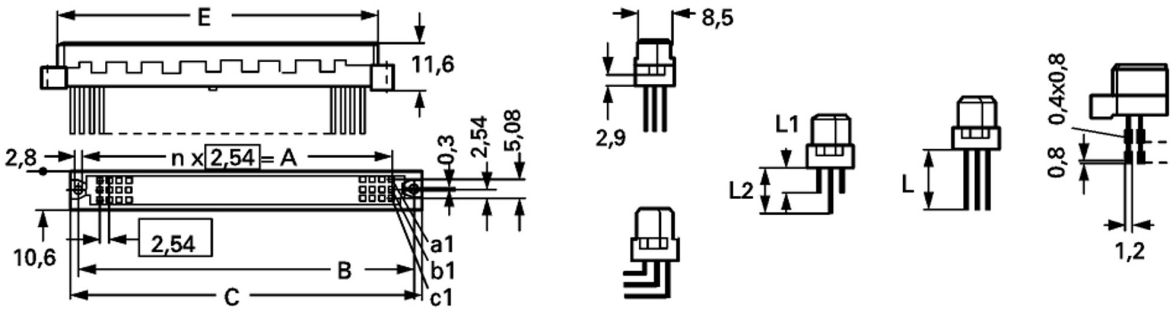


Abbildung 4.14.: Rückwand Messerleiste DIN 41612C [OHG07].

Pin	Signalname	Signalbeschreibung
1	O1	Stromversorgung Sensoren
2	I3	Stromversorgung Sensoren
3	O2	Stromversorgung Bauteile
4	I2	Stromversorgung Bauteile
5	IO1	Steuersignal – C++ Methodenaufruf
6	IO2	Steuersignal – El. Signal
7	I1	Temperatur – Mech. Signal
8-96		Restliche Signale

Tabelle 4.5.: Pin-Belegung der Buchse (Messerleiste) DIN 41612C des Mainboards.

* *Recheneinheit*
Stellvertretend für die mechanischen Eigenschaften der Komponente *M.Recheneinheit* des Pakets *Mainboard* wird hier der *maximal zulässige Temperaturbereich* der Baugruppe angegeben, in dem sie verbaut bzw. eingesetzt werden kann. Die Baugruppe darf nur in einer Umgebung verwendet werden, die innerhalb von $-40°\mathrm{C}$ bis $+120°\mathrm{C}$ liegt. Der Einsatz in allen anderen Umgebungen ist nicht zulässig. Die Überprüfung des gültigen Temperaturbereichs

findet nur auf Modellebene statt, da der angegebene Temperaturbereich nur zur Auslegung der Baugruppen dient. Das heißt: Im laufenden realen System wird die Temperatur nicht abgefragt oder ausgewertet. Aus diesem Grund ist auch, wie bereits erwähnt, keine elektrische Verbindung vorhanden, über die die eigentlichen Daten versendet werden[282].

* *Mainboard*
  Der letzte Abschnitt der Beschreibung der kompatibilitätsrelevanten mechanischen Eigenschaften des Pakets *Mainboard* ist die Beschreibung der verschiedenen Abmessungen der Baugruppe. In der Abbildung 4.15 wird eine vereinfachte CAD Darstellung des Frontblechs der Baugruppe Mainboard gezeigt. Aus dieser Zeichnung wurden zum Teil die Werte, die in der Tabelle 4.6 aufgeführt sind entnommen. Die anderen Abmessungen stammen aus den CAD Zeichnungen des Frontanschlusses DIN 41612F bzw. des Rückwandanschlusses DIN 41612C bzw. weiteren CAD Abbildungen der Baugruppe.

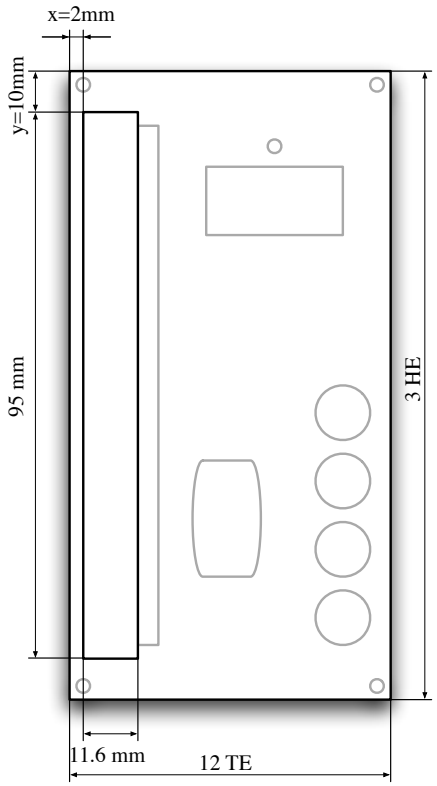


Abbildung 4.15.: CAD Zeichnung des 19-Zoll Frontblechs des Mainboards.

Bezeichnung	**Eigenschaften**	**Kommentar**
Gehäuse	Höhe: 3HE, Breite: 12TE, Tiefe: 160mm	Außenmaße des Gehäuses
Anschluss der Sensoren	Höhe: 11,6mm, Breite: 95mm, Tiefe: 10,6mm	Frontanschluss DIN 41612F für Sensoren
Position	x: 2mm y: 10mm	Position der Buchse DIN 41612F auf dem Frontblech
Rückwandanschluss	Höhe: 11,6mm, Breite: 95mm, Tiefe: 10,6mm	Rückwand BUS-Systemanschluss
Position	x: 2mm, y: 10mm	Position der Buchse DIN 41612C auf der Rückseite
Temperatur	$-40° \ldots +120°\mathrm{C}$	Maximal zulässiger Temperaturbereich

Tabelle 4.6.: Mechanische Daten des Mainboards.

– **Software**
  Nachdem nun alle kompatibilitätsrelevanten elektrischen/elektronischen sowie die mechanischen Eigenschaften der Baugruppe *Mainboard* in den letzten beiden Abschnitten dieses Kapitels erläutert wurden, folgt nun die Identifikation und Beschreibung der Softwarefunktionen des Mainboards. In der Abbildung 4.16 auf der nächsten Seite ist die Klassenstruktur des stark vereinfachten Gleitschutzsystems exemplarisch als UML/UML2-Klassendiagramm[283] dargestellt.

[282] Siehe hierzu: Kapitel „3.6.4.1 *Modellierung von mechanischen Signalen in (U)CML*“ ab Seite 229.
[283] Siehe hierzu: Kapitel „3.4.3 *Die Unified Modelling Language – UML/UML2*“ ab Seite 131.

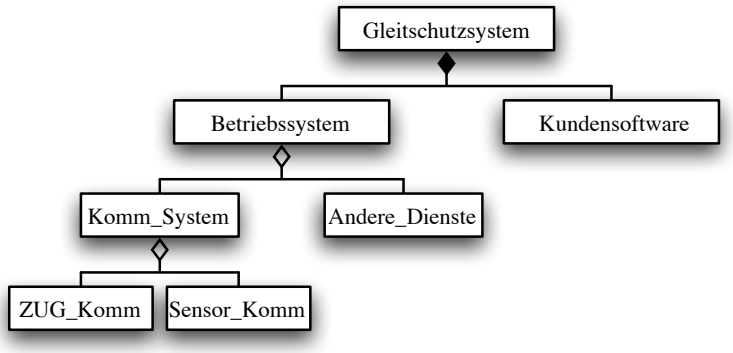


Abbildung 4.16.: UML/UML2-Diagramm des Aufbaus und der Struktur des Softwaresystems Gleitschutz.

In dem UML-Klassendiagramm (Abb.: 4.16) ist der grundsätzliche Aufbau sowie die Struktur der Klassen des vereinfachten Gleitschutzsystems dargestellt. Sie wurde aus den vorhandenen *.cpp bzw. *.h Dateien[284] des Systems bzw. aus Textdokumenten, wie z.B. Klassen- und Modulbeschreibungen extrahiert bzw. aus dem Quellcode reengineered.

Für die Fallstudie Gleitschutzsystem sind lediglich die beiden Klassen `Sensor_Komm` und `Kundensoftware` von Bedeutung. In der Klasse `Sensor_Komm` ist die C++ Methode `getCurrentValue()` enthalten. Mit ihrer Hilfe wird die aktuelle Umdrehungsgeschwindigkeit des Polrads des Gleitschutzsensors ermittelt. In der Klasse `Kundensoftware` wird die Methode `getCurrentValue()` aufgerufen und ausgewertet. Im nachfolgenden Listing ist die Klasse `Sensor_Komm` mit der Methode `getCurrentValue()` exemplarisch abgebildet.

```
1  class Sensor_Komm {
2    ...
3    public float getCurrentValue() {
4      // Lese Umdrehungsrate aus Speicherzelle
5      ...
6    }
7    ...
8  }
```

Für die Kompatibilitätsbestimmung in (U)CML sind außer der C++ Methodendeklaration noch weitere Informationen notwendig:

* **Datentypname**
  Der verwendete Datentyp der C++ Methode `getCurrentValue` ist **`float`** (Zeile 3).
* **Definitionsbereich des Datentyps**
  Für die Bestimmung des maximal zur Verfügung stehenden Definitionsbereichs des Datentyps **`float`** wird die Breite der Register der verwendeten Hardware benötigt. Das ESRA-Gleitschutzsystem arbeitet intern mit 32-Bit breiten Registern. Daraus leitet sich ein maximaler Wertebereich von $1.175494 \ldots 10^{-38}$ bis $3.402824 \ldots 10^{+38}$ für den Datentyp **`float`** ab [285]. Zusätzlich zur Registerbreite ist die Datentypausrichtung für die Kompatibilitätsbestimmung essentiell. Das Gleitschutzsystem liegt als „little endian“ System[286] vor.

  Zusätzlich zum Definitionsbereich des Mainboards wird der Wertebereich des Gleitschutzsensors benötigt. Der Gleitschutzsensor hat ebenfalls den Definitionsbereich **`float`**, jedoch wird davon nicht der gesamte Wertebereich ausgeschöpft, den der Datentyp **`float`** bietet. Der Sensor GI5 nutzt lediglich den Bereich von $+10$ bis $+10000$ für die Repräsentation der Umdrehungsgeschwindigkeit (siehe Tab.: 4.7 auf Seite 277).
* **Einheit**
  Dem Datentyp **`float`** der C++ Methode `getCurrentValue()` ist die Einheit Hz zugeordnet.
* **Startwert und Stepping**
  Der kleinste Wert, den der Gleitschutzsensor GI5 liefet, ist 10Hz. Dieser Wert wird als Startwert für die Kompatibilitätsbestimmung verwendet. Der Sensor liefert kontinuierliche Werte. Aus diesem Grund wird das Feld Stepping nicht genutzt.
* **Methodenaufruf**
  Zuletzt wird der „Aufrufer“ der Methode `getCurrentValue()` aus der Klasse `Kundensoftware` benötigt. Denn nur so kann überprüft werden, ob dieser ebenfalls die hier definierten Eigenschaften aufweist. Dieser Schritt ist jedoch nicht einfach zu realisieren, da im Allgemeinen in einem UML-Modell der Software zwar die Beziehungen zwischen den einzelnen Klassen modelliert und dargestellt werden können, jedoch der eigentliche Methodenaufruf

[284] Die *.cpp und *.h Dateien bilden zusammen den Quellcode des Systems. Aus diesem wird mit Hilfe eines Compilers das ausführbare Programm erzeugt.
[285] Nähere Informationen zum IEEE Zahlensystem bzw. zum Definitionsbereich von Datentypen in der Programmiersprache C++ finden Sie unter [IEE06], [Wik06f], [Pre06], [Böh06] sowie [Wik06f].
[286] Siehe hierzu: Kapitel „2.4.2.1 *Statisches projektspezifisches Regelwerk*“ ab Seite 84.

nicht[287]. Aus diesem Grund müssen diese Beziehungen aus dem Quellcode extrahiert werden, so dass sie anschließend in (U)CML modelliert werden können. Im nachfolgenden Listing ist ein Auszug aus dem Quellcode der Klasse `Kundensoftware` abgebildet, der die Methode `BerechneAktuelleRadgeschwindigkeit()` enthält.

```
1  class Kundensoftware {
2    private:
3      Sensor_Komm sensor;
4    ...
5    public void BerechneAktuelleRadgeschwindigkeit() {
6      float aktRadGeschwindigkeit = 0.0;
7
8      ...
9      // Hole aktuelle Radgeschwindigkeit vom Sensor
10     aktRadGeschwindigkeit = sensor.getCurrentValue();
11     ...
12   }
13   ...
14 }
```

Aus dem obigen Listing ist ersichtlich, dass die aufrufende Methode `BerechneAktuelleRadgeschwindigkeit()` (Zeile 5) als Ergebniswert ebenfalls **`float`** (Zeile 6) erwartet. Alle anderen Angaben sind aus dem Listing nicht zu entnehmen und müssen wieder aus der Dokumentation der Klassen bzw. der Hardwarebeschreibung entnommen werden. Im hier gewählten Beispiel stimmen alle kompatibilitätsrelevanten Eigenschaften überein.

Anmerkung:
Für die Modellierung von Software in (U)CML ist es hilfreich, wenn bereits im Quellcode Angaben über den verwendeten Wertebereich bzw. eine Zuordnung von Datentypen zu Einheiten, z.B. als Kommentar eingetragen ist. Dadurch wird nicht nur die Modellierung in (U)CML, sondern auch die „klassische“ Fehlersuche erleichtert, so dass ein Fehler, wie auf Seite 58 beschreiben, nicht auftreten kann.

Baugruppe Sensor
Der Gleitschutzsensor dient zur Messung der aktuellen Umdrehungsgeschwindigkeit eines Rades bzw. eines Radsatzes des Drehgestells. Der Sensor ist wiederum ein eingebettetes System mit einem elektrischen und einem mechanischen Anteil. Im Gegensatz zum Mainboard besitzt der Sensor keine eigene Software. Trotzdem ist es sinnvoll, in den Sensor eine „virtuelle Softwarefunktion“ zu integrieren, um die Kommunikation zwischen dem Sensor auf der einen Seite und dem Mainbaord auf der anderen besser beschreiben zu können. Ohne die virtuelle Softwarefunktion müsste das Mainboard die Umwandlung des elektrischen Signals in eine Softwarefunktion übernehmen[288] und dadurch die Kompatibilität zwischen dem Sensor und dem Mainboad sicherstellen. Auf Modellebene ist es sinnvoller, die Umwandlung des elektrischen/elektronischen Signals vom Sensor erledigen zu lassen, da dieser besser weiß, wie das elektrische Signal in einen Softwarewert überführt werden kann („objektorientierte Herangehensweise[289]“).

In der nachfolgenden Aufzählung werden die kompatibilitätsrelevanten Eigenschaften des Sensors in Abhängigkeit von der jeweiligen Domäne aufgeführt und erläutert.

– **Elektrotechnik**
  Der elektrische/elektronische Anteil des Gleitschutzsensors sorgt für die Umsetzung der mechanischen Umdrehung des Polrads (vgl. Abb. 4.3 auf Seite 264) in eine frequenzabhängige Rechteckspannung (Abb.: 4.18 auf der nächsten Seite links). Die Rechteckspannung wird an das Mainboard übertragen und dort von einer C++ Methode ausgewertet (s.o. Klasse *Kundensoftware*). Im (U)CML-Modell des Gleitschutzsensors wird zusätzlich zur elektrischen/elektronischen Schnittstelle die oben angesprochene virtuelle Softwarefunktion eingeführt, mit deren Hilfe der ermittelte Wert der Umdrehungsgeschwindigkeit des Polrads in ein Softwaresignal transformiert wird. Dieses wird zusätzlich zum elektrischen/elektronischen Signal an das Mainboard übermittelt. Der hier beschriebene Sachverhalt wird im Abschnitt Software des Sensors noch einmal aufgegriffen und ausführlich diskutiert. Die Abbildung 4.17 auf der nächsten Seite zeigt den logischen Schaltplan des Sensors GI5 inklusive der zusätzlichen Softwarefunktion.

  In der nachfolgenden Tabelle 4.7 auf der nächsten Seite werden die kompatibilitätsrelevanten elektrischen/elektronischen Eigenschaften des Gleitschutzsensors GI5 quantifiziert. Alle Eigenschaften wurden der technischen Beschreibung des Sensors entnommen.

  Die Umdrehungsgeschwindigkeit des Polrads wird vom Sensor in ein Rechtecksignal mit unterschiedlichen Frequenzen umgesetzt. Dabei gilt: Je höher die Umdrehungsgeschwindigkeit des Polrads des Sensors, desto höher ist die Frequenz des Rechtecksignals. Im Stillstand ($v = 0\mathrm{Km/h}$) wird der Startwert $f = 10\mathrm{Hz}$ übertragen. Die Frequenz steigt dann, in Abhängigkeit von der aktuellen Radgeschwindigkeit, linear bis zum Maximalwert der Frequenz $f = 10\mathrm{KHz}$ an (bei $v = 400\mathrm{Km/h}$). In der Abbildung 4.18 auf der nächsten Seite (rechts) wird der Zusammenhang zwischen der aktuellen Geschwindigkeit und der Frequenz des Rechtecksignals als Diagramm dargestellt.

[287] Anmerkung:
Wenn in UML außer der Klassenstruktur auch Sequenzdiagramme zur Modellierung der Aufrufbeziehungen benutzt werden, kann diese Information direkt in (U)CML überführt werden.
[288] Siehe hierzu: Kapitel „3.6.4 *Modellierung von Signalen in (U)CML*“ ab Seite 229.
[289] Siehe hierzu: Kapitel „2.2 *Modellbildung*“ ab Seite 31.

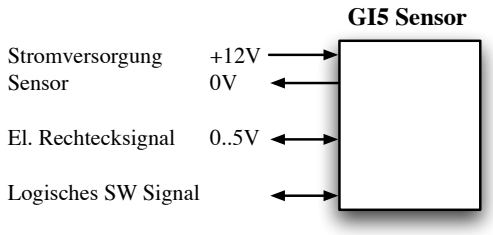


Abbildung 4.17.: Logischer Schaltplan des Sensors GI5.

Sig. Name	Richtung	Signalpegel		Kommentar
I1	IN	$+12V, \pm 0.5V; 2A, \pm 0.1A$		Stromversorgung Sensor
O1	OUT	0V		Stromversorgung Sensor
IO1	IN/OUT	C++ Methode		„virtuelle Softwarefunktion“ (IO2)
IO2	IN/OUT	Rechtecksignal	$Low : 0 \ldots 2V, \pm 0.2V$ $High : 3 \ldots 5V, \pm 0.2V$ $f_{min} = 10\text{Hz}$ $f_{max} = 10\text{Khz}$	Rechtecksignal der Umdrehungsgeschwindigkeit des Polrads

Tabelle 4.7.: Signalbeschreibung Sensor GI5.

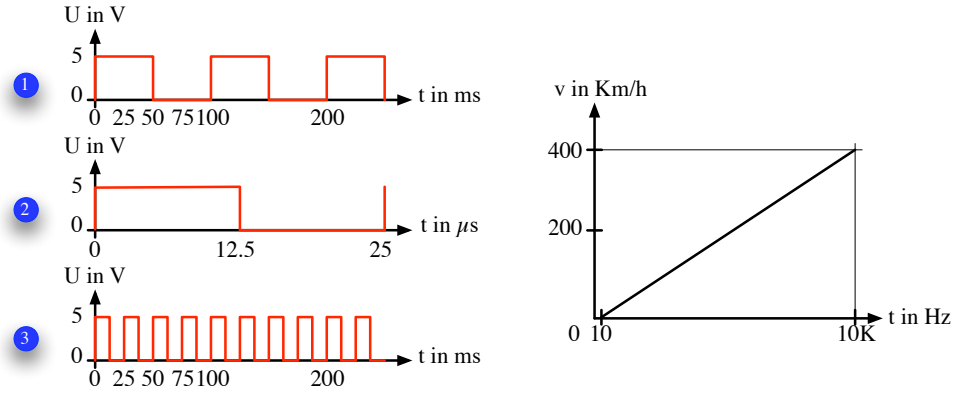


Abbildung 4.18.: Zusammenhang zwischen der Radgeschwindigkeit und Frequenz der Rechteckspannung.

Aus dem Diagramm (Abb. 4.18) lässt sich die Periodendauer der Rechteckschwingung sowie die Formel für die Berechnung der Werte der Schwingung herleiten.

1. Stillstand: $v = 0\text{Km/h}$
   $f = 10\text{Hz} : f = \frac{1}{T}$: Periodendauer $T = \frac{1}{f} = \frac{1}{10\text{Hz}} = 100\text{ms}$
2. Maximalgeschwindigkeit: $v = 400\text{Km/h}$
   $f = 10\text{KHz} : T = \frac{1}{10\text{KHz}} = 25\mu\text{s}$
3. Aktuelle Geschwindigkeit:
   Geradensteigung: $m = \frac{400-0}{10000-10} = 0,04$
   Achsenabschnitt: $y = mx + t : t = -0.4 : y = 0.04 * x - 0.4$
   daraus folgt: $T_{akt} = 25\text{ms}, f = \frac{1}{25\text{ms}} = 40\text{Hz} \mathrel{\widehat{=}} 1.2\text{Km/h}$

Anmerkung:
Die Formel $v(t) = 0.04 * x - 0.4$ ist vor allem für die Simulation des Systems wichtig. Im stark vereinfachten (U)CML-Modell des Gleitschutzsystems, wird sie nicht weiter verwendet.

In der Abbildung 4.19 auf der nächsten Seite ist das elektrische Rechtecksignal des Sensors (Ausgang: IO2) mit allen Werten aus den angegebenen Tabellen dargestellt. Außerdem ist in der Abbildung die Transformation des elektrischen Rechtecksignals in ein logisches Signal abgebildet. In der nachfolgenden Aufzählung sind die logischen (Bool´schen) Werte mit ihren Gültigkeitsbereichen für das elektrische Signal des Gleitschutzsensors definiert:

1. **High**
   Im Bereich von $3 - 5\text{V} \pm 0.2\text{V}$ hat das Rechtecksignal den logischen Zustand „high“.

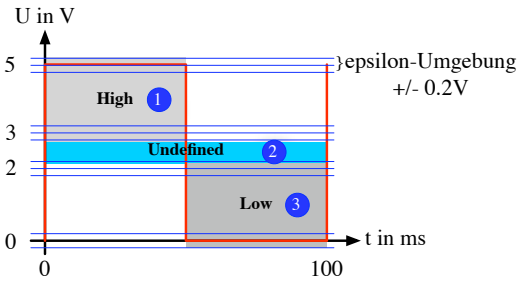


Abbildung 4.19.: Beschreibung der Rechteckspannung.

2. **Undefined**
   Im Bereich von $2.2 - 2.8\text{V}$ ist das logische Signal nicht definiert. Alle Werte in diesem Bereich können keinem logischen Wert zugeordnet werden.
3. **Low**
   Von $0 - 2\text{V} \pm 0.2\text{V}$ hat das Signal den logischen Zustand „low“.

Die beiden logischen Werte *high* und *low* bzw. die korrespondierenden elektrischen Wertebereiche sind für die Bestimmung der Kompatibilität eines elektrischen Signals entscheidend. Sind z.B. zwei Komponenten miteinander verbunden und der Wertebereich des elektrischen bzw. logischen Signals stimmt nicht exakt überein, so kann es zu erheblichen Kommunikationsproblemen zwischen diesen beiden Komponenten kommen (vgl. Abbildung 4.20).

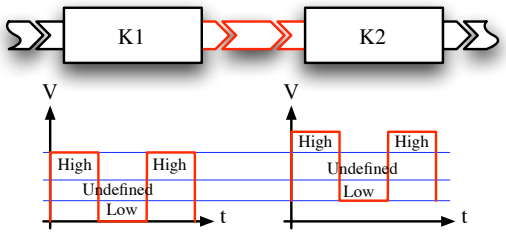


Abbildung 4.20.: Unterschiedliche Wertebereiche zweier elektrischer Signale.

Bei der elektrischen Signalübertragung von der Komponente *K1* zur Komponente *K2* kommt es zu erheblichen Kommunikationsproblemen, weil die beiden Wertebereiche der Komponenten bzw. der Stecker und Buchsen nicht übereinstimmen. Wenn die Komponente *K1* eine logische 0 auf dem Ausgangsstecker (*low*) sendet, so interpretiert die Komponente *K2* (Eingangsbuchse) diesen ebenfalls als *low*. Der logische Wert *high* der Komponente *K1* wird von der Komponente *K2* als *undefined* interpretiert, da er im nicht definierten Bereich liegt. Daraus folgt, dass die Komponente *K2* nie den logischen Wert *high* empfangen kann, da der logische *high*-Bereich der Komponente *K2* nie erreicht wird.

– **Mechanik**
Die mechanischen Eigenschaften des Sensors GI5 sind zum Teil aus den technischen Zeichnungen des Sensors bzw. aus Datenblättern entnommen. Die Abbildung 4.21 auf der nächsten Seite zeigt die schematische CAD Zeichnung der Außenmaße des Sensors GI5 inklusive der Anschlussbuchse. Die Außenmaße aus der Zeichnung wurden in die Tabelle 4.8 eingefügt. Zusätzlich sind die mechanischen Eigenschaften wie z.B. der maximal zulässige Temperaturbereich der Komponente *Sensor GI5*, in der Tabelle aufgeführt.

Eigenschaft	Werte	Kommentar
Gehäusegröße	Höhe: 100, Breite: 50, Tiefe: 50	Außenmaße in mm
Masse	$1,6\text{Kg}$	Masse ohne Stecker
Temperatur	$-45° \ldots 85°\text{C}$	
Luftfeuchtigkeit	$0\% \ldots 95\%$	Unkondensiert
Steckergröße	Höhe: 53, Breite: 73, Tiefe: 43	Außenmaße in mm
Steckertyp	Klemmadapter	10-polig
Buchsengröße	Höhe: 28, Breite: 93, Tiefe: 43	Außenmaße in mm
Buchsentyp	Klemmadapter mit Haltebügel	

Tabelle 4.8.: Mechanische Eigenschaften der Komponente Sensor GI5.

In der Abbildung 4.22 auf der nächsten Seite ist die am Sensor angebrachte Buchse (*Revos Basic Zweihandverriegelung 10-polig plus PE*) sowie der dazugehörige Stecker dargestellt. Aus dieser Abbildung wurden ebenfalls die Abmessungen der Buchse übernommen, die in der CAD Zeichnung bzw. in der Tabelle aufgeführt sind.

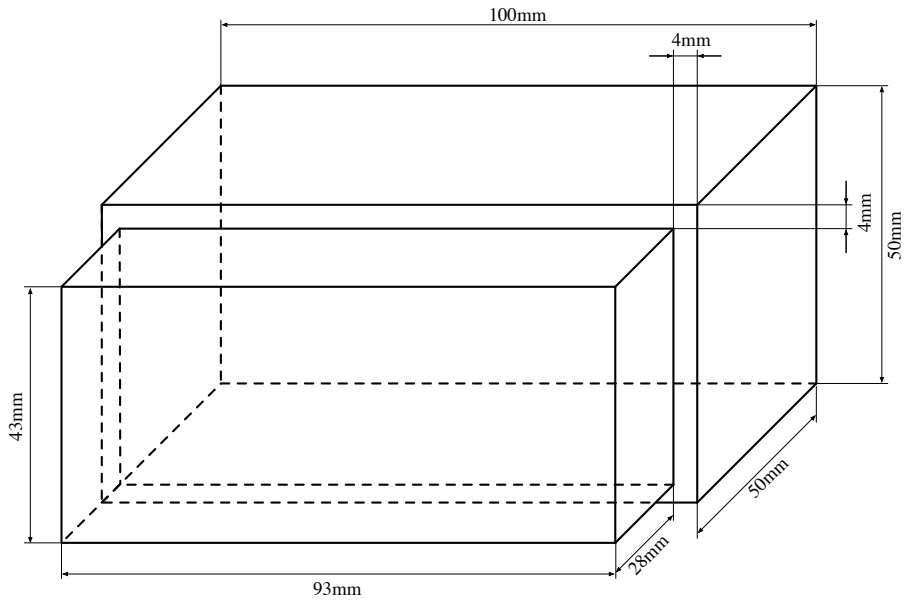


Abbildung 4.21.: CAD Zeichnung: Sensor GI5.

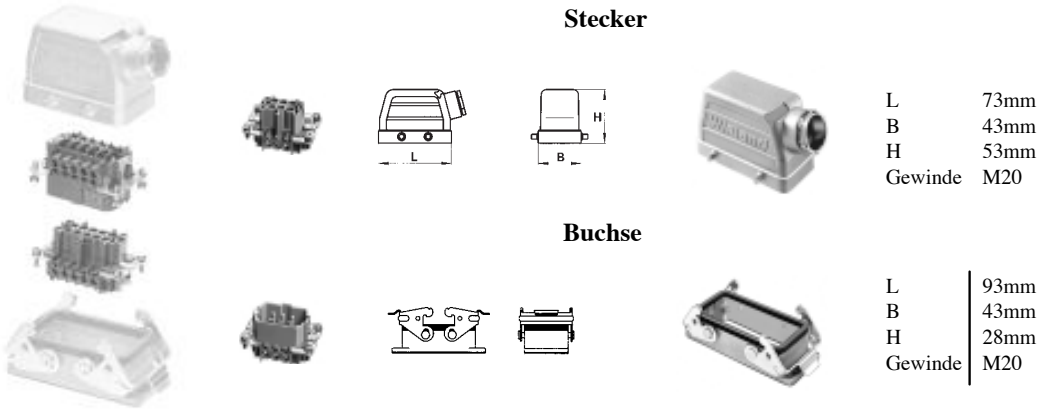


Abbildung 4.22.: Anschluss Sensor GI5 [Ele, 4,10,12].

Zuletzt folgt nun die Zuordnung der logischen elektrischen/elektronischen Signale zu den realen Pins der Buchse bzw. des Steckers. Die Tabelle 4.9 zeigt die Zuordnung der logischen Signale zu den realen Pins der Buchse *Revos Basic*. Der entsprechende Stecker ist spiegelbildlich zur Buchse beschalten und wird nicht extra aufgeführt.

Pin	Signalname	Signalbeschreibung
1	I1	Stromversorgung Sensor
2	O1	Stromversorgung Sensor
3	IO2	Rechtecksignal des Sensors
4-10		Unbelegte Pins
11-12		Schutzleiteranschluss (PE)

Tabelle 4.9.: Pin-Belegung der Anschlussbuchse des Sensors.

Anmerkung:
Die virtuelle Softwarefunktion zur Transformation bzw. Übertragung der Umdrehungsgeschwindigkeit des Sensors wird nur auf Modellebene vom Sensor an das Mainboard übertragen. Aus diesem Grund muss die Softwarefunktion nicht über den Stecker übertragen werden.

– **Software**
Der real existierende Gleitschutzsensor GI5 enthält, wie bereits in der Einleitung beschrieben, *keine* eigene Software. Für die Modellierung und vor allem die Kompatibilitätsbestimmung des Sensors in (U)CML ist es jedoch hilfreich, in das Modell des Sensor eine virtuelle Softwarefunktion zu integrieren, die die Transformation des elektrischen/elektronischen Messwerts des Sensors in ein Softwaresignal ausführt.

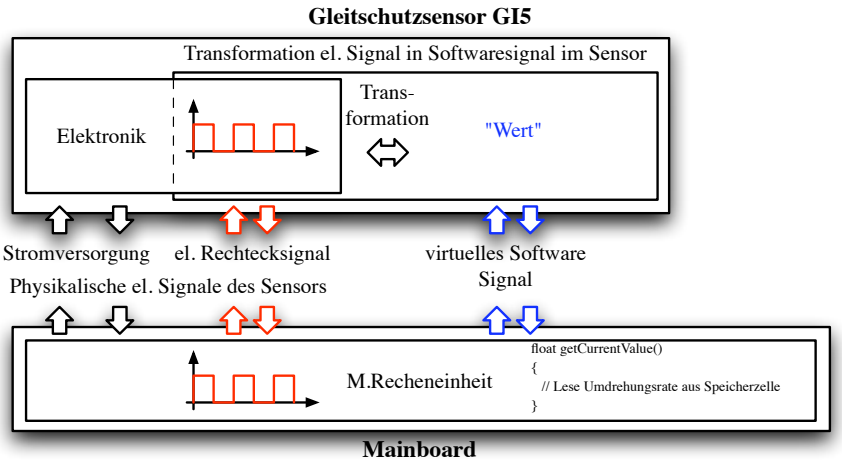


Abbildung 4.23.: Integration einer virtuellen Softwarefunktion in den Gleitschutzsensor GI5.

Durch diese Art der Modellierung ist es möglich sowohl das elektrische/elektronische Signal als auch das Softwaresignal exakt zu beschreiben und auf Kompatibilität zu untersuchen. Die Abbildung 4.23 zeigt das schematische Modell des Gleitschutzsensors GI5 mit einer virtuellen Softwarefunktion „Wert“. Im linken Teil der Darstellung ist der physikalische Anschluss (3-Pol: Stromversorgung und Rechtecksignal) dargestellt, über den die elektrischen/elektronischen Signale vom und zum Sensor übertragen werden. Auf der rechten Seite der Abbildung ist die virtuelle Softwarefunktion „Wert“ abgebildet. Sie wurde zusätzlich in das Modell des Sensors integriert, so dass das angeschlossene Mainboard diesen Wert mit Hilfe der Softwarefunktion `Sensor_Komm::getCurrentValue()` abfragen kann, ohne selbst die Wandlung des elektrischen Signals in ein Softwaresignal vornehmen zu müssen.

Im Stillstand, also bei $v = 0$, liefert der Sensor einen Wert von $+10$ und bei voller Fahrt, also $v = 400\mathrm{Km/h}$, den Wert $+10000$.

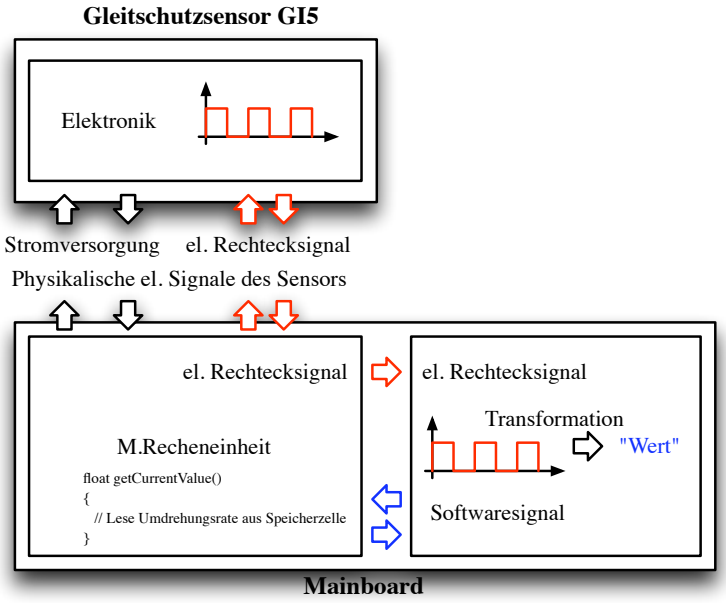


Abbildung 4.24.: Modellierung Gleitschutzsensor GI5 ohne eigene Softwarefunktion.

In (U)CML ist es auch möglich, den Sensor ohne eine zusätzliche virtuelle Softwarefunktion zu modellieren (Abb.: 4.24). Dann muss, im Gegensatz zur vorherigen Modellierungsmethode, die Transformation des elektrischen Rechtecksignals des Sensors in ein Softwaresignal durch eine zusätzliche Komponente *Transformation* im Mainboard durchgeführt werden. Die Transformationskomponente bekommt das elektrische/elektronische Rechtecksignal des Sensors als Eingabe und wandelt dieses in ein Softwaresignal um. Das Softwaresignal kann anschließend von der Komponente *M.Recheneinheit* des Mainboards verarbeitet werden.

Die zweite Modellierungsmethode entspricht genauer dem realen Sensoranschluss an das Mainboard. Allerdings muss bei dieser Modellierungsart eine zusätzliche Transformationskomponente in das System integriert werden, da nur so sichergestellt werden kann, dass sowohl das elektrische Signal des Sensors als auch die Transformation des Signals auf Kompatibilität hin untersucht werden kann.

Identifikation der Verbindungen zwischen den Komponenten des Mainboards sowie dem Mainboard und dem Gleitschutzsensor GI5
Bevor mit der eigentlichen Modellierung des vereinfachten Gleitschutzsystems begonnen werden kann, müssen die noch fehlenden Verbindungen zwischen dem Mainboard, dem Gleitschutzsensor und innerhalb des Mainboards identifiziert und beschrieben werden. In der Abbildung 4.25 sind die unterschiedlichen elektrischen/elektronischen- und mechanischen Verbindungen innerhalb des Mainboard, sowie dem Mainboard und dessen Umwelt (Rack und Einbauort) dargestellt. Alle aufgeführten Verbindungen wurden aus den unterschiedlichen Stromlaufplänen bzw. technischen Zeichnungen der entsprechenden Baugruppen entnommen.

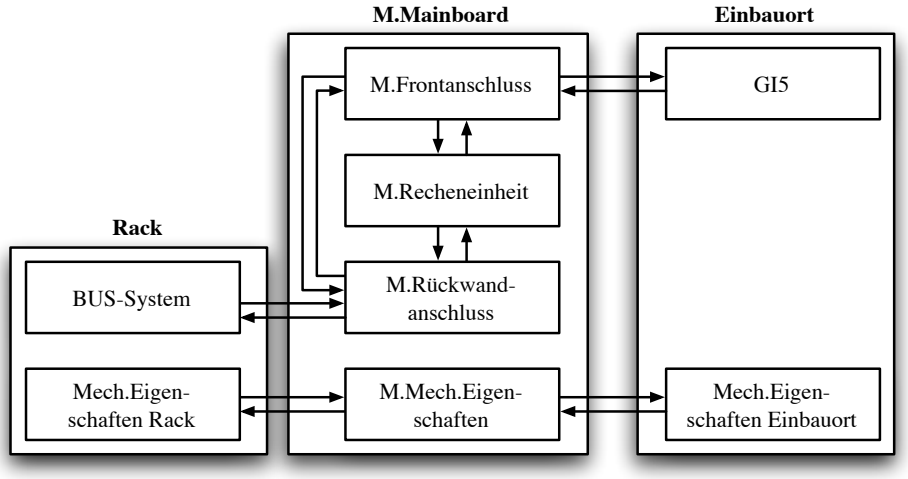


Abbildung 4.25.: Schematischer Verbindungsplan der Baugruppen des Gleitschutzsystems.

Die Abbildung 4.25 zeigt die Verbindungen zwischen den einzelnen Elementen des Gleitschutzsystems. Die Verbindungen lassen sich auch mit Hilfe der aus dem Systems Engineering bekannten Element-Element-Matrix bzw. der (U)CML-Matrixdarstellung darstellen. Die Tabelle 4.10 zeigt die Element-Element-Matrix des vereinfachten Gleitschutzsystems aus der Abbildung 4.25 für die elektrischen/elektronischen sowie die mechanischen Verbindungen.

Element-Element-Matrix	BUS-System	Mech.Eigenschaften Rack	M.Frontanschluss	M.Recheneinheit	M.Rückwandanschluss	M.Mech.Eigenschaften	GI5	Mech.Eigenschaften Einbauort
BUS-System	**0**	0	0	0	1	0	0	0
Mech.Eigenschaften Rack	0	**0**	0	0	0	1	0	0
M.Frontanschluss	0	0	**0**	0	1	0	1	0
M.Recheneinheit	0	0	1	**0**	1	0	0	0
M.Rückwandanschluss	1	0	1	0	**0**	0	0	0
M.Mech.Eigenschaften	0	1	0	0	0	**0**	0	1
GI5	0	0	1	0	0	0	**0**	0
Mech.Eigenschaften Einbauort	0	0	1	0	0	0	0	**0**

Tabelle 4.10.: Element-Element-Matrix bzw. (U)CML-Matrixdarstellung der elektrischen/elektronischen und mechanischen Verbindungen des Gleitschutzsystems.

Weder in der Abbildung 4.25, noch in der Tabelle 4.10 sind die Softwareverbindungen aufgelistet. Das liegt daran, dass es im Allgemeinen keine „Verbindungspläne“ für Softwaresysteme gibt. In UML/UML2 wird meistens nur die Klassenstruktur, nicht jedoch die Sequenz der Methodenaufrufe modelliert. Aus diesem Grund müssen die für das (U)CML-Modell notwendigen Verbindungen zwischen den einzelnen C++ Methoden aus dem Quellcode reengineered werden, um sie in das Modell einzeichnen zu können.

Die Tabelle 4.11 zeigt die aus dem Quellcode des Gleitschutzsystems extrahierten Softwareverbindungen zwischen den einzelnen Baugruppen des Systems. Zusätzlich zu den realen Verbindungen sind in der Tabelle auch die virtuellen Verbindungen zwischen den Elementen des Systems eingetragen.

Element-Element-Matrix	BUS-System	Mech.Eigenschaften Rack	M.Frontanschluss	M.Recheneinheit	M.Rückwandanschluss	M.Mech.Eigenschaften	GI5	Mech.Eigenschaften Einbauort
BUS-System	**0**	0	0	0	1	0	0	0
Mech.Eigenschaften Rack	0	**0**	0	0	0	0	0	0
M.Frontanschluss	0	0	**0**	0	1	0	1	0
M.Recheneinheit	0	0	1	**0**	1	0	0	0
M.Rückwandanschluss	1	0	1	0	**0**	0	0	0
M.Mech.Eigenschaften	0	0	0	0	0	**0**	0	0
GI5	0	0	1	0	0	0	**0**	0
Mech.Eigenschaften Einbauort	0	0	0	0	0	0	0	**0**

Tabelle 4.11.: Element-Element-Matrix der Softwareverbindungen des Gleitschutzsystems.

Nachdem die Verbindungen des Systems identifiziert wurden, folgt im nächsten Schritt die Beschreibung der Verbindungen. Begonnen wird mit der Beschreibung der Verbindungen innerhalb der Baugruppe *Mainboard*. Daran anschließend folgt die Beschreibung der Baugruppe *Sensor*.

– **Baugruppe Mainboard intern**
Das Mainboard der Fallstudie Gleitschutzsystems besteht aus vier unterschiedlichen Komponenten *M.Frontanschluss, M.Recheneinheit, M.Rückwandanschluss* sowie der Komponente *M.Mech.Eigenschaften*. Die ersten drei Komponenten *M.Frontanschluss, M.Recheneinheit* und *M.Rückwandanschluss* sind dabei untereinander verbunden. Die Komponente *M.Mech.Eigenschaften* ist nur mit der Komponente *Mech.Eigenschaften* des Racks sowie der Komponente *Mech.Einbauort* des Einbauorts verbunden (vgl. Abb.: 4.25 bzw. Tab.: 4.10 und 4.11). Die aufgeführten Verbindungen lassen sich in drei unterschiedliche Kategorien unterteilen:

* *Elektrotechnik*
Zwischen den einzelnen Komponenten des Mainboards existieren verschiedene elektrische/elektronische Verbindungen. Die in der Tabelle 4.10 enthaltenen elektrischen/elektronischen Verbindungen werden in der Abbildung 4.26 weiter verfeinert und analysiert.

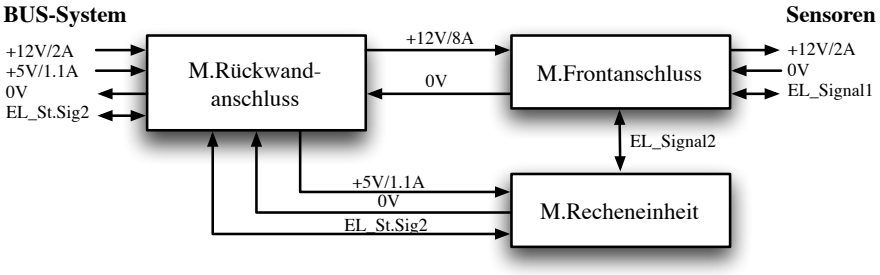


Abbildung 4.26.: Elektrische/elektronische Verbindungen des Mainboards.

Die Abbildung stellt eine Sammlung aller bis jetzt vorhandenen elektrischen/elektronischen Eigenschaften der internen Komponenten der Baugruppe *Mainboard* dar. Außerdem sind die Verbindungen zwischen den einzelnen Komponenten eingezeichnet. Alle in der Zeichnung enthaltenen elektrischen/elektronischen Verbindungen werden anschließend bei der Modellierung in (U)CML übernommen. Dort werden zusätzlich die kompatibilitätsrelevanten Eigenschaften der Anschlüsse der Komponenten in das Modell eingetragen.

* *Mechanik*
In der Abbildung 4.25 bzw. der Tabelle 4.10 sind außer den elektrischen/elektronischen Verbindungen zwischen den Mainboardinternen Komponenten auch die mechanischen Verbindungen eingezeichnet. Für die mechanischen Verbindungen wird in (U)CML keine „Trägerverbindung“, wie z.B. bei Softwaresignalen, benötigt. Im Beispielsystem

sind die mechanischen Eigenschaften bzw. Verbindungen nur für die Auslegung notwendig, so dass sie im späteren realen System nicht explizit enthalten sind. Aus diesem Grund existieren für die mechanischen Verbindungen auch keine Trägerverbindungen zwischen den beteiligten Komponenten.

* *Software*
  Die Modellierung der Softwareverbindungen erfolgt nach der Analyse der einzelnen Quellcodes des Systems. In der Tabelle 4.11 auf der vorherigen Seite sind alle Softwareverbindungen zwischen den einzelnen Komponenten des Mainboards aufgeführt. Diese Verbindungen werden in der Abbildung 4.27 weiter verfeinert.

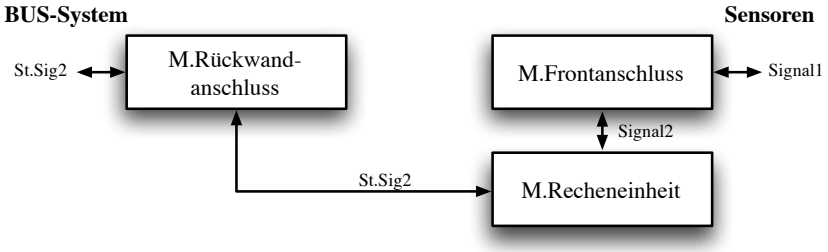


Abbildung 4.27.: Softwareverbindungen des Mainboards.

Die Softwareverbindungen zwischen den internen Komponenten des Pakets *Mainboard* werden über die entsprechenden elektrischen/elektronischen Trägersignale übertragen, d.h. das Mainboard enthält keine virtuelle Softwareverbindung, die nur auf Modellebene existiert. Die Verbindung *Signal1* stellt hier eine Ausnahme dar. Das Softwaresignal bzw. die Softwareverbindung *Signal1* ist innerhalb des Mainboards *keine* virtuelle Verbindung, da ein entsprechendes elektrisches/elektronisches Trägersignal existiert (*EL_Signal1*). Das Softwaresignal *Signal1* wird jedoch von einer virtuellen Softwarefunktion innerhalb des Sensors GI5 erzeugt und an das Mainboard übertragen.

– **Verbindung Sensor – Mainboard**
  Der Gleitschutzsensor GI5 ist mit dem Frontanschluss des Mainboards über eine dreiadrige flexible Leitung verbunden. In der Abbildung 4.28 ist auf der linken Seite der schematische Stromlaufplan zwischen dem Frontanschluss des Mainboards und dem Gleitschutzsensor GI5 dargestellt.

  Wie bereits mehrfach erwähnt, enthält der Sensor GI5 eine virtuelle Softwarefunktion, die jedoch im realen System nicht über die Verbindung zwischen dem Mainboard und dem Sensor übertragen wird. Aus diesem Grund ist auch keine eigene Verbindungsleitung dafür vorgesehen.

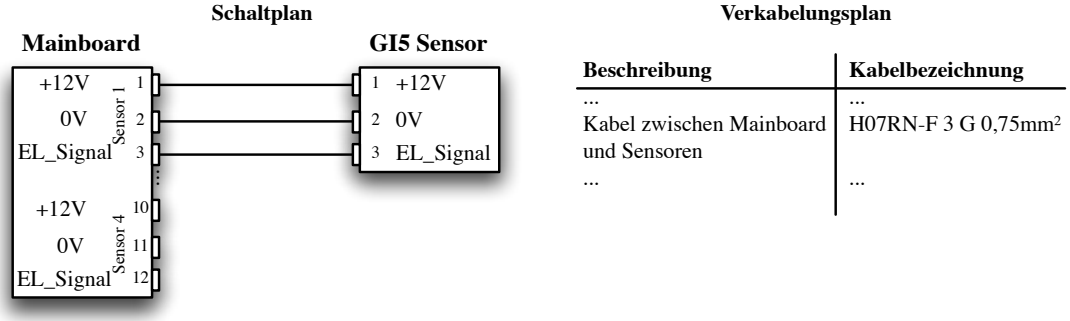


Abbildung 4.28.: Schalt- und Verkabelungsplan: Anschluss GI5 Sensor an das Mainboard.

In der nachfolgenden Aufzählung wird die Verbindung zwischen dem Mainboard (genauer: dem Frontanschluss des Mainboards) und dem GI5 Sensor detailliert beschrieben, so dass sie im nächsten Schritt, der Modellierung, in (U)CML umgesetzt werden kann.

* *Elektrotechnik*
  Im *Schaltplan* (Abb. 4.28) ist die elektrische Verbindung des Mainboards mit dem Sensor GI5 abgebildet (links). Die Verbindung zwischen den beiden Komponenten wird durch drei einzelne Leitungen hergestellt, die in einem gemeinsamen Kabel zusammengefasst sind. Aus dem Schaltplan ist nicht ersichtlich, welche Kabelart für die Verbindung benutzt wird. Diese Information muss aus dem *Verkabelungsplan* (Abb.: 4.28 rechts[290]) entnommen werden.
* *Mechanik*
  Der Gleitschutzsensor ist mit dem Frontstecker des Mainboards elektrisch verbunden (s.o.). Zusätzlich zur elektrischen Verbindung existiert zwischen dem Mainboard und dem Gleitschutzsensor eine mechanische Verbindung (die Verbindungsleitung). Über diese mechanische Verbindung wird z.B. die maximale Leiterlänge bzw. der verwendete Steckertyp ausgetauscht. Beide Verbindungen sind nur auf Modellebene vorhanden. Im realen Gleitschutzsystem

[290] Kabelbezeichnungen siehe [Wik07x] sowie [DIH89, 291].

sind diese mechanischen Verbindungen nicht vorhanden. Für die Bestimmung der Kompatibilität sind beide Informationen jedoch sehr wichtig, da nur so sichergestellt werden kann, dass die Anforderungen an z.B. die Verbindungsleitung zwischen dem Mainboard und dem Sensor von beiden eingehalten werden.

* *Software*
  Zwischen dem Sensor und dem Mainboard gibt es *keine* reale Softwareverbindung. Es existiert lediglich eine virtuelle Softwareverbindung, die nur auf Modellebene sicherstellt, dass die Kompatibilität zwischen dem Sensor und dem Mainboard getestet werden kann.

Nachdem nun alle Elemente, die dazugehörigen kompatibilitätsrelevanten Eigenschaften sowie die Verbindungen zwischen den Elementen des Gleitschutzsystems beschrieben wurden, folgt im nächsten Abschnitt die Modellierung des Gleitschutzsystems in (U)CML.

4.1.1.1.2. Modellierung des Gleitschutzsystems in (U)CML

Im ersten Schritt des (U)CML-Modellbildungsprozesses wurden sämtliche vorhandenen Daten und Informationen des Systems Gleitschutz untersucht sowie die kompatibilitätsrelevanten Eigenschaften identifiziert und aufbereitet. Nach Abschluss der Aufbereitungsphase folgt nun die prototypische Modellierung des Gleitschutzsystems mit Hilfe der Kompatibilitätsmodellierungssprache (U)CML.

In den nachfolgenden Unterabschnitten dieses Kapitels werden sämtliche Komponenten des Gleitschutzsystems aus den gesammelten Daten in (U)CML modelliert (qualitative Modellierung). Dabei werden, wie im letzten Kapitel, lediglich das Mainboard und der Gleitschutzsensor GI5 vollständig modelliert. Alle anderen Pakete und Komponenten des Gleitschutzsystems können später auf die gleiche Weise modelliert werden.

Vollständige Modellierung des Mainboards und Gleitschutzsensors GI5

In diesem Abschnitt werden die beiden Baugruppen *Mainboard* und *Gleitschutzsensor GI5* des ESRA-Gleitschutzsystems qualitativ, d.h. ohne Werte, in (U)CML modelliert. Dabei wird jeder Schritt der Modellierung einzeln aufgeführt und erläutert. Die für die Modellierung notwendigen kompatibilitätsrelevanten Daten wurden im letzten Abschnitt identifiziert und den einzelnen Baugruppen des Gleitschutzsystems zugeordnet. Aufbauend auf diesen Daten wird nun das Modell jedes Pakets bzw. jeder Komponente des Systems schrittweise aufgebaut:

1. Modellierung des *Strukturmodells* – Pakete und Komponenten – der Baugruppe.
2. Modellierung der *Paket- und Komponentenanschlüsse*.
3. Modellierung der *Verbindungen innerhalb jedes Pakets*.

Diese drei Modellierungsschritte werden nun für die beiden Baugruppen *Mainboard* und *Sensor* einzeln ausgeführt. Begonnen wird wieder mit der Modellierung der Baugruppe Mainboard:

- **Baugruppe Mainboard**
  Die Modellierung des Mainboads in (U)CML unterteilt sich in drei Schritte. Im ersten Schritt wird die Struktur der Baugruppe in (U)CML modelliert. Daran anschließend folgt die Modellierung der Paketschnittstellen sowie der Stecker und Buchsen der Komponenten. Im letzten Schritt der qualitativen Modellierung folgt die Modellierung der internen Verbindungen zwischen den einzelnen Steckern und Buchsen der Komponenten sowie die Verbindung zu den Paketschnittstellen.

  Die drei Modellierungsschritte sind in der nachfolgenden Aufzählung aufgeführt:

  1. **Strukturmodell**
     Die Baugruppe Mainboard besteht aus den vier atomaren Einheiten – *M.Frontanschluss, M.Rückwandanschluss, M.Mech.Eigenschaften* und *M.Recheneinheit*. Diese Baugruppen werden in (U)CML als Komponenten modelliert, die in dem Paket *Mainboard* enthalten sind. In der Abbildung 4.29 ist das Paket *Mainboard* mit den darin enthaltenen Komponenten dargestellt.

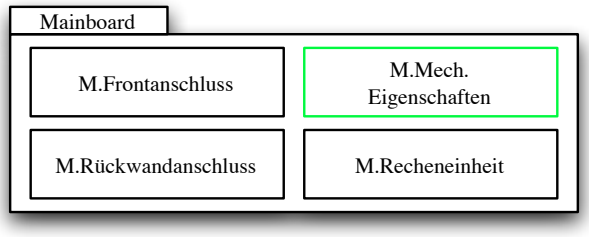


Abbildung 4.29.: Statisches (U)CML-Modell des Pakets *Mainboard*.

  2. **Paket- und Komponentenanschlüsse**
     Nachdem im letzten Schritt die grundlegende Struktur des Pakets *Mainbaord* modelliert wurde, folgt nun die Modellierung der Schnittstellen des Mainboards nach außen sowie der Stecker und Buchsen der im Paket enthaltenen Komponenten.

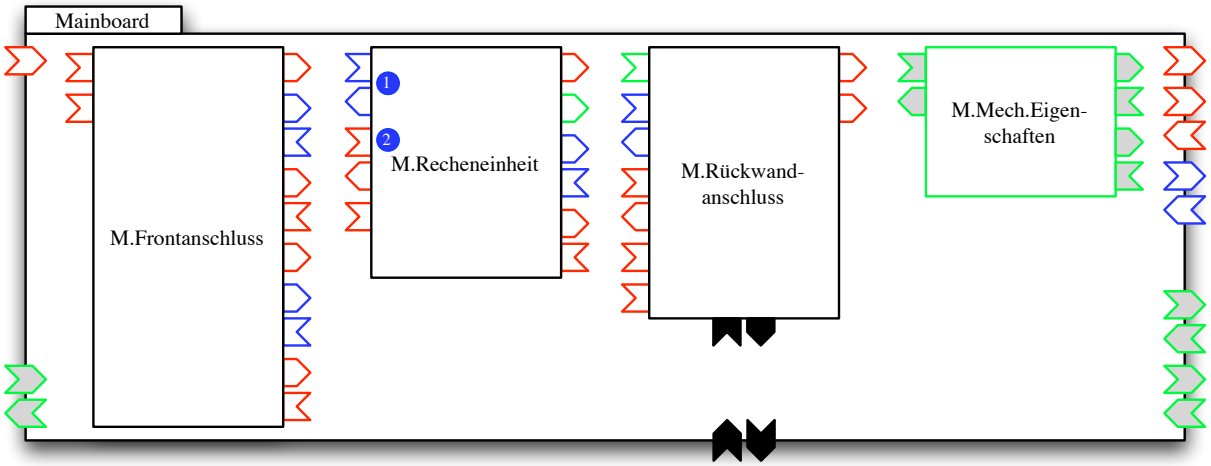


Abbildung 4.30.: Verfeinerung des Pakets *Mainboard*.

In der Abbildung 4.30 ist das Paket *Mainboard* mit allen Paketschnittstellen nach außen sowie den im Paket enthaltenen Komponenten mit allen Steckern und Buchsen dargestellt. Dabei sind alle Anschlüsse bereits ihrem Verwendungszweck nach – Elektronik, Mechanik oder Software – entsprechend eingefärbt.

Die Einfärbung der Paketschnittstellen und der Stecker und Buchsen der Komponenten ergibt sich aus der Identifikation der Domänen (Eigenschaften), wie sie im vorherigen Kapitel identifiziert wurden. Beispielsweise wird der Kommunikationsstecker (1) blau eingefärbt, weil über ihn das Softwaresignal *Signal2* übertragen werden soll (vgl. IO3 aus Tabelle 4.1 auf Seite 270). Der Kommunikationsstecker (2) wiederum ist rot eingefärbt, da er zur Übertragung des elektrischen/elektronischen Signals (IO4) verwendet wird.

3. **Verbindungen innerhalb des Pakets**
   Nachdem das Strukturmodell mit allen Anschlüssen (Steckern, Buchsen und Paketschnittstellen) in (U)CML modelliert wurde, folgt nun die Modellierung der internen Verbindungen (Flusspfeile) innerhalb des Pakets *Mainboard* – also der Verbindungen zwischen den Steckern und Buchsen der Komponenten sowie den Paketschnittstellen des Pakets *Mainboard*. Alle dafür notwendigen Daten und Informationen wurden bereits im Abschnitt „Identifikation der Verbindungen“ ab Seite 281 für die drei Domänen Elektrotechnik, Mechanik und Software getrennt aufbereitet.

   Die Abbildung 4.31 auf der nächsten Seite zeigt die Baugruppe Mainboard mit allen Komponenten und Verbindungen innerhalb des Pakets. Die Verbindung zwischen den einzelnen Komponenten und den Paketschnittstellen erfolgt aufgrund der Auswertung der elektrischen Schaltpläne (vgl. Abb. 4.26), der mechanischen Eigenschaften sowie der Analyse der Software (vgl. Abb.: 4.27).

   Exemplarisch für alle Verbindungen innerhalb des Pakets *Mainboard* werden an dieser Stelle zwei Verbindungen 1 und 2 herausgegriffen:

   – *Softwareverbindung* (1):
     In der Abbildung 4.27 auf Seite 283 wurde der „Verbindungsplan“ der unterschiedlichen Softwaremethoden innerhalb des Mainboards dargestellt. Die bidirektionale Softwareverbindung *Signal2* verbindet die Komponente *M.Recheneinheit* mit der Komponente *M.Frontanschluss*. Als bidirektionale Verbindung wird Verbindung 1 in (U)CML durch einen zusammengesetzten Kommunikationsflusspfeil[291] modelliert.
   – *Elektrische/elektronische-Verbindung* (2):
     Mit Hilfe der elektrischen/elektronischen Verbindung *EL_Signal2* (vgl. Abbildung: 4.26 auf Seite 282) wird das Rechtecksignal zur Ansteuerung des Gleitschutzsensors von der Komponente *M.Recheneinheit* zur Komponente *M.Frontanschluss* übertragen. Die elektrische/elektronische Verbindung wird dabei über die (U)CML-Kommunikationsverbindung (2) als zusammengesetzter Kommunikationsflusspfeil modelliert.

   Alle anderen Verbindungen innerhalb des Pakets *Mainboard* können auf die gleiche Art und Weise modelliert und in Abhängigkeit von ihrer Verbindungsart in (U)CML dargestellt werden.

- **Baugruppe Sensor**
  Als nächstes folgt die ausführliche Transformation der gesammelten Daten und Informationen der Baugruppe Sensor in ein statisches qualitatives (U)CML-Modell. Dabei wird zuerst das Strukturmodell mit Paket- und Komponentenanschlüssen in (U)CML modelliert. Abschließend folgt die Modellierung der Verbindungen innerhalb des Pakets.

  1. **Strukturmodell**
     Die ESRA-Baugruppe Sensor besteht aus drei unterschiedlichen Sub-Baugruppen – *El.Signale, Sw.Signale* und *Mech.EigenschaftenSensor*. Der Sensor wird als (U)CML-Paket, und die drei Sub-Baugruppen als (U)CML Komponenten modelliert und dargestellt. In der Abbildung 4.32 auf Seite 287 ist das statische Strukturmodell des Sensors abgebildet.

[291] Siehe hierzu: Kapitel „3.6.2.5.5 *Zusammengesetzter (U)CML-Pfeil*“ ab Seite 198.

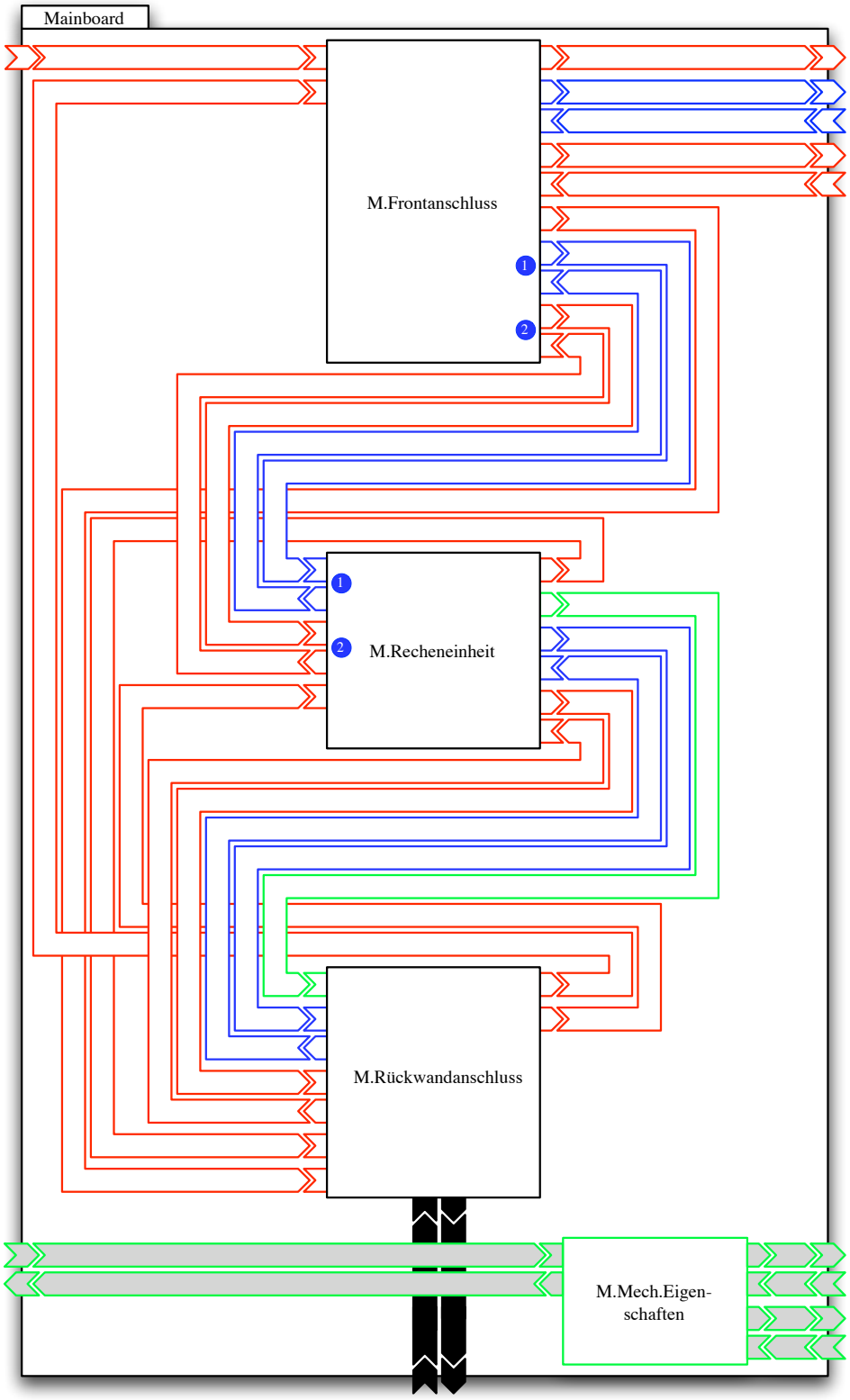


Abbildung 4.31.: Verfeinerung des Pakets *Mainboard* um Verbindungen zwischen den Komponenten des Pakets bzw. den Komponenten und den Paketschnittstellen.

Die drei Komponenten des Sensors sind dabei einer Domäne fest zugeordnet. Zum Beispiel sind an der Komponente *Mech.Eigenschaften* nur mechanische Stecker und Buchsen angebracht. Aus diesem Grund wird die Komponente als „mechanische Komponente“ mit einem grünen Rand dargestellt.

2. **Paket- und Komponentenanschlüsse**
   Die Abbildung 4.33 auf der nächsten Seite (links) zeigt die Verfeinerung des Pakets *Sensor* aus der vorangegangenen Abbildung. In dieser werden sowohl die Paketschnittstellen als auch die Stecker und Buchsen der Komponenten in Abhängigkeit von der jeweiligen Domäne modelliert.

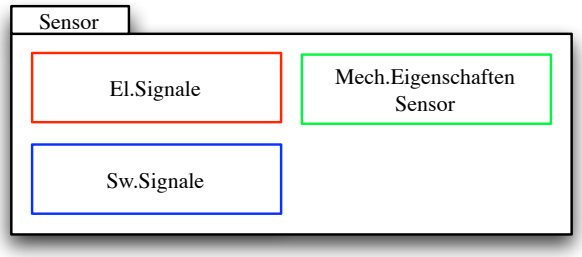


Abbildung 4.32.: Statisches (U)CML-Modell des Pakets *Sensor*.

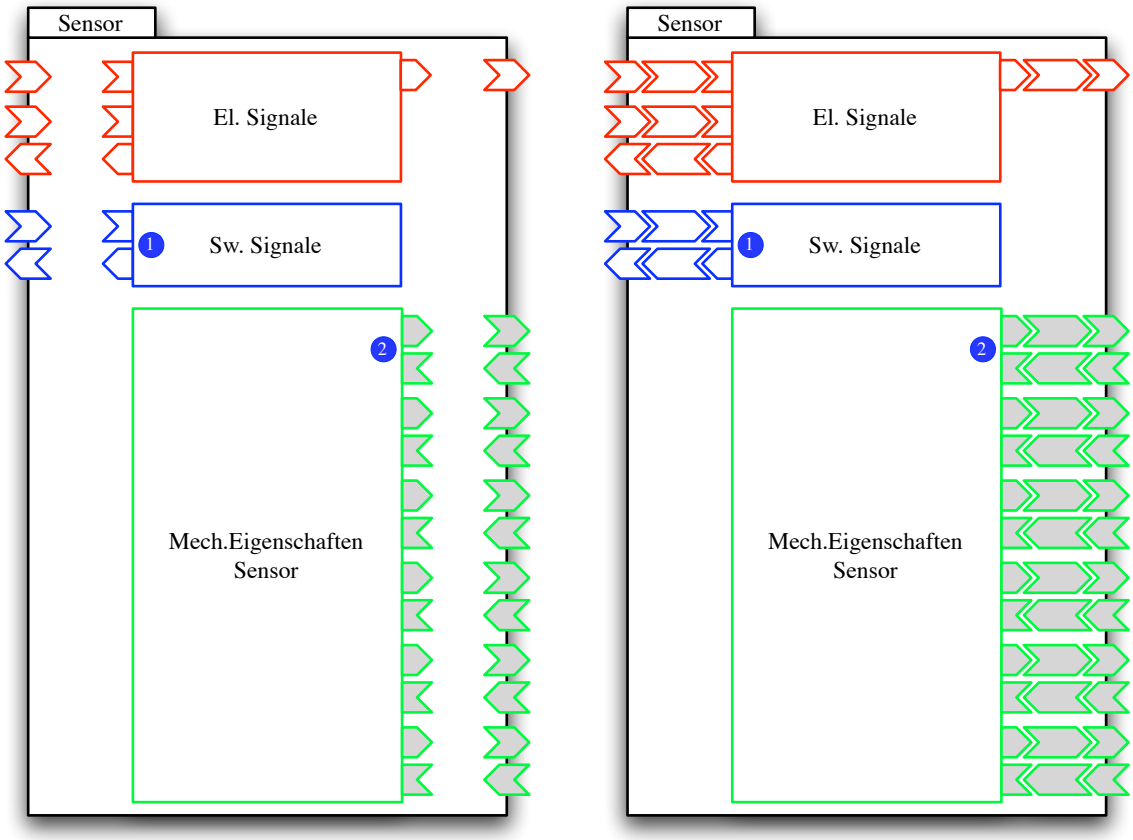


Abbildung 4.33.: Verfeinerung des (U)CML-Pakets *Sensor*. Links: Ohne Verbindungspfeile, rechts: mit Verbindungsflusspfeilen

Wenn alle Stecker und Buchsen an einer Komponente der gleichen Domäne angehören, wird die entsprechende Komponente in der Domänenfarbe eingefärbt. Beispielsweise sind an der Komponente *Mech.EigenschaftenSensor* nur mechanische Kommunikationsstecker angebracht, so dass diese Komponente in der Domänenfarbe grün (Mechanik) eingefärbt und dargestellt werden kann.

Stellvertretend für alle Anschlüsse innerhalb des Pakets *Sensor* sind in der nachfolgenden Aufzählung zwei Komponentenanschlüsse herausgegriffen:

- *Softwarekommunikationsstecker* (1):
  Das Softwaresignal *Signal1* ist ein virtuelles Signal, d.h. im realen System existiert es nicht; es ist nur auf Modellebene vorhanden. Das virtuelle Softwaresignal *Signal1* (IO1) wurde der Abbildung 4.17 auf Seite 277 bzw. der Tabelle 4.7 auf Seite 277 entnommen und als Softwarekommunikationsstecker in (U)CML modelliert.

- *Mechanische Kommunikationsstecker* (2):
  Der optionale Kommunikationsstecker (2) dient zur Übertragung der verwendeten *Steckergröße* des Sensors. Dabei wurde die Steckergröße als zusammengesetzter Datentyp (Tripel: X, Y, Z) realisiert. Für diesen mechanischen Anschluss gilt außerdem: Er ist ebenfalls nur auf Modellebene vorhanden und dient während der Modellierungsphase des Systems zur Überwachung der benötigten Steckerform für das Anschlusskabel.

  Außerdem ist der mechanische Kommunikationsstecker optional, so dass er im Modell nicht angeschlossen sein muss und das Modell trotzdem keinen Fehler enthält.

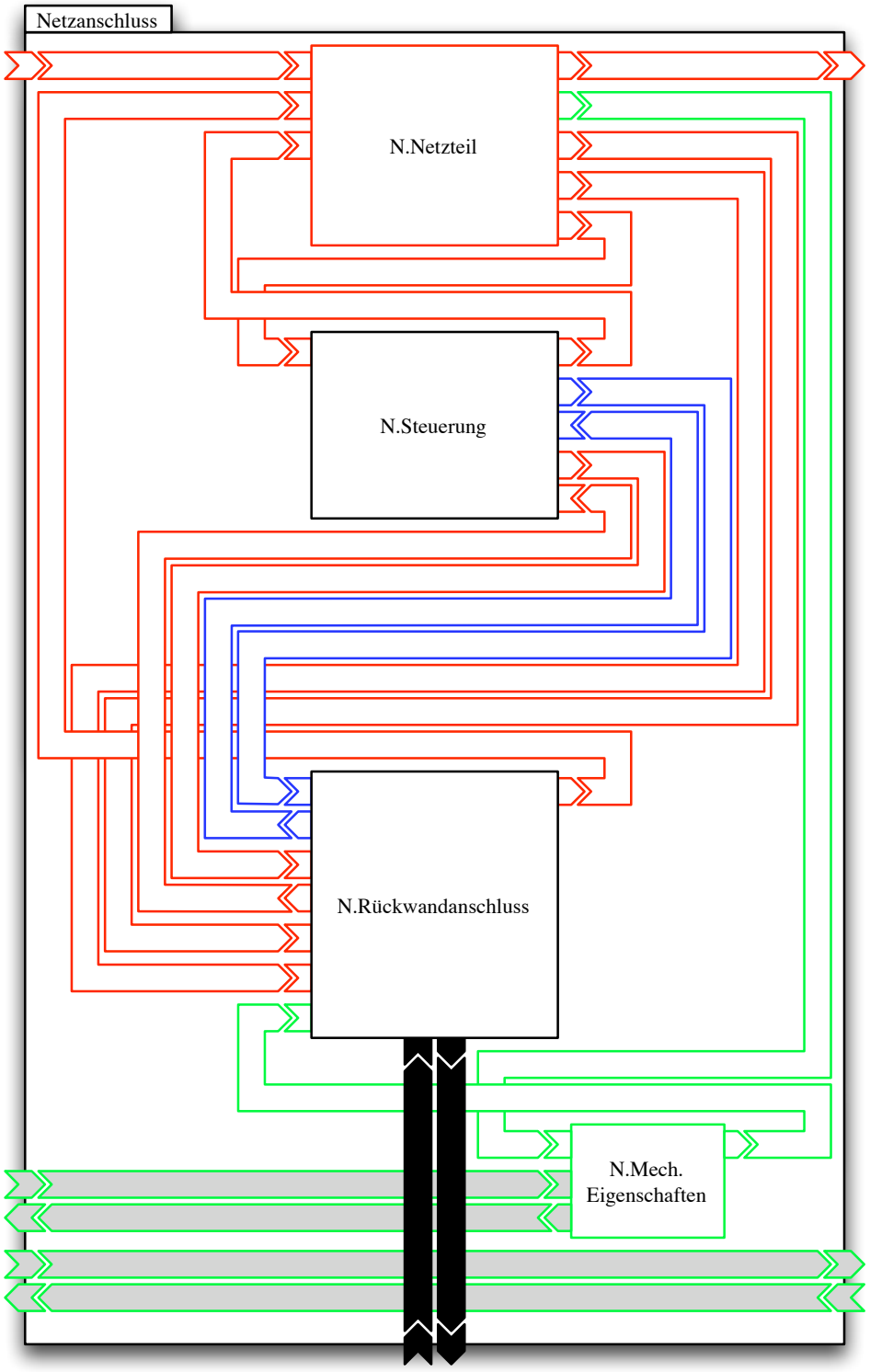


Abbildung 4.34.: (U)CML-Modell des Netzanschlusses.

3. **Verbindungen innerhalb der Pakete**
   Nachdem alle Paketschnittstellen sowie die Stecker und Buchsen der Komponenten modelliert worden sind, folgt die Verbindung der Stecker und Buchsen mit den Paketschnittstellen des Pakets *Sensor*. In der Abbildung 4.33 auf der vorherigen Seite (rechts) ist das Paket *Sensor* mit allen internen Verbindungen dargestellt.

   Auch in diesem Fall wurden wieder zwei Verbindungen exemplarisch für alle anderen herausgegriffen:

   – *Softwareverbindung* (1):
   Mit Hilfe der virtuellen Softwareverbindung *Signal1* wird die Softwarefunktion „Wert“ vom Sensor an die Sensorumwelt bzw. an das Mainboard übergeben. Dort wird der Wert von einer C++ Methode ausgewertet, wie bei der Identifikation des kompatibilitätsrelevanten Eigenschaften des Sensors bzw. des Mainboards gezeigt wurde.

– *Mechanische Verbindung* (2):
Die mechanische optionale Kommunikationsverbindung (2) wird zur mechanischen Kommunikation zwischen dem Sensor auf der einen Seite und dem Einbauort auf der anderen verwendet. Dabei ist die mechanische Kommunikationsverbindung nur auf Modellebene vorhanden, d.h. im realen System ist diese Verbindung nicht existent. Sie dient lediglich zur Kompatibilitätsbestimmung während der Modellierungsphase. Aus diesem Grund ist auch keine reale physikalische Verbindung vom Sensor zum Einbauort vorhanden.

Anmerkung:
Im Paket *Sensor* gibt es keine Verbindungen zwischen den einzelnen realen Baugruppen. Aus diesem Grund haben die (U)CML-Komponenten auch keine Verbindungen untereinander.

Nachdem nun sowohl das Mainboard als auch der Sensor qualitativ in (U)CML modelliert worden sind, folgt im nächsten Abschnitt dieses Kapitels die Modellierung der restlichen Baugruppen des Gleitschutzsystems.

Modellierung der restlichen Pakete und Komponenten des Gleitschutzsystems
In diesem Abschnitt werden die restlichen Baugruppen des ESRA-Gleitschutzsystems in (U)CML modelliert und im Flussdiagramm dargestellt. Dabei wird mit der Modellierung des Netzanschlusses begonnen. Anschließend folgt die Modellierung des Racks sowie des eigentlichen Gleitschutzsystems.

- **Netzanschluss**
Das Netzteil dient zur Stromversorgung des gesamten ESRA-Gleitschutzsystems. In der Abbildung 4.34 auf der vorherigen Seite ist das (U)CML-Modell des Pakets *Netzanschluss* mit allen enthaltenen Komponenten, den Anschlüssen sowie den Flusspfeilen dargestellt. Sämtliche Daten, die während der Identifikationsphase gesammelt worden sind, wurden in das statische qualitative (U)CML-Modell des Netzanschlusses integriert, wie z.B. die Belegung des Rückwandsteckers (*N.Rückwandanschluss*).

- **Rack**
Die 19-Zoll Baugruppe Rack, in der sämtliche 19-Zoll Einschübe des ESRA-Gleitschutzsystems enthalten sind, wird in (U)CML als Paket modelliert. Das Paket *Rack* enthält das BUS-System, an dem alle 19-Zoll Einschubsteckkarten des Gleitschutzsystems angeschlossen sind sowie die Komponente *R.Mech.Eigenschaften* mit deren Hilfe die mechanischen Eigenschaften des Racks modelliert und abgebildet werden. In der nachfolgenden Abbildung 4.35 ist das Paket *Rack* mit den drei Paketen *Netzanschluss, Mainboard* und *Restliche Boards* in Black-Box Darstellung abgebildet.

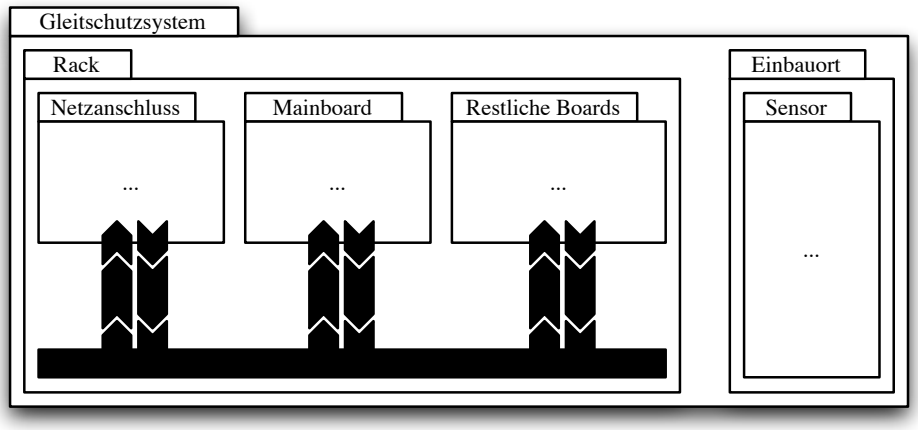


Abbildung 4.35.: Paketstruktur des Gleitschutzsystems.

Die Abbildung 4.36 auf der nächsten Seite zeigt das gesamte Gleitschutzsystem, in dem sich das Paket *Rack* befindet. Außerdem sind sämtliche Schnittstellen des Racks gegenüber dem Gleitschutzsystem modelliert. Die Paketschnittstellen des Racks sind notwendig damit das Rack die innen liegenden Pakete und Komponenten z.B. mit Strom oder Daten versorgen kann. Wenn ein Paket oder eine Komponente, die sich innerhalb des Pakets *Rack* befindet, Daten mit der Umwelt des Paktes *Rack* austauschen will, muss es dafür eine entsprechende Paketschnittstelle geben (z.B. Strom- und Datenversorgung des Sensors).

- **Gleitschutzsystem und Einbauort**
Der Einbauort des Sensors ist als Paket *Einbauort* innerhalb des Pakets *Gleitschutzsystem* modelliert (Abb.: 4.35). Innerhalb des Pakets *Einbauort* befindet sich das Paket *Sensor*, das im letzten Abschnitt vollständig in (U)CML modelliert wurde (vgl. Abb.: 4.33 auf Seite 287 rechts). In der Abbildung 4.36 auf der nächsten Seite ist das Paket *Sensor* mit allen Paketschnittstellen als Black-Box innerhalb des Pakets *Einbauort* dargestellt. Insbesondere ist der mechanische Teil der Verbindung zwischen dem Sensor und dem Einbauort modelliert, mit dessen Hilfe die mechanische Kompatibilität des Sensors mit dem Einbauort sowie die Kompatibilität der Verbindung zwischen dem Sensor und dem Mainboard sichergestellt wird.

  Das Paket *Gleitschutzsystem* repräsentiert das gesamte ESRA-Gleitschutzsystem. In ihm sind auf der obersten Ebene das Paket *Rack* sowie das Paket *Einbauort* enthalten. Des Weiteren stellt das Paket *Gleitschutzsystem* die Schnittstelle zwischen der Umwelt und dem System Gleitschutz zur Verfügung.

Gleitschutzsystem mit allen Paketen, Komponenten und Verbindungen in Black-Box Ansicht
Nachdem nun alle für das Gleitschutzsystem benötigten Pakete und Komponenten in (U)CML modelliert worden sind, folgt in diesem Modellierungsschritt, die Integration der einzelnen Pakete und Komponenten zu einem vollständigen Modell. Bevor jedoch die Integration abgeschlossen ist, müssen noch fehlende Verbindungen zwischen den einzelnen Paketschnittstellen modelliert werden. Im Abschnitt „Identifikation der Verbindungen zwischen den Komponente des Mainboards sowie dem Mainboard und dem Gleitschutzsensor GI5“ ab Seite 281 wurden die Verbindungen zwischen dem Mainboard und dem Sensor identifiziert und aufbereitet. Die Abbildung 4.36 zeigt das Gleitschutzsystem mit allen Verbindungen zwischen den einzelnen Paketen des Systems. Dabei werden die Pakete *Netzanschluss*, *Mainboard*, *Restliche Boards* sowie das Paket *Sensor* in Black-Box Darstellung angezeigt.

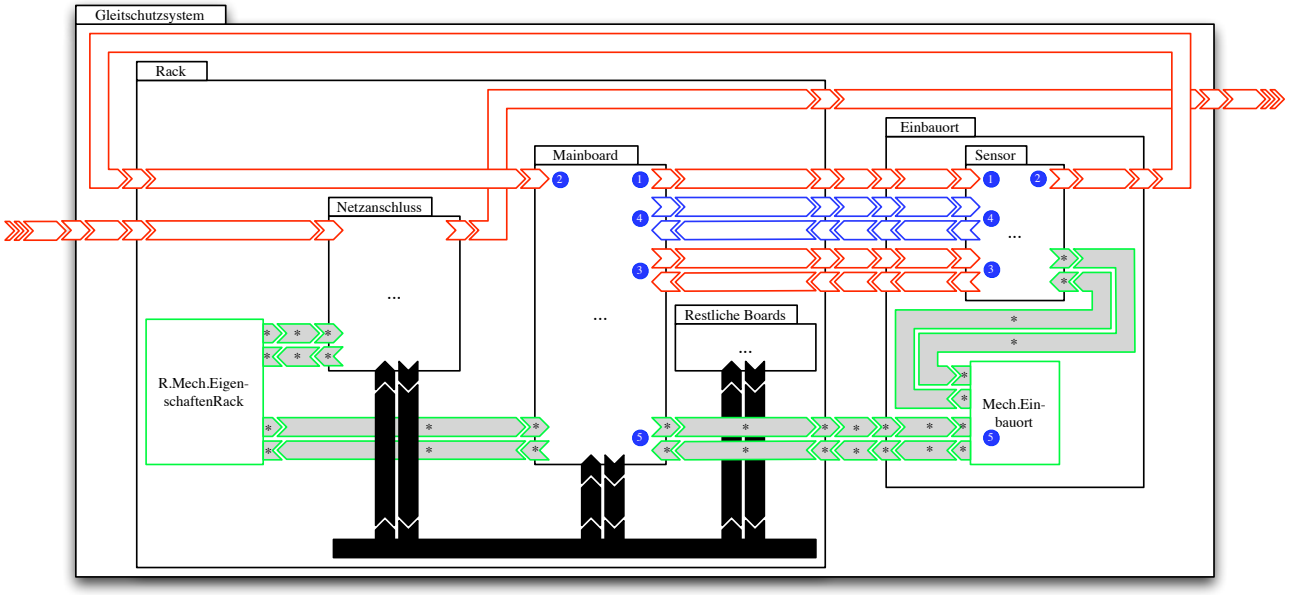


Abbildung 4.36.: Gleitschutzsystem mit allen Komponenten, Verbindungen und Subpaketen in Black-Box Ansicht.

Grundlage für die Modellierung der Flusspfeile innerhalb des Gleitschutzsystems ist der schematische elektrische Schaltplan 4.25 auf Seite 281 sowie die beiden Tabellen 4.10 und 4.11 auf Seite 282. Die Verbindungen zwischen dem Paket *Mainboard* und dem Paket *Sensor* werden nun ausführlich beleuchtet. Dabei wird die Modellierung wieder in die drei Domänen – Elektrotechnik, Mechanik und Software – unterteilt.

Im Abschnitt „Identifikation der Verbindungen“ ab Seite 281 wurden die verschiedenen kompatibilitätsrelevanten Verbindungen zwischen dem Mainboard und dem Gleitschutzsensor identifiziert und für die Modellierung vorbereitet.

- **Elektrotechnik**
  Die elektrischen/elektronischen Verbindungen zwischen dem Mainboard und dem Sensor unterteilen sich in zwei unterschiedliche Verbindungsarten (vgl. Abb.: 4.28 auf Seite 283): Zum einen in die Stromversorgung, über die der Sensor mit Strom bzw. Spannung versorgt wird und zum andern in die Verbindungen zur Übertragung der elektrischen Signale:
  - Stromversorgung
    Die Stromversorgung des Sensor setzt sich aus zwei Flusspfeilen zusammen: die Strom-/Spannungsversorgung $+12V, 2A$ (1) und die Rückleitung $0V$ (2). Beide Verbindungen zusammen bilden die Stromversorgung des Sensors.
  - Rechtecksignal
    Zwischen dem Sensor und dem Mainboard werden die Messwerte des Sensors mit Hilfe des elektrischen Rechtecksignals *EL_Signal*[292] zwischen dem Mainboard und dem Sensor ausgetauscht (3).
- **Mechanik**
  Der Sensor tauscht auf Modellebene Daten mit dem Mainboard aus (5). Diese Daten sind für die Bewertung der Kompatibilität der mechanischen Verbindung entscheidend. Trotzdem sind alle mechanischen Kommunikationsflusspfeile optional modelliert, so dass sie nicht zwingend angeschlossen werden müssen. Dadurch wird das Modell sehr flexibel und für verschiedene Sensortypen gleichermaßen verwendbar. Im hier modellierten Fall tauschen das Mainboard und der Sensor den *verwendeten Steckertyp* sowie die *maximale Leiterlänge* über den zusammengefassten mechanischen Kommunikationsflusspfeil aus.
- **Software**
  Zwischen dem Sensor und dem Mainboard existiert auf Modellebene eine virtuelle Softwarekommunikationsverbindung (4). Über diese Softwareverbindung wird der zulässige Umdrehungswertebereich des Polrads des Sensors an das Paket *Mainboard* übertragen. Der Softwarewert wird dabei vom Ausgang des Pakets *Sensor* bzw. vom Beschreibungsfeld des Kommunikationssteckers zur Verfügung gestellt und für die Übertragung aufbereitet[293].

[292] Siehe Tabelle 4.9 auf Seite 279.
[293] Vgl. Abschnitt „Software“ auf Seite 279.

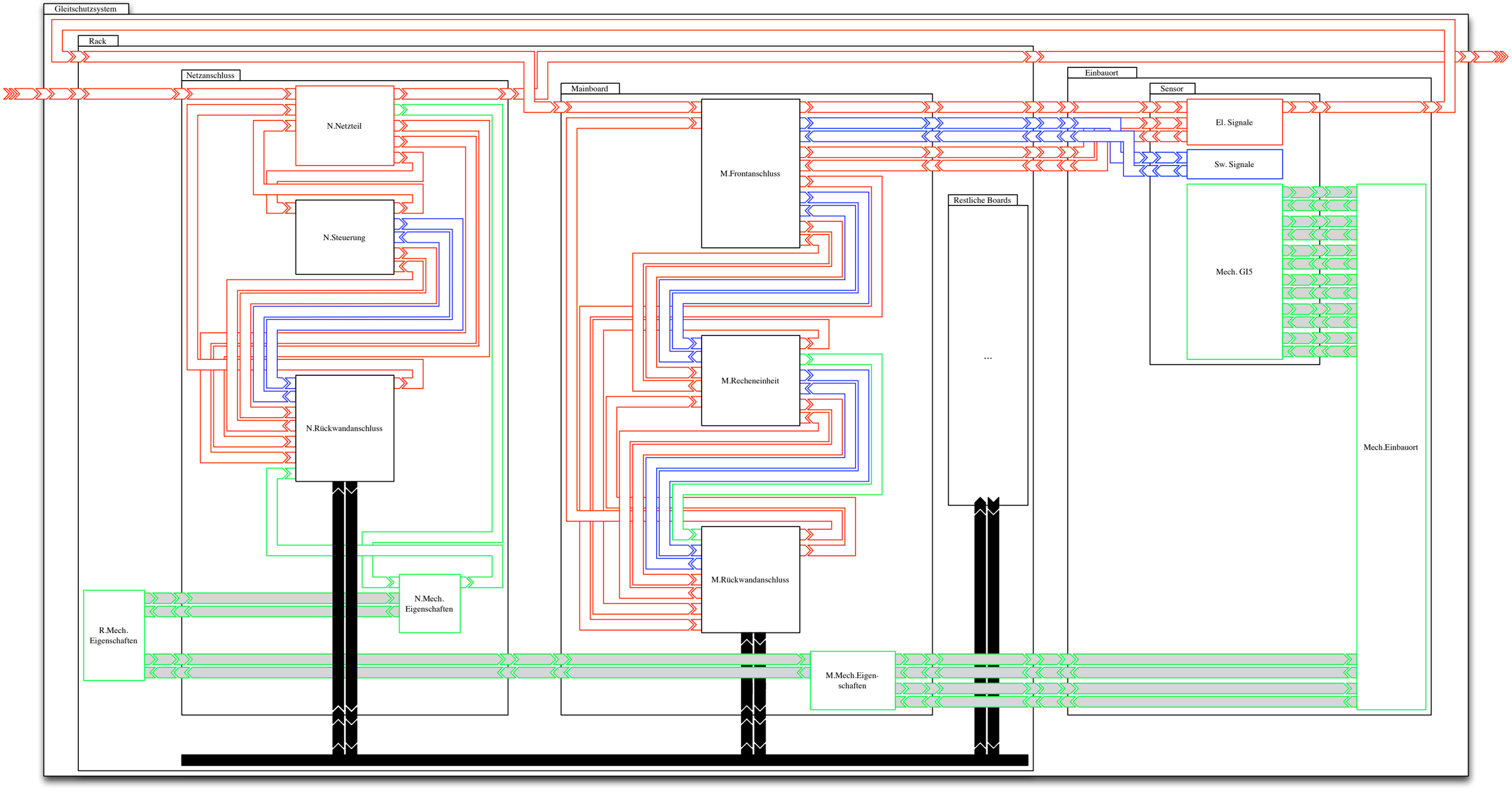


Abbildung 4.37.: Verfeinerung des Fallbeispiels Gleitschutzsystem aus Abbildung 4.36 auf der vorherigen Seite.

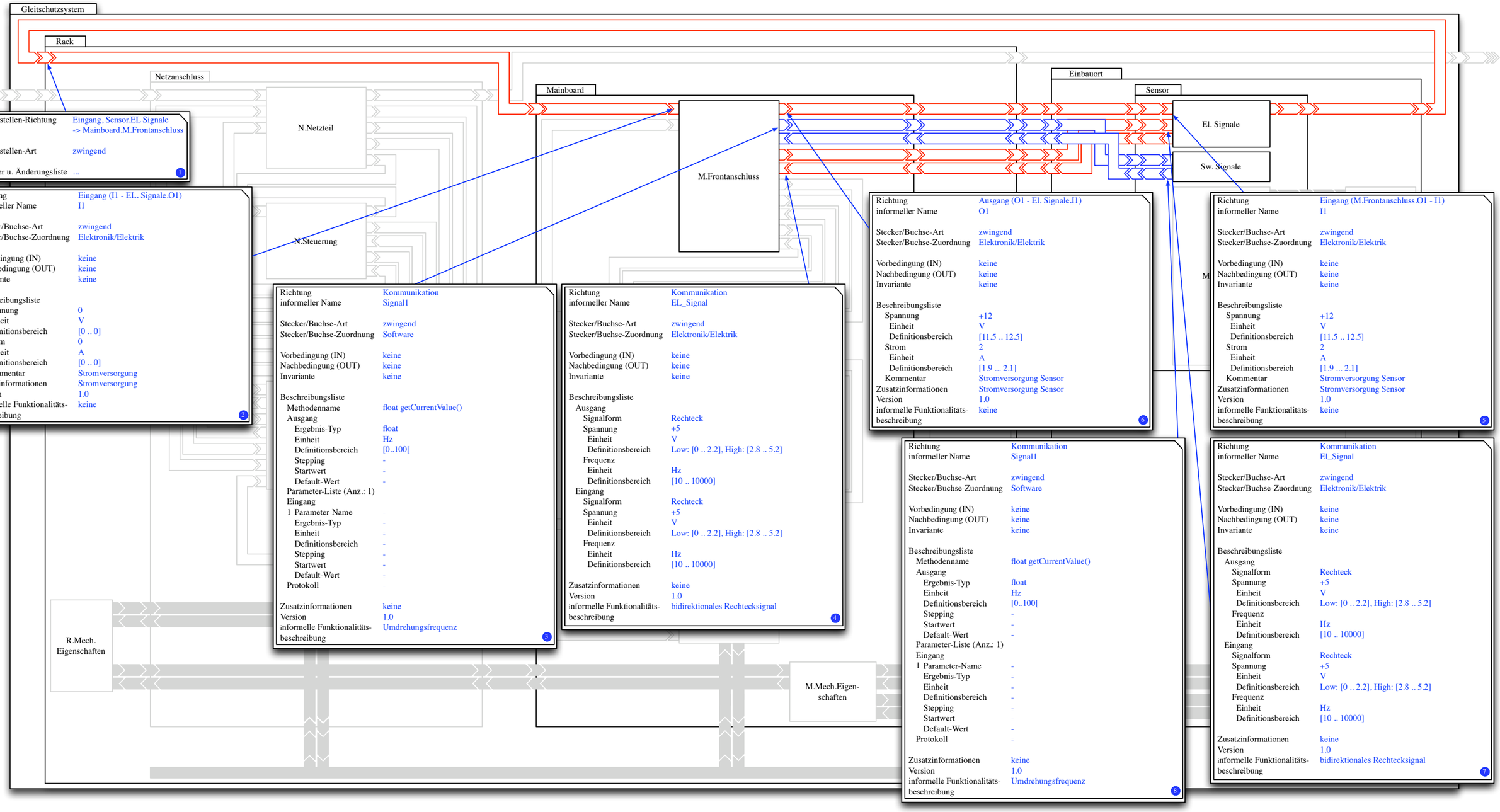


Abbildung 4.38.: Gleitschutzsystem Modell mit Beschreibungsfeldern.

Das vollständige qualitative (U)CML-Modell des Gleitschutzsystems in Flussansicht
Im Anschluss an die Modellierung der Verbindungen zwischen den einzelnen Baugruppen des Gleitschutzsystems folgt nun die Darstellung des vollständigen (U)CML-Modells der Fallstudie Gleitschutzsystem. In der Abbildung 4.37 auf Seite 291 ist das gesamte Gleitschutzsystem in Glass-Box Darstellung, mit allen Flusspfeilen, abgebildet (Ausnahme: das Paket *Restliche Boards*).

Im Modell sind sämtliche Pakete, Komponenten sowie alle notwendigen Verbindungen modelliert, die für die Kompatibilitätsprüfung notwendig sind. Aufbauend auf dem qualitativen (U)CML-Modell des Gleitschutzsystems folgt nun, im nächsten Abschnitt die Transformation des *qualitativen* (U)CML-Modells in ein *quantitatives Modell*, in dem alle identifizierten kompatibilitätsrelevanten Eigenschaften quantitativ in die Beschreibungsfelder der Pakete, Komponenten sowie der Stecker und Buchsen eingetragen werden.

4.1.1.1.3. Befüllung der Beschreibungsfelder mit den Eigenschaften des Systems

Im letzten Abschnitt wurde das vollständige qualitative (U)CML-Modell des Gleitschutzsystems modelliert und im Flussdiagramm dargestellt. Im nächsten Schritt des Modellbildungsprozesses folgt nun die Befüllung der Beschreibungsfelder der Pakete, Komponenten sowie der Stecker und Buchsen mit konkreten Werten. Die relevanten Eigenschaften wurden bereits im ersten Abschnitt während der Identifikation der kompatibilitätsrelevanten Eigenschaften quantifiziert. Diese Werte werden nun in die entsprechenden Beschreibungsfelder eingetragen. In der Abbildung 4.39 auf der nächsten Seite wurde exemplarisch für alle anderen Bestandteile des Systems die Komponente *M.Recheneinheit* des Mainboards herausgegriffen. Anhand der Komponente *M.Recheneinheit* wird die Befüllung der (U)CML-Beschreibungsfelder erläutert.

Die Abbildung 4.39 zeigt in der Mitte die Komponente *M.Recheneinheit*. Auf der linken Seite der Abbildung ist das Beschreibungsfeld für die Stromversorgung der Baugruppe *M.Recheneinheit* dargestellt. In diesem Beschreibungsfeld sind alle für die Beschreibung des Eingangs I1 – Stromversorgung – notwendigen kompatibilitätsrelevanten Daten eingetragen, die während der Identifikationsphase gesammelt worden sind. So wurden z.B. aus der Tabelle links oben die Strom- und Spannungswerte sowie die Schwankungsbreiten (Gültigkeitsintervall) in die entsprechenden Felder des Beschreibungsfelds eingetragen.

Das Beschreibungsfeld auf der rechten Seite der Abbildung beschreibt das elektrische Rechtecksignal des Sensors. Auch in diesem Fall wurden die entsprechenden Werte aus der Tabelle links oben entnommen. Das untere Beschreibungsfeld beschreibt einen Softwarekommunikationsstecker. Über diesen Stecker wird der Definitionsbereich des Sensors an die Komponente *M.Recheneinheit* übertragen. Auch hier sind, wie bei den anderen beiden Anschlüssen, wieder sämtliche kompatibilitätsrelevanten Eigenschaften der Software in das Beschreibungsfeld eingetragen worden. Als Quelle für das Beschreibungsfeld dient die Methodendeklaration rechts oben in der Abbildung.

Auf die gleiche Art und Weise werden nun alle Beschreibungsfelder des Modells mit konkreten Werten befüllt. Die Abbildung 4.38 auf der vorherigen Seite zeigt das vollständige (U)CML-Modell des Gleitschutzsystems mit acht Beschreibungsfeldern:

- **Beschreibungsfeld 1 – Paketschnittstelle Stromversorgung (Rack)**
  Paketschnittstelle des Pakets *Rack* zur Übertragung des elektrischen/elektronischen Signals 0V von der Komponente *El. Signal* zur Komponente *M.Recheneinheit*. Dieses Beschreibungsfeld wird beispielsweise durch das Werkzeug *(U)CML-ed* [294] automatisch befüllt.

- **Beschreibungsfeld 2 – Buchse Stromversorgung (M.Frontanschluss)**
  Beschreibung der elektrischen/elektronischen Eingangsbuchse *I1* der Komponente *M.Frontanschluss*. Alle Werte sind aus folgenden Quellen entnommen: Abbildung 4.10 sowie Tabelle 4.1 auf Seite 270.

- **Beschreibungsfeld 3 und 8 – Kommunikationsstecker/-buchse Softwaresignal**
  Über die Softwarekommunikationsverbindung zwischen der Komponente *M.Frontanschluss* (IO1) und der Komponente *Sw. Signal* (IO2) wird das virtuelle Softwaresignal *Signal1* übertragen. Alle Eigenschaften des Signals sind aus der Tabelle 4.1 sowie dem Abschnitt „Software“ ab 274 für das Mainboard und aus der Tabelle 4.7 bzw. der Abbildung 4.17 für den Sensor entnommen worden.

- **Beschreibungsfeld 4 und 7 – Kommunikationsstecker/-buchse Rechtecksignal**
  Die elektrische/elektronische Kommunikationsverbindung dient zur Übermittlung technischer Daten des Rechtecksignals. Die Daten des Mainboards stammen aus der Tabelle 4.1 auf Seite 270, die Daten des Sensors aus der Abbildung 4.17 bzw. der Tabelle 4.7 (Seite 277 ff).

- **Beschreibungsfeld 5 und 6 – Stecker/Buchse Stromversorgung**
  Das Steckerbeschreibungsfeld des Steckers *O1* beschreibt die elektrischen Eigenschaften der Stromversorgung des Sensors. Das Buchsenbeschreibungsfeld *I1* beschreibt ebenfalls die Eigenschaften der Stromversorgung des Sensors, jedoch aus Sensorsicht. Die Daten stammen aus den bisher aufgeführten Abbildungen und Tabellen.

In den Beschreibungsfeldern sind die für die Kompatibilität des Sensors mit dem Mainboard notwendigen konkreten (quantitativen) Werte eingetragen, so dass im nächsten Schritt – basierend auf den konkreten Werten in den Beschreibungsfeldern – die Kompatibilität des Systems Gleitschutz bewertet werden kann.

Bevor jedoch mit der Kompatibilitätsbewertung begonnen werden kann, müssen noch die projektspezifischen Kompatibilitätsregeln, definiert werden.

[294] Siehe hierzu: Kapitel „3.8 *(U)CML-ed, ein graphischer Editor für die Kompatibilitätsmodellierungssprache (U)CML*“ ab Seite 254.

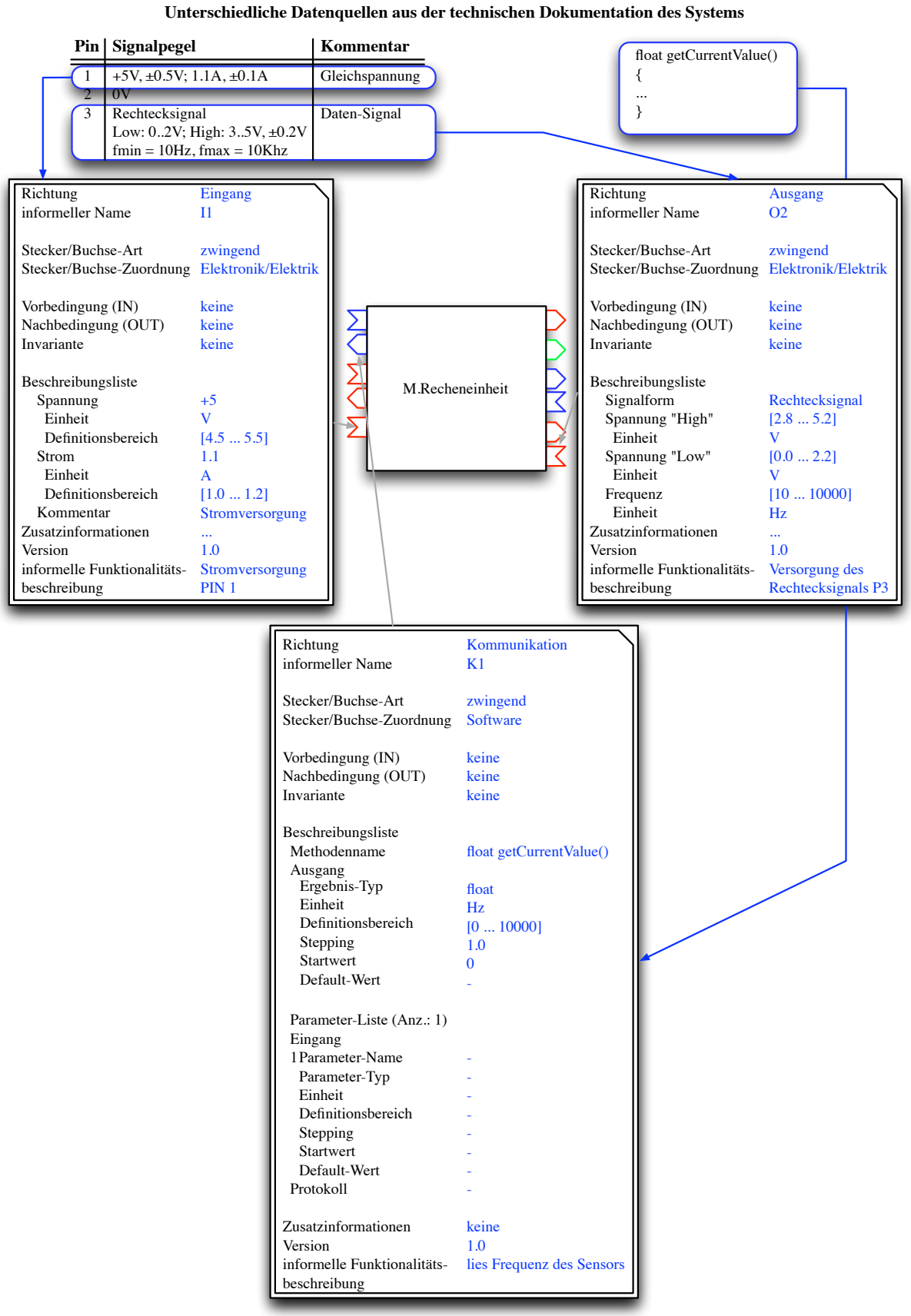


Abbildung 4.39.: Befüllung des Beschreibungsfelds *M.Recheneinheit* des Pakets *Mainboard* mit konkreten Werten.

4.1.1.2. Regelwerk erstellen

Nachdem die schrittweise Modellierung des ESRA-Gleitschutzsystems in (U)CML abgeschlossen ist, folgt in diesem Abschnitt die Festlegung der Kompatibilitätsregeln, nach denen das quantitative (U)CML-Gleitschutzsystemmodell überprüft werden soll.

Wie bereits im Kapitel „3.6.5 *(U)CML-Sprach- und Kompatibilitätsregelwerk*“ ab Seite 236 gezeigt worden ist, besteht das (U)CML-Kompatibilitätsregelwerk aus drei Teilen: Dem (U)CML-*Sprachregelwerk*, in dem sämtliche die (U)CML betreffenden Regeln zusammengefasst sind, dem *projektunabhängige Regelwerk*, in dem alle Kompatibilitätsgrundregeln wie z.B. Präfixe, Einheiten und Intervalle enthalten sind sowie dem *projektabhängigen Regelwerk*. Das projektabhängige Regelwerk dient dabei zur Ergänzung und Erweiterung des projektunabhängigen Regelwerks (zum Beispiel um die Definition neuer projektspezifischer Präfixe und Einheiten). Das projektunabhängige Regelwerk kann **nicht** an das zu bearbeitende Projekt angepasst werden, das projektabhängige Regelwerk jedoch schon, sofern es spezielle Kompatibilitätsregeln benötigt, die noch nicht im projektunabhängigen Regelwerk enthalten sind.

In der nachfolgenden Aufzählung wird das (U)CML-Regelwerk in seine drei Bestandteile – (U)CML-Sprachregelwerk, projektunabhängiges- und projektabhängiges Regelwerk – unterteilt beschrieben. Dabei wird besonders auf die Anforderungen des Gleitschutzsystems an das Regelwerk eingegangen.

- **(U)CML-Sprachregelwerk**
  Im (U)CML-Sprachregelwerk[295] sind sämtliche Regeln enthalten, mit deren Hilfe der Aufbau sowie die Struktur der (U)CML beschrieben werden[296]. In der nachfolgenden Aufzählung sind einige Beispiele für (U)CML-Sprachregeln aufgelistet:
  - In einem System darf es nur genau ein Systempaket geben.
  - Ein (U)CML-Paket darf beliebig viele weitere Pakete enthalten.
  - Jedes (U)CML-Paket bzw. jede (U)CML-Komponente muss mindestens einen Eingang und mindestens einen Ausgang haben.
  - Alle zwingenden Stecker/Buchsen und Schnittstellen müssen angeschlossen sein.
  - Die Signatur (Beschreibungsfeld) einer Quelle muss mit der Signatur der Senke übereinstimmen.

  Das (U)CML-Sprachregelwerk ist unabhängig von dem zu modellierenden System. Eine Anpassung des Sprachregelwerks an ein Projekt ist nicht vorgesehen und nicht erlaubt.
- **Projektunabhängiges Regelwerk**
  Im projektunabhängigen Regelwerk sind sämtliche allgemein gültigen Präfixe, Einheiten sowie Kompatibilitätsregeln hinterlegt, die für die Bestimmung der Kompatibilität eines in (U)CML modellierten Systems benötigt werden[297]. In der nachfolgenden Aufzählung sind einige Beispiele für das projektunabhängige Regelwerk, aufgeteilt nach Domänen, aufgeführt:
  - Elektrotechnik
    Für die Bestimmung der elektrischen/elektronischen Kompatibilität von Signalen sind für das Gleitschutzsystem folgende Kompatibilitätsregeln notwendig:
    * Übereinstimmung der Signalform:
      Die Signalform der Quelle muss der der Senke entsprechen.
    * Übereinstimmung der Intervalle:
      Die Intervalle für den logischen Wert „low“ sowie für den logischen Wert „high“ müssen deckungsgleich sein.
    * Präfixe und Einheiten:
      Für die Modellierung des Gleitschutzsystems werden folgende elektrische/elektronische Präfixe und Einheiten benötigt:

Einheiten	**Präfixe**
[V] olt	$10^{1} = 1$
[A] mpere	$10^{3} = 1\mathrm{k}$
[Hz] Herz	$10^{-3} = 1\mathrm{m}$
  - Mechanik
    Für die Überprüfung der mechanischen Komponenten sowie der mechanischen Stecker, Buchsen und Schnittstellen des Gleitschutzsystems sind ebenfalls speziell auf diese Domäne angepasste Kompatibilitätsregeln notwendig. Ein Beispiel für eine mechanische Kompatibilitätsregel ist z.B. die Modellierung von Toleranzen mit Hilfe von Intervallen. Dabei gilt: zwei mechanische Werte sind kompatibel, wenn
    * der Wert,
    * die Präfixe gleich sind,
    * die Einheiten gleich sind,
    * der Toleranzbereich (Intervalle) gleich ist.

    Wenn alle vier Teile gleich sind, ist die mechanische Stecker-/Buchsenverbindung kompatibel.

[295] Das (U)CML-Sprachregelwerk wird auch als (U)CML-Syntax bezeichnet.
[296] Siehe hierzu: Kapitel „3.6.5.1 *(U)CML-Sprachregelwerk*“ ab Seite 236.
[297] Siehe hierzu: Kapitel „3.6.5.2.1 *Projektunabhängiges Kompatibilitätsregelwerk*“ ab Seite 239.

Beispiel:

Quelle: $10\text{km} \pm 500\text{m}$
Senke: $10\text{km}, [9.5 \ldots 10.5]\text{km}$
Ergebnis: Quelle und Senke sind kompatibel zueinander. Es wird jedoch eine Warnung ausgegeben, da die Einheiten der Toleranzangabe nicht übereinstimmen.

– Software
Die Datentypkompatibilität der Softwaredatentypen lässt sich in (U)CML in zwei Tabellen zusammen fassen. In der programmiersprachenunabhängigen generischen Datentypmatrix (2.7) sind die allgemein gültigen Datentypen der (U)CML aufgelistet, während in der Tabelle 2.6 die Datentypkompatibilität für C/C++ aufgelistet ist.

Die folgende Tabelle enthält einige der schon definierten Datentypen, die auch für die Bestimmung der Kompatibilität des Gleitschutzsystems benötigt werden:

Quelle	→	**Senke**	**Bewertung**
`float`	→	`float`	OK
`float`	→	`double`	Warnung
`float`	→	`integer`	Fehler: Datenverlust!

Dabei ist zu beachten, dass es bei der Zuweisung in der dritten Zeile der Tabelle zu einem Datenverlust kommt, da die beiden Datentypen nicht verträglich sind.

- **Projektabhängiges Regelwerk**
Das projektabhängige Regelwerk wird zur Erweiterung bzw. Anpassung des projektunabhängigen Regelwerks eingesetzt. Hier können zusätzliche Einheiten, Präfixe und Regeln hinterlegt werden, die die bereits vorhandenen erweitern bzw. ergänzen. Des Weiteren ist es möglich, Regeln aus dem projektunabhängigen Regelwerk zu ersetzen[298].

  Für die Bestimmung der Kompatibilität des Gleitschutzsystems reicht das projektunabhängige Regelwerk aus. Aus diesem Grund werden keine projektspezifischen Regeln benötigt.

4.1.1.3. Durchführung des Kompatibilitätstests

Aufbauend auf dem statischen quantitativen (U)CML-Modell sowie den im letzten Abschnitt definierten Kompatibilitätsregeln, folgt nun die Durchführung der Kompatibilitätsbestimmung. Der Kompatibilitätstest unterteilt sich dabei in die drei Schritte – Überprüfung des *(U)CML-Sprachregelwerks*, des *projektunabhängiges Regelwerk* sowie des *projektabhängigen Regelwerks*.

Die Abbildung 4.40 zeigt den Ablauf des (U)CML-Kompatibilitätstests als Programmablaufplan.

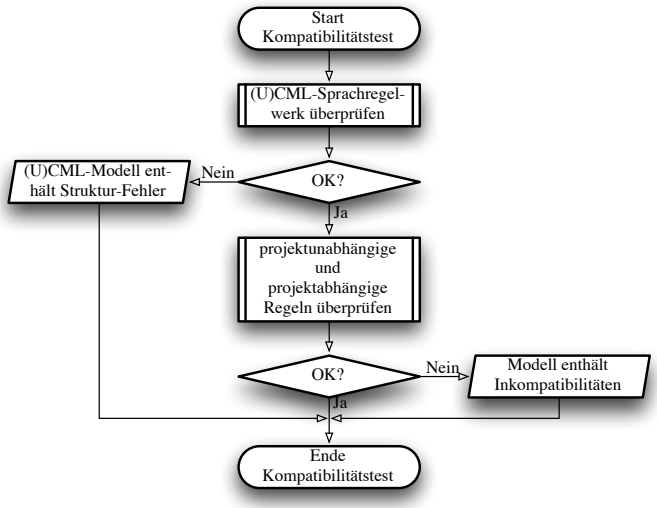


Abbildung 4.40.: Ablauf des (U)CML-Kompatibilitätstests.

Der Regelprüfungsalgorithmus unterteilt sich dabei in die zuvor angesprochenen drei aufeinander aufbauenden Teilbereiche. Diese werden im Anschluss anhand der Fallstudie Gleitschutzsystem exemplarisch erläutert.

Überprüfung des Systems anhand des (U)CML-Sprachregelwerks
Als erstes wird das modellierte System anhand des (U)CML-Sprachregelwerks untersucht. Dabei wird die Einhaltung aller Sprachregeln der (U)CML überprüft (vgl. Kapitel „3.6.5.1 *(U)CML-Sprachregelwerk*“ ab Seite 236). In der nachfolgenden Aufzählung sind vier (U)CML-Sprachregelgruppen exemplarisch für alle anderen aufgeführt:

[298] Siehe hierzu: Kapitel „3.6.5.2.2 *Projektabhängiges Kompatibilitätsregelwerk*“ ab Seite 242.

- **Pakete und Komponenten**
  Haben alle Pakete und Komponenten mindestens einen zwingenden Eingang und einen zwingenden Ausgang?

  Ergebnis: Alle Pakete und Komponenten des Systems haben mindestens einen zwingenden Ein- und Ausgang. Die Regel ist somit erfüllt – das System enthält keine Fehler oder Warnungen.

- **Stecker, Buchsen und Schnittstellen**
  Sind alle zwingenden Stecker, Buchsen und Schnittstellen angeschlossen?

  Ergebnis: Alle zwingenden Stecker, Buchsen und Schnittstellen des Systems sind angeschlossen. Außerdem sind alle optionalen Anschlüsse ebenfalls angeschlossen.

- **Externe Systemeingangs- und externe Systemausgangspfeile**
  Hat das System mindestens einen externen Systemeingangspfeil und mindestens einen externen Systemausgangspfeil?

  Ergebnis: Das Systempaket *Gleitschutzsystem* hat einen externen Systemeingangs- und einen Systemausgangspfeil – die Kommunikation mit der Umwelt ist somit sichergestellt.

- **Flusspfeile**
  Sind alle Flusspfeile an beiden Seiten (Quelle/Senke) angeschlossen?

  Ergebnis: Im Systempaket gibt es keine nicht beidseitig angeschlossenen Flusspfeile.

Ergebnis der (U)CML-Sprachregelwerksüberprüfung:
Das System *Gleitschutz* ist wohlgeformt und enthält keine Fehler oder Warnungen.

Nachdem nun die syntaktische Korrektheit des modellierten Systems sichergestellt worden ist wird das System anhand des projektunabhängigen bzw. des projektabhängigen Regelwerks verifiziert. Dabei ist das grundsätzliche Vorgehen bei beiden identisch. In der Abbildung 4.41 ist das Vorgehen bei der Regelwerksverifikation schematisch abgebildet.

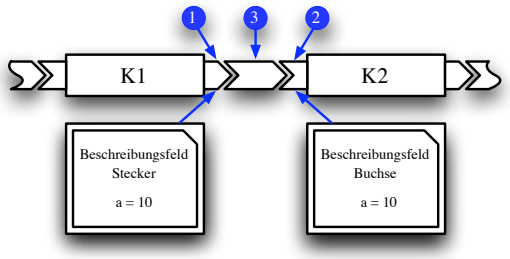


Abbildung 4.41.: Verifikation einer Verbindung zwischen zwei Komponenten.

Zuerst werden die Stecker (1) und Buchsen (2) der Komponenten auf logische Fehler in den Beschreibungsfeldern wie z.B. die Intervallverletzung oder fehlende Einträge überprüft. Daran anschließend wird die Verbindung (3) zwischen der Quelle und der Senke geprüft. Dazu werden die in den entsprechenden Beschreibungsfeldern eingetragenen Werte miteinander anhand des Regelwerks verglichen.

Das hier vorgestellte Verfahren wird nun zur Verifikation des Systems anhand des projektunabhängigen sowie projektabhängigen Regelwerks angewendet.

Überprüfung des Systems anhand des projektunabhängigen Regelwerks
Im projektunabhängigen Regelwerk sind sämtliche in (U)CML vordefinierten Kompatibilitätsregeln, Präfixe und Einheiten enthalten[299]:

- **Vordefinierte Präfixe und Einheiten**
  Im Abschnitt Präfixe und Einheiten des projektunabhängigen Regelwerks sind sämtliche in der Elektrotechnik, Mechanik und Software gebräuchlichen Präfixe und Einheiten vordefiniert. Diese können im gesamten System benutzt werden. In den Tabellen 2.3, 2.4 sowie 2.5 sind einige Beispiele für projektunabhängige Präfixe und Einheiten aufgeführt worden.

- **Vordefinierte Kompatibilitätsregeln**
  Mit Hilfe der vordefinierten projektunabhängigen Kompatibilitätsregeln kann ein in (U)CML modelliertes System auf Kompatibilität untersucht werden, insofern das System keine speziellen Kompatibilitätsregeln benötigt. In der Tabelle 2.6 sind die grundlegenden vordefinierten Softwaredatentypenkompatibilitätsregeln für die Programmiersprachen C/C++ enthalten. In der Abbildung 2.45 auf Seite 74 hingegen sind die Grundregeln für die Schachtelung von Intervallen dargestellt, die die (U)CML unterstützt.

  Weitere vordefinierte Kompatibilitätsregeln beschreiben die Signatur einer Softwaremethode. Dabei gilt für alle Programmiersprachen, dass stets die Signatur, also der Methodenname, die Parameter sowie der Ergebniswert identisch sein müssen, damit die Methode gefunden und aufgerufen werden kann[300].

[299] Siehe hierzu: Kapitel „3.6.5.2.1 *Projektunabhängiges Kompatibilitätsregelwerk*“ ab Seite 239.
[300] Eine Ausnahme bildet hier z.B. die *ellipsis* der Programmiersprache C bzw. C++. Wenn die ellipsis verwendet werden soll, muss diese im projektabhängigen Regelwerk explizit beschrieben werden, da die ellipsis ein „sehr schwer zu überprüfendes Konstrukt“ ist und deshalb in (U)CML bzw. nach MISRA [MIS07] Standard nicht verwendet werden sollte.

Verifikation der Stecker und Buchsen der Komponenten
Die projektunabhängigen Regeln und Definitionen werden nun auf die Fallstudie angewendet. In der Abbildung 4.38 auf Seite 292 ist das Gleitschutzsystem mit acht Beschreibungsfeldern abgebildet. In diesen Beschreibungsfeldern sind einige der vordefinierten Einheiten, Präfixe, Softwaredatentypen und Intervalle benutzt worden. Zum Beispiel enthalten die beiden Beschreibungsfeldern vier und sieben die vordefinierten Einheiten V für Volt und Hz für Herz. Des Weiteren sind mehrere Gültigkeitsintervalle angegeben, wie z.B. das Intervall $[10 \ldots 10000]$ für den zulässigen Definitionsbereich des Kommunikationssteckers.

Im Beschreibungsfeld vier bzw. acht ist die Signatur der Softwaremethode `float getCurrentValue()` definiert worden. Zusätzlich zur Signatur der Methode ist der Definitionsbereich als Intervall angegeben.

Verifikation der Verbindungen
Nachdem nun sämtliche Einträge in den Beschreibungsfeldern der Stecker und Buchsen überprüft wurden, folgt nun die Prüfung der Kompatibilität der Einträge von über einen Flusspfeil verbundenen Steckern und Buchsen.

Beispiel: Kompatibilität der elektrischen/elektronischen Kommunikationsverbindung zwischen der Komponente *M.Frontanschluss* des Pakets *Mainboard* und der Komponente *El. Signale* des Pakets *Sensor* (vgl. Beschreibungsfelder 4,7 und 6,5 der Abbildung 4.38 auf Seite 292).

Beschreibung	Quelle	Senke	Bewertung
Spannung	$+5$	$+5$	OK
Einheit	V	V	OK
Definitionsbereich	Low: $]0 \ldots 2.2]$, High: $[2.8 \ldots 5.2]$	Low: $]0 \ldots 2.2]$, High: $[2.8 \ldots 5.2]$	OK
Frequenz			
Einheit	Hz	Hz	OK
Definitionsbereich	$[0 \ldots 10000]$	$[0 \ldots 10000]$	OK

Tabelle 4.12.: Kompatibilitätsprüfung der elektrischen Verbindung zwischen der Komponente *M.Frontanschluss* und *El. Signale*.

Beispiel: Kompatibilität der Softwareverbindung (Beschreibungsfeld vier und acht):
Auch die Softwareverbindung *Signal1* zwischen den beiden Komponenten *M.Frontanschluss* und *SW. Signal* ist kompatibel, da die beiden definierten Signaturen der C/C++ Methode und die definierten Wertebereiche identisch sind.

Ergebnis der projektunabhängigen Regelwerksverifikation:
Das System *Gleitschutz* enthält keine Fehler oder Warnungen. Sämtliche verwendeten projektunabhängigen Präfixe, Einheiten sowie die vordefinierten Kompatibilitätsregeln wurden eingehalten.

Überprüfung des Systems anhand des projektabhängigen Regelwerks
Nachdem das (U)CML-Modell anhand des allgemein gültigen projektunabhängigen Regelwerks überprüft wurde, folgt nun die Verifikation des Modells anhand des projektabhängigen Regelwerks. Das projektabhängige Regelwerk der (U)CML dient zur Erweiterung des projektunabhängigen Regelwerks um projektabhängige Eigenschaften und Regeln (vgl.: „3.6.5.2.2 *Projektabhängiges Kompatibilitätsregelwerk*“ ab Seite 242).

Das Modell des Gleitschutzsystems enthält keine projektabhängigen Präfixe, Einheiten oder Kompatibilitätsregeln. Für die Bestimmung der Kompatibilität des Systems reichen die im projektunabhängigen Regelwerk definierten Einheiten, Präfixe und Regeln aus.

Nachdem das gesamte Gleitschutzsystem anhand der definierten Kompatibilitätsregeln überprüft worden ist, folgt im letzten Schritt die Bewertung der Ergebnisse.

4.1.1.4. Bewertung der Ergebnisse

Nachdem der Kompatibilitätstest durchgeführt worden ist, folgt als letzter Schritt die Bewertung der Ergebnisse. Sollte das Modell keine Fehler oder Warnungen enthalten, so kann an dieser Stelle abgebrochen werden. Enthält das Modell jedoch Fehler oder Warnungen, muss nun bewertet werden, ob der Fehler *kritisch* oder *zu vernachlässigen* ist. In der nachfolgenden Aufzählung sind die drei Fehlerarten noch einmal aufgelistet und entsprechende Maßnahmen zur Beseitigung der Fehler bzw. Warnungen angegeben.

- **Modell enthält keine Fehler und Warnungen**
  Wenn nach dem Kompatibilitätstest weder Fehler noch Warnungen im Modell enthalten sind, so ist das Modell in sich (unter Anwendung aller spezifizierten Regeln) kompatibel. Weitere Maßnahmen sind nicht notwendig.
- **Modell enthält Fehler und Warnungen**
  Enthält das Modell Fehler oder Warnungen, so müssen die Fehler und Warnungen bewertet und geeignete Maßnahmen zur Beseitigung der Fehler und Warnungen ergriffen werden.
  - *Fehler*
    Grundsätzlich gilt: Fehler im Modell müssen beseitigt werden, da sonst die Kompatibilität des Modells nicht gewährleistet ist. Zur Beseitigung der Kompatibilitätsfehler stehen unterschiedliche Maßnahmen zur Verfügung, wie z.B. die Anpassung der Schnittstellen zwischen zwei Komponenten oder die „Lockerung“ des Regelwerks. Beide Maßnahmen sind jedoch mit äußerster Vorsicht durchzuführen, da es ansonsten zu versteckten Inkompatibilitäten kommen kann.

    In der Fallstudie enthält das Modell keine Fehler, so dass keine Gegenmaßnahmen ergriffen werden müssen.

– *Warnungen*
Grundsätzlich gilt: Warnungen können ignoriert werden, weil es sich „nur“ um eine Anmahnung handelt[301]. Im Allgemeinen sollten Warnungen in Modell mit besonderer Vorsicht behandelt werden. Zwar können Warnungen grundsätzlich ignoriert werden, es ist aber dennoch ratsam, diese zu beseitigen oder zumindest zu dokumentieren.

Das Gleitschutzmodell der Fallstudie enthält keine Warnungen. Aus diesem Grund sind auch keine Gegenmaßnahmen einzuleiten.

4.1.2. Szenario 2: Austausch einer Komponente aus einem in (U)CML modellierten System

In diesem Szenario wird eine nicht mehr lieferbare Komponente, der Gleitschutzsensor *GI5*, durch ein neue Komponente *GI6* mit unterschiedlichen Eigenschaften ersetzt. Um den Austausch durchführen zu können, muss zunächst wieder ein (U)CML-Modell der neuen Komponente erstellt werden. Das Modell des Sensors *GI6* wird dann in das, im Szenario 1 erstellte Referenzmodell integriert und das gesamte Gleitschutzsystem wieder auf Kompatibilität untersucht und die Ergebnisse bewertet. Bevor jedoch mit der eigentlichen Modellierung des Sensors in (U)CML begonnen werden kann, müssen zunächst die kompatibilitätsrelevanten Eigenschaften des neuen Sensors ermittelt werden. Dies geschieht nach der im Abschnitt „4.1.1.1.1 *Aufbereitung der vorhandenen Daten aus unterschiedlichen Quellen*“ ab Seite 267 gezeigten Methode.

4.1.2.1. Modellierung der neuen Komponente

Für die Modellierung des neuen Sensors *GI6* müssen zunächst wieder die kompatibilitätsrelevanten Eigenschaften des Sensors ermittelt und so aufbereitet werden, dass sie leicht in ein (U)CML-Modell überführt werden können.

Anmerkung:
Die einzelnen Schritte, von der Datensammlung über die Modellierung des Systems bis hin zum abschließenden Kompatibilitätstest, werden hier nur noch „stichpunktartig“ aufgeführt, da alle dafür notwendigen Schritte bereits im letzten Kapitel („4.1.1 *Szenario 1: Modellierung eines neuen Systems bzw. einer Baugruppe in (U)CML*“ ab Seite 266) ausführlich dargestellt worden sind.

Aufbereitung der vorhandenen Daten des Sensors GI6
Die Aufbereitung der vorhandenen Daten aus den unterschiedlichen technischen Dokumenten des Sensors *GI6* erfolgt auf die gleiche Weise, wie im Abschnitt „Aufbereitung der vorhandenen Daten aus unterschiedlichen Quellen“ für den Gleitschutzsensor *GI5* gezeigt. Begonnen wird wieder mit der Analyse der elektrischen/elektronischen Eigenschaften des Sensors:

- **Elektrotechnik**
  In der nachfolgenden Abbildung (Abb.: 4.42) ist der logische Schaltplan des Sensors *GI6* abgebildet.

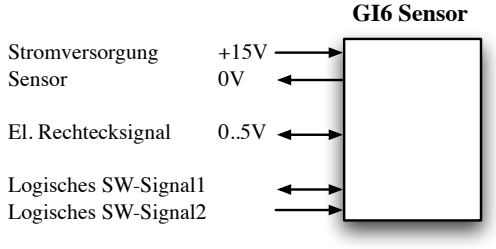


Abbildung 4.42.: Logischer Schaltplan des Sensors GI6.

  Im Gegensatz zum Gleitschutzsensor *GI5* (vgl. Abb.: 4.17) besitzt der Sensor *GI6* einen zweiten logischen Softwaresignaleingang (I2) und eine um 3 Volt höhere Versorgungsspannung (I1). In der Tabelle 4.13 sind sämtliche elektrischen/elektronischen Eigenschaften sowie der zweite logische Softwarekommunikationsanschluss beschrieben.

  Alle anderen elektrischen/elektronischen Werte des Sensors *GI6* stimmen mit den Werten des Sensor *GI5* überein.

- **Mechanik**
  Der Gleitschutzsensor *GI6* hat die gleichen Abmessungen wie der Sensor *GI5* und benutzt den selben Anschlussstecker. Neu hinzugekommen ist das optionale mechanische Signal *max. Leiterlänge*. Mit Hilfe dieses Signals kann statisch getestet werden, ob der Sensor innerhalb der maximal zulässigen Leitungslänge angeschlossen worden ist. Bei diesem mechanischen Signal handelt es sich ebenfalls um ein virtuelles Signal, für das im konkreten realen System keine Leitung existiert. Das Signal ist lediglich auf Modellebene vorhanden.

- **Software**
  Die neu hinzugekommene virtuelle Softwaremethode `void Reset()` des Sensors *GI6* dient zum Zurücksetzen des Sensors auf seinen Startwert. Dazu wird der interne Speicherplatz für den aktuellen Umdrehungswert des Polrads („Wert“) auf den Startwert ($v = 0$ Stillstand) zurückgesetzt.

[301] Vergleiche Warnungen bei einem C oder C++ Compiler. Das Programm ist trotz Warnungen lauffähig, es ist jedoch ratsam, Warnungen ernst zu nehmen, da sie auf potentielle Fehlerquellen hinweisen.

Sig. Name	Richtung	Signalpegel		Kommentar
I1	IN	$+15\text{V}, \pm 0.5\text{V}$; 2A, $\pm 0.1\text{A}$ (*)		Stromversorgung Sensor
O1	OUT	0V		Stromversorgung Sensor
IO1	IN/OUT	C++ Methode		„virtuelle Softwarefunktion“ (IO2)
I2	IN	C++ Methode		„virtuelle Softwarefunktion“ (IO2)
IO2	IN/OUT	Rechtecksignal	$Low : 0 \ldots 2\text{V}, \pm 0.2\text{V}$ $High : 3 \ldots 5\text{V}, \pm 0.2\text{V}$ $f_{min} = 10\text{Hz}$ $f_{max} = 10\text{Khz}$	Rechtecksignal der Umdrehungsgeschwindigkeit des Polrads
(*) Der Gleitschutzsensor *GI6* kann auch mit einer Versorgungsspannung von $+12\text{V}, \pm 0.5\text{V}$ betrieben werden. Dazu muss jedoch der Widerstand R1 im Sensor entfernt werden.				

Tabelle 4.13.: Signalbeschreibung Sensor GI6.

Anmerkung:
Die Softwaremethode **`void`** `Reset()` ist bereits ein fester Bestandteil der Steuersoftware des Gleitschutzsystems (Klasse: Sensor_Komm). Sie war jedoch im Referenzmodell mit dem GI5 Sensor nicht angeschlossen, da der Sensor *GI5* die Funktionalität „Reset“ nicht anbietet. Im (U)CML-Modell ist C++ Methode **`void`** `Reset()` in der Komponente *M.Recheneinheit* des Pakets *Mainboard* implementiert.

Modellierung des Sensors GI6 in (U)CML
Nachdem nun alle für die Modellierung des Sensors benötigten Daten gesammelt und aufbereitet wurden, folgt die Transformation der kompatibilitätsrelevanten Eigenschaften des Sensors *GI6* in ein qualitatives (U)CML-Modell.

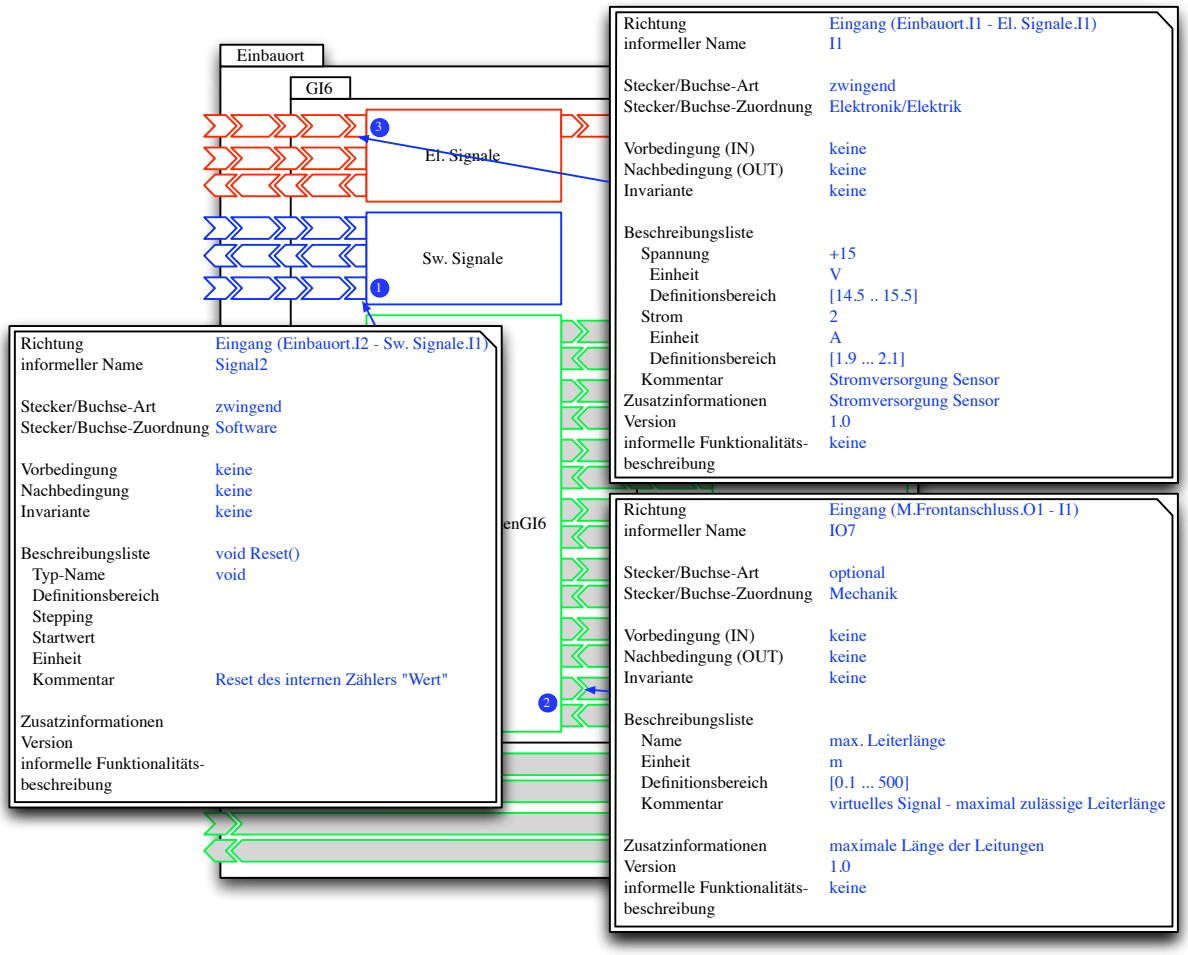


Abbildung 4.43.: Statisches Modell des Sensors „GI6“ in (U)CML mit drei Beschreibungsfeldern für die neu hinzugekommenen Stecker.

Die Abbildung 4.43 auf der vorherigen Seite zeigt das (U)CML-Modell des Gleitschutzsensors *GI6* sowie das Modell des Einbauorts, an dem der Sensor eingebaut wird. Im Modell existiert die neu hinzugekommene virtuelle Softwaremethode (**`void`** `Reset()`) als Eingangsbuchse (1). Zusätzlich wurde das Modell des Einbauorts (Paket *Einbauort*) sowie das Paket *GI6* um eine Standardschnittstelle und den entsprechenden Standardflusspfeil dazwischen erweitert. Die zusätzliche mechanische Eigenschaft *max. Leiterlänge* (2) wurde als optionaler Kommunikationsstecker modelliert und die entsprechende optionale Kommunikationsschnittstelle sowie der optionale Kommunikationspfeil zwischen dem Kommunikationsstecker und der Kommunikationsschnittstelle dem Paket *GI6* hinzugefügt.

Befüllung der Beschreibungsfelder des Sensors GI6
Anhand der im ersten Schritt identifizierten kompatibilitätsrelevanten Eigenschaften werden nun die entsprechenden (U)CML-Beschreibungsfelder des Pakets *GI6* bzw. der darin enthaltenen Stecker und Buchsen der Komponenten mit quantitativen Werten befüllt.

In der Abbildung 4.43 auf der vorherigen Seite ist das (U)CML-Modell des Gleitschutzsensors *GI6* mit seinen drei Komponenten *El. Signale, Sw. Signale* sowie *Mech. GI6* abgebildet. Für die beiden im Vergleich zum Sensor *GI5* neu hinzugekommenen Anschlüsse der Komponente, *El. Signale* sowie der Komponente *Sw. Signale* sind die entsprechenden Beschreibungsfelder (1) und (2) ebenfalls in der Abbildung enthalten. Das Softwarebeschreibungsfeld der Eingangsbuchse (1) beschreibt die neu hinzugekommene virtuelle Softwaremethode **`void`** `Reset()`. Im mechanischen Beschreibungsfeld (2) wird der virtuelle mechanische Kommunikationsstecker *max. Leiterlänge* beschrieben. Als letztes muss das Beschreibungsfeld der Stromversorgung (3) mit den Werten aus der Tabelle 4.13 befüllt werden.

4.1.2.2. Austauschprozess einer Komponente

Nachdem im vorangegangenen Kapitel der Gleitschutzsensor *GI6* in (U)CML modelliert wurde, folgt in diesem Kapitel der Austausch des nicht mehr lieferbaren Sensors *GI5* aus dem Referenzmodell durch den neuen Sensor *GI6*. Der Ablauf des Austauschprozesses ist in der Abbildung 4.44 dargestellt. Auf der linken Seite der Abbildung ist der Ablauf der Ersetzung einer Komponente durch eine andere innerhalb eines Pakets abgebildet. Auf der rechten Seite der Abbildung ist ein einfaches (U)CML-Modell als Beispiel für den Austausch einer Komponente dargestellt (Austausch K1 gegen K3).

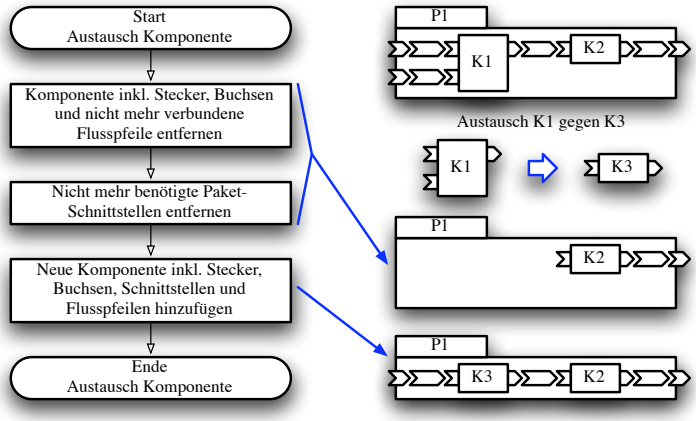


Abbildung 4.44.: Ablauf des Austausches einer Komponente in einem (U)CML-Modell.

Austausch der Komponente *GI5* durch die Komponente *GI6*
Die Abbildung 4.45 auf der nächsten Seite illustriert die drei Schritte, die für den Austausch einer Komponente innerhalb eines Pakets notwendig sind, anhand des Austauschs des Gleitschutzsensors *GI5* durch den Sensor *GI6*. Für die schematische Darstellung des Austauschs des Pakets *Sensor* (GI5) gegen das Paket *GI6* ist nur der dafür wesentliche Ausschnitt aus dem gesamten Gleitschutzsystemmodell abgebildet. Auf der linken Seite der Abbildung ist das Paket *Einbauort* sowie das darin enthaltene Paket *Sensor* mit allen Flusspfeilen dargestellt. Auf der rechten Seite ist das Paket *Sensor* durch das neue Paket *GI6* inklusive aller benötigten Schnittstellen und Flusspfeile ersetzt worden. In der Abbildung 4.46 auf der nächsten Seite ist das vollständige Gleitschutzsystem mit dem neuen Sensor *GI6* dargestellt. Die Abbildung 4.47 auf Seite 303 zeigt das gesamte Gleitschutzsystem in Glass-Box Darstellung mit dem neu eingebauten Sensor *GI6*.

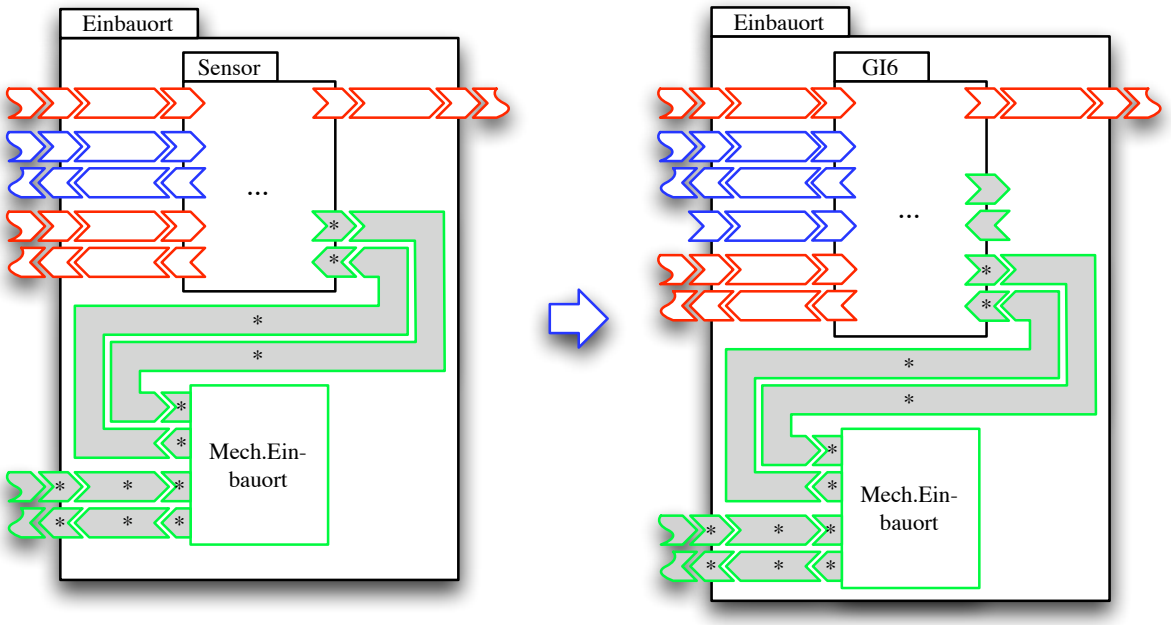


Abbildung 4.45.: Ersetzung des Sensors GI5 durch den Sensor GI6 auf Modellebene (Auszug aus dem Gleitschutzsystem).

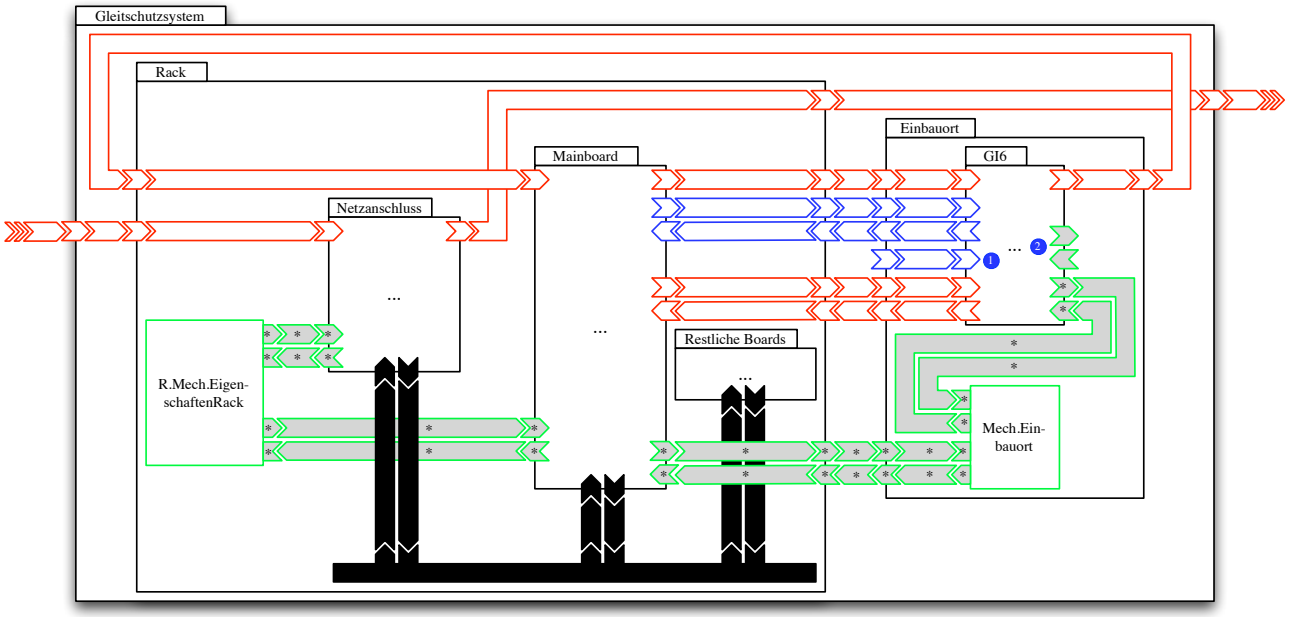


Abbildung 4.46.: Austausch des Sensors GI5 durch den Sensor GI6 auf Modellebene.

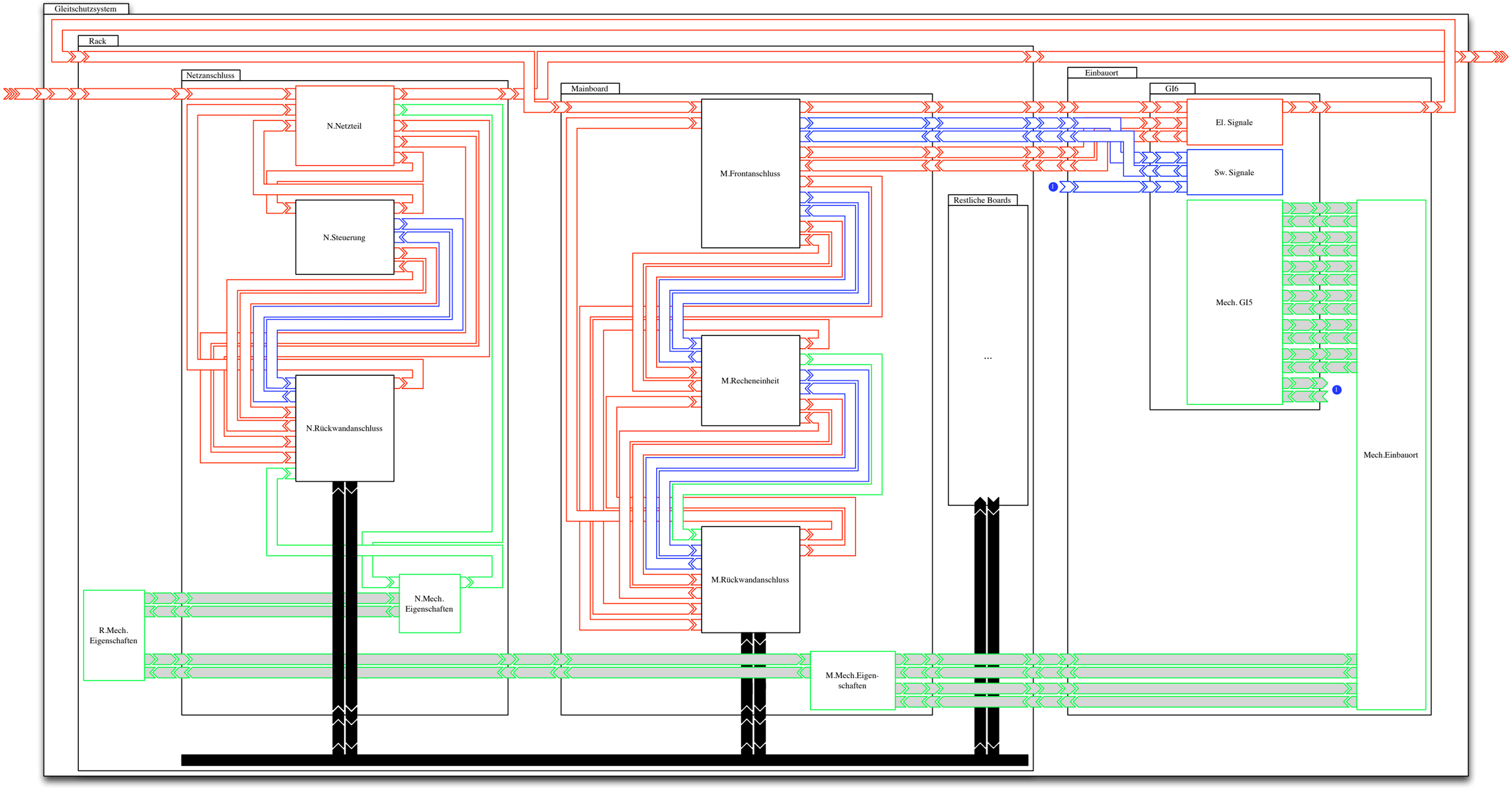


Abbildung 4.47.: Vollständiges Modell des Gleitschutzsystems.

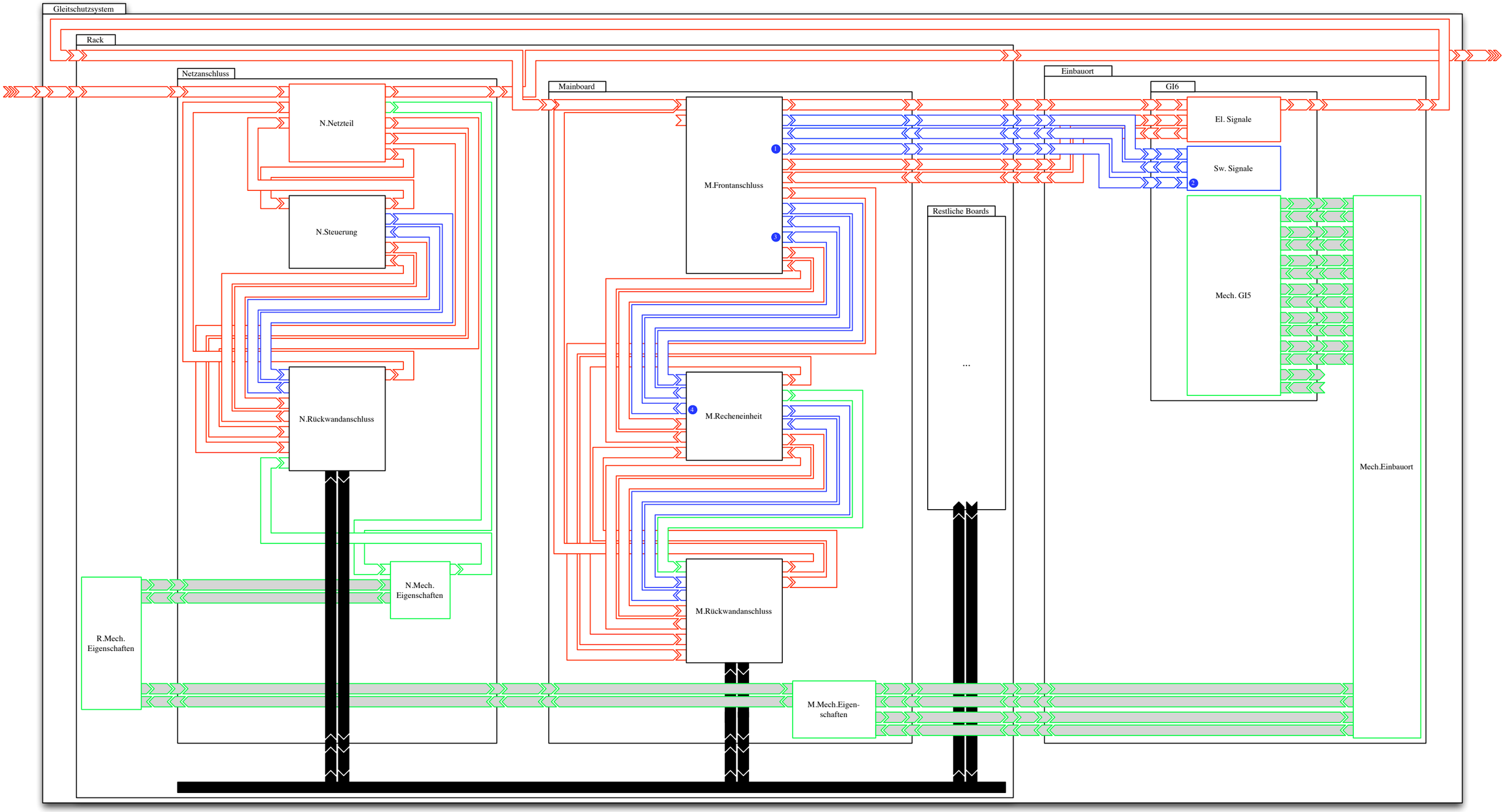


Abbildung 4.48.: Vollständiges Modell ohne Fehler.

4.1.2.3. Kompatibilitätstest und Bewertung der Ergebnisse

Nachdem Abschluss der Modellierung des Gleitschutzsystems mit dem neuen Sensor *GI6* in (U)CML folgt nun die Durchführung des Kompatibilitätstests. Dabei wird zuerst wieder die syntaktische Korrektheit des Modells anhand des (U)CML-Sprachregelwerks überprüft. Im Anschluss folgt die Verifikation des Modells anhand des vorher definierten projektunabhängigen und projektabhängigen Regelwerks. Abgeschlossen wird dieses Kapitel mit der Bewertung des Kompatibilitätstests.

Durchführung des Kompatibilitätstestes
Für die Durchführung des Kompatibilitätstests wird das selbe Regelwerk benutzt, das auch für die Kompatibilitätsbestimmung des Referenzsystems verwendet wurde. Dabei unterteilt sich der gesamte Kompatibilitätstest wieder in die nachfolgenden drei Einzeltests (vgl. Abb.: 4.49):

- **Überprüfung des Systems anhand des (U)CML-Sprachregelwerks**
  Ergebnis: Die zwingende Paketschnittstelle (1) des Pakets *Einbauort* ist nicht angeschlossen. Der optionale Kommunikationsstecker (2) ist nicht angeschlossen. Dies ist jedoch kein Fehler, da optionale Ein- und Ausgänge nicht angeschlossen sein müssen.
- **Überprüfung des Systems anhand des projekt(un)abhängigen Regelwerks**
  Ergebnis: Die Flussverbindung (3) zwischen dem Paket *Mainboard* – Ausgangsstecker 1 – und der Eingangsbuchse des Pakets *GI6* enthält einen Fehler. Der Ausgang des Pakets *Mainboard* liefert $+12$V Spannung, während der Eingang des Pakets *GI6* einen Spannungswert von $+15$V erwartet (vgl. Abb.: 4.38 auf Seite 292 sowie die Abb.: 4.43 auf Seite 300).

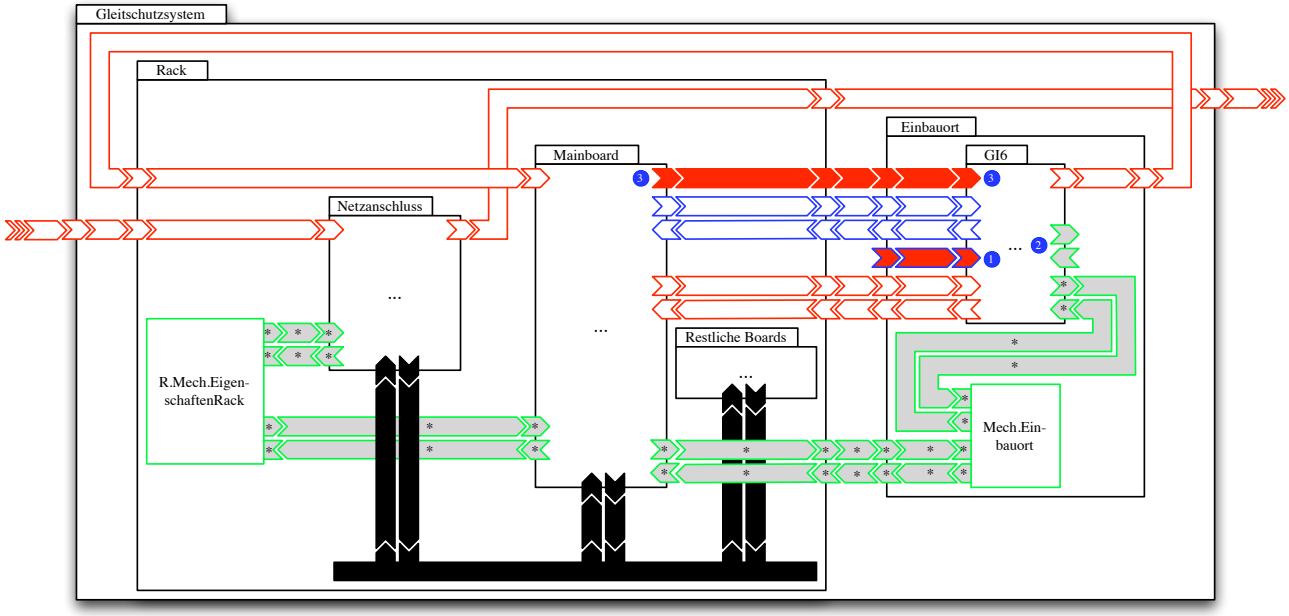


Abbildung 4.49.: (U)CML-Modell des Gleitschutzsystems mit Kompatibilitätsfehlern.

Alle im (U)CML-Modell enthaltenen Fehler werden rot, alle Warnungen gelb markiert, so dass sie einfach wiedergefunden werden können. Da das Modell mehrere Fehler enthält, folgt nun die Fehlerbewertung sowie die Fehlerbeseitigung.

Fehlerbewertung und -beseitigung
Der im letzten Schritt durchgeführte Kompatibilitätstest des Gleitschutzsystems hat zwei unterschiedliche Fehler aufgedeckt – (U)CML-Sprachregelverletzungen und Regelwerksverletzungen. In diesem Abschnitt werden die identifizierten Fehler bewertet und im Anschluss beseitigt, so dass wieder ein vollständig fehlerfreies Modell des Gleitschutzsystems entsteht.

(U)CML-Sprachregelfehler
In (U)CML gilt: Alle in einem Modell enthaltenen Verletzungen des Sprachregelwerks müssen zwingend beseitigt werden, weil sonst die Kompatibilität des gesamten Systems nicht gewährleistet werden kann[302]. Um den im Gleitschutzmodell enthaltenen syntaktischen Sprachfehler zu beseitigen, bieten sich zwei unterschiedliche Lösungsmöglichkeiten an:

1. **Die *zwingende* Eingangsbuchse der Komponente *Sw. Signale optional* setzen**
   Ein möglicher Lösungsansatz besteht darin, die zwingende Eingangsbuchse der Komponente *Sw. Signale* auf optional zu setzen, so dass er nicht angeschlossen sein muss. Dieser Lösungsansatz ist jedoch nur möglich, wenn die Softwarefunktion `void Reset()` des Sensors nicht benutzt werden soll bzw. diese Funktion für den korrekten Betrieb des Sensors nicht zwingend benötigt wird.

   Im neuen Gleitschutzsystems mit GI6 Sensor soll die neu hinzugekommene Eigenschaft „Reset“ genutzt werden. Aus diesem Grund ist dieser Lösungsansatz unbrauchbar und wird verworfen.

[302] Dies ist vergleichbar mit der Übersetzung z.B. eines C++ Programms. Enthält der Quellcode syntaktische Fehler, so ist das erzeugte Programm nur bedingt ablauffähig. Es müssen zuerst alle syntaktischen Fehler beseitigt werden, bevor das Programm einwandfrei (abgesehen von logischen Fehlern) funktioniert.

2. **Erweiterung der Komponente *M.Frontanschluss* des Pakets *Mainboard***
   Wenn das Softwaresignal **`void`** `Reset()` des Sensors *GI6* benutzt werden soll bzw. für den Betrieb des Sensors benötigt wird, muss das Signal vom Mainboard geliefert werden. Dazu müssen das Paket *Mainboard* sowie die beiden Komponenten *M.Frontanschluss* und *M.Recheneinheit* um die entsprechenden Anschlüsse erweitert werden.

   - Verbindung $1 \rightarrow 2$
     Mit Hilfe des zusammengesetzten Flusspfeils $1 \rightarrow 2$ wird der Ausgangsstecker der Komponente *M.Frontanschluss* des Pakets *Mainboard* mit der Eingangsbuchse der Komponente *Sw. Signale* des Pakets *GI6* über die Paketschnittstellen der Pakete *Rack*, *Einbauort* sowie *GI6* verbunden.
   - Verbindung $3 \rightarrow 4$
     Um die Reset-Funktion des Sensors *GI6* nutzen zu können, muss zusätzlich zur Verbindung $1 \rightarrow 2$ noch eine zusätzliche Verbindung im Modell des Gleitschutzsystems hinzugefügt werden. Durch den zusammengesetzten Flusspfeil $3 \rightarrow 4$ wird der Ausgangsstecker (3) der Komponente *M.Frontanschluss* mit der Eingangsbuchse (4) der Komponente *M.Recheneinheit* verbunden.

   In der Abbildung 4.48 auf Seite 304 sind die beiden beschriebenen Softwareverbindungen in das Modell des Gleitschutzsystems eingezeichnet worden.

Nachdem nun beide Verbindungen ($1 \rightarrow 2$ und $3 \rightarrow 4$) in das (U)CML-Modell integriert worden sind (vgl. Abb.: 4.48 auf Seite 304), kann die Softwaremethode des Sensors *GI6* durch die Komponente *M.Recheneinheit* aufgerufen und der Sensor dadurch gesteuert werden. Im Anschluss an die Integration der beiden neu hinzugekommenen Verbindungen im Modell, müssen beide Verbindungen ebenfalls auf Kompatibilität untersucht werden.

Ergebnis: Die beiden neu hinzugekommenen zusammengesetzten Flusspfeile $1 \rightarrow 2$ und $3 \rightarrow 4$ enthalten keine Kompatibilitätsfehler.

Definitionsbereichsverletzung
Der zweite Fehler, der während des Kompatibilitätstests identifiziert wurde, betrifft den zusammengesetzten Flusspfeil 3 (Abb.: 4.49 auf der vorherigen Seite). Bei diesem Fehler handelt es sich um eine so genannte „Definitionsbereichsverletzung“, also eine Verletzung des Gültigkeitsintervalls eines Wertes.

Im Referenzmodell (vgl. Abb.: 4.38 auf Seite 292) bzw. in der Tabelle 4.1 auf Seite 270 wurden die elektrischen/elektronischen Eigenschaften des Mainboards und insbesondere der beiden Komponenten *M.Frontanschluss* sowie *M.Recheneinheit* spezifiziert. Aus der Tabelle 4.1 geht die Versorgungsspannung der Sensoren $+12\text{V}$ mit einer Toleranzabweichung von $\pm 0.5\text{V}$ hervor. Der neue Gleitschutzsensor *GI6* benötigt laut Datenblatt[303] eine Versorgungsspannung von $+15\text{V} \pm 0.5\text{V}$. Der Sensor kann jedoch laut Datenblatt auch mit $+12\text{V} \pm 0.5\text{V}$ betrieben werden, wenn dazu der interne Widerstand R1 des Sensors entfernt wird.

Um die Kompatibilität zwischen dem Sensor und dem Mainboard zu erreichen, können zwei unterschiedliche Anpassungen des (U)CML-Modells vorgenommen werden:

- **Anpassung der Speisespannung des Sensors**
  Eine möglich Lösung, die Gültigkeitsverletzung der Versorgungsspannung des Sensors zu lösen, besteht in der Anpassung der Versorgungsspannung des Sensors. Dazu sind folgende Anpassungen notwendig:
  - **Mainboard**
    - *M.Frontanschluss*
      Ausgangsstecker (O1) – Feld: Spannung: $+15\text{V}$
      Eingangsbuchse (I2) – Feld: Spannung: $+15\text{V}$
    - *M.Rückwandanschluss*
      Ausgangsstecker (O1) – Feld: Spannung: $+15\text{V}$
      BUS-System-Anschluss

  Eine Anpassung des BUS-Systems bewirkt, dass alle daran angeschlossenen Baugruppen ebenfalls angepasst werden müssen. Daraus folgt, dass auch die Pakete *Netzanschluss* und *Restliche Boards* angepasst werden müssen.

  - **Netzanschluss**
    - *N.Rückwandanschluss*
      Eingangsbuchse (I2) – Feld: Spannung: $+15\text{V}$
      BUS-System-Anschluss
    - *N.Netzteil*
      Wenn die Komponente eine zusätzliche bzw. angepasste Versorgungsspannung liefern soll so muss sie an die geänderten Anforderungen angepasst werden. Diese Anpassung hat auch eine Anpassung der realen Stromversorgung zur Folge. Die Stromversorgung muss nun zusätzlich eine Spannung von $+15\text{V}$ mit einer Genauigkeit von $\pm 0.5\text{V}$ liefern (O4).
  - **Restliche Boards**
    Da die restlichen Boards des Gleitschutzsystems in der Fallstudie nicht modelliert sind, kann auch keine Aussage über die Auswirkungen auf die darin enthaltenen Pakete und Komponenten gemacht werden.

  Ergebnis: Die Anpassung der Versorgungsspannung des Mainboards hat weitreichende Folgen für das gesamte Gleitschutzsystem. Aus diesem Grund sollte diese Anpassung nur durchgeführt werden, wenn es keine andere Alternative gibt.

[303] Siehe hierzu: Kapitel „4.13 *Signalbeschreibung Sensor GI6*“ ab Seite 300.

- **Anpassung des Sensors an die vom Mainboard gelieferte Stromversorgung**
  Aufgrund der Eigenschaft des Sensors, mit verschiedenen Versorgungsspannungen betrieben werden zu können, wird diese Möglichkeit im Modell implementiert. Dazu ist lediglich die Anpassung des Beschreibungsfelds (3) in der Abbildung 4.43 auf Seite 300 notwendig. Der Eintrag *Spannung* wird auf $+12$V angepasst und im Feld *Kommentar* der Eintrag „Widerstand R1 aus Sensor entfernt“ hinzugefügt.

Sobald alle Anpassungen am (U)CML-Modell des Gleitschutzsystems mit Sensor GI6 vorgenommen worden sind, wird es noch einmal auf Kompatibilität überprüft, um sicherzustellen, dass alle Änderungen am Modell den gewünschten Effekt hatten. In der Abbildung 4.48 auf Seite 304 ist das gesamte fehlerfreie Gleitschutzsystem mit dem Sensor *GI6* abgebildet.

4.2. Bewertung der Ergebnisse der Fallstudie

Im Abschnitt „Fallstudie – Gleitschutzsystem“ wurde zunächst die schrittweise Modellierung des real existierenden ESRA-Gleitschutzsystem der Firma Knorr-Bremse AG in (U)CML demonstriert. Dabei wurde mit der Identifikation der kompatibilitätsrelevanten Eigenschaften des Systems begonnen. Daran anschließend folgte die schrittweise Modellierung des Systems – qualitative und quantitative Modellierung. Nachdem die Modellierungsphase abgeschlossen wurde, folgte die Definition des Kompatibilitätsregelwerks sowie die Durchführung und Bewertung des Kompatibilitätstests.

Im zweiten Teil dieses Kapitels wurde der Austausch einer Komponente aus dem zuvor modellierten System gezeigt. Dabei wurden noch einmal alle zuvor durchgeführten Schritte, von der Identifikation der kompatibilitätsrelevanten Eigenschaften über die Modellierung bis hin zum Test des Systems auf Kompatibilität mit nachfolgender Bewertung der Ergebnisse, wiederholt. Des Weiteren wurden in diesem Kapitel zwei mögliche Lösungen zur Beseitigung der aufgezeigten Inkompatibilitäten demonstriert und ebenfalls bewertet.

Ergebnisse der Fallstudie
Die Durchführung der Fallstudie hat gezeigt, dass die (U)CML sowohl bei der strukturierten Modellierung eines neuen Systems bzw. dem Austausch einer Komponente eines bestehenden Systems als auch bei der Bestimmung der Kompatibilität eingesetzt werden kann. Vor allem hat sich während der Fallstudie herausgestellt, dass aufgrund des einfachen Aufbaus der (U)CML alle an der Entwicklung beteiligten Teams – Elektriker/Elektroniker, Mechaniker und Informatiker – gemeinsam an einem Modell arbeiten konnten, ohne dass eine lange Einarbeitungsphase notwendig war. Dadurch wurden viele „kleine Hürden“, die während der Entwicklung eines eingebetteten Systems auftreten vermindert bzw. konnten ganz beseitigt werden.

Während der Fallstudie hat sich außerdem gezeigt, dass der einmal zu leistende Mehraufwand für die vollständige Modellierung eines Systems in (U)CML nicht zu unterschätzen ist. Der hier geleistete Mehraufwand zahlt sich jedoch auf lange Sicht aus; wenn beispielsweise eine Komponente eines Systems ersetzt werden muss, kann diese Ersetzung wesentlich leichter durchgeführt werden, aufgrund der Tatsache, dass der Kompatibilitätstest nahezu „mitgeliefert“ wird. Ein weiteres Ergebnis der Fallstudie ist, dass für den Einsatz von (U)CML ein leistungsfähiges Werkzeug benötigt wird, da ansonsten die Vorteile der (U)CML bei großen Flussdiagrammen verloren gehen. Aus diesem Grund wurde bereits nach dem Abschluss der Fallstudie mit der prototypischen Implementierung eines graphischen Werkzeugs für die (U)CML begonnen[304].

[304] Siehe hierzu: Kapitel „3.8 *(U)CML-ed, ein graphischer Editor für die Kompatibilitätsmodellierungssprache (U)CML*“ ab Seite 254.

Teil III.

Zusammenfassung und Ausblick

In diesem Abschnitt wird zunächst eine ausführliche Zusammenfassung der Arbeit und im Anschluss ein Ausblick auf weitere artverwandte Themenbereiche gegeben, die im Rahmen dieser Arbeit nicht verfolgt wurden.

Kapitel:

Kapitel 5.

Zusammenfassung der Arbeit

Die adäquate Versorgung mit kompatiblen Ersatzteilen für komplexe technische Systeme wurde und wird immer schwieriger. Dies liegt größtenteils an den kürzer werdenden Produktlebenszeiten, von zum Teil nur wenigen Monaten, sowie der daraus resultierenden sehr großen Menge an unterschiedlichen Varianten eines Produkts am Markt. Im Gegensatz zu den kurzlebigen Produkten, wie beispielsweise Mobiltelefonen oder MP3-Playern, gibt es jedoch auch Produkte wie z.B. Schienenfahrzeuge, mit einer Produktlebenszeit von zum Teil deutlich mehr als 30 Jahren. Während dieser extrem langen Lebenszeit muss beispielsweise das System Schienenfahrzeug kontinuierlich mit passenden Ersatzteilen versorgt werden, um dessen Funktionsfähigkeit zu gewährleisten. Diese Aufgabe gestaltet sich jedoch sehr schwierig, aufgrund der Tatsache, dass entweder die benötigten Baugruppen des Systems über einen sehr langen Zeitraum eingelagert oder die vorhandenen Baugruppen des Systems möglichst kompatibel zur Vorgängerversion weiterentwickelt werden müssen. Vor allem die kompatible Weiterentwicklung von Baugruppen gestaltet sich durch die Komplexität der Baugruppen sowie die ständig anwachsende Vernetzung als schwer zu bewältigende Aufgabe.

Um dieser Entwicklung Rechnung zu tragen, wurde im Rahmen dieser Arbeit die domänenübergreifende, einfach zu erlernende Kompatibilitätsmodellierungssprache (U)CML entwickelt, mit deren Hilfe technische Systeme von Ingenieur-Teams aus unterschiedlichen Fachrichtungen gemeinsam modelliert und auf Kompatibilität an den Schnittstellen überprüft werden können. Ein weiteres Ziel dieser Arbeit war die nahtlose Integration des neu entwickelten Kompatibilitätsmanagements in das umfassende Systems Engineering Umfeld, um angefangen bei der kompatibilitätskonformen Systemauslegung über die Wartungsphase bis hin zur Entsorgung den kompletten Lebenszyklus eines Systems begleiten zu können. Dabei wurden zunächst die grundlegenden Begriffe und Definitionen sowohl aus der Domäne des Systems Engineerings als auch aus dem Kompatibilitätsumfeld vorgestellt, die für die interdisziplinäre Modellierung von technischen Systemen bestehend aus Mechanik, Elektrik/Elektronik und Software notwendig sind. Nach der Vorstellung der grundlegenden Begriffe und Definitionen, erfolgte die Einführung und Erweiterung des objektorientierten Modellbildungsparadigmas für technische Systeme aus Hard- und Software sowie die Anwendung speziell auf die Domäne der eingebetteten softwarelastigen Systeme. Begleitend zur Vorstellung des objektorientierten Modellbildungsparadigmas wurde sowohl der generische Prozesses zur Identifikation der kompatibilitätsrelevanten Eigenschaften eines technischen Systems als auch der damit verbundene übergreifende objektorientierte kompatibilitätskonforme Entwicklungsprozess für eingebettete softwarelastige Systeme vorgestellt und die Einbettung in das umfassende Systems Engineering Umfeld diskutiert. Im Anschluss wurde das allgemein gültige objektorientierte Modellbildungsparadigma so erweitert, dass es für die Modellierung und Bewertung der kompatibilitätsrelevanten Eigenschaften von eingebetteten Systemen angewendet werden kann.

Nachdem sämtliche Grundlagen der objektorientierten Kompatibilitätsmodellierung eingeführt worden sind, wurden zunächst Anforderungen an eine Kompatibilitätsmodellierungssprache für komplexe technische Systeme festgelegt. Im Anschluss wurde stellvertretend für alle existierenden Modellierungssprachen eine Auswahl von vier unterschiedlichen Modellierungssprachen und -konzept vorgestellt, mit denen technische Systeme modelliert werden können. Dabei wurde insbesondere die Fähigkeit jeder vorgestellten Modellierungssprache untersucht, die kompatibilitätsrelevanten Eigenschaften eines Systems in der jeweiligen Notation der Modellierungssprache zu modellieren sowie das entstandene Systemmodell auf Kompatibilität zu untersuchen. Insbesondere wurde bei dem Vergleich der unterschiedlichen Modellierungssprachen zwischen den beiden Szenarien: „Modellierung der Struktur sowie des Verhaltens eines Systems“ sowie der „Austausch- und Ersetzungskompatibilität“ unterschieden. Abgeschlossen wurde das Kapitel mit der Vorstellung der neu entwickelten Kompatibilitätsmodellierungssprache (U)CML, dem dazugehörigen objektorientierten (U)CML-Modellbildungsprozess sowie der abschließenden Bewertung der fünf vorgestellten Modellierungssprachen und -konzepte für die Modellierung und Bewertung von Kompatibilität anhand der zuvor festgelegten Anforderungen an eine Kompatibilitätsmodellierungssprache für technische Systeme.

Nach den beiden theoretischen Abschnitten wurde der praktische Einsatz und Nutzen der Kompatibilitätsmodellierungssprache (U)CML anhand des stark vereinfachten Beispielsystems Gleitschutz untersucht. Dabei musste das eingebettete softwarelastige Gleitschutzsystem so in (U)CML modelliert werden, dass sowohl die Modellierung der Struktur des Systems, als auch der anschließende Austausch/Ersetzung einer Komponente des Systems auf Kompatibilität untersucht werden konnte. Dabei stellte sich heraus, dass die (U)CML sowohl während der (kompatibilitätskonformen) Modellierungsphase eines Systems als auch für die Modellierung der Austausch-/Ersetzungskompatibilität gleichermaßen eingesetzt werden kann.

Kapitel 6.

Ausblick

Der in dieser Arbeit vorgestellte generische, objektorientierte, interdisziplinär anwendbare Ansatz zur Modellierung von kompatibilitätsrelevanten Eigenschaften eines technischen Systems sowie die damit verbundene Kompatibilitätsmodellierungssprache (U)CML, bieten noch reichlich Potential für weiterführende Arbeiten. Dies gilt insbesondere für die Verfeinerung des objektorientierten Prozesses zur Identifikation der kompatibilitätsrelevanten Eigenschaften eines Systems, das graphische Kompatibilitätsmodellierungswerkzeug *(U)CML-ed* sowie dem allem zugrunde liegende Formalismus zur Beschreibung der kompatibilitätsrelevanten Eigenschaften. So sollte beispielsweise die eingeführte formale Notation zur Beschreibung der Eigenschaften eines Systems auf weitere Domänen, wie zum Beispiel die Anforderungserhebung (Requirements-Engineering), die Beschreibung von physikalischen Eigenschaften oder die Chemie, angepasst und erweitert werden. Durch die Einführung weiterer Erweiterungen können dann beispielsweise nicht nur eingebettete Systeme modelliert werden, die aus mechanischen, elektrischen/elektronischen und Softwarekomponenten bestehen, sondern zusätzlich physikalische oder chemische Eigenschaften aufweisen. Dadurch würde der „Kompatibilitätsgedanke“ auch weitere Bereiche der modernen Produktentwicklung erfassen.

In den nachfolgenden Unterpunkten werden weitere lohnenswerte Ansatzpunkte aufgezeigt, die in dieser Arbeit nicht angesprochen wurden.

Anpassung und Erweiterung der Kompatibilitätsmodellierungssprache (U)CML
Die in dieser Arbeit vorgestellte Kompatibilitätsmodellierungssprache (U)CML sowie der dazugehörige objektorientierte Modellbildungsprozess, lassen sich problemlos, wie bereits gezeigt, auf weitere Domänen ausweiten. Zu den lohnendsten Erweiterungen zählt insbesondere die Anpassung der (U)CML für die strukturierte Modellierung von Anforderungen. Durch die „direkte Verknüpfung“ einer vom Kunden vorgegebenen Anforderung aus dem Pflichtenheft mit einer bestimmten Eigenschaft oder Funktion einer Komponente des Systemmodells lässt sich genau nachweisen, dass das Systemmodell mindestens die Kundenanforderungen erfüllt. Dadurch lassen sich zum Teil entstehende Dissonanzen zwischen dem Kunden und dem Entwickler des Systems vermeiden, da das aus den Anforderungen entstandene Pflichtenheft vollständig formal beschrieben und nicht wie sonst üblich, in verbaler Form vorliegt.

Um die interdisziplinäre Systementwicklung mit Hilfe der (U)CML weiter zu verbessern, sollten weitere Domänen, wie beispielsweise die Chemie oder die Physik in die Kompatibilitätsmodellierung aufgenommen werden. Durch die Hinzunahme sowohl der Chemie als auch der Physik wird die interdisziplinäre Kompatibilitätsmodellierung weiter verbessert. So können nach der Erweiterung beispielsweise auch Systeme modelliert und auf Kompatibilität untersucht werden, die bestimmte physikalische oder chemische Eigenschaften aufweisen, die für die Kompatibilitätsbestimmung notwendig sind.

Die wichtigste Erweiterung der (U)CML ist jedoch die Anpassung der kompatibilitätskonformen Modellierung von dynamischen Aspekten eines Systems und insbesondere die Entwicklung einer grundsätzlichen generischen Methodik für die Simulation der (U)CML-Modelle. Dazu zählt insbesondere die Entwicklung eines grundlegenden generischen Simulationsmodells, das über die Darstellung der MSCs hinaus reicht, um insbesondere unterschiedliche Simulationswerkzeuge, wie beispielsweise das in den Ingenieurwissenschaften beliebte Matlab Simulink [Inc07a], nahtlos in das (U)CML-Modell zu integrieren bzw. in das graphische Modellierungswerkzeug *(U)CML-ed* einzubinden.

Das gemeinsame Ziel sämtlicher Erweiterungen der (U)CML ist es, die (U)CML von einer auf bestimmte Domänen zugeschnittenen Kompatibilitätsmodellierungssprache kontinuierlich zu einer interdisziplinär einsetzbaren Modellierungssprache für Kompatibilität zu verbessern und erweitern.

Erweiterung des Editors *(U)CML-ed*
Eng mit der Anpassung und Erweiterung der (U)CML ist die Weiterentwicklung des graphischen Werkzeugs *(U)CML-ed* verbunden. Im Idealfall sollten sämtliche Konzepte der (U)CML vollständig im Editor enthalten sein. Aus diesem Grund zählt zu den wichtigsten Aufgaben die Implementierung der noch fehlenden Sprachkonzepte der (U)CML. Dazu zählen unter anderem das BUS-System, die zusammengefassten Flusspfeile sowie die Implementierung der MSCs zur dynamischen Simulation des Modells. Darüber hinaus sollte der Editor um eine dezidierte generische Schnittstelle zur Anbindung von Simulationswerkzeugen, wie beispielsweise von Matlab Simulink, erweitert werden, um die externe Simulation der (U)CML-Modelle zu ermöglichen.

Des Weiteren muss die graphische Bedienoberfläche des Editors weiter verbessert werden. Zu den wichtigsten Verbesserungen der Oberfläche zählen unter anderem „Komfortfunktionen“ wie beispielsweise das Zeichnen von Flusspfeilen über Paketgrenzen hinweg sowie die Implementierung von Tooltips[305], in denen die wichtigsten Eigenschaften eines (U)CML-Objekts kompakt zusammengefasst dargestellt werden. Außerdem muss die aktuelle Layout-Engine, die für die graphische Darstellung der unterschiedlichen Diagrammarten (Fluss-, Graphen-, Matrix- sowie Baumdarstellung) zuständig ist, weiter optimiert und Darstellungsfehler beseitigt werden.

[305] Bei einem *Tooltip* innerhalb eines Programms oder einer Web-Seite handelt es sich, um eine Kurzinformation über das Objekt, über dem der Mauszeiger gerade steht. Genauere Informationen zu Tooltips finden Sie unter [Wik08q].

Eine weitere lohnenswerte Erweiterungsmöglichkeit des *(U)CML-ed* besteht in der Abkehr von der XML-Persistenzschicht des Editors hin zu einer vollständig transparenten Datenbankanbindung. Durch die Abkehr von der dateibasierten Persistenzschicht lassen sich zwei unterschiedliche Konzepte gleichermaßen verwirklichen. Zum einen ist durch die Verwendung einer Datenbank das Concurrent-Engineering [Wik08o][Age08][DIS07] möglich, also das simultane Arbeiten von mehreren Teams an einem gemeinsamen Datenmodell des Systems. Zum anderen können weitere Werkzeuge einfacher auf das zentrale Datenmodell des Systems zugreifen, wodurch die Erweiterungsmöglichkeiten des Editors erheblich verbessert werden.

Entwicklung eines zentralen generischen domänenübergreifenden Systemmodells
Die wohl wichtigste Erkenntnis, die sich aus dem in dieser Arbeit vorgestellten Kompatibilitätsmodellierungskonzept und insbesondere aus dem domänenübergreifenden Systemmodell ableiten lässt, sind die zahlreichen positiven Eigenschaften des einheitlichen domänenübergreifenden Systemmodells auf die fächerübergreifende Kooperation bei der Systementwicklung. Zum einen kann das gemeinsame Systemmodell von der Vorstudie bis zur Entsorgung des Systems domänenübergreifend von allen an der Systementwicklung beteiligten Entwicklern gleichermaßen eingesetzt werden, da alle Entwickler das Systemmodell lesen und verstehen können. Zum anderen lassen sich sämtliche Daten des Systems in diesem Modell ablegen, wie beispielsweise die Entwicklungsdauer, geschätzte Kosten oder technische Zeichnungen des Systems, um diese Daten, in einem nächsten Schritt so zu vernetzen, dass beispielsweise eine Änderung an einer Eigenschaft einer Komponente des Systems instantan an alle angebundenen Baugruppen des Systemmodells mitgeteilt wird.

Bereits während dieser Arbeit wurde mit der Planung des Forschungsprojekts INTERXPHASE begonnen, dessen Ziel es ist, ein zentrales generisches integriertes domänenübergreifendes Systemmodell, aufbauend auf den grundlegenden Konzepten des integrierten Datenmodells der (U)CML, zu entwickeln, das über alle Entwicklungsphasen eines Systems gleichermaßen eingesetzt werden kann. Insbesondere kann durch das integrierte vernetzte Datenmodell des Systems die Kompatibilität eines Systems, bzw. der unterschiedlichen Systembestandteile einfach verifiziert werden, da sämtliche Bestandteile des Systems in der zentralen Datenbank enthalten sind. Dadurch lässt sich viel Zeit bei der kompatibilitätskonformen Weiterentwicklung von Systembestandteilen einsparen. Zusätzlich zur Zeiteinsparung wird die Produktqualität verbessert, da jede Änderung am Systemmodell sofort und automatisch auf Kompatibilität untersucht und bewertet werden kann.

Im Projekt INTERXPHASE sollen im Gegensatz zur (U)CML nicht nur die technischen Parameter eines Systems hinterlegt werden können, sondern beispielsweise auch Geschäftsprozesse oder Kostenmodelle. Ein weiteres Merkmal des zentralen domänenübergreifenden Datenbankmodells ist die Vernetzung der unterschiedlichen Datenbestände der Datenbank. So kann beispielsweise das Kostenmodell zu einer technischen Baugruppe hinterlegt werden, das diese Baugruppe aus wirtschaftswissenschaftlicher Sichtweise beschreibt. Des Weiteren unterstützt der INTERXPHASE Ansatz das im letzten Abschnitt angesprochene Concurrent-Engineering Paradigma, die parallele Entwicklung eines Systems.

Zusammengenommen ist das Forschungsprojekt INTERXPHASE die wichtigste Erweiterung der grundlegenden Konzepte dieser Arbeit.

Teil IV.

Anhang

Kapitel:

Anhang A.

Mathematische Grundlagen und Definitionen

Quellenangabe: [Sto][Stö93][Bro91][Wik06i][Wik06g][Wik07s][Pri06][Ohn03][Lan06]

A.1. Mengen und Tupel

Definition A.1 ***Mengen***
Menge, die Zusammenfassung bestimmter wohlunterscheidbarer Objekte (Elemente der Menge) zu einem Ganzen. Mengen werden mit großen Buchstaben (z.B. X), Elemente der Menge mit kleinen Buchstaben (z.B. p, x) gekennzeichnet.
Man schreibt $p \in X$, wenn p Element von X ist und $p \notin X$, wenn p nicht Element der Menge X ist.

Lemma A.1 ***Darstellung und Eigenschaften von Mengen***

- ***Aufzählung*** *ihrer Elemente $X = \{1, 2, 3, 4\}$*
- ***Charakterisierung*** *durch eine definierte Eigenschaft E der Elemente p, man schreibt*
$$X = \{p \mid p \text{ genügt der Eigenschaft } E\}.$$
- ***Leere Menge*** $\emptyset = \{\}$
- ***Gleichheit von Mengen:*** *Mann nennt zwei Mengen X und Y gleich, wenn jedes Element aus X auch Element von Y ist und auch jedes Element aus Y Element von X ist. Schreibweise: $X = Y$*

Definition A.2 ***n-Tupel***
Das $n-Tupel$ ist ein Begriff aus der Mathematik. Er bezeichnet eine geordnete Zusammenstellung von Objekten, im Gegensatz zu Mengen, deren Elemente keine festgelegte Reihenfolge haben. $n-Tupel$ werden üblicherweise durch runde Klammern angegeben.
Die Objekte werden als Elemente, Komponenten oder Einträge des $n-Tupels$ bezeichnet. Dadurch, dass bei einem $n-Tupel$ jedem seiner Elemente ein eindeutiger Platz zugeordnet ist, kann es auch mehrfach dasselbe Element enthalten. n bezeichnet hierbei die Anzahl der Elemente des $n-Tupels$. Diese Anzahl muss endlich sein. Ein 3-Tupel wird oft auch als Tripel bezeichnet.
Notationskonflikt: Oft werden auch offene Intervalle als (a,b) geschrieben. Ob ein Intervall oder ein Paar gemeint ist, ist aus dem Kontext zu ersehen[Wik07w].

A.2. Relationen

Binäre Relation (Relationstypen)

Definition A.3
Eine binäre Relation $R \in A \times A$ auf A heißt:

- *reflexiv falls $(a, a) \in R$ für jedes $a \in A$ gilt.*
- *irreflexiv falls für kein $a \in A$ gilt $(a, a) \in R$.*
- *symmetrisch falls aus $(a, b) \in R$ immer auch $(b, a) \in R$ folgt.*
- *asymmetrisch falls aus $(a, b) \in R$ immer auch $(b, a) \notin R$ folgt.*
- *antisymmetrisch falls aus $(a, b), (b, a) \in R$ folgt $a = b$.*
- *transitiv falls aus $(a, b) \in R$ und $(b, c) \in R$ immer auch $(a, c) \in R$ folgt.*

Die natürlichste binäre Relation auf einer Menge A ist die Identität (oder Gleichheitsrelation, Diagonale) $id_A = (a, a) | a \in A$

Binäre Relationen $R \in A \times A$ auf einer Menge A werden manchmal auch homogene (binäre) Relationen genannt.

Ordnungsrelation (Mathematische Definition)
Verallgemeinerung der „kleiner-gleich“-Beziehung. Sie erlauben es, Elemente einer Menge miteinander zu vergleichen. Eine Ordnungsrelation ist formal eine zweistellige Relation $R \subseteq M \times M$ auf einer Menge M mit bestimmten Eigenschaften.

Ist eine Menge M mit einer Ordnungsrelation R gegeben, dann wird das Paar (M, R) eine geordnete Menge genannt. Statt $a\ R\ b$ wird meistens (je nach Art der Ordnung) $a \leq b$ oder $a < b$ geschrieben.

Eine (totale) Ordnung auf einer Menge liefert eine Anordnung der Elemente in einer bestimmten Reihenfolge, z.B. die Anordnung der Buchstaben A bis Z im lateinischen Alphabet. Die Reihenfolge der Buchstaben ist willkürlich festgelegt. Jede andere Reihenfolge wäre ebenfalls eine Ordnung.

Ordnungsrelation (System-Engineering Element-Konzept)
Im Systems Engineering Element-Konzept wird mit Hilfe von Ordnungsrelationen die Struktur eines Systems festgelegt. In der Abbildung A.1 ist ein System, bestehend aus vier Elementen dargestellt. Das *Element A* steht dabei auf der höchsten Hierarchieebene des Systems (Hierarchieebene 1). Wenn nur ein Element auf der obersten Hierarchieebene ist, so wird dies oft auch als *Top-Level Element* bezeichnet. Die drei *Elemente B*, *C* und *D* sind alle auf der Hierarchieebene zwei. Des Weiteren sind alle dem *Element A* untergeordnet.

Ordnungsrelationen lassen sich graphisch als *gerichtete Pfeile* zwischen zwei Elementen darstellen. Alle Elemente eines Systems bilden zusammen einen Graphen, genauer einen *gerichteten Graphen*.

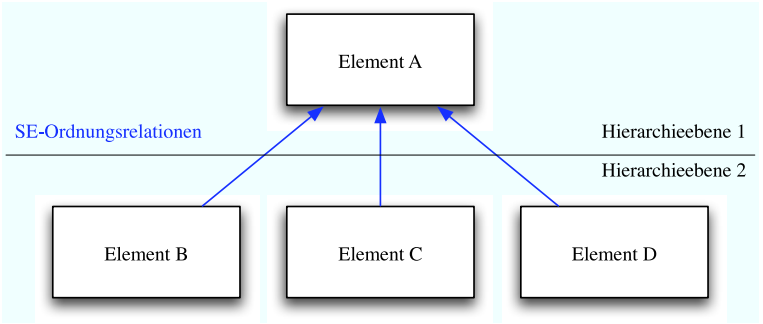


Abbildung A.1.: Ordnungsrelation.

Eigenschaften der SE Ordnungsrelation:

- Ordnungsrelationen bilden eine (hierarchische) Struktur.
- Ordnungsrelationen haben „keine Länge“. D.h. es ist unerheblich wie lange die Verbindungslinie zwischen zwei Elementen ist.
- Ordnungsrelationen sind irreflexive, asymmetrische eins zu eins Verbindungen.

Flussrelation (System-Engineering Element-Konzept)
Als Flussrelation wird im SE Element-Konzept eine binäre Relation $R \subseteq A \times B$ mit $a \in A$ und $b \in B$ bezeichnet, mit deren Hilfe Informationen I (Daten, Material, Energie etc.[306]) von einem Element A zu einem anderen Element B, mit $A \neq B$, unverändert übertragen werden. Genauer: Die Information $i \in I$ vom Ausgang a des Elements A wird zum Eingang b des Elements B unverändert transportiert $(a \xrightarrow{i} b)$ (vgl. [Ohn03] Definition einer Flussrelation in Petri-Netzen bzw. Streams in FOCUS [PDBS01, 57ff].).

In der Abbildung A.2 ist der oben beschriebene Sachverhalt dargestellt.

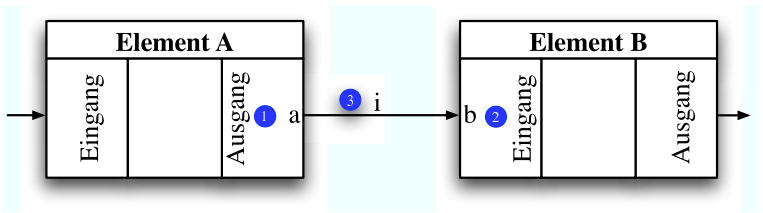


Abbildung A.2.: Graphische Darstellung der SE-Flussrelation:
(1) repräsentiert den Ausgang des Elements A
(2) den Eingang des Elements B
(3) Flussrelation, die den Ausgang a des Element A mit Eingang b des Elements B verbindet

[306] In der Informatik wird für Daten aller Art, auch Material oder Energie, der Oberbegriff *Datum* verwendet.

Eigenschaften der Flussrelation:

- Flussrelationen haben „keine Länge“. D.h. es ist unerheblich wie lange die Verbindungslinie zwischen dem Ausgang eines Elements und dem Eingang eines anderen Elements dargestellt ist.
- Flussrelationen haben keine Funktionalität, sie transportieren Daten, Material oder elektrischen Strom, ohne diesen zu beeinflussen oder zu verändern.
- Der Transport eines Datums benötigt keine Zeit ($t = 0$), d.h. sobald das Datum am Ausgang eines Elements anliegt, wird es ohne Zeitverzögerung zum Eingang des mit ihm verbundenen Elements transportiert.

  Anmerkung:
  In realen existierenden Systemen ist die Zeit t für den Transport des Datums vom Ausgang A eines Elements zum Eingang B eines anderen stets größer Null ($t > 0$)! Die Annahme, dass die Zeit für den Transport eines Datums von einem Element zu einem anderen Null ist gilt ausschließlich auf der theoretischen Modellierungsebene. Sollte die reale Zeit, die das Datum für den Transport benötigt für das System wichtig sein, so muss diese explizit, in einem Element z.B. als Attribut, modelliert werden.
- Flussrelationen sind gerichtet. Sie verbinden stets genau einen Ausgang eines Elements mit genau einem korrespondierenden Eingang eines anderen Elements. Die Pfeilrichtung gibt dabei die Flussrichtung des Datums vor.
- Flussrelationen sind irreflexiv und asymmetrisch.

A.3. Mathematische Definition des Systems Engineering Element-Konzepts

Satz A.1 ***Aus Definition 3.3 folgt unmittelbar:***
Sei $i \in \mathbb{N} \setminus \{0\}$ und $2 \leq i < \infty$. Ein System S besteht aus mindestens zwei Elementen E_i (z.B. $E_1, E_2, \ldots, E_i$). Die Elemente des Systems müssen nicht paarweise verschieden sein, d.h. in einem Systems S darf es mehrere gleiche Elemente geben.

Satz A.2 ***Elemente E, Attribute A und Relationen R***
Elemente E, Attribute A und Relationen R können als Vektoren bzw. Mengen geschrieben werden, wenn gilt: $i \in \mathbb{N} \setminus \{0\}$ mit $2 \leq i < \infty$ und $j, k \in \mathbb{N}$ mit $2 \leq j, k < \infty$

$$\vec{E} = (E_1, E_2, \ldots, E_i)^\top = \begin{pmatrix} E_1 \\ E_2 \\ \vdots \\ E_i \end{pmatrix} \quad \vec{A} = (A_1, A_2, \ldots, A_j)^\top = \begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_j \end{pmatrix} \quad \vec{R} = (R_1, R_2, \ldots, R_k)^\top = \begin{pmatrix} R_1 \\ R_2 \\ \vdots \\ R_k \end{pmatrix}$$

$$\mathscr{E} = \{E_1, E_2, \ldots, E_i\}; \quad \mathscr{A} = \{A_1, A_2, \ldots, A_j\}; \quad \mathscr{R} = \{R_1, R_2, \ldots, R_k\}$$

Lemma A.2 ***Aus den beiden Sätzen A.1 und A.2 folgt:***
Sei $i \in \mathbb{N} \setminus \{0\}$ mit $2 \leq i < \infty$ und $k \in \mathbb{N}$ mit $0 \leq k < \infty$. Ein System S kann sowohl als Vektor $S = \{\vec{E}\}$ als auch als Menge $S = \{E_1, E_2, \ldots, E_i\}$ von Elementen geschrieben werden. Soll zusätzlich betont werden, dass ein System S nicht nur aus Elementen E_i besteht, sondern die Elemente mittels Relationen R_k verbunden sind, dann kann ein System S auch als Menge, bestehend aus Elementen E_i und Relationen R_k des Systems S geschrieben werden $S = \{\vec{E}, \vec{R}\}$ bzw. $S = \{E_1, E_2, \ldots, E_i, R_1, R_2, \ldots, R_k\}$.

Satz A.3 ***Elemente E haben Attribute A:***
Sei $i \in \mathbb{N} \setminus \{0\}$ mit $2 \leq i < \infty$, $j \in \mathbb{N}$ mit $0 \leq j < \infty$, so gilt: Jedes Element E_i eines Systems S kann ***keine*** *oder* ***beliebig viele*** *Attribute A_j enthalten ($E_i = \{A_j\}$).*

Anmerkung:
Nach dem Lemma A.2 sind die beiden Schreibweisen $E_i = \{\vec{A}\}$ bzw. $E_i = \{A_1, A_2, \ldots, A_j\}$ äquivalent, wenn gilt: $i \in \mathbb{N} \setminus \{0\}$ mit $2 \leq i < \infty$ und $j \in \mathbb{N}$ mit $0 \leq j < \infty$.

Definition A.4 ***Notation der Attribute eines Elements***
Jedes Attribut A eines Elements E lässt sich formal schreiben als:

$$A_{\text{ATTRIBUTTYP } \text{ATTRIBUT\_INDEX}\ \{:\ \text{EIGENSCHAFTSTYP}\}}$$

mit

ATTRIBUTTYP = *Eigenschaften (≙ ATTR) ∨ Funktionen (≙ FKT)*
EIGENSCHAFTSTYP = *variabel (≙ VAR) ∨ konstant (≙ CONST)*
ATTRIBUT_INDEX = *Index des Attributs*

Lemma A.3 ***Verfeinerung der Attribute A_j des Elements E_i:***
Sei $e, f, j \in \mathbb{N}$ mit $0 \leq e, f, j < \infty$ und $i \in \mathbb{N} \setminus \{0\}$ mit $0 \leq i < \infty$. Des Weiteren sei j die Anzahl der Attribute des i-ten Elements E. Dann lassen sich die Attribute A_j des Elements E_i weiter unterteilen in e Eigenschaften und f Funktionen des Elements E_i mit $j = e + f$.

- *Eigenschaften (*ATTR*):*
  Sei e die Anzahl der Eigenschaften des Elements E_i. Alle Eigenschaften eines Elements E_i können mit einem optionalen Attribut versehen werden:
  - *variable Eigenschaft:* VAR
  - *konstante Eigenschaft:* KONST

  Durch die Eigenschaften VAR *und* KONST *können die Elementinternen Zugriffsrechte auf einzelne Eigenschaften eingeschränkt werden. So kann z.B. auf eine konstante Eigenschaft nur lesend zugegriffen werden, während auf eine variable Eigenschaft sowohl lesend- als auch schreibend zugegriffen werden kann.*

- *Funktionen (*FKT*):*
  *Sei f die Anzahl aller Funktionen des Elements E_i. Mit Hilfe der Funktionen (*FKT*) eines Elements E_i wird die Funktionalität des Elements E_i beschrieben. Alle Funktionen eines Elements E_i sind stets variabel; konstante Funktionen sind nicht erlaubt. Die einzelnen Funktionen FKT_f sind genauso definiert wie Funktionen in der Mathematik (vgl. [Stö93, 4, 104, 730]).*

Anmerkungen zur Definition A.4 sowie dem Lemma A.3:

- Ein Attribut A eines Elements E lässt sich folglich schreiben als:

$$E_{\text{ELEMENT\_INDEX}}.A_{\text{ATTRIBUTTYP }\text{ATTRIBUT\_INDEX}\{:\text{EINGENSCHAFTSTYP}\}}$$

- Die Attribute eines Elements werden manchmal auch als „Parameter“ bezeichnet.
- Jede Eigenschaft A_j eines Elements E_i kann entweder „qualitativen“ oder „quantitativen“ Charakter haben. Das heißt: Eine Eigenschaft eines Elements kann sowohl einen Wert enthalten oder als Platzhalter für Werte dienen.
- Die Attribute (ATTR und FKT) sind einem Element fest zugeordnet. Die Eigenschaften (ATTR) und Funktionen (FKT) des Elements sind von außerhalb nicht erreichbar (vgl. Datenkapselung bei C++/JAVA Klassen). Eine Kommunikation mit dem Element ist nur über dessen Eingänge und Ausgänge möglich.
- Die Ein- und Ausgänge eines Elements bilden die „Schnittstelle“ des Elements.

Definition A.5 ***Notation der SE-Relationen***
Jede Relation R zwischen zwei Elementen E_{QUELLE} und E_{ZIEL} eines System S lässt sich formal schreiben als:

$$R_{\text{RELATIONSTYP }\text{INDEX}} : \text{QUELLE} \rightarrow \text{ZIEL}$$

mit

RELATIONSTYP = *Ordnungsrelation ($\hat{=}$ O) $\vee$ Flussrelation ($\hat{=}$ F)*
INDEX = *Index der Relation*
QUELLE *und* ZIEL *sind zwei unterschiedliche Elemente E (*QUELLE $\neq$ ZIEL*) des Systems S*

Lemma A.4 ***Relationen R zwischen Elementen E eines Systems S:***
Sei $f, k, o \in \mathbb{N}$ mit $0 \leq f, k, o < \infty$ und $i, j \in \mathbb{N} \setminus \{0\}$ mit $i, j \geq 2 \wedge i, j \leqslant \mid S \mid$ und $i \neq j \iff E_i \neq E_j$. Des Weiteren gilt: Die Anzahl der Ordnungsrelationen o plus der Anzahl der Flussrelationen f des Systems ist gleich der Anzahl der Relationen k des Systems S ($k = o + f$).

- *Sei o die Anzahl der Ordnungsrelationen (A.2 auf Seite 318) R_O des Systems S:*
  Die Elemente E_i eines Systems S stehen über Ordnungsrelationen $R_{O_o:E_i \rightarrow E_j}$ in Verbindung. Für alle Ordnungsrelationen $R_{O_o:E_i \rightarrow E_j}$ zwischen dem Element E_i und dem Element E_j gilt: Die Ordnungsrelation R_O ist irreflexiv, asymmetrisch und gerichtet (vgl. A.2 auf Seite 317). Daraus folgt: Es gibt maximal $n - 1$ Ordnungsrelationen in einem System S, wenn n gleich der Anzahl der Elemente des Systems S entspricht ($n = \mid S \mid$).

- *Sei f die Anzahl der Flussrelationen R_F (A.2 auf Seite 318) des System S:*
  Eine Flussrelation $R_{F_f:E_i \rightarrow E_j}$ verbindet genau einen Ausgang eines Elements E_i mit genau einem Eingang eines anderen Elements E_j. Die Flussrelation zwischen den Elementen E_i und E_j kann formal auch $E_i \xrightarrow{I} E_j$ geschrieben werden, wobei I die zu übertragende Information (Datum) repräsentiert. Flussrelationen R_F sind irreflexiv, asymmetrisch und gerichtet. Außerdem gilt: Zwischen zwei Elementen E_i und E_j eines Systems S kann es keine oder beliebig viele Flussrelationen R_F geben.

Anmerkung zum Lemma A.4:

- Ist es für die Bewertung eines Systems S unerheblich, ob zwei Elemente o.B.d.A., z.B. E_1 und E_2 über eine Ordnungs- oder Flussrelation verbunden sind, so kann kurz auch $R_{E_1 \to E_2}$ bzw. $R_{k,E_1 \to E_2}$ geschrieben werden.
- Des Weiteren müssen nicht alle Elemente E eines System S miteinander in Relation stehen (vgl. Lemma A.4). Grundsätzlich kann es auch Elemente in einem System geben, die mit keinem anderen Element verbunden sind („isolierte Elemente"). Gibt es in einem System isolierte Elemente, so können diese aus dem System entfernt werden, ohne das sich die Gesamtfunktionalität des Systems verändert.

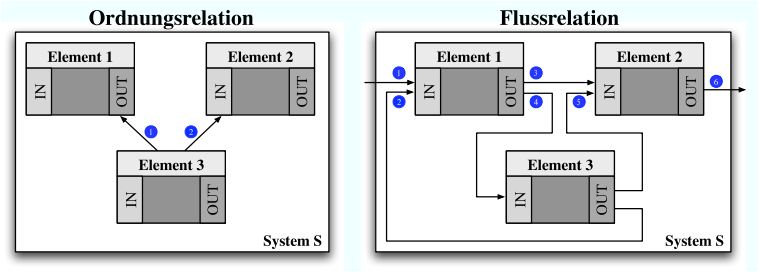


Abbildung A.3.: Ordnungs- und Flussrelationen im SE-Elementkonzept.

Beispiel 82: (Definition A.5 und Lemma A.4)
In der Abbildung A.3 ist ein System S bestehend aus $n =\mid S \mid = 3$ Elementen dargestellt. Die drei Elemente des Systems heißen $S = \{\text{ELEMENT 1}, \text{ELEMENT 2}, \text{ELEMENT 3}\}$. Auf der linken Seite sind die Ordnungsrelationen, auf der rechten Seite die Flussrelationen in das Systems S eingetragen.

- Ordnungsrelation:
  Die drei Elemente des Systems S sind mittels $n - 1 = 3 - 1 = 2$ Ordnungsrelationen verbunden und bilden die hierarchische Struktur des Systems S. Nach Definition A.5, können die zwei Ordnungsrelation R_{O_2} formal geschrieben werden:

$$R_{\text{O}_2} : \text{ELEMENT 3} \to \text{ELEMENT 1} \qquad R_{\text{O}_3} : \text{ELEMENT 3} \to \text{ELEMENT 2}$$

- Flussrelation:
  Aus der Zeichnung lässt sich leicht entnehmen, dass der Eingang des ELEMENT 1 mit zwei unterschiedlichen Flussrelationen verbunden ist. Eine Flussrelation (1) kommt von der Umwelt (U $\xrightarrow{\text{i}}$ ELEMENT 1), die anderen (2) vom Ausgang des ELEMENT 3 (ELEMENT 3 $\xrightarrow{\text{i}}$ ELEMENT 1). Außerdem ist der Ausgang des ELEMENT 1 über zwei Flussrelationen mit dem Eingang des ELEMENT 2 (3) (ELEMENT 1 $\xrightarrow{\text{i}}$ ELEMENT 2) und dem Eingang des ELEMENT 3 (4) (ELEMENT 1 $\xrightarrow{\text{i}}$ ELEMENT 3) verbunden. Über die Flussrelationen kommuniziert das Element mit anderen Elementen im System bzw. mit der Umwelt.
  Nach Definition A.5, können die sechs Flussrelationen des Systems S formal geschrieben werden:

$$\begin{array}{ll} R_{\text{F}_1} : \text{UMWELT} \to \text{ELEMENT 1} & R_{\text{F}_2} : \text{ELEMENT 3} \to \text{ELEMENT 1} \\ R_{\text{F}_3} : \text{ELEMENT 1} \to \text{ELEMENT 2} & R_{\text{F}_4} : \text{ELEMENT 1} \to \text{ELEMENT 3} \\ R_{\text{F}_5} : \text{ELEMENT 3} \to \text{ELEMENT 2} & R_{\text{F}_6} : \text{ELEMENT 2} \to \text{UMWELT} \end{array}$$

  In diesem Beispiel nehmen die beiden Flussrelationen R_{F_1} und R_{F_6} eine Sonderrolle ein. Mit Hilfe dieser beiden Relationen kommuniziert das System mit seiner Umwelt.

Satz A.4 ***SE-Matrixschreibweise für Elemente eines Systems:***
Die Elemente E eines Systems $S = \{E_1, E_2, \ldots, E_i\}$ mit $i \in \mathbb{N} \setminus \{0\}, i =\mid S \mid$ lassen sich in eine quadratische Element-Element-Matrix $M_E(i \times i)$ der Dimension i eintragen[307]*. Dazu werden alle Elemente E_i eines Systems S in die Index-Zeilen und anschließend in die Index-Spalten der $i \times i$-Matrix M_E eingetragen.*

$$M_E(i \times i) = \begin{array}{c|cccc} & E_1 & E_2 & \cdots & E_i \\ E_1 & \boldsymbol{0} & e_{1,2} & \cdots & e_{1,i} \\ E_2 & e_{2,1} & \boldsymbol{0} & \cdots & e_{2,i} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ E_i & e_{i,1} & e_{i,2} & \cdots & \boldsymbol{0} \end{array}$$

Im nächsten Schritt wird an jeder Stelle der Matrix M_E eine „1" eingetragen, an der zwei Elemente E_a und E_b ($a, b \in \mathbb{N} \setminus \{0\} : a, b \geq 1 \wedge a, b \leqslant \mid S \mid$ und $a \neq b$) in Verbindung (Relation) stehen ($R_{E_a \to E_b}$), sonst eine „0". Auf der Hauptdiagonalen der Element-Element-Matrix steht stets „0", da Elemente, genauer die Relationen (Ordnungs- und Flussrelationen), nicht reflexiv sind.

[307] Vgl. Adjazenzmatrix [Wik06j].

Lemma A.5 ***Erweiterung des Satzes A.4 (SE-Matrixschreibweise):***
Die allgemeine Element-Element-Matrix M_E lässt sich aufteilen in eine Matrix für die Struktur bildenden Ordnungsrelationen M_{E_O} und in eine Matrix für die informationsübertragenden Flussrelationen M_{E_F}. Dabei gilt: Die elementweise ODER*-Funktion $\otimes$[308] der beiden Matrizen M_{E_O} und M_{E_F} ergibt die allgemeine Element-Element-Matrix $M_E = M_{E_O} \otimes M_{E_F}$.*

$$\underbrace{\begin{vmatrix} \boldsymbol{0} & e_{1,2} & \cdots & e_{1,i} \\ e_{2,1} & \boldsymbol{0} & \cdots & e_{2,i} \\ \vdots & \vdots & \ddots & \vdots \\ e_{i,1} & e_{i,2} & \cdots & \boldsymbol{0} \end{vmatrix}}_{M_E} = \underbrace{\begin{vmatrix} \boldsymbol{0} & e_{1,2} & \cdots & e_{1,i} \\ e_{2,1} & \boldsymbol{0} & \cdots & e_{2,i} \\ \vdots & \vdots & \ddots & \vdots \\ e_{i,1} & e_{i,2} & \cdots & \boldsymbol{0} \end{vmatrix}}_{M_{E_O}} \otimes \underbrace{\begin{vmatrix} \boldsymbol{0} & e_{1,2} & \cdots & e_{1,i} \\ e_{2,1} & \boldsymbol{0} & \cdots & e_{2,i} \\ \vdots & \vdots & \ddots & \vdots \\ e_{i,1} & e_{i,2} & \cdots & \boldsymbol{0} \end{vmatrix}}_{M_{E_F}}$$

Lemma A.6 ***Erweiterung der allgemeinen Element-Element-Flussrelationsmatrix M_{E_F} um externe Flüsse:***
Sollen in einem System S auch die externen Ein- bzw. Ausgangsflussrelationen modelliert werden, so ist die allgemeine Element-Element-Matrix M_{E_F} für Flussrelationen um die entsprechenden Ein- und Ausgangsflüsse zu erweitern. Sei F_{IN} die in ein System S fließenden Flüsse und F_{OUT} die das System S verlassenden Flüsse, dann gilt: $m =\mid F_{IN} \mid, n =\mid F_{OUT} \mid$.

$$M_E(i \times (m+n+i)) = \begin{array}{c|cccc|ccccccc|} & E_1 & E_2 & \cdots & E_i & F_{1:IN} & \cdots & F_{m:IN} & F_{1:OUT} & \cdots & F_{n:OUT} \\ E_1 & \boldsymbol{0} & e_{1,2} & \cdots & e_{1,i} & & & & & & \\ E_2 & e_{2,1} & \boldsymbol{0} & \cdots & e_{2,i} & & & & & & \\ \vdots & \vdots & \vdots & \ddots & \vdots & & & & & & \\ E_i & e_{i,1} & e_{i,2} & \cdots & \boldsymbol{0} & & & & & & \end{array}$$

Hat ein Element E_i eine Verbindung (Flussrelation) zu einem Ein- oder Ausgang, so wird an der entsprechenden Stelle in der erweiterten Matrix $M_E(i \times (m+n+i))$ eine „1“ eingetragen, sonst eine „0“.

Beispiel 83: Anwendung des Satz A.4 sowie der Lemmata A.5 und A.6 auf das Beispielsystem A.3:
Im Beispiel A.3 war das System S vollständig beschrieben worden. Es hat sich gezeigt, dass die Beschreibung der Relationen R des Systems unübersichtlich war. Die Übersichtlichkeit lässt sich durch die Verwendung der Matrixschreibweise deutlich verbessern.

Ordnugsrelationen des Systems:

$$M_{E_O}(3 \times 3) = \begin{array}{c|ccc|} & Element1 & Element2 & Element3 \\ Element1 & \mathbf{0} & 0 & 0 \\ Element2 & 0 & \mathbf{0} & 0 \\ Element3 & 1 & 1 & \mathbf{0} \end{array}$$

Flussrelationen des Systems:

$$M_{E_F}(3 \times 3) = \begin{array}{c|ccc|} & Element1 & Element2 & Element3 \\ Element1 & \mathbf{0} & 1 & 1 \\ Element2 & 0 & \mathbf{0} & 0 \\ Element3 & 1 & 1 & \mathbf{0} \end{array}$$

Erweiterte Darstellung der Flussrelationen mit Ein- und Ausgängen:

$$M_{E_F}(3 \times 3 + 2) = \begin{array}{c|ccccc|} & Element1 & Element2 & Element3 & Eingang1 & Ausgang1 \\ Element1 & \mathbf{0} & 1 & 1 & 1 & 0 \\ Element2 & 0 & \mathbf{0} & 0 & 0 & 1 \\ Element3 & 1 & 1 & \mathbf{0} & 0 & 0 \end{array}$$

Zusammenfassung der Fluss- und Ordnungsrelationen:

$$M_E(3 \times 3) = \begin{vmatrix} \mathbf{0} & 0 & 1 \\ 0 & \mathbf{0} & 0 \\ 1 & 1 & \mathbf{0} \end{vmatrix} = \begin{vmatrix} \mathbf{0} & 1 & 0 \\ 0 & \mathbf{0} & 0 \\ 1 & 1 & \mathbf{0} \end{vmatrix} \otimes \begin{vmatrix} \mathbf{0} & 1 & 1 \\ 0 & \mathbf{0} & 0 \\ 1 & 1 & \mathbf{0} \end{vmatrix}$$

[308] Elementweise ODER-Funktion bedeutet, dass z.B. $M_E(1,1) = M_{E_O}(1,1)$ ODER $M_{E_F}(1,1)$ berechnet wird. Allgemein: Sei $\forall i, j \in \mathbb{N} \setminus \{0\}$ mit $i, j \geq 1 \wedge i, j \geq \mid S \mid$, dann ist die elementweise ODER-Funktion $\otimes$ definiert als $M_E(i,j) = M_{E_O}(i,j) \vee M_{E_F}(i,j)$.

A.4. Formale Darstellung und Definition des KOMP SE Element-Konzepts

Definition A.6 ***Erweiterung der ursprünglichen verbalen SE-Element Definition[309] für die Modellierung von Kompatibilität***

i *Ein System/Element hat mindestens einen Eingang*

ii *Ein System/Element hat mindestens einen Ausgang*

iii *Attribute haben stets einen eineindeutigen (Daten-) Typ*

iv *Attribute können mit (SI-) Einheiten versehen werden*

v *Attribute und Funktionen können entweder „öffentlich“ oder „privat“ sein*

vi *Flussrelationen sind stets „zwingend“ oder „optional“*

Definition A.7 ***Aufbau und Struktur eines KOMP SE Attributs***

{Sichtbarkeit} {Eigenschaftstyp} Datentyp Variablenname {[in Teiler Einheit {, Intervall}]} {= Zahl {[in Teiler Einheit {, Intervall}]}};

Sichtbarkeit = {private, protected, public}
Eigenschaftstyp = {const, var}
Datentyp = {void, char, short, integer, long, float, double, string, unsigned short, unsigned int, unsigned long}
Teiler = dekadischer Teiler | Informatik Teiler
Dekadische Teiler = $x \in \mathbb{Z} : 10^x$
Informatik Teiler = $x \in \mathbb{N} : 2^x$
Einheit = SI Einheit | Informatik Einheit
SI Einheit = Meter, Kilogramm, Sekunde, Ampere, Kelvin, Mol, Candela[310]
Informatik Einheit = Bit
Intervall = [obere Grenze, Untere Grenze]

[309] Vgl. Definition 3.2 auf Seite 118.
[310] Siehe auch Tabelle 2.2 auf Seite 70.

Anhang B.

Weitere Kompatibilitätsbegriffe und Definitionen

In diesem Abschnitt werden weitere Kompatibilitätsdefinitionen bzw. Kompatibilitätsbegriffe eingeführt und erläutert, die über die bereits im Kapitel „1.4 *Der Kompatibilitätsbegriff*“ ab Seite 18 eingeführten Definitionen hinaus gehen, jedoch im Kompatibilitätsumfeld häufig verwendet werden. Sämtliche Begriffe und Definitionen stammen aus [Koß09].

B.1. Kompatibilitätsarten

Definition B.1 ***Abwärtskompatibilität [Koß09]***
Eine Komponente ist abwärtskompatibel zu einem System, wenn folgende Bedingungen gelten:

1. *Die Komponente ist von neuerer Version, als die Komponente, gegen die sie getauscht wird.*
2. *Die Version des Systems ist älter, als die aktuelle Version des Systems.*
3. *Die statische und dynamische Kompatibilität sind erfüllt.*

Definition B.2 ***Aufwärtskompatibilität [Koß09]***
Eine Komponente ist aufwärtskompatibel zu einem System, wenn folgende Voraussetzungen erfüllt werden:

1. *Die Komponente ist von älterer Version, als die Komponente, gegen die sie getauscht wird.*
2. *Die Version des Systems ist neuer, als die Version des Systems, in dem die Komponente verwendet werden soll.*
3. *Die statische und dynamische Kompatibilität sind erfüllt.*

B.2. Weiterführende Begriffe und Definitionen

Definition B.3 ***System***
Ein System ist eine von seiner Umgebung abgegrenzte Anordnung aufeinander einwirkender Komponenten (nach [PDB94, 1]).

Definition B.4 ***System***
Ein System ist eine Ansammlung von Systembausteinen, die gemeinsam ein Ziel verfolgen, das von den Einzelelementen nicht erreicht werden kann. Ein Baustein kann aus Software, Hardware, Personen oder beliebigen anderen Einheiten bestehen (nach [Wei06, 10] bzw. [McK06, 10]).

Definition B.5 ***Syntax***

- ***Lehre vom Satzbau:*** *[Wis08b]*
  Syntax *kommt aus dem griechischen und bedeutet die Lehre vom Satzaufbau. Es handelt sich also um die Sprachregelung und die eindeutige Verständigungsweise, die durch die Folge von Zeichen beschrieben wird. Syntaktische Regeln sind u.a. die Interpunktion, die Groß- und Kleinschreibweise, die Schreibweise am Satzbeginn und das Zeichen am Satzende, usw.*
- ***Syntax formaler Sprachen:*** *[Wik08p]*
  Unter der Syntax einer formalen Sprache (z. B. Programmiersprachen in der Informatik; Kalküle in der Logik) versteht man ein System von Regeln, nach denen erlaubte Konstruktionen bzw. Ausdrücke (wohlgeformte Ausdrücke) aus einem grundlegenden Zeichenvorrat („Alphabet“) gebildet werden – wobei von der inhaltlichen Bedeutung der Zeichen abgesehen wird bzw. werden kann. Aus einer formalen Syntax können Syntax Graphen als grafische Repräsentation erstellt werden.

Definition B.6 ***Semantik [Wis08a]***
Semantik ist die Lehre von der Wortbedeutung. Übertragen auf die Informatik bedeutet dies, dass die Programmzeilen eines Programms eine exakte Syntax und eine eindeutige Semantik umfassen müssen, da Rechner keine Interpretationsmöglichkeiten haben. Zu diesem Zweck werden die semantischen Sprachelemente durch Zeichen, Ziffern und Befehle dargestellt.

Definition B.7 ***Schnittstelle (nach Prof. Dr. Dr. hc. Broy)***
An einer Schnittstelle treffen zwei abgegrenzte Systeme zusammen um in geregelter Weise zu interagrieren und Informationen (z.B. Nachrichten, Signale, Daten usw.) auszutauschen. Es gibt zwei Sichten auf eine Schnittstelle, die syntaktische und die semantische.

1. ***Syntaktische Schnittstelle:***
   Die syntaktische Schnittstelle gibt an, welche grundsätzlichen Möglichkeiten für Informationsaustausch an der Schnittstelle bestehen.
2. ***Semantische Schnittstelle:***
   Die semantische Schnittstelle beschreibt das Schnittstellenverhalten, dass die Kausalität und Abhängigkeiten im Informationsaustausch festlegt (die semantische Schnittstelle legt die Interaktionsmuster einer Komponente fest, welche Einzelschritte dabei auftreten ist in der syntaktischen Schnittstelle festgelegt).

Definition B.8 ***Konfiguration [Koß09]***
Eine Konfiguration ist eine Zusammensetzung von Versionen verschiedener Komponenten zu einem spezifischen Zeitpunkt im Rahmen einer Architektur zu einem System. Sie umfasst das Gesamtsystem aller Komponenten.

Anhang C.

Weiterführende Begriffe und Definitionen aus dem Systems Engineering

Im Kapitel „1.2 *Was hat Systems Engineering mit modellbasierter Kompatibilitätsbewertung zu tun?*“ ab Seite 3 wurden einige grundlegende Begriffe und Definitionen des Systems Engineerings eingeführt und erläutert. In diesem Abschnitt des Anhangs sind weitere unterschiedliche Definitionen des Begriffs Systems Engineering aufgelistet, die verdeutlichen, dass das SE eine sehr weitreichende Disziplin ist, die sich nicht mit Hilfe nur einer allgemein gültigen Definition vollständig beschreiben lässt.

- **Definition Systems Engineering nach Prof. Dr. Wünsche [PDW07]:**

  Definition C.1 ***Systemtechnik***
  Die Systemtechnik ist ein Methodengebäude zur Behandlung von Problemen mit hoher Komplexität. Sie befasst sich mit sämtlichen Lebensphasen von Systemen, d.h. mit der Planung und Realisierung einschließlich der Organisation der Nutzung sowie Außerdienststellung derselben, welche
  - *die ihnen zugedachten Funktionen voll erfüllen,*
  - *sich optional in ihre Umwelt einfügen sowie*
  - *deren Subsystem „reibungslos“ zusammenwirken.*

- **Definition Systems Engineering nach Prof. Dr. Haberfellner:**

  Definition C.2 ***Systems Engineering***[311]
  Systems Engineering ist eine Methodik, die hilft, den Prozess der Lösung von komplexen Problemen effizienter zu gestalten. Systems Engineering ist dann notwendig, wenn viele Lösungen denkbar sind und es keinen vorgezeichneten Lösungsweg gibt. Die Anwendung von Systems Engineering gibt noch keine Gewähr für optimale Lösungen, schafft aber bessere Voraussetzungen dafür.

- **Definition Systems Engineering nach Wikipedia [Wik07v]:**

  Definition C.3 ***Systems Engineering***
  Die Systemtechnik versucht, mit einem ganzheitlichen Ansatz an den Entwurf komplexer Systeme heranzugehen. Der Systemtechniker beschäftigt sich also mit dem Entwurf komplexer Gesamtsysteme, im Unterschied zu den Spezialisten, die sich auf den Entwurf der Teilsysteme konzentrieren.

- **Definition Systems Engineering nach Dr. Armin Schulz:**

  Definition C.4 ***Systems Engineering***
  Systems Engineering ist eine strukturierte Vorgehensweise zur zeit- und kostengerechten sowie marktorientierten Entwicklung komplexer technischer Systeme. Systems Engineering integriert dabei die notwendigen unterschiedlichen Rollen/Disziplinen, betrachtet die Problemstellung über ihren gesamten Lebenszyklus und stellt geeignete Prozesse, Methoden/Techniken sowie organisatorische Strukturen bereit, um die Problemstellung zu lösen.

C.1. Weiterführende Begriffe und Definitionen aus der objektorientierten Modellierung

Definition C.5 ***Mathematisches Objekt [Wik07l]***
Als mathematisches Objekt bezeichnet man Gegenstände mathematischer Untersuchungen, wie Funktionen, Zahlen, Mengen, Gruppen, Polynome, Vektoren, Restklassen usw. Um die Mathematik auf eine einheitliche Grundlage zu stellen, hat man zum Beispiel Anfang des 20. Jahrhunderts damit begonnen, für alle mathematischen Objekte Darstellungen als Mengen zu finden. Dies kann man heute zur Definition verwenden: Alles was sich als Menge darstellen lässt, ist ein mathematisches Objekt.

[311] Die Ursprünglich Definition stammt von Prof. Dr. R. Haberfellner [Hab73]. Sie wurde von Prof. Dr. G. Patzak in seinem Buch „Systemtechnik – Planung komplexer innovativer System, Grundlagen, Methoden, Techniken“ wieder aufgegriffen und verfeinert.

Definition C.6 ***Objekt in der Programmierung [Wik06h]***
Ein Objekt bezeichnet in der Objektorientierung ein Exemplar oder eine Instanz einer bestimmten Klasse. Als Objekt[312] *bezeichnet man in der Informatik ein Exemplar eines beliebigen Datentyps. Der Datentyp beschreibt das Muster aller Objekte, die zu ihm gehören. Dieser Datentyp kann ebenso ein elementarer (z. B. ein Integer) sein, wie auch eine Klasse bei objektorientierter Programmierung. Durch Konstruierung bzw. durch Instantiation wird von einer Klasse ein Objekt erzeugt, das die der Klasse eigenen Attribute und Methoden, jedoch objektspezifische Attributwerte besitzt.*

Definition C.7 ***Objekt in der objektorientierten Softwareentwicklung [Neu95]***
Ein Objekt ist zunächst ein Gegenstand, eine Rolle, ein Konzept oder ein Prozess im realen Problem- und Anwendungsbereich eines Softwaresystems (reales Objekt). Durch einen geeigneten Abstraktionsprozess wird daraus ein Objekt abgeleitet, das als Basismodul in die Ablaufstruktur eines Softwaresystems eingebunden wird. Ein Objekt hat genau eine Identität, einen Typ, einen Zustand und einen spezifizierten Lebenszyklus.

Anmerkung zu Definition C.7:
In der obigen Definition eines Objekts kann der Begriff „Softwaresystem“ durch den Begriff „System“ ersetzt werden. Durch diese Substitution kann die obige Objektdefinition auch in allen anderen Bereichen eingesetzt werden.

Definition C.8 ***Klasse nach [Neu95] auf Seite 571***
Eine Klasse ist das Ergebnis der Abstraktion von realen Objekten, die bestimmte Eigenschaften und Komponenten aufweisen und ein definiertes Verhalten zeigen. Die Abstraktion wird nach folgendem Muster vorgenommen:

- *die elementaren Eigenschaften (Attribute und Zustandsgrößen) der Objekte werden auf geeignete Datenobjekte bzw. auf definierte elementare Objekte abgebildet,*
- *die nicht-elementaren Komponente des Objekte werden durch Komponentenobjekte dargestellt,*
- *das agierende Verhalten der Objekte wird durch eine Steuerprozedur nachgebildet, das reagierende Verhalten der Objekte durch Operationen, für die je nach Bedarf Eingangs- und Ausgangsgrößen definiert werden.*

Eine Klasse besteht im Allgemeinen aus:

- *einem Spezifikationsteil, in der die strukturellen Komponenten (Daten- bzw. Komponentenobjekte) und die zugeordneten Operationen spezifiziert sind,*
- *einem Ausführungsteil, in dem die spezifizierten Operationen implementiert sind.*

Eine Klasse verkörpert:

- *ein logisches Modell für die Menge von gleichartigen Objekten,*
- *eine physikalische Schablone, welche die Struktur des Speichers festlegt, der einem Objekt zur Speicherung von aktuellen Werten während seiner Lebenszeit zur Verfügung steht,*
- *einem Typ, der ein Objekt der Klasse charakterisiert und bestimmt, ob das Objekt mit anderen Objekten hinsichtlich seiner Schnittstelle kompatibel ist.*

Aus einer Klasse können nach Bedarf beliebig viele konkrete Objekte eines Typs abgeleitet werden. Klassen sind erweiterbar und modifizierbar. Sie können also auch dann wiederverwendet werden, wenn die aus ihnen abgeleiteten Objekte nicht in vollem Umfang den vorgegebenen Anforderungen entsprechen.

[312] *Anmerkung:*
Aus sprachwissenschaftlicher Sicht ist ein Objekt deshalb keine Instanz einer Klasse, weil dies eine Fehlübersetzung des englischen Begriffes Instance ist. Tatsächlich ist die richtige Übersetzung des englischen Begriffes Instance etwa Objekt, Exemplar oder Ausprägung. Den abgeleiteten Verbformen wie instantiieren oder ähnlichen ist dementsprechend ausprägen, Objekt erzeugen oder Objekt anlegen vorzuziehen. Der Begriff Instanz hat sich im Kontext der Objektorientierung allerdings soweit als Synonym des englischen Begriffes Instance etabliert, dass dieser rein formalen Nuance heutzutage keine Bedeutung mehr beizumessen ist. In der Praxis ist die Aussage Ein Objekt ist eine Instanz einer Klasse daher nicht als fachlich falsch anzusehen. Dieses Paradigma wird durch die Tatsache unterstützt, dass der Begriff Instanz in der Objektorientierung nicht vorbelegt ist, was die Eindeutigkeit der oben getroffenen Aussage hilfsweise unterstreicht.

Definition C.9 ***Grundeigenschaften objektorientierter (Programmier-) Sprachen nach [tec]***
Nach Booch [Boo94a] muss eine objektorientierte (Programmier-) Sprache die nachfolgenden vier Sprachelemente bzw. Spracheigenschaften aufweisen, damit sie als objektorientierte Sprache angesehen werden kann:

1. ***Abstraktion:***
   Die Möglichkeit, Objekte oder Klassen mit klar definierten Grenzen zu bilden.
2. ***Kapselung:***
   Kapselung ist das Komplementäre zur Abstraktion. In objektorientierten Sprachen muss es grundsätzlich möglich sein, Details wie zum Beispiel Attribute oder Methoden zu verstecken.
3. ***Modularität:***
   Ein Programm/System muss sich in einzeln klar abgegrenzte übersetzbare/simulierbare Einheiten aufteilen lassen.
4. ***Hierarchie:***
   Die Möglichkeit Klassenhierarchien bilden zu können (Vererbung) (siehe hierzu: 3.4.3).

Anmerkung zur Definition C.9:
Sprachen, die die ersten drei Eigenschaften (1-3) aufweisen, jedoch nicht die Hierarchiesierung (4) werden als *Objekt-basierte* (Programmier-) Sprachen bezeichnet.

Definition C.10 ***Grundeigenschaften objektorientierter (Programmier-) Sprachen nach Prof. Dr. M. Broy [PDB95]***
Typische objektorientierte Programmiersprachen unterstützen folgende Konzepte:

1. ***Datenkapselung*** *(Persistenz)*
2. ***Klassen und Vererbung***
3. ***Objekte und Instanzen*** *(dynamisches Kreieren neuer Objekte)*
4. ***Methodenentwurf und Nachrichtenaustausch***

Anhang D.

KOMP Systems Engineering Element-Konzept

Nachdem in den beiden Kapiteln 3.4.2 und 3.4.2.2 sowohl das *klassische* als auch das *erweiterte SE Element-Konzept* erläutert und anhand von einigen Beispielen ausführlich beschrieben wurde, folgt nun in diesem Kapitel die Erweiterung des Systems Engineering Element-Konzepts zur Modellierung und Bewertung von Kompatibilität in einem System – das „KOMP SE Element-Konzept“. Um in einem System Kompatibilität modellieren zu können, sind einige Erweiterungen und Ergänzungen des Element-Konzepts notwendig, z.B. die explizite Modellierung der Ein- und Ausgänge der Elemente, um ein System auf Kompatibilität untersuchen zu können. Ohne die hier vorgestellten Erweiterungen des SE Element-Konzepts ist es nicht möglich zu bewerten, ob ein System kompatibel ist.

In den nachfolgenden Unterpunkten dieses Kapitels wird das KOMP SE Element-Konzept beschrieben. Begonnen wird mit dem Aufbau und der Struktur des KOMP SE-Elements. Daran anschließend werden alle Änderungen und Erweiterungen gegenüber dem klassischen und dem erweiterten SE Element-Konzept erläutert und anhand von einfachen Beispielen vertieft.

Aufbau und Struktur des KOMP SE Elements
Das KOMP SE Element-Konzept ist eine Anpassung und Erweiterung des klassischen- bzw. des erweiterten Systems Engineering Element-Konzepts. In der Abbildung D.1 ist der generische Aufbau und die Struktur des KOMP SE-Elements dargestellt. Auf den ersten Blick hat sich gegenüber dem klassischen SE-Element (vgl. Abbildung: 3.10 auf Seite 118), so wie es im Kapitel „3.4.2 *Das Systems Engineering Element-Konzept – „Die Münchner Schule““* ab Seite 116 eingeführt wurde, nicht viel verändert. Lediglich die Begriffe *Attribut* und *Funktion* in den Abschnitten (3) und (4) wurden durch die Begriffe *Variablen* und *Methoden* ersetzt. Des Weiteren fällt auf, dass ein KOMP SE-Element kein Subsystem/Subelement mehr in (4) enthalten darf. Alle weiteren Änderungen und Erweiterungen gegenüber dem klassischen SE Element-Konzept sind auf dieser Abstraktionsebene nicht sichtbar.

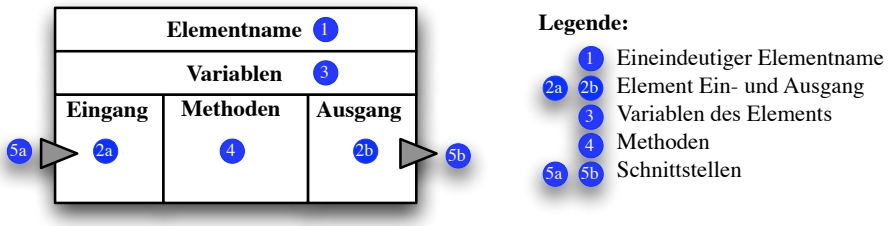


Abbildung D.1.: Darstellung des KOMP SE Elements

Unterschiede und Erweiterungen des klassischen bzw. des erweiterten objektorientierten SE Element-Konzepts von Dr.-Ing. L. Schrepfer und dem KOMP SE Element-Konzept
Die nachfolgende Aufzählung enthält die wesentlichen Ergänzungen und Erweiterungen des KOMP SE Element-Konzepts gegenüber dem klassischen- bzw. dem erweiterten Element-Konzept. In der Aufzählung sind vor allem die Unterschiede bzw. die Erweiterungen zu den beiden bestehenden Konzepten aufgeführt und anhand von einfachen Beispielen erklärt.

- **Umbenennung der Funktionen in Methoden sowie der Attribute in Variable**
  Um eine Verwechselung zwischen dem klassischen- bzw. dem erweiterten objektorientierten SE Element-Konzept und dem KOMP SE-Element zu vermeiden, werden die *Funktionen* des klassischen SE Element-Konzepts in *Methoden* und die *Attribute* in *Variablen* umbenannt.

  Die Umbenennung ist vor allem während der Entwurfsphase eines Systems/Elements wichtig, da im klassischen und erweiterten objektorientierten Systems Engineering Element-Konzept die Attribute und Funktionen des Elements keine ausgezeichneten Datentypen haben. Im KOMP SE-Element müssen alle Parameter[313] mit eindeutigen Datentypen versehen werden.

- **Ein KOMP SE-Element enthält keine Subsysteme**
  Eine Schachtelung der Elemente wie im klassischen SE Element-Konzept (vgl. Abschnitt 3.4.2.1 auf Seite 120) ist nicht erlaubt. Das enthalten sein eines Elements in einem anderen kann im KOMP SE nur durch explizite Modellierung der entsprechenden Abhängigkeiten (Ordnungsrelationen) zwischen den Elementen dargestellt werden.

- **Ein Element hat mindestens einen Eingang und mindestens einen Ausgang**
  Diese neu hinzugekommene Regel sorgen dafür, dass jedes Element mindestens einen definierten Ein- bzw. Ausgang besitzt. Ohne die Verwendung der Regeln ist es möglich, Elemente zu modellieren die sich nicht „naturkonform“ verhalten. So ist es

[313] Im KOMP SE Element-Konzept werden unter dem Begriff *Parameter* sowohl die Variablen in (3) als auch die Parameter einer Methode (4) zusammengefasst.

beispielsweise möglich, ein Element zu modellieren, das weder einen Ein- bzw. noch einen Ausgang hat. Solche Systeme kommen in der Natur nicht vor. Ein Element das keinen Eingang, jedoch einen Ausgang hat, wäre eine unerschöpfliche Energiequelle, während ein System mit einem Eingang, jedoch ohne einen Ausgang, sich wie ein Schwarzes Loch [Wik06k] verhalten würde. Forschungsergebnisse von Prof. Stephen Hawking haben jedoch gezeigt, dass selbst Schwarze Löcher Strahlung, die so genannte „Hawking-Strahlung“ [314], abgeben. Als Folgerung aus den beiden obigen Begründungen, ist es zwingend notwendig, dass ein Element mindestens einen Ein- und einen Ausgang haben muss.

- **Variablen und Methoden können mit einem „Sichtbarkeitsattribut“ versehen werden**
  Die Variablen und Methoden eines KOMP SE-Elements können mit einem optionalen Schlüsselwort **`private, protected`** oder **`public`**[315] versehen werden, das sowohl die Sichtbarkeit als auch den Zugriff auf einzelne Bestandteile des Elements reglementiert:

  – ***public*** – öffentlich sichtbar (Standard Einstellung):
  Die Variable bzw. Methode ist für alle anderen Elemente des Systems ohne Beschränkungen sichtbar und kann, sofern eine Flussrelation zwischen den Elementen existiert, beliebig aufgerufen und im Falle einer Variablen verändert[316] werden.

  Alle öffentlich sichtbaren Variablen und Methoden eines Elements werden sowohl in den Ein- als auch den Ausgang des Elements (die Schnittstelle) eingetragen.

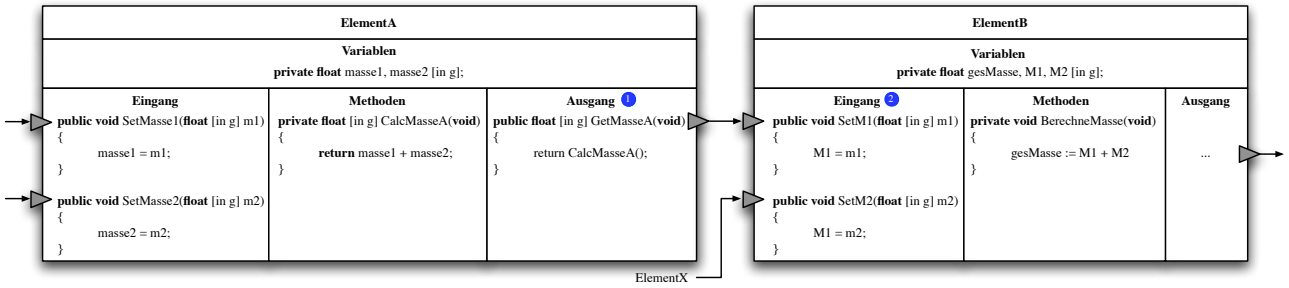


Abbildung D.2.: Zugriff auf öffentliche Methoden.

In der Abbildung D.2 sind zwei Elemente *ElementA* und *ElementB* dargestellt, die über zwei Relationen miteinander verbunden sind. Genauer: Die beiden Elemente sind mittels zweier Flussrelationen verbunden, die den Ausgang des *ElementsA* mit dem Eingang des *ElementsB* verbinden. Über diese beiden Flussrelationen tauschen die beiden Elemente Daten aus. Im *ElementeA* ist die Variable *public float masse [in g] = 20.0 [in g];* und die Methode *public float [in g] GetMasse(void) public* deklariert und definiert. Im *ElementB* sind sämtliche Variablen und Methoden *private*.

  – ***protected*** – nur in abgeleiteten Elementen bzw. in Subelementen sichtbar:
  Die Variable bzw. Methode ist nur innerhalb des Elements, in davon abgeleiteten Elementen bzw. in Subsystemen des Elements sichtbar und veränderbar. Für alle anderen Elemente ist das Attribut bzw. die Methode nicht sichtbar.

  – ***private*** – nur innerhalb des Elements sichtbar:
  Die Variable bzw. Methode ist nur innerhalb des Elements sichtbar. Für kein anderes Element des Systems ist die Variable bzw. die Methode sichtbar. Dies ist die stärkste Einschränkung einer Variablen bzw. einer Methode.

  Wird kein spezielles Sichtbarkeitsattribut angegeben, so wird stets *public* (ohne Beschränkung sichtbar) angenommen[317].

- **Parameter und Variablen haben stets einen eindeutigen (Daten-) Typ[318]**
  Ein Parameter bzw. eine Variable eines Elements kann stets nur genau einem (Daten-) Typ angehören. Es ist nicht erlaubt, dass ein Parameter/Variable keinen oder mehr als einen (Daten-) Typ besitzt. Außerdem kann der (Daten-) Typ des Parameters/Variable nach der Deklaration im Variablen Teil des Elements nicht mehr verändert werden. Durch dieses Axiom ist sichergestellt, dass nach der Deklaration des Parameters/Variablen dieser nachträglich nicht mehr verändert werden kann.

  Im KOMP SE Element-Konzept wird auf die interne Repräsentation der Datentypen nicht weiter eingegangen. Sollte sie jedoch für die Kompatibilität eines Systems wichtig sein, so müssen die speziellen Eigenschaften der Zielplattform als Parameter modelliert werden.

  Beispiel 84: Eigenschaften eines Elements mit eindeutigem (Daten-) Typ
  Die Abbildung D.3 auf der nächsten Seite zeigt das Element *Element A* mit vier Variablen, einer Eingangs- und einer Ausgangsfunktion, sowie einer Methode in KOMP Systems Engineering Element Darstellung.

[314] Nähere Informationen zu Stephen Hawking bzw. der „Hawking Strahlung“ finden Sie unter [Wik06m].
[315] Im englischen Sprachraum (und in Anlehnung an C++/JAVA) werden oft die Begriffe „public“, „protected“ und „private“ für „öffentlich“, „geschützt“ und „privat“ verwendet.
[316] Eine Variable kann durch dritte nur verändert werden, wenn sie *public* und *var* deklariert ist. Ist die Variable *const* deklariert, so kann sie nur gelesen werden.
[317] Dies wird aus Gründen der Abwärtskompatibilität zum klassischen SE Element-Konzept so definiert.
[318] In der Informatik wird dieses SE-Axiom oft auch als „Typsicherheit“ bezeichnet.

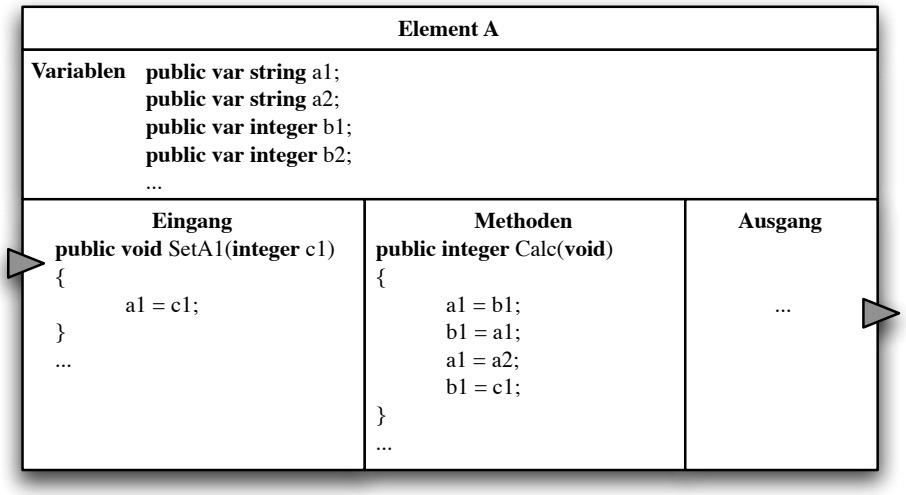


Abbildung D.3.: KOMP SE-Element mit Variablen- und Schnittstellendefinition.

Das Element *Element A* enthält vier öffentliche zugängliche typisierte Variablen:

```
1 public var string a1;
2 public var string a2;
3 public var integer b1;
4 public var integer b2;
```

Die beiden Variablen `a1` und `a2` sind vom Datentyp **`string`**, während die beiden Variablen `b1` und `b2` vom Datentyp **`integer`** sind. Im nachfolgenden Listing ist der Rumpf der Methode **`public integer`** `Calc(`**`void`**`)` dargestellt.

```
1  // Zuweisungsfehler:
2  // Einer Zeichenkette kann keine Zahl zugewiesen werden!
3  a1 = b1;
4
5  // Zuweisungsfehler:
6  // Einer Zahl kann keine Zeichenkette zugewiesen werden!
7  b1 = a1;
8
9  // Keine Fehler!
10 a1 = a2;
11 b1 = c1;
```

Im obigen Listing sind vier Zuweisungen enthalten wovon zwei eine Datentypverletzung verursachen. In der Zeile 3 wird beispielsweise versucht einer Zeichenkette vom Datentyp **`string`** eine Zahl vom Datentyp **`integer`** zuzuweisen. Dies führt zu einem Kompatibilitätsfehler, da im Allgemeinen die Datentypen **`string`** und **`integer`** nicht zuweisungskompatibel sind[319]. Die beiden Zuweisungen in den Zeilen 10 und 11 sind in Ordnung da hier die Datentypen kompatibel zueinander sind. □

Anmerkung:
Die Typisierung der Parameter/Variablen ist universell nutzbar, d.h. es können nicht nur Informatik spezifische Datentypen abgebildet werden, sondern auch Typen aus der Mechanik und Elektrotechnik.

- **Parameter und Variablen können mit SI- oder Informatik-Einheiten und Teilern versehen werden**
  Alle Parameter/Variablen eines Elements können entweder mit einer genormten physikalischen Einheit, den so genannten SI-Einheiten, oder einer in der Informatik üblichen Einheit versehen werden.

  – *SI-Einheiten*
    Im KOMP SE Element-Konzept dürfen alle sieben durch das BIPM genormten Basiseinheiten und Basisteiler und deren Kombinationen als Suffixe für Parameter/Variablen verwendet werden. Es ist jedoch nicht zwingend notwendig, dass ein Parameter/Variablen eine Einheit zugeordnet ist. Für die Modellierung von Kompatibilität ist es jedoch von Vorteil, wenn ein Parameter/Variable eine eineindeutige Einheit hat, da dadurch Einheitenfehler vermieden werden können. In der Tabelle 2.2 auf Seite 70 sind die vom SI Gremium genormten sieben Basiseinheiten aufgelistet, aus denen sich alle anderen Einheiten als Kombination der Basiseinheiten ableiten lassen.

[319] Siehe hierzu die beiden Tabellen 2.6 auf Seite 87 bzw. 2.7 auf Seite 88.

Beispiel 85: Zuweisung von Variablen mit und ohne Einheit
Variablendefinition ohne Einheiten- und Teilerdeklaration:

```
1  float masse1 = 10.0;  // Masse in kg
2  float masse2 = 20.0;  // Masse in g
3  float ges_masse = masse1 + masse2;
```

In der Zeile 3 wird die Gesamtmasse *ges_masse* aus den beiden Teilmassen *masse1* und *masse2* berechnet. Dabei passiert ein Fehler bei der Zuweisung da „vergessen" wurde, dass beide Massen unterschiedliche Einheiten haben. Die Information, dass *masse1* in kg und *masse2* in g angegeben ist, steht nur als Kommentar in den beiden Zeilen 1 und 2. Es wird also das falsche Ergebnis *30* berechnet, anstatt *10.2*. Dieser Fehler kann vermieden werden, wenn zu jedem Parameter ein eineindeutige Einheit und Teiler mit angegeben und bei der Zuweisung überprüft wird.
Variablendefinition mit expliziter Einheiten- und Teilerdeklaration:

```
1  float masse3 [in kg] = 10.0 [in kg];
2  float masse4 [in g] = 20.0 [in g];
3
4  // Fehler: Falscher Einheitenteiler!
5  float ges_masse2 [in kg] = masse3 + masse4;
```

In diesem Beispiel kann der obige Zuweisungsfehler aus Zeile 3 nicht mehr vorkommen weil jede Variable (Zeile 1 und 2) eine eineindeutige Einheit und einen Teiler besitzt. Der Zuweisungsfehler in Zeile 5 kann durch die Erweiterung der Parameter/Variablen um Einheiten und Teiler entdeckt und beseitigt werden. □

– *Informatik Präfixe und Einheiten*
Ähnlich zu den genormten physikalischen SI-Einheiten gibt es in der Informatik ebenfalls „ein" genormtes Einheitensystem das verschiedene Präfixe definiert. Im Gegensatz zu den strengen physikalischen SI-Normen gibt es in der Informatik zwei unterschiedlich Einheitenpräfixsysteme. Zum einen die aus der SI-Norm bekannten Präfixe K(ilo), M(ega), G(iga), T(era) usw., die als gemeinsame Basis das metrische Zahlensystem (10^x) haben. Zum anderen die speziell für die Informatik neu eingeführten Präfixe Kibi [Ki], Mebi [Mi], Gibi [Gi] und Tebi [Ti]. Diese Präfixe beziehen sich auf das binäre Zahlensystem (2^x), so wie es in der Elektrotechnik und Informatik gebräuchlich ist. In der Literatur und vor allem in der Werbung hat sich das metrische Einheitensystem trotz der Ungenauigkeit auch in der Informatik durchgesetzt.

Beispiel 86: Unterschied zwischen metrischem und binären Präfixen
In physikalisch-technischen Systemen repräsentiert z.B. das Präfix [k] Kilo den Wert $10^3 = 1000$. Im binären System bedeutet das gleiche Präfix [k] jedoch $2^{10} = 1024$. Um Verwirrungen und Inkompatibilitäten bei den Präfixen zu vermeiden, sollte in der Informatik das Präfix *Ki* statt *k* verwendet werden. □

- **Zwingende und optionale Element-Schnittstellen**
Ein SE-Element kann sowohl zwingende als auch optionale Schnittstellen besitzen. Eine zwingende Schnittstelle wird mittels eines grau ausgefüllten Dreiecks dargestellt, während eine optionale Schnittstelle nicht ausgefüllt dargestellt wird. Im Unterschied zu einer zwingenden Schnittstelle muss ein optionale Schnittstelle nicht angeschlossen, also mit einem Flusspfeil verbunden, sein.

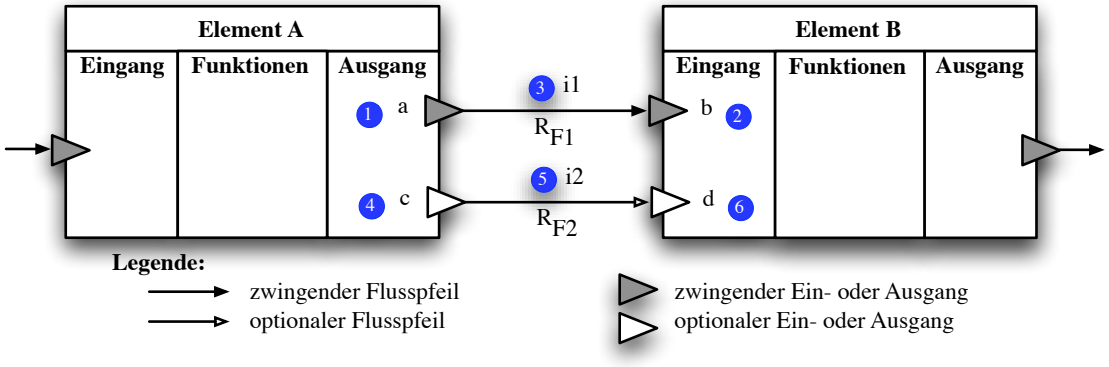


Abbildung D.4.: Zwei Elemente mit zwingenden- und optionalen Flussrelationen sowie zwingenden- und optionalen Element-Schnittstellen.

In Abbildung D.4 ist ein System, das aus zwei Elementen *Element A* und *Element B* besteht, dargestellt. Die beiden Elemente verfügen sowohl über eine zwingende- (1,2), als auch über eine optionale Schnittstelle (4,5). Im obigen Fall ist sowohl die zwingende als auch die optionale Schnittstelle mittels eines Flusspfeils verbunden. Der Flusspfeil (5) der optionale Schnittstelle könnte entfernt werden, ohne dass es zu einer (Schnittstellen-) Inkompatibilität kommt. Dies gilt nicht für den zwingenden Flusspfeil (3). Wird er entfernt, so sind die beiden zwingenden Schnittstellen (1) und (2) nicht mehr verbunden und des kommt zu einem Schnittstellenfehler.

Anmerkung:
Eine ausführliche Beschreibung des Optionalitätsprinzips finden Sie unter „3.6.3 *Das Optionalitätsprinzip der (U)CML*“ ab Seite 218.

- **Flussrelationen können zwingend oder optional sein**
  In einem System kann es sowohl zwingende (3) als auch optionale (5) Flussrelationen geben. Eine zwingende Flussrelation (3: R_{F_1}) verbindet genau eine zwingende Ausgangsschnittstellen (1: a) mit einer zwingenden Eingangsschnittstelle (2: b), also $R_{F_1 : a \xrightarrow{i1} b}$. Ebenso verbindet eine optionale Flussrelation (5: R_{F_2}) zwei optionale Schnittstellen miteinander ($R_{F_2 : c \xrightarrow{i2} d}$).
  Des Weiteren ist es möglich, eine optionale Eingangs- oder Ausgangsschnittstelle mit einer zwingenden Eingangs- oder Ausgangsschnittstelle mittels einer (zwingenden) Flussrelation zu verbinden. In diesem Fall wird sowohl die optionale Schnittstelle als zwingende Schnittstelle als auch die optionale Flussrelation als zwingende gezeichnet. Wird die zwingende Flussrelation wieder gelöst, so kehrt die ursprünglich optionale Schnittstelle wieder in diesen Zustand zurück.

- **Erweiterung des Schnittstellenbegriffs**
  Für die Kompatibilitätsmodellierung und -bestimmung müssen die Schnittstelleneigenschaften, also die Ein- und Ausgänge eines Elements, erweitert werden. Wie bereits in den beiden Abbildungen D.2 bzw. D.3 dargestellt, sind sämtliche von außerhalb des Elements erreichbaren Variablen und Methoden, in den Ein- und Ausgängen des jeweiligen Elements gespeichert. So kann zum Beispiel auf die Methode `public void SetMasse1(float [in g] m1)` (Abb.: D.2) von außerhalb des Elements zugegriffen werden, während auf die private Methode `private float [in g] CalcMasseA(void)` von außerhalb des Elements nicht zugegriffen werden kann.

  Im KOMP SE Element-Konzept gibt es im Gegensatz zum erweiterten Element-Konzept, keine inneren Element-Schnittstellen. Aufgrund der Tatsache, dass es im KOMP SE Element-Konzept keine inneren Elemente (Subsysteme) mehr gibt, sind die von Dr.-Ing. L. Schrepfer eingeführten inneren Schnittstellen obsolet.

- **Kompatibilitäts-Regelwerk**
  Um die Kompatibilität eines Systems bestimmen zu können, ist es notwendig, Regeln zu definieren, mit deren Hilfe genau definiert wird, wann zwei Elemente kompatibel zueinander sind. In der folgenden Aufzählung sind einige Kompatibilitätsregeln exemplarisch aufgelistet, die in den obigen Modellen verwendet worden sind.

  – Ein Element muss mindestens einen Ein- und Ausgang haben.
  – Optionale Ein- bzw. Ausgänge müssen im Gegensatz zu den zwingenden Ein- und Ausgängen nicht angeschlossen sein.
  – Flussrelationen verbinden stets Ausgänge mit Eingängen von Elementen.
  – Eine implizite Datentyp Konvertierung ist nicht erlaubt.
  – Einheiten von Quelle und Senke müssen übereinstimmen.
  – usw.

  Anmerkung:
  Eine ausführliche Beschreibung der Kompatibilitätsregeln für eingebettete softwarelastige Systeme finden Sie im Kapitel „3.6.5 *(U)CML-Sprach- und Kompatibilitätsregelwerk*“ ab Seite 236.

Anhang E.

Weiterführende Techniken – (U)CML

In diesem Abschnitt werden einige wichtige Ergänzungen zur im Kapitel „3.6 *Einführung in die Kompatibilitätsmodellierungssprache (U)CML*“ ab Seite 153 vorgestellten graphischen Kompatibilitätsmodellierungssprache (U)CML vorgestellt und erläutert. Begonnen wird mit der knappen Einführung der MIDL – ***M**odell and **I**nterface **D**escription **L**anguage* – einer textuellen Notationssprache zur textuellen Modellierung von Systemen in (U)CML. Im Anschluss daran wird die CCL – ***C**ompatibility **C**onstraint **L**anguage* – vorgestellt, mit deren Hilfe sowohl Bedingungen in (U)CML als auch einfache funktionale Programme innerhalb des Editors *(U)CML-ed* ausgeführt werden können. Abgeschlossen wird diese Kapitel mit der beispielhaften Überführung eines UML/UML2-Diagramms in das entsprechende Flussdiagramm in (U)CML.

E.1. Einführung in MIDL

In (U)CML ist es möglich, ein System bestehend aus Paketen, Komponenten, Flusspfeilen usw. auf zwei unterschiedliche Arten zu modellieren. Zum einen bietet die (U)CML bzw. der dazugehörige graphische Editor *(U)CML-ed* eine graphische Oberfläche mit dessen Hilfe sowohl die unterschiedlichen Sprachelemente der (U)CML gezeichnet als auch das Regelwerk editiert werden kann. Und zum anderen kann in (U)CML ein System vollständig textuell in der Modellierungssprache MIDL beschrieben werden. Die MIDL ähnelt dabei, der im Kapitel „2.3.2.2 *Erweiterung der Eigenschaftsdeklaration von Objekten/Klassen für die Modellierung von Kompatibilität*“ ab Seite 67 vorgestellten erweiterten Notation zur Beschreibung der Eigenschaften und Methoden eines Objekts/Klasse. Das nachfolgende Listing zeigt einen kleinen Ausschnitt aus der formalen Beschreibungssprache MIDL zur textuellen Modellierung eines Pakets/Komponente.

```
1  // Datentypen
2  TYPE_NAME = eineindeutiger Identifikator
3  TYPE_ZUORDNUNG = Elektrotechnik, Mechanik oder Software
4  TYPE = z.B. int, bool, etc. (evtl. Objekt)
5  CONDITION = Vor- und Nachbedingungen (OCL)
6  RANGE = Wertebereich
7  UNIT = TYPE_NAME {DEC_TEILER | INF_GROESSE}
8  FUN_NAME = STRING (eineindeutige Zeichenkette)
9  PARAMETER_LISTE = LET:(var TYPE_NAME:TYPE)
10
11 // Definition Paket/Komponente
12 Paket/Komponente:
13 {
14   def: (
15     {
16       {FUN_NAME;}
17       LET:(var/const TYPE_NAME:TYPE);
18       INV/PRE/POST:(CONDITION);
19       RANGE:[TYPE_NAME:ARRAY];
20       UNIT:(TYPE_NAME:{DEC_TEILER | INF_GROESSE} EINHEIT)
21     }+
22     |
23     {
24       FUN:(FUN_NAME:[PARAMETER_LISTE*])
25     }
26   )
27 }
```

Nach dieser knappen Einführung in MIDL, folgt nun die Anwendung der MIDL zur Modellierung einer Komponente. Das folgende Listing zeigt auszugsweise die textuelle Notation der Komponente *Komp01*, aus der Abbildung E.1 auf der nächsten Seite, in MIDL. Dabei fällt auf, dass in der textuellen Notation der Komponente die gleichen Informationen enthalten sind wie in der graphischen Beschreibung der Komponente. In der graphischen Repräsentation der Komponente sind die Informationen jedoch teilweise in den Stecker-/Buchsenbeschreibungsfelder enthalten und nicht in der Beschreibung der Komponente. Die graphische Darstellung ist jedoch im Gegensatz zur textuellen Notation für den Benutzer einfacher zu erlernen.

```
1  Komp01:
2  {
3    def: (
4      LET:(I1:long, I2:long, O1:long);
5      PRE:(I1>=0 && I1<=10 && I2>=0 && I2<=10);
6      POST:(O2<=20);
7      RANGE:(I1:[0...10], I2:[0...10], O1:[0...20]);
8      UNIT:(I1:long, I2:long, O1:long)
9    )
10 }
```

Anmerkung:
Eine genaue Beschreibung der MIDL finden Sie unter [BK05].

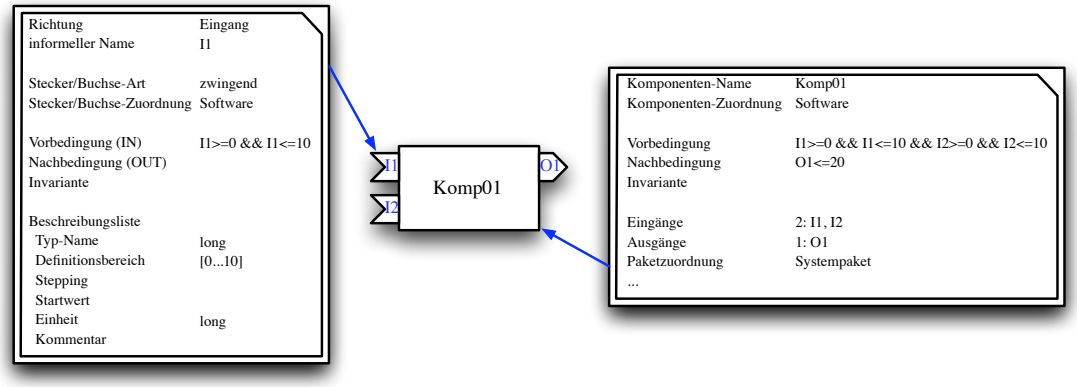


Abbildung E.1.: Darstellung der Komponente *Komp01* in (U)CML.

E.2. Einführung in CCL

Mit Hilfe der ***C**ompatibility **C**onstraint **L**anguage* (CCL) können in einem (U)CML-Modell sowohl die Vor- und Nachbedingungen (inklusive der Invarianten) textuell beschrieben, als auch einfache Funktionen innerhalb des Editors *(U)CML-ed* programmiert werden. Die Abbildung E.2 zeigt einen kleinen Auszug aus einem in (U)CML modellierten Modell. In diesem Modell muss der Ausgangsstecker *O1* der Komponente *Stromversorgung* mit dem Eingangsbuchse *I1* der Komponente *Komp01* verbunden sein. Des Weiteren muss der Ausgangsstecker *O2* der Komponente *Stromversorgung* mit dem Eingangsbuchse *I1* der Komponente *Komp02* verbunden sein, damit das System fehlerfrei funktioniert.

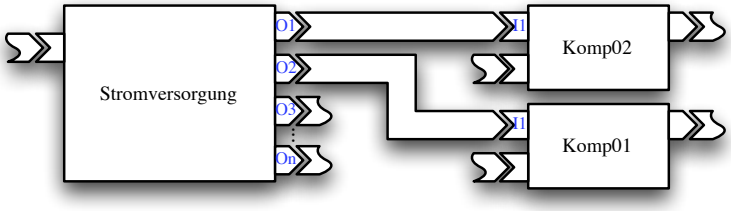


Abbildung E.2.: Auszug aus einem Systemmodell in (U)CML.

Zusätzlich darf die maximale Ausgangslast der Komponente *Stromversorgung* von 5A nicht überschritten werden. Um beide Anforderungen zu erfüllen, kann jeweils ein CCL Ausdruck angegeben werden. Mit Hilfe des Ausdrucks:

```
1  context Stromversorgung
2  pre
3  {
4    let Plug destplug1=findDestination(plug("O1"));
5    let Plug destplug2=findDestination(plug("O2"));
6
7    destplug1.component==component("Komp02");
8    destplug1.name=="I1";
9
10   destplug2.component==component("Komp01");
11   destplug2.name=="I1";
12 }
```

kann beispielsweise sichergestellt werden, dass der Ausgang *O1* der Komponente *Stromversorgung* mit dem Eingang *I1* der Komponente *Komp01* verbunden ist. Ist die Verbindung nicht vorhanden, so würde bei der Durchführung des Kompatibilitätstests (z.B. im *(U)CML-ed*) eine Fehlermeldung ausgegeben. Auf den ersten Blick scheint hier jedoch der CCL Ausdruck „überflüssig“ zu sein, da ja bereits eine Verbindung zwischen den beiden Komponenten mittels eines Flusspfeils hergestellt worden ist. Soll jedoch sichergestellt werden, dass der Ausgang *O1* der Komponente *Stromversorgung* auf jeden Fall mit dem Eingang *I1* der Komponente *Komp01* verbunden ist, so kann zusätzlich zur Flussverbindung ein CCl Ausdruck hinterlegt werden, der sicherstellt, dass diese Verbindung tatsächlich existiert. Denn in der Flussdarstellung könnte beispielsweise auch der Ausgang *O1* mit dem Eingang der *I1* der Komponente *Komp02* verbunden sein. Auf den Ersten Blick wäre das Systemmodell dann ebenfalls korrekt, sofern sämtliche Eigenschaften zwischen dem Sender und dem Empfänge übereinstimmen. Durch den CCL Ausdruck ist jedoch sichergestellt, dass die beiden „richtigen“ Komponente miteinander verbunden sind. Dadurch wird die Sicherheit innerhalb des Modells erheblich verbessert.

Zusätzlich kann im *(U)CML-ed* mit Hilfe der CCL einfache Berechnungen angestellt werden. Dürfen beispielsweise alle angeschlossenen Komponenten an der Stromversorgung maximal 5A Strom ziehen, so kann der aktuelle Verbrauch der angeschlossenen Komponenten durch den folgenden CCL Ausdruck berechnet werden.

```
1 context Stromversorgung
2 pre
3 {
4   let Integer valueOfPlug(Plug p):=p.value;
5   sum(plug("O.*),valueOfPlug)<5;
6 }
```

E.3. Von UML / UML 2 zu (U)CML

Eine (U)CML-Komponente entspricht im Wesentlichen einer UML/UML2-Klasse bzw. der Instanz einer Klasse (Objekt), da eine (U)CML-Komponente sowohl quantitative als auch qualitative Eigenschaften enthalten kann. Um die in einer UML/UML2-Klasse/Objekt enthalten Informationen in eine (U)CML-Komponente zu transferieren, müssen lediglich die in der UML/UML2-Klasse/Objekt enthaltenen Attribute und Methoden in ihre jeweiligen Entsprechungen in (U)CML überführt werden. Die Abbildung E.3 zeigt das grundsätzlich Vorgehen bei der Transformation der Informationen einer UML/UML2-Klasse/Objekt in eine (U)CML-Komponente.

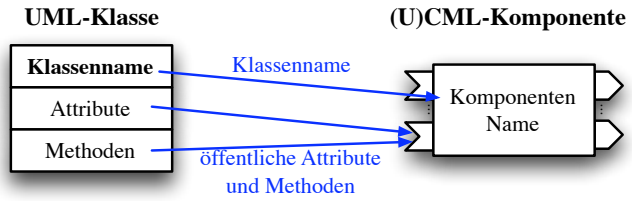


Abbildung E.3.: Allgemeines Vorgehen bei der Überführung einer UML/UML2-Klasse/Objekt in eine (U)CML-Komponente.

Dabei wird der Klassenname direkt als Komponentenname verwendet, während die Attribute und Methoden der UML/UML2-Klasse/Objekt in Stecker und Buchsen der (U)CML-Komponente überführt werden. Im Allgemeinen werden dabei für jedes öffentliche Attribut der UML/UML2-Klasse/Objekt ein eigener Stecker/Buchse erzeugt, je nachdem, ob das Attribut der Klasse von außerhalb der Klasse gelesen oder geschrieben werden soll. Genauso wird mit den Methoden der UML/UML2-Klasse verfahren. Auch hier werden wieder die öffentlichen Methoden in entsprechende Kommunikationsstecker übersetzt. Die Abbildung E.4 zeigt die Transformation einer UML/UML2-Klasse in eine entsprechende (U)CML-Komponente.

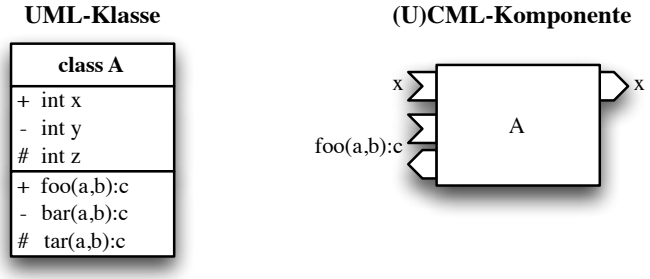


Abbildung E.4.: Transformation einer UML/UML2-Klasse in eine (U)CML-Komponente.

Zusätzlich zur Überführung der Klassen/Objekte müssen in (U)CML die Verbindungen zwischen den UML/UML2-Klassen/Objekten mit Hilfe der (U)CML-Flusspfeile hergestellt werden. In einem UML/UML2-Klassendiagramm sind jedoch meistens die Verbindungen zwischen den Klassen/Objekten des Systems nicht explizit modelliert und dargestellt. Aus diesem Grund können die Verbindungen nicht direkt aus dem UML/UML2-Klassendiagramm in das (U)CML-Flussdiagramm überführt werden.

Soll zusätzlich die Vererbungsstruktur der in einem UML/UML2-Klassendiagramm enthaltenen Klassen in die Komponentenstruktur innerhalb eines (U)CML-Flussdiagramm überführt werden, so muss die dafür notwendige Information aus dem Klassendiagramm extrahiert und spezielle Komponenten daraus erzeugt werden, weil es zum gegenwärtigen Zeitpunkt in (U)CML keine Vererbungsbeziehungen gibt. Die Abbildung E.5 auf der nächsten Seite zeigt beispielhaft, wie aus einer Vererbungsbeziehung zwischen zwei Klassen eine Komponente modelliert werden kann.

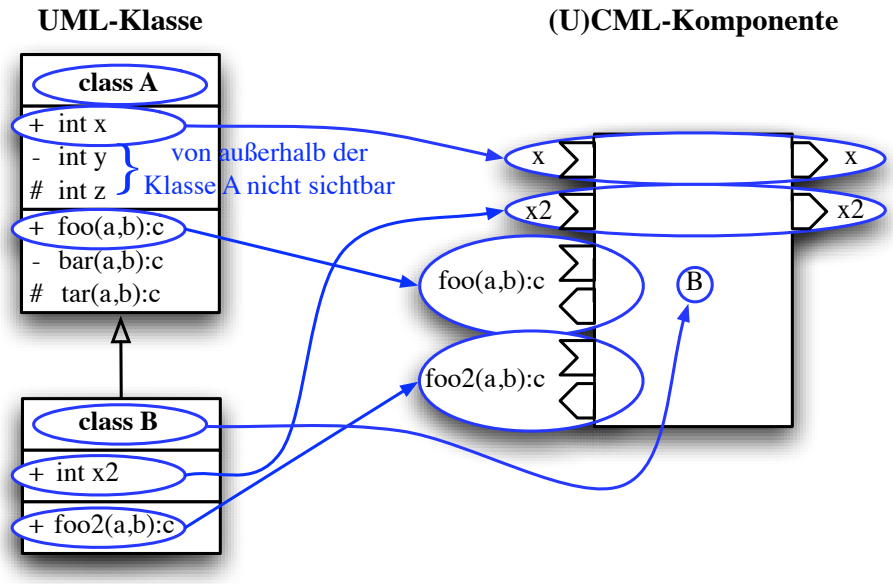


Abbildung E.5.: Überführung einer UML/UML2-Vererbungsbeziehung in eine (U)CML-Komponente.

Anhang F.

Modellierung von Signalen

F.1. Modellierung von elektrotechnischen Signalen

Der wohl wichtigste elektrische Signalverlauf bzw. die wesentlichste Signalart, wenn es um die Modellierung und Beschreibung von eingebetteten softwarelastigen Systemen geht, ist das so genannte Rechtecksignal. Mit Hilfe einer Folge von Rechtecksignalen werden in den meisten eingebetteten Systemen digitale Werte von einer Baugruppe zu einer anderen übertragen. Dabei repräsentieren die elektrischen Rechtecksignale direkt die entsprechend kodierten digitalen (binären) Zahlen. In den meisten Systemen stellt dabei der Signalpegel „high“ die logische eins, während der Signalpegel „now“ den logischen Wert Null darstellt. Auch eine Vertauschung der beiden Werte sowie der damit verbundenen Signale ist möglich. In diesem Fall spricht man meistens von der umgekehrten-, inversen- oder Eins-Logik. Die folgende Abbildung F.1 zeigt links den Signalverlauf eines Rechtecksignals sowie die dazugehörige Transformation des elektrischen Signalpegels in ein entsprechendes logisches Signal (rechts).

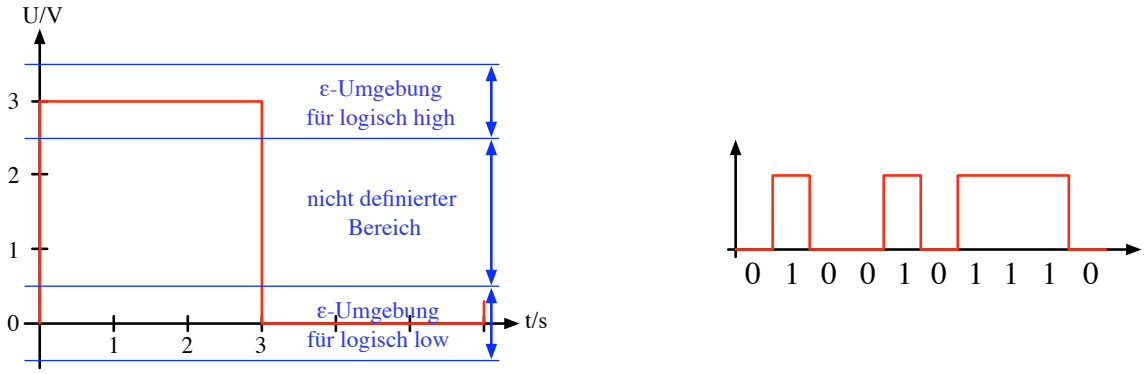


Abbildung F.1.: Transformation eines Rechtecksignals.

Auch für die Transformation eines elektrischen Rechtecksignals in ein entsprechendes binäres Digitalsignal kommt wieder die schon im Kapitel *„Dynamisches projektspezifisches Regelwerk“* eingeführte ϵ Schreibweise zum Einsatz. Liegt das Signal, genauer der Signalwert (Amplitude) innerhalb der oberen ϵ Umgebung, so wird dieser Wert als logische eins interpretiert. Im unteren Bereich als logische Null. Alle Werte zwischen der oberen- sowie der unteren Schranke werden als (Signal-) Fehler interpretiert da sie weder als logische Null oder eins aufgefasst werden können.

Bis jetzt wurde stets angenommen, dass ein elektrisches Rechtecksignal eine eindeutige Rechteckform, wie beispielsweise in der obigen Abbildung F.1 dargestellt, besitzt. In der Natur hingegen gibt es keine solchen ideal geformten Rechtecksignale – hier entstehen Rechtecksignale aus der Überlagerung von unendlich vielen einzelnen Sinus-Schwingungen. Die folgende Abbildung F.2 auf der nächsten Seite zeigt den Aufbau eines Rechtecksignals bestehend aus unterschiedlichen Sinus-Schwingungen mit unterschiedlichen, stetig anwachsenden Frequenzen sowie das „ideale Rechtecksignal“, an das sich die Summe der einzelnen Sinus-Schwingungen immer mehr annähert.

Mathematisch kann eine Rechteckschwingung als eine unendliche Summe von einzelnen Sinus-Schwingungen beschrieben werden. Die folgende Formel beschreibt mathematisch ein Rechtecksignal:

$$f(x) = \frac{4}{\pi} * \sum_{n=1}^{\infty} \sin\Big(\frac{(2*n-1)*x}{2*n-1}\Big)$$

Nach dieser Formel wurde auch das Rechtecksignal aus der Abbildung F.2 berechnet. Es setzt sich aus den folgenden Einzelschwingungen zusammen:

Grundschwingung: $f(x) := \frac{4}{\pi} * \sin(x)$
1. Oberschwingung: $f(x) := \frac{4}{\pi} * (\sin(x) + \frac{1}{3} * \sin(3*x))$
2. Oberschwingung: $f(x) := \frac{4}{\pi} * (\sin(x) + \frac{1}{3} * \sin(3*x) + \frac{1}{5} * \sin(5*x))$
3. Oberschwingung: $f(x) := \frac{4}{\pi} * (\sin(x) + \frac{1}{3} * \sin(3*x) + \frac{1}{5} * \sin(5*x) + \frac{1}{7} * \sin(7*x))$
4. Oberschwingung: $f(x) := \frac{4}{\pi} * (\sin(x) + \frac{1}{3} * \sin(3*x) + \frac{1}{5} * \sin(5*x) + \frac{1}{7} * \sin(7*x) + \frac{1}{9} * \sin(9*x))$
23. Oberschwingung: $f(x) := \frac{4}{\pi} * (\sin(x) + \frac{1}{3} * \sin(3*x) + \frac{1}{5} * \sin(5*x) + \ldots + \frac{1}{23} * \sin(23*x))$

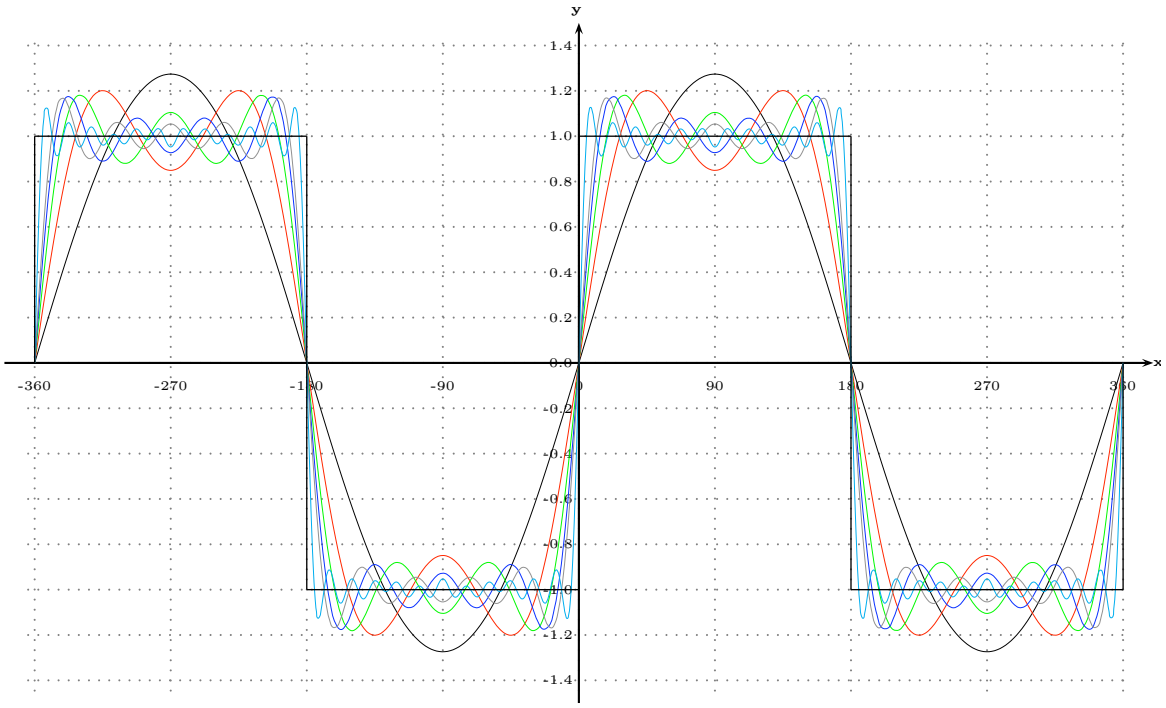


Abbildung F.2.: Rechtecksignal bestehend aus mehreren Sinus-Schwingungen.

Zusammen bilden sie das obige Rechtecksignal. Jedoch auch dieses Rechtecksignal entspricht nicht den realen Gegebenheiten innerhalb eines Systems. Aufgrund von äußeren Einflüssen, wie beispielsweise der Dämpfung einer Leitung, unterscheidet sich der reale Signalverlauf zum Teil erheblich von dem oben beschriebenen Ideal. Die Abbildung F.3 zeigt einen möglichen realen Signalverlauf (rot) eines Rechtecksignals bzw. das dazugehörige ideale Rechtecksignal (blau).

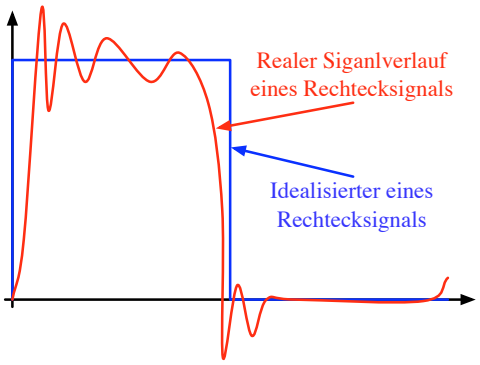


Abbildung F.3.: Gegenüberstellung: Realer- idealer Signalverlauf eines realen Rechtecksignals.

Innerhalb eines Modells werden jedoch stets alle elektrischen Signale als ideal angenommen um sie einfach modellieren und beschreiben zu können. Für die Kompatibilitätsmodellierung und -beschreibung hingegen muss das reale Signal so genau wie möglich beschrieben werden, um reale Inkompatibilitäten von elektrischen Signalen identifizieren zu können. Aus diesem Grund ist hier die Nutzung der im Kapitel „*Dynamisches projektspezifisches Regelwerk*“ eingeführten ϵ-Umgebung besonders wichtig.

Erweiterung auf nicht kontinuierliche elektrische Signale
Ein Beispiel für ein elektrisches Bauteil das eine so genannte „Sprungstelle“ enthält und somit keinen kontinuierlichen Signalverlauf besitzt, ist der so genannte Schmitt-Trigger[320]. Die Abbildung F.4 auf der nächsten Seite zeigt links das genormte Schaltzeichen des Schmitt-Triggers sowie rechts den dazugehörigen Signalverlauf. Der Schmitt-Trigger erzeugt aus einem nahezu beliebig geformten elektrischen Eingangssignal wieder ein Rechtecksignal am Ausgang. Dazu enthält der Schmitt-Trigger zwei feste Grenzwerte. Eingangssignale die einmal über der oberen Grenze liegen werden solange als logische eins interpretiert, bis sie die untere Grenze unterschreiten. Ab diesem Zeitpunkt wird dann das Ausgangssignal auf logisch Null festgelegt. Dabei hängen die Höhe des Eingangs- bzw. Ausgangssignals (Pegel) vom verwendeten Schmitt-Trigger-Typ ab. Für TTL Schaltungen beispielsweise beträgt der Ausgangspegel $+5\mathrm{V}$ für das logische eins Signal.

[320] Der Name *Schmitt-Trigger* geht auf seinen Erfinder Otto Schmitt aus dem Jahre 1934 zurück. Nähere Informationen zum Schmitt-Trigger finden Sie unter [Wik08n], [Eng] sowie [Rös88, 294ff].

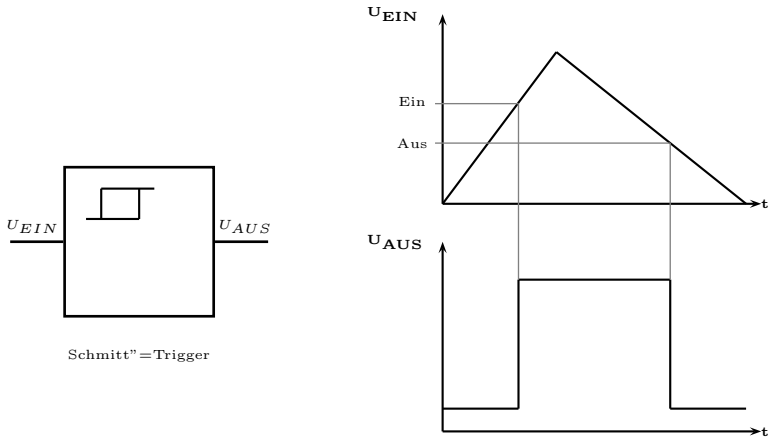


Abbildung F.4.: Schaltbild und Kennlinie eines Schmitt-Triggers.

Stichwortverzeichnis

Symbole

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Glossar

(U)CML-Baumdiagramm	Kompakte Darstellung der hierarchischen Struktur eines Systems., 158
(U)CML-Flussdiagramm	Wichtigste Diagrammart der Modellierungssprache (U)CML. Im Flussdiagramm wird der Aufbau und die hierarchische Struktur des Systems dargestellt. Des Weiteren wird im Flussdiagramm der *Fluss* zwischen Komponenten explizit dargestellt., 157
(U)CML-Graphendiagramm	Kompakte Darstellung der Struktur und der agreggierten Verbindungen zwischen den Komponenten eines Systems., 159
(U)CML-Matrixdiagramm	Kompakte Darstellung der statischen Struktur eines Systems (Pakete, Komponenten und Verbindungen), ähnlich der **D**esign **S**tructure **M**atrix (DSM) bzw. einer Adjazenzmatrix., 160
Abwärtskompatibilität	Eine Komponente ist abwärtskompatibel zu einem System, wenn folgende Bedingungen gelten: 1. Die Komponente ist von neuerer Version, als die Komponente, gegen die sie getauscht wird. 2. Die Version des Systems ist älter, als die aktuelle Version des Systems. 3. Die statische und dynamische Kompatibilität sind erfüllt., 325
Arbitrierung	Die Arbitration ist ein Zugangsverfahren bei dem sich die Nutzer nach einer gegenseitigen Vereinbarung das Zugangsrecht gewähren. Jedes an das Netzwerk angeschlossene Gerät hat generell die gleichen Rechte. Erst die Verhandlung eines Gerätes mit allen anderen sichert diesem den temporären Zugang [Wis07]., 272
Attribut	*(v. lat: attribuere = zuteilen, zuordnen)* Ein Attribut beschreibt die Eigenschaften eines Elements. Alle Eigenschaften eines Elements müssen durch seine Attribute und Funktionen vollständig beschrieben sein., 119
Aufwärtskompatibilität	Eine Komponente ist aufwärtskompatibel zu einem System, wenn folgende Voraussetzungen erfüllt werden: 1. Die Komponente ist von älterer Version, als die Komponente, gegen die sie getauscht wird. 2. Die Version des Systems ist neuer, als die Version des Systems, in dem die Komponente verwendet werden soll. 3. Die statische und dynamische Kompatibilität sind erfüllt., 325
Aushungern	Aushungern liegt vor, wenn ein Prozess unendlich lange warten muss, obwohl immer wieder die Möglichkeit besteht weiter zu arbeiten. Siehe auch [PDB94, 62]., 113
Axiom	*(v. griech.: tà tôn progónon axiómata = als wahr angenommener Grundsatz)* Als Axiom bezeichnet man eine Aussage bzw. Grundregel, die innerhalb ihres System nicht bewiesen werden kann. Zum Beispiel lassen sich die Axiome der Mathematik nicht beweisen, und müssen als gegeben angenommen werden (Jede natürliche Zahl n hat genau einen Nachfolger n' ist ein Axiom der Arithmetik)., 118
Backus-Naur-Form (BNF)	Die Backus-Naur-Form oder Backus-Normalform (kurz BNF) ist eine kompakte formale Metasprache, die benutzt wird, um kontextfreie Grammatiken (= Typ-2-Grammatiken, vgl. Chomsky-Hierarchie) darzustellen. Hierzu zählt die Syntax gängiger höherer Programmiersprachen. Sie wird auch für die Notation von Befehlssätzen und Kommunikationsprotokollen verwendet. Durch die Backus-Naur-Form im Algol 60 Report wurde es erstmals möglich, die Syntax einer Programmiersprache formal exakt, also ohne die Ungenauigkeiten natürlicher Sprachen, darzustellen [Wik06b]., 52
Bezeichner/Identifikator	Anhand eines Bezeichners/Identifikators kann ein (U)CML-Paket, -Komponente bzw. ein -Stecker/-Buchse eineindeutig innerhalb des Systems identifiziert werden. Bei Steckern/-Buchsen und Schnittstellen ist zusätzlich zu deren Name der Name des Eigentümers notwendig, da Stecker/Buchsen und Schnittstellennamen nicht global eineindeutig, sondern nur lokal eineindeutig sind., 161
BIPM	Das BIPM wurde mit der Meterkonvention von 1875 gegründet und arbeitet unter der Aufsicht der internationalen Kommission für Maß und Gewicht. Alle vier Jahre findet die Generalkonferenz für Maß und Gewicht (CGPM, französisch Conférence générale des poids et mesures) statt, bei der die zentralen Angelegenheiten des BIPM entschieden werden., 70
Black-Box Ansicht	Bei der Black-Box Ansicht auf ein System ist die exakte interne Struktur bzw. der interne Aufbau des Systems für das Verständnis des Systems „uninteressant“. Es sind lediglich die äußeren Schnittstellen, sowie das Verhalten des Systems nach außen für dessen Verständnis von Interesse. Die Black-Box Ansicht wird vor allem dann eingesetzt, wenn die innere Struktur eines Systems unbekannt, zu komplex oder nicht von Belang für das Verständnis der Struktur oder Funktionalität des Systems ist., 17
C++	C++ ist eine objektorientierte Erweiterung der Programmiersprache C. C++ wurde von Bjarne Stroustrup [Str90] ab 1979 bei AT&T entwickelt., 126
CPU	Das engl. Akronym *CPU* steht für Central Processing Unit – Hauptprozessor., 181
DIBMUK Prozess	Das Akronym DIBMUK steht für Definition, Identifikation, Bewertung, Maßnahmen, Umsetzung und Kontrolle. Mit Hilfe des DIBMUK-Prozesses kann die Kompatibilitätswahrung während des gesamten Entwicklungsprozesses bis hin zur Wartung und Pflege abgedeckt werden. Nähere Informationen zum DIBMUK-Prozess finden Sie unter [BDW06] und [BKB+07]., 7
Dynamische Kompatibilität	Dynamische Kompatibilität ist die Verträglichkeit der Komponenten und ihrer Schnittstellen zueinander zur Laufzeit in einem System., 25

Eigenschaft (Objekt) — Ein Objekt/Klasse besitzt Eigenschaften, auch Attribute genannt, mit deren Hilfe die Merkmale eines Objekts/Klasse festgelegt und beschrieben werden. Die Eigenschaften des Objekts/Klasse können dabei sowohl konstanter-, als auch veränderlicher Natur sein., 51

Eineindeutigkeit — Im Systems Engineering Kontext bedeutet eineindeutig, dass ein System einen Namen/Objekt/Element genau ein Mal im System existiert. In der Mathematik bedeutet eineindeutig: Die Abbildung einer Menge M in eine Menge N, bei der jedem Element von M genau ein Element von N und jedem Element von N genau ein Element von M entspricht., 12

Eingebettete Systeme — Als eingebettetes System, auch engl. embedded system genannt, bezeichnet man in der Technik Systeme, die sowohl aus Hard- als auch Software bestehen. Die Hardware lässt sich dabei noch in Elektronik- und Mechanikbaugruppen unterteilen. Ein besonderes Kriterium für eingebettete Systeme ist die enge Verzahnung der einzelnen Komponenten des Systems. Nähere Informationen zu eingebetteten Systemen finden Sie unter [Wik08f] und [Wik08g]., 3

Eingebettetes softwarelastiges System — Bei eingebetteten softwarelastigen Systemen nimmt der Softwareanteil innerhalb des Systems eine dominante Rolle gegenüber den „restlichen" Komponenten des Systems ein., 3

Element — Ein Element ist der kleinste Bestandteil eines Systems. Elemente können auch als Entitäten bzw. atomare Einheiten aufgefasst werden., 122

Entität — Philosophisch: Dasein im Unterschied zum Wesen eines Dinges (nach [Dud05])., 19

Formalisierungskomponente — Unter den Formalisierungskomponenten eines Systems werden sämtliche Attribute des Systems bezeichnet. Die Attribute werden unterteilt in Funktionen (aktiver Teil des Systems) und Parameter [PDII06, 8]., 119

Funktionale Dekomposition — Die Funktionale Dekomposition (FKTD) hat zum Ziel, schrittweise ein System zu zerlegen, beginnend bei der Sicht auf die Hauptfunktion eines Systems über die Zwischenebenen bis zur Ebene elementarer Funktionen. Auf einer Ebene wird jeweils von Details der darunter liegenden Ebene abstrahiert. Die Teilfunktionen zusammengenommen ergeben vollständig die aufgegliederte Funktion (Funktionshierarchie) [Fre07] bzw. [PDSPDS, 31ff]., 39

Glass-Box Ansicht — Im Gegensatz zur Black-Box Ansicht, in der die innere Struktur des Systems vollständig verborgen bleibt, ist die interne Struktur bei der Glass-Box Ansicht uneingeschränkt von außerhalb des Systems sichtbar. Die Glass-Box Ansicht wird vor allem dort eingesetzt, wo die interne Struktur des Systems für dessen Erweiterung oder Anwendung von Interesse ist., 17

Grey-Box Ansicht — Die Grey-Box Ansicht stellt einen Kompromiss zwischen den beiden extremen Sichten auf ein System dar. Sie ist eine Kombination aus Black-Box und Glass-Box Ansicht. Hier kann ein wichtiger Systembestandteil vollständig dargestellt sein (Glass-Box), während eine unwichtige Komponente nur als Black-Box dargestellt wird. Die meisten Systeme werden als Grey-Box Modelle dargestellt, da hier ein guter Kompromiss zwischen der Darstellung wichtiger Systembestandteilen und der Abstraktion erreicht wird., 17

Hierarchie — Als Hierarchie bezeichnet man ein System von Elementen (Teil- bzw. Subsysteme), die einander über- bzw. untergeordnet sind. Ist dabei jedem Element höchstens ein anderes Element unmittelbar übergeordnet, so spricht man von einer Monohierarchie, während bei einer Polyhierarchie auch mehrere übergeordnete Elemente möglich sind. Jede Hierarchie lässt sich mathematisch als Ordnungsrelation beschreiben. Dabei gilt: Ein Baum definiert eine Monohierarchie, während ein gerichteter azyklischer Graph eine Polyhierarchie definiert., 13

Information Hiding — Das Verbergen von Implementierungsdetails vor dem Benutzer wird als Information Hiding bezeichnet. Der Benutzer eines Systems muss nichts über dessen inneren Aufbau oder Struktur wissen, um es benutzen/bedienen zu können. Für den Benutzer ist lediglich die äußere Schnittstelle des Systems von Interesse., 49

Instantiierung — Als Instantiierung wird der Prozess bezeichnet, der aus einer (allgemeinen) Klasse ein konkretes Objekt generiert., 58

Instanz — Unter einer Instanz einer Klasse versteht man in objektorientierten Programmiersprachen das erzeuge Objekt der Klasse., 126

Invariante — Informatik: Eine Invariante ist eine Bedingung, die in jedem Programmzustand erfüllt sein muss. Bei der Implementierung einer Schleife muss beispielsweise innerhalb dieser immer gelten, dass die Schleifenvariable kleiner ist als der Wert der Abbruchbedingung [PDB95, 173]. Mathematik: Die Invariante bezeichnet jede Funktion, Zahl oder Eigenschaft, die bei gewissen Transformationen oder allgemeiner bei Abbildungen unverändert, d.h. invariant, bleibt., 203

Klasse — Eine Klasse ist eine Menge von Objekten, die eine gemeinsame Struktur und ein gemeinsames Verhalten aufweisen., 48

Kompatibilität (austauschbar/ersetzbar) — Eigenschaft, in einem System eine (beliebige) Einheit durch eine andere Einheit zu ersetzen/auszutauschen, ohne dass die korrekte Funktionsweise (Syntax und Semantik) des Systems beeinträchtigt wird., 20

Kompatibilität (Identität) — Wird in einem System S eine Einheit A gegen eine identische Einheit B ($A \equiv B$) ausgetauscht/ersetzt, so ist die neue Einheit strikt kompatibel, genauer Austausch- bzw. Ersetzungskompatibel, zum System S, aufgrund der Tatsache, dass sich die beiden Einheiten A und B weder syntaktisch noch semantisch unterscheiden., 23

Kompatibilität (Verträglichkeit) — Nach DIN ISO 8402 wird Kompatibilität als „Eignung einer Einheit (eine Einheit kann eine Tätigkeit oder ein Prozess sein ebenso wie ein Produkt, eine Organisation, ein System oder eine Person, oder irgendeine Kombination daraus) unter spezifischen Bedingungen zusammen benutzt zu werden, um relevante Forderungen zu erfüllen" bezeichnet., 20

Kompatibilitätsregelwerk
Im Kompatibilitätsregelwerk sind sämtliche Kompatibilitätsregeln aller an der Systementwicklung beteiligter Domänen hinterlegt. Mit Hilfe dieser Kompatibilitätsregeln kann das zu untersuchende System, genauer die miteinander verbundenen Schnittstellen zwischen den beteiligten Teilsystemen des Systems, domänenübergreifend auf Kompatibilität hin untersucht und bewertet werden., 28

Kompatibilitätsregelwerks (Eigenschaften)
Das Kompatibilitätsregelwerk enthält sowohl statische als auch dynamische Kompatibilitätsregeln aus den unterschiedlichen Domänen., 28

Komplexität
In der Literatur finden sich zahlreiche Definitionen des Begriffs Komplexität. Im Allgemeinen ist die Komplexität eines Systems definiert als die Anzahl der Elemente des Systems. Zum Beispiel ist die Komplexität einer Menge $M = \{a, b, c, d\}$ gleich 4. Eine ausführliche Definition der *Komplexität von Systemen* finden Sie unter [PDII06, 17ff, 50ff]. Komplexität in der Informatik: Hier wird vor allem das Verhalten (Speicherverbrauch, Laufzeit) von Algorithmen als auch deren Klassifikation (P oder NP) untersucht. Siehe hierzu: [Wik07k][Wik07j][FH], 31

Konfiguration
Eine Konfiguration ist eine Zusammensetzung von Versionen verschiedener Komponenten zu einem spezifischen Zeitpunkt im Rahmen einer Architektur zu einem System. Sie umfasst das Gesamtsystem aller Komponenten., 326

Kontext
Ein Kontext ist eine Menge von Umgebungseigenschaften, die relevant für die Interaktionen einer Entität (System, Komponente) sind. Der Kontext einer Entität ist im Allgemeinen frei definierbar, abhängig von der Sichtweise auf die jeweilige Entität., 19

Modell
Ein Modell ist eine vereinfachte Nachbildung eines existierenden oder gedachten Systems in einem anderen begrifflichen oder gegenständlichen System. Es wird genutzt, um eine bestimmte Aufgabe zu lösen, deren Durchführung mittels direkter Operationen am Original nicht möglich oder zu aufwendig wäre., 35

Modellbildung
Unter Modellbildung verstehen wir die Festlegung einer formale Darstellung, die die Struktur und das Verhalten des zu modellierenden realen Systems so genau wie notwendig beschreibt (nach [PDIG])., 32

Nachbedingungen
Die Nachbedingungen einer Funktion oder eines Programms geben an, welche Aussagen nach der Ausführung gelten müssen, falls zuvor die Vorbedingungen erfüllt waren. Die Nachbedingung ist Teil der formalen Spezifikation der Funktion (bzw. des Programms) und dient der Verifikation: Wenn die Vorbedingung gilt, so müssen nach Ausführung der Funktion alle Nachbedingung erfüllt sein, sonst ist das Programm nicht korrekt., 203

Nachbedingungen (allgemein)
Das Konzept von Vor- und Nachbedingungen wird vor allem in der formalen Semantik benutzt: Es stellt die Basis der axiomatischen Semantik dar. Das Ziel ist es dabei, aus den Vor- und Nachbedingungen der einzelnen Teile des Programms logisch die gewünschte Nachbedingung für das gesamte Programm zu folgern. Auch bei dem weniger formalen Testen von Software spielen Nachbedingungen eine wesentliche Rolle, da das Ergebnis von Testläufen leicht mit den Nachbedingungen verglichen werden kann. Das wird vor allem für den so genannten Unit-Test verwendet [Wik07o]., 203

Nicht strikte Kompatibilität
Einheit B ist nicht strikt kompatibel zu einem System, wenn Einheit A durch Einheit B im System ersetzt wird, sich das Verhalten des Systems ändert, dies aber nicht die korrekte (der Spezifikation genügende) Funktionsweise des Systems beeinträchtigt., 23

Objekt
Ein Objekt ist zunächst ein Gegenstand, eine Rolle, ein Konzept oder ein Prozess im realen Problem- und Anwendungsbereich eines Softwaresystems (reales Objekt). Durch einen geeigneten Abstraktionsprozess wird daraus ein Objekt abgeleitet, das als Basismodul in die Ablaufstruktur eines Softwaresystems eingebunden wird (nach [Neu95, 574])., 47

Objekt
Ein Objekt bezeichnet in der Objektorientierung (z.B. in der Programmiersprache C++) ein Exemplar einer bestimmten Klasse., 126

Qualitativ
Im SE-Element-Konzept bedeutet qualitative Beschreibung, dass ein Parameter lediglich als Platzhalter dient. Er jedoch noch nicht mit einem konkreten Wert belegt ist., 119

Quantitativ
Im SE-Element-Konzept bedeutet quantitative Beschreibung, dass ein Parameter einen festen Wert zugewiesen bekommen hat. Dieser kann sich jedoch im Verlauf der Simulation des Systems verändern, sofern er nicht als konstant definiert ist., 119

RAM
Das engl. Akronym RAM steht für Random Access Memoy – Wahlfreier Schreib-/Lese-Speicher., 181

Relation
(v. lat. relatio: „das Zurücktragen"). Allgemein bezeichnet eine Relation eine Beziehung oder Verbindung zwischen Dingen/Mengen. In der Mathematik gibt es verschiedene ausgezeichnete Relationsarten, die bestimmte Eigenschaften von Dingen/Mengen ausdrücken. Eine ausführliche mathematische Definition der verschiedenen Relationstypen finden Sie im Anhang A bzw. unter [Wik07s] und [Pri06]., 124

Schnittstelle
An einer Schnittstelle treffen zwei autonome Systeme zusammen um in geregelter Weise etwas auszutauschen (z.B. Nachrichten, Signale, Daten, Informationen usw.). Es gibt zwei Sichten auf eine Schnittstelle: die syntaktische und die semantische., 326

Schnittstellenkompatibilität
Systeme/Teilsysteme/Komponenten sind an einer gemeinsamen Schnittstelle kompatibel, wenn sie statisch und dynamisch kompatibel sind., 22

SE-Element
Das SE Element-Konzept basiert lediglich vier Axiomen: 1. Ein System besteht aus Elementen, 2. Elemente haben Attribute, 3. Elemente stehen über Relationen miteinander in Wechselwirkung/Verbindung, 4. Ein Element kann selbst wieder ein System sein., 116

Semantik
Nach [Dud05]: Semantik: (Sprachwissenschaftlich) 1. Teilgebiet der Linguistik, das sich mit den Bedeutungen sprachlicher Zeichen und Zeichenfolgen befasst. 2. Bedeutung, Inhalt (eines Wortes, Satzes od. Textes)., 22

Semantische Schnittstelle	Die semantische Schnittstelle beschreibt das Schnittstellenverhalten, dass die Kausalität und Abhängigkeiten im Informationsaustausch festlegt (die semantische Schnittstelle legt die Interaktionsmuster einer Komponente fest, welche Einzelschritte dabei auftreten ist in der syntaktischen Schnittstelle festgelegt)., 326
SI-Einheit(en)	Das Akronym SI (Système international d´unités) beschreibt das international genormte Einheitensystem. Das SI-Einheitensystem wurde 1960 auf der 11. Generalkonferenz für Maß und Gewicht eingeführt [Bun06a] [Bun06b] [Wik06l]., 69
Signatur	Eine Signatur (oder Methodensignatur) definiert in der Programmierung die formale Schnittstelle einer Funktion. Sie besteht aus dem Namen der Funktion, der Anzahl, Reihenfolge und Typen ihrer Parameter und dem Typ des oder der Funktionsrückgabewerte ([Wik07t]). Siehe auch: [Dum08] sowie [PDB98b]., 53
SIL Level	Das Akronym SIL steht für Safety-Integrety-Level (Sicherheits-Integritätslevel). Mit Hilfe des SIL Levels wird die „Funktionale Sicherheit sicherheitsbezogener elektrischer/elektronischer/programmierbar elektronischer Systeme“ beschrieben (Norm: IEC 61508). Nähere Informationen zu SIL finden Sie unter: [Kem], [RAM07], [Wik07f] sowie [IEC07]., 110
Simulation	In der Simulationstechnik versteht man unter Experiment die gezielte empirische Untersuchung des Modellverhaltens durch wiederholte Simulationsläufe mit systematischen Parametervariationen., 44
Statische Kompatibilität	Statische Kompatibilität ist die Verträglichkeit der Schnittstellen der Komponenten zueinander in einem System, ohne dass das System ausgeführt werden muss., 25
Strikte Kompatibilität	Einheit B ist strikt kompatibel zu einem System und einem definierten Kontext, wenn Einheit A durch Einheit B im System ersetzt wird und B das gleiche Verhalten wie A im System und dem Kontext garantiert., 23
Syntaktische Schnittstelle	Die syntaktische Schnittstelle gibt an, welche grundsätzlichen Möglichkeiten für Informationsaustausch an der Schnittstelle bestehen., 326
Syntax	Nach [Dud05]: Syntax: (Sprachwissenschaftlich) 1. in einer Sprache übliche Verbindung von Wörtern zu Wortgruppen und Sätzen; korrekte Verknüpfung sprachlicher Einheiten im Satz. 2. Lehre vom Bau des Satzes als Teilgebiet der Grammatik; Satzlehre. 3. wissenschaftliche Darstellung der Syntax (2)., 22
System	Ein System kann ein beliebiger, abgegrenzter Gegenstand unseres Denkens sein. Die Systemgrenze trennt das betrachtete System von der Umwelt, in die es eingebettet ist [DIN98, 66]., 12
System-Engineering Element-Konzept	Das Systems Engineering Element-Konzept, kurz SE-Element, wurde im Jahre 1982 von Prof. Dr. G. Patzak entwickelt und 1992/93 von Prof. Dr.-Ing. E. Igenbergs am Lehrstuhl für Raumfahrttechnik – Systems Engineering Group – der Technischen Universität München weiterentwickelt. Mit Hilfe des SE Elementkonzeps lassen sich komplexe Systeme auf einfache Art und Weise beschreiben., 116
Systemstruktur	Die Systemstruktur ist die Menge der Komponenten eines Systems und die Menge der, die Komponenten miteinander verbindenden, Relationen (aus [FM06])., 15
Systemverhalten	Das Systemverhalten ist die Menge der zeitlich aufeinanderfolgenden Zustände eines Systems (aus [FM06])., 16
Teil- oder Subsystem	Ein Teil- bzw. Subsystem ist ein abgegrenzter Bereich innerhalb eines Systems, der selbst Merkmale eines Systems aufweist., 13
UML	Die UML ist in den 1990er Jahren von den „drei Amigos“ – Grady Booch, Ivar Jacobson und James Rumbaugh – maßgeblich entwickelt worden, um das aufkommende Paradigma der objektorientierten Programmiersprachen zu unterstützen. Im Jahre 2005 wurde die zweite Version der UML (UML 2.x) veröffentlicht. Heute ist die OMG – Object Management Group – für die Wartung und Pflege der UML zuständig., 131
Validierung	Überprüfen der hinreichenden Übereinstimmung von Modell und System. Es ist sicherzustellen, dass das Modell das Verhalten des Originalsystems im Hinblick auf die Untersuchungsziele genau genug und fehlerfrei widerspiegelt., 45
VDI	VDI steht für „Verein Deutscher Ingenieure“, einem nationalen Zusammenschluss von Ingenieuren verschiedener Fachrichtungen zum Zweck des Wissenstransfers bzw. zur Schaffung von Normen und Richtlinien., 35
Verbindung zwischen Systemen	Systeme können mittels Verbindungen, auch Relationen genannt, an ihren Schnittstellen miteinander verbunden werden. Dabei werden grundsätzlich zwei Verbindungsarten unterschieden: 1. Gerichtete Verbindung: Mit Hilfe von gerichteten Verbindungen, auch Flussrelationen genannt, werden Daten und Informationen stets in Pfeilrichtung vom Sender zum Empfänger hin unverändert übertragen. Des Weiteren werden gerichtete Verbindungen weiter unterschieden in einseitig- und beidseitig gerichtete Verbindungen. Dabei kann jede beidseitig gerichtete Verbindung durch zwei einseitig gerichteten Verbindungen dargestellt werden. 2. Ungerichtete Verbindung: Ungerichtete Verbindungen haben keine eindeutige Flussrichtung., 14
Verklemmung	Eine Verklemmung liegt vor, wenn z.B. ein Prozess A auf das Ergebnis des Prozesses B wartet und der Prozess B auf ein Ergebnis des Prozesses A wartet. Von einer Verklemmung wird ebenfalls gesprochen, wenn ein oder mehrere Prozesse auf ein exklusives Betriebsmittel warten, sich dieses durch die Nutzung selbst blockieren oder dieses vom eigenen oder einem anderen Prozess (dauerhaft) belegt ist. Siehe auch [PDB94, 47]., 113
Vorbedingung	Die Vorbedingung einer Funktion oder eines Programms gibt an, unter welchen Bedingungen das Verhalten der Funktion definiert ist. Die Vorbedingung ist Teil der formalen Spezifikation der Funktion (bzw. des Programms) und dient der Verifikation: Wenn sie gilt, so müssen nach Ausführung der Funktion alle Nachbedingungen erfüllt sein, sonst ist das Programm nicht korrekt [Wik07z]., 203

Literaturverzeichnis

[Ado07] `http://www.adobe.com/de/` 257

[AG] AG, Knorr-Bremse:*Elektronik*.
URL: `http://www.knorr-bremse.de/download/com_de/schiene/SfS_Elektronik.pdf` 4

[AG06] AG, Knorr-Bremse: *ESRA - Systemkomponenten*. 2004-2006. – Interne Dokumentation 4.6, 4.7, 4.8, 4.9

[Age08] `http://www.esa.int/esaMI/CDF/SEM1OF1P4HD_0.html` 6

[AHS08] ARIOLA, Zena M. ; HERBELIN, Hugo ; SABRY, Amr:*A Proof-Theoretic Foundation of Abortive Continuations (Extended version)*.
URL: `http://grouchy.cs.indiana.edu/l/www/ftp/techreports/TR608.pdf` 63, 2.5.1

[Alt07] `http://www.altova.com/` 3.6

[BDW06] BORNEMANN, Falk ; DR. WENZEL, Stefan: Managing compatibility throughout the product life cycle of embedded systems – Definition and application of an effective process to control compatibility. In: *INCOSE* (2006) 1.2, F.1

[BE08] BRANDSTÄTTER, Markus ; ECKL, Carolin: Multi-disciplinary System Engineering and the Compatibility Modeling Language (U)CML. In: MALPICA, Freddy (Hrsg.) ; TREMANTE, Andrés (Hrsg.) ; WELSCH, Friedrich (Hrsg.) ; TAIT, Bill (Hrsg.) ; IFSR (Veranst.): *International Multi-Conference on Engineering and Technological Innovation* Bd. 1 IFSR, 2008, S. 189ff 18, 82, 85, 2.6, 204

[Bec92] BECHTOLD, Hubert: *Tabellenbuch Kommunikationselektronik*. Verlag Europa Lehrmittel, 1992 (3. Auflage 3-8085-3373-0). – Co-Autoren: Dipl.-Ing. Ulrich Freyer, Dr.-Ing. Gerog Häberle, Dipl.-Gwl. Heiz Häberle, Elektro-Ing. Oskar Huber, Dipl.-Ing. Gerhard Mangold, Dipl.-Ing. Klaus Rieger, Dipl.-Ing. Heinz Ruckriegel, Dipl.-Ing. Dietrich Schad, Dipl.-Ing. Bernd Schiemann, Dipl.-Ing. Dietmar Schmid, Dipl.-Ing. Dieter Schnell, Dipl.-Ing. Manfred Schuh, Dipl.-Gwl. Frank-Dieter Stricker 2.4

[BK05] BRANDSTÄTTER, Markus ; KOSS, Dagmar: *(U)CML - Ein allgemeiner Modellierungsansatz zur Darstellung von Kompatibilität*. TUM, 2005 3.6.1, 229, 3.6.5, E.1

[BKB+07] BORNEMANN, Falk ; KOSS, Dagmar ; BRANDSTÄTTER, Markus ; DR. CÄSAR, Michael ; DR. SLOTOSCH, Oscar: Schlussbericht - MOKOMA / BMBF. 2007 (1). – Abschlussbericht 14, F.1

[BMI06] `http://www.kbst.bund.de/cln_012/nn_836802/Content/Standards/V__Modell__xt/v__modell__xt__node.html__nnn=true` 1.2

[Böh06] `http://www.imb-jena.de/~gmueller/kurse/c_c++/c623.html` 285

[Boo94a] BOOCH, Grady: *Object Oriented Analysis and Design with Applications*. 2nd. Addison Wesley Professional, 1994 (The Addison-Wesley Object Technology Series 0-8053-5340-2) C.9

[Boo94b] BOOCH, Grady: *Objektorientierte Analyse und Design*. Addison-Wesley, 1994 (The Addison-Wesley Object Technology Series 978-3893196739) 2.5, 2.13, 2.7, 2.14, 2.15

[BPDN02] BÜRGMAYR, Johannes ; PROF. DR. NIPKOW, Tobias: MARS CLIMATE ORBITER LOSS. In: *Technische Universität München* (27. November 2002) 2.3.2

[BRJ98] BOOCH, Grady ; RUMBAUGH, James ; JACOBSON, Ivar: *The Unified Modeling Language User Guide*. 1ed. Addison-Wesley Professional, 1998 (The Addison-Wesley Object Technology Series 0201571684) 58, 3.4.3, 3.6

[Bro91] BRONSTEIN, Semendjajew: *Taschenbuch der Mathematik*. Verlag Nauka Moskau, 1991 (1 ISBN: 3-87144-492-8) A

[Büc83] BÜCHEL, A. ; ING. DAENZER, W.F. Prof. Dr. h.c. d. (Hrsg.): *Systems Engineering - Leitfaden zur methodischen Durchführung umfangreicher Planungsvorhaben*. Verlag Industrielle Organisation, Zürich, 1983 (3 3 85743 864 9) 54

[Bul06] `http://www.bullhost.de/k/kompatibel.html` 28

[Bun06a] `http://www.ptb.de/de/wegweiser/einheiten/si/` 2.3.2.2.1.1, 90, 2.3.2.2.1.2, F.1

[Bun06b] BUNDESANSTALT, Physikalisch-Technische:*SI-Einheiten*.
URL: `http://www.ptb.de/de/publikationen/download/pdf/einheiten.pdf` 90, 2.4, F.1

[Car] CARIT, BMW:*Kompatibilitätsmanagement*.
URL: `http://www.bmw-carit.de/pdf/konferenz.pdf` 28

[CKS03] CHRISSIS, M. B. ; KONRAD, M. ; SHRUM, S. ; INC, Pearson E. (Hrsg.): *CMMI - Guidelines for Process Integration and Product Improvement*. Carnegie Mellon Software Engineering Institute, 2003 28

[com06] http://www.iec.ch/ 2.3.2.2.1.2

[Cor06] http://etna.int-evry.fr/COURS/UML/notation/index.html 3.6.2.6.3

[CS04] CHONOLES, Michael J. ; SCHARDT, James A. ; BÄCKMANN, Marcus (Hrsg.): *UML 2 für Dummies*. mitp, 2004 58, 60, 2.12, 2.16, 73, 2.3.1.4, 3.27, 3.4.3, 159, 3.28, 3.6, 3.34, 3.35, 3.4.3.1.1, 3.4.3.1.2

[DAC06] http://www.quality.de/lexikon/kompatibilitaet.htm 28

[Dan09] DANGL, Markus: Interdisziplinäres Projekt (IDP) - Integration der CCL (Compatibility Constraint Language) in (U)CML-ed / Technische Universität München. 2009 (1). – Interdisziplinäres Projekt 3.6.2.6.3

[das07a] http://www.elektronik-kompendium.de/sites/com/0309161.htm 215

[das07b] http://www.elektronik-kompendium.de/sites/com/0309191.htm 216

[DC04] DR. CAESAR, Michael: *Review Präsentation*. Dezember 2004. – MOKOMA Review I Präsentation 4.1, 4.2, 4.3, 4.4

[DE07] http://products.astrium.eads.net/corp/prod/00000853.htm 3.6.2.3.2

[DH02] DAENZER, W.F. Prof. Dr. sc. t. ; HUBER, F.: *Systems Engineering, Methodik und Praxis*. Verlag Industrielle Organisation, 2002 1.2, 16

[Die06] DIETRICH, Arno: *Spezifikation, Auslegung, Kodierung und Test eines Simulationsmodells für das TanDEM-X Kaltgassystem*, Technische Universität München - Lehrstuhl für Raumfahrttechnik, Diplomarbeit, 2006 3.6.2.3.2

[DIF98] DR.-ING. FRICKE, Ernst: *Der Änderungsprozess als Grundlage einer nutzerzentrierten Systementwicklung*, Technische Universität München - Lehrstuhl für Raumfahrttechnik, Diss., 1998 3.4.2

[DIH89] DR.-ING. HÄBERLE, Gregor: *Tabellen und Schaltungen*. Kohl + Noltemeyer und Co., 1989 290

[DIN] DIN:*Informationsverarbeitung: Sinnbilder für Datenfluss- und Programmablaufpläne*.
URL: http://www.fh-jena.de/~kleine/history/software/DIN66001-1966.pdf 111

[DIN98] DR.-ING. NEGELE, Herbert: *Systemtechnische Methodik zur ganzheitlichen Modellierung am Beispiel der integrierten Produktentwicklung*, Technische Universität München - Lehrstuhl für Raumfahrttechnik, Diss., 11.1998 1.2, 2.2, 2.2, 142, 3.4.2.1, F.1

[DIP08] DR.-ING. PEUKERT, Andreas: *Spacecraft Architectures Using Commercial Off-The-Shelf Components*, Technische Universität München - Lehrstuhl für Raumfahrttechnik, Diss., 2008 55

[DIQ01] DR.-ING. QUIRMBACH, Oliver: *Integration der System- und Konstanentwicklung am Beispiel eines Satellitensystems*, Technische Universität München - Lehrstuhl für Raumfahrttechnik, Diss., 2001 54, 2.2.1

[DIS99] DR.-ING. SCHREPFER, Lutz: *Modellierung und Simulation von Systemen dynamischer Struktur*. Copyright: Herbert Utz Verlag GmbH, 2006, Technische Universität München - Lehrstuhl für Raumfahrttechnik, ISBN 3-8316-0382-0, 1999. http://www.utzverlag.de/shop.php?bn=40382 54, 2.2.1, 2.2.1, 3.4.2.1, 3.4.2.2

[DIS07] DR.-ING. SCHIFFNER, Michael: *Eine objektorientierte Modellierungsmethode für die simultane Systementwicklung*, Technische Universität München, Diss., 2007 143, 187, 6

[DIS08] DR.-ING. SCHIFFNER, Michael:*Lehrstuhl für Raumfahrttechnik*.
URL: http://www.lrt.mw.tum.de/de/wissenschaft/projekte.phtml 187

[DIV01] DR.-ING. VOLLERTHUN, Andreas: *Integration von Konzeptentwurf und Marketing*, Technische Universität München - Lehrstuhl für Raumfahrttechnik, Diss., 2001 3.4.2

[DIW93] DR.-ING. WALTHER, Christian: *Systemtechnische Zusammenhänge zwischen Eigenschaten und Funktionen großer Systeme*, Technische Universität München - Lehrstuhl für Raumfahrttechnik, Diss., 1993 3.9, 3.4.2.1

[DJ08] DR. JUNKER, Markus:*Vorlesungsskript "Formale Logik"*.
URL: home.mathematik.uni-freiburg.de/junker/ws05/ws05-logik-skript-KAP3.pdf 63, 2.5.1

[DL] DR. LEWANDOWSKI, Stefan:*Einführung in die Informatik*.
URL: http://www.fmi.uni-stuttgart.de/fk/lehre/ws06-07/autip1/folien/Skript-1.pdf 74

[DN06] http://home.zait.uni-bremen.de/~mn/old/glossar/g/node13.html 3.4.2.1

[Dud05] DUDENVERLAG: *Das Fremdwörterbuch (CD-ROM)*. Bd. 8. Dudenverlag, 2005 1.11, F.1

[Dud07] http://www.dudkowski.de/beschr/scsin.htm 241

[Dum06] http://ivs.cs.uni-magdeburg.de/~dumke/UML/20.htm 3.4.3.1.2

[Dum08] http://ivs.cs.uni-magdeburg.de/sw-eng/agruppe/lehre/ead.shtml F.1

[Eck06] ECKL, Carolin: Entwicklung einer GUI für (U)CML / Technische Universität München. TUM. Boltzmannstraße 15, 2006. – Forschungsbericht 266

[Eck07] ECKL, Carolin: *Untersuchung und Anpassung von MSCs zur Überprüfung von Verhaltenskompatibilität*. Boltzmannstraße 3, Technische Universität München, Diplomarbeit, 2007 37, 76, 2.3.2.3, 2.5.2, 163, 3.4.4.1.2, 205

[Eck12] ECKL, Carolin: *InterXPhase - Unveröffentlicht*, Technische Universität München - Lehrstuhl für Raumfahrttechnik, Diss., voraussichtlich Ende 2012 82, 2.6, 204

[ECS08] http://www.ecss.nl/ 13

[Ele] ELECTRONIC, Wieland:*Industrie Steckverbinder*.
URL: http://www.wieland-electric.de/uploads/tx_ffwedownloadcenter/0075.0_revos_BASIC_web.pdf 4.22

[Eng] ENGINEERING, Krucker:*Diverse Dokumente*.
URL: www.krucker.ch/DiverseDok/SchmittTrigger.pdf 320

[Eng07] ENGINEERING, The University of Q. o.:*Functional Analysis*.
URL: http://www.catalyst.uq.edu.au/designsurfer/functional_analysis_basics.pdf 54

[FH] FORTNOW, Lance ; HOMER, Steve:*A Short History of Computational Complexity*.
URL: http://people.cs.uchicago.edu/~fortnow/papers/history.pdf F.1

[FM06] FACHSCHAFT MASCHINENBAU, Foliensammlung 6. ; TECHNISCHE UNIVERSITÄT MÜNCHEN (Hrsg.): *Vorlesung Modellbildung und Simulation*. 1. Boltzmannstraße 15: Technische Universität München, 2006 1.8, 1.9, 1.15, 2.2, 2.1, 2.2, 46, F.1

[FMS] FRIEDENTHAL, Sanford ; MOOR, Alan ; STEINER, Rick:*OMG Systems Modeling Group - OMG SysML Tutorial*.
URL: http://www.omgsysml.org/INCOSE-2007-OMG-SysML-Tutorial.pdf 3.4.4.1

[Fow00] FOWLER, Martin: *UML konzentriert - Eine kompakte Einführung in die Standard-Objektmodellierung*. Addison-Wesley, 2000 (1 3827316170) 2

[Fre07] http://www.informatik.uni-bremen.de/gdpa/methods_d/m-fctd.htm F.1

[Gaj06] GAJDZIK, Jakob ; TUM (Hrsg.): *(U)CML ed Entwicklung eines grafischen Modellierungswerkzeuges in Java*. 1. Boltzmannstraße 15: TUM, 2006 266

[GfS07] http://www.gfse.de/se/index.html 1.2

[Gie06] http://www-aix.gsi.de/~giese/swr/index.html 2.3.2

[Glo07] GLOCKNER, Dipl.-Ing. M.: *Methoden zur Analyse von Rückwärtskompatibilität von Steuergeräten*, Technische Universität Chemnitz, Diss., 2007 2.8

[Gmb07] GMBH juris:*Gesetz über die elektromagnetische Verträglichkeit*.
URL: http://www.gesetze-im-internet.de/bundesrecht/emvg_1998/gesamt.pdf 1.4

[GRe07] http://www.ing.iac.es/~eng/standards/software/sof-std-2/sof-std-2.htm 3.6.2.6.3

[Gri07] http://www.spaceref.com/news/viewsr.html?pid=23775 1.2

[Gro06a] http://www.uml.org/ 73, 3.4.3

[Gro06b] http://www.omg.org/ 73, 3.4.3

[Gro06c] http://www.omg.org/mda/ 3.4.3

[Hab73] HABERFELLNER, Reinhard Prof. Dr. sc. t.: Systems Engineering. Eine Methodik zur Lösung komplexer Probleme. In: *io Management Zeitschrift 42* (1973), Nr. Nr. 7, 373-386 1.2, 311

[HEB08] http://autofocus.in.tum.de 4, 3.6

[HK07] http://www.sakowski.de/arb-r/arb-r27.html 233

[IEC07] http://www.iec.ch/zone/fsafety/Preview.htm 128, F.1

[IEE06] http://www.ieee.org/portal/site 285

[Inc07a] http://www.mathworks.com/ 6

[Inc07b] http://www.adobe.com/devnet/pdf/pdf_reference.html 257

[ita06] http://www.it-administrator.de/lexikon/kibi.html 2.3.2.2.1.2

[JR85] JANSEN, Host ; RÖTTER, Heinrich: *Fachkunde Fernmeldetechnik*. Europa Lehrmittel, 1985 (1 ISBN 3-8085-3342-0) 208

[KB07] KOSS, Dagmar ; BRANDSTÄTTER, Markus: (U)CML - A Modeling Language for Modeling and Testing Compatibility. In: SMITH, J. (Hrsg.): *Software Engineering and Applications SEA 2007* , 2007 266

[KE01] KEMPER, Alfons ; EICKLER, Andre ; 4 (Hrsg.): *Datenbanksysteme*. Oldenbourg, 2001 (1 3486257064) 164

[Kem] KEMMANN, Sören:*Die IEC/EN Norm 61508-3:2001*.
URL: http://wwwagse.informatik.uni-kl.de/teaching/qsm/ws2004/material/Norm_61508-3_final.pdf 128, F.1

[Ker05] KERN, Carsten: *MSCan - ein tool zur Analyse von Message Sequence Charts*, Fakultät für Mathematik, Informatik und Naturwissenschaften der Rheinisch-Westfälischen Technischen Hochschule Aachen, Diplomarbeit, 2005 76, 3.6

[Kni] KNIGHT, John:*Glitches and Hazards in Digital Circuits*.
URL: http://www.doe.carleton.ca/~jknight/97.267/267_04W/Asch1HazShortJ.pdf 207

[Kon06] http://www.software-kompetenz.de/?28860 138

[Koß09] KOSS, Dagmar: *Kompatibilität und Kompatibilitätsmanagement - Methoden und Ansätze -*. Boltzmannstraße 3, Technische Universität München, Unveröffentlicht, 2009 1.4, 1.12, 1.13, 1.14, 1.15, 1.16, 1.17, 1.18, 1.19, 37, 2.3.2.3, 2.5.2, 163, 3.4.4.1.2, 205, B, B.1, B.2, B.8

[Krü00] KRÜGER, Ingolf H.: *Distributed System Design with Message Sequence Charts*, Technische Universität München, Diss., 2000. http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2000/krueger.pdf 76, 3.6

[KS03] KOSSIAKOFF, Alexander ; SWEET, William N.: *Systems Engineering - Principles and Practice*. Wiley-Interscience, 2003 (1 0 471 23443 5) 54, 2.2.1, 2.3.1.1

[Lab08] http://www.thechiselgroup.org/shrimp 272

[Lan06] http://www.inf.fh-flensburg.de/lang/algorithmen/grundlagen/menge.htm A

[LFPM00] LYKINS, Howard ; FRIEDENTHAL, Sanford ; PH.D. MEILICH, Abraham:*Adoption UML for an Object Oriented Systems Engineering Method (OOSEM)*.
URL: http://www.omg.org/docs/syseng/02-06-11.pdf 57

[Mäd07] MÄDER, Andreas:*VHDL Kompakt*.
URL: http://tech-www.informatik.uni-hamburg.de/vhdl/doc/kurzanleitung/vhdl.pdf 3.6.9

[Mag06] http://www.projektmagazin.de/glossar/gl-0824.html 1.2

[McK06] http://www.incose.org/ProductsPubs/products/sehandbook.aspx B.4

[Mes06] http://www.bipm.org/ 2.3.2.2.1.1

[Mic08] http://java.sun.com/ 267

[MIS07] http://www.misra.org.uk/ 231, 300

[Mol94] MOLL, Karl-Rudolf: *Informatik Management*. Springer-Verlag, 1994 (1 ISBN 3-540-57458-1) 3.4.1

[MPDHDFDM07] MATTHIAS, Glockner. ; PROF. DR. HARDT, Wolfram ; DR. FUCHS, Maximilian ; DR. MACHT, Michael: Kompatibilitätsanalyse dynamischen Verhaltens von integrierten Automobil-Steuergeräten. In: *Wissenschaftliche Schriftenreihe Eingebettete, selbstorganisierende Systeme*. http://archiv.tu-chemnitz.de/pub/2007/0083/index.html : TU Chemnitz, 2007 2.8

[MS07] MEROTH, A. ; SCHWEGLER, U.:*Lösung komplexer Probleme durch systematisches Denken*.
URL: www.fh-heilbronn.de/diehochschule/hs_aktivitaeten/ethik/veranstaltungen/muz/muzmeroth.pdf 1.2

[Ne08] http://www.din.de/cmd?level=tpl-home&contextid=din 41

[Neu95] NEUMANN, Horst A.: *Objektorientierte Entwicklung von Software-Systemen*. Addison-Wesley, 1995 2.3.1.1, 2.6, C.7, C.8, F.1

[OHG07] http://www.buerklin.com/ 4.12, 4.13, 4.14

[Ohn03] OHNMACHT, Anke:*Ablaufmodellierung mit Petri-Netzen*.
URL: http://www.informatik.uni-ulm.de/dbis/01/lehre/ss03/gs/p_ablauf/Petri-Netze.pdf A, A.2

[OMG] OMG:*Final Adopted Specification*.
URL: http://www.sysml.org/docs/specs/OMGSysML-FAS-06-05-04.pdf 3.49, 3.50, 3.51, 3.4.4.1.1, 3.52, 3.53, 3.54, 3.55, 3.56

[OMG07a] http://www.omg.org/ 157

[OMG07b] http://www.uml.org/ 3.6.2.6.3

[OMG07c] OMG:*Unified Modelling Language: Superstructure*.
URL: www.omg.org/docs/formal/07-02-05.pdf 73, 3.4.3, 3.4.3.1.1, 3.4.3.1.2, 3.39

[OOS07] http://www.oose.de/syseng_was_ist_systems_engineering.htm 1.1, 9

[O'R08] http://www.xml.com/ 268

[Pat82] PATZAK, Gerold Dipl.-Ing. Dr. t.: *Systemtechnik - Planung komplexer innovativer System, Grundlagen, Methoden, Techniken*. Springer-Verlag, 1982 (1 0-387-11783-0) 2.2, 3.4.2, 3.1

[PDB94] PROF. DR. BROY, Manfred: *Informatik - Eine grundlegende Einführung, Teil III*. Springer-Verlag, 1994 77, 131, 132, B.3, F.1

[PDB95] PROF. DR. BROY, Manfred: *Informatik - Eine grundlegende Einführung, Teil IV*. Springer-Verlag, 1995 68, C.10, F.1

[PDB98a] PROF. DR. BAHILL, Terry: The Design-Methods Comparison Project. In: *IEEE Transactions on Systems* (1998). http://www.sie.arizona.edu/sysengr/methods2/index.html, Abruf: 25.04.2008 2.2, 53, 2, 2.2.1, 2.3.1.1

[PDB98b] PROF. DR. BROY, Manfred: *Informatik - Eine grundlegende Einführung, Teil I*. Springer-Lehrbuch, 1998 3.6.2.6.3, 3.6.2.6.3, F.1

[PDB05] PROF. DR. BALZERT, Heide: *UML 2 in 5 Tagen*. W3L GmbH Bochum, 2005 (1 3-937 137-61-0) 58, 2.3.1.4, 3.4.3, 3.28

[PDB07] PROF. DR. BROY, Manfred ; C.A.R., Hoare (Hrsg.) ; GRÜNBAUER, J. (Hrsg.): *Specifying, Relating and Composing Object Oriented Interfaces and Components into Architectures*. Nieuwe Hemweg 6B, 1013 BG Amsterdam, The Netherlands : IOS Press, 2007 http://www.iospress.nl/loadtop/load.php?isbn=nics 28, 33, 34

[PDBS01] PROF. DR. BROY, Manfred ; STOLEN, Ketil ; GRIES, David (Hrsg.): *Specification and development of interactive systems*. Springer-Verlag, 2001 (1 0-387-95073-7) A.2

[PDDP06] http://www.donau-uni.ac.at/de/universitaet/whois/03925/index.php 3.4.2

[PDIG] PROF. DR.-ING. GRASS, Werner:*Technische Grundlagen der Informatik*. URL: http://lrs2.fmi.uni-passau.de/skripten/SS05/TI_1_Kap2.3_mathe.pdf 2.1, F.1

[PDIG08] PROF. DR.-ING. GHEORGHIU, Victor:*Technische Thermodynamik und Strömungslehre (TTS)*. URL: http://www.mp.haw-hamburg.de/pers/Gheorghiu/Vorlesungen/TTS/Skript/1.3.4/TTS_1.3.4.htm 93

[PDII06] PROF. DR.-ING. IGENBERGS, Eduard:*Einführung in die Systemtechnik*. URL: www.lrt.mw.tum.de/download/veranstaltungen/veranstaltungen_4/skripte/Einfhrung_in_Systems-Engineering.pdf 1.2, 141, 3.6.2.2, F.1

[PDM08] http://wwwmayr.in.tum.de/lehre/2006SS/info4/ 77

[PDN08] PROF. DR. NIEHUS, Horst:*Experimentalphysik I –Teil Wärmelehre 1/2*. URL: http://asp2.physik.hu-berlin.de/Vorlesung_Exp/1_2_EX_Folien.pdf 93

[PDSPDS] PROF. DR. STUDER, R. ; PROF. DR. STUCKY, W.:*Objektorientierte Softwareentwicklung mit der Unified Modeling Language*. URL: http://www.aifb.uni-karlsruhe.de/Lehrangebot/Winter2001-02/AngInformatik1/folien/uml_WS2001.pdf F.1

[PDW07] PROF. DR. WÜNSCHE, J.: Vorlesung Systemtechnik / Universität der Bundeswehr München Neubiberg. 2007 (1). – Forschungsbericht C

[PMA99] PÖHL, Frank ; MAYER, Volker ; ANHEIMER, Walter: *ATPG für Verzögerungsfehler in Schaltungen mit konventionellem Prüfpfad unter Berücksichtigung von Tri-State-Gattern*. Institut für Theoretische Elektrotechnik und Mikroelektronik Universität Bremen, 1999 http://www.item.uni-bremen.de/research/papers/paper.pdf/Frank.Poehl/itg99/itg99.pdf 207

[Pre06] http://www.galileo-press.de/openbook/c_von_a_bis_z/c_007_005.htm 285

[Pri06] http://www.math.tu-cottbus.de/INSTITUT/lsgdi/DM/BinRel.html A, F.1

[PS00] PROF. STEPHAN, Peter F.: Denken am Modell / Kunsthochschule München. Version: 2000. http://mgnt.khm.de:8080/gems/hypermedia/dam_de.pdf. 2000. – Forschungsbericht 16

[PT] PANDIKOW, Asmus ; TORNE, Anders:*Support for Object-Orientation in AP-233*. URL: http://www.ida.liu.se/~rtslab/publications/2001/pandikow+01a.pdf 3.4.4.1

[Ram06a] http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht8Ht/bnf-ebnf 74

[Ram06b] http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht8Ht/c++_grammatik_de 74

[RAM07] http://www.rams.de/beratung/safety/61508/index.html 128, F.1

[RD08] http://www.dlr.de/qp/desktopdefault.aspx/tabid-3105/4701_read-6899/ 13

[Roc06] http://www.informatik.hu-berlin.de/~roch/VModell/ 1.2

[Rös88] RÖSLER, André: *Grundlagen der Elektrotechnik*. Medien-Institut Bremen, 1988 320

[Rot07] ROTHMUND, Tobias:*Bedeutung von Hardware in the loop in der Automobilindustrie*. URL: http://www.mechatronics-net.de/gerworkgroup/sitzung8/8ak-rothmund.pdf 3.6.2.3.2

[Sch95] SCHADE, Christian: *Kompatibilitätskriterien, Kompatibilitätsmanagement und Projektselektion*. 1. Wiesbaden : Deutscher Universitätsverlag, 1995. – ISBN 3–8244–6118–8 28

[Sch06] http://www.schulz-koengen.de/biblio/bnfebnf.htm 74

[SE07] http://www.incose.org/ 1.2

[Spi] SPIESS, Hase M.:*Neue Erkenntnisse zum Gleitschutzverhalten elektrischer Triebzüge*. URL: http://www.eurailpress.com/archiv/showpdf.php?datei=/erparchiv/etr2005/ETR_10_Hase_Muether_Spiess.pdf 4

[sta96] STANDARDIZATION, European C. s.:*Raumfahrt-Projektmanagement*. URL: http://www.dlr.de/qp/Portaldata/44/Resources/dokumente/einrichtung/ecss-m-30ag.pdf 1.2, 1.3

[Sta08] STATISTA:*Statistik: Handybesitz*. URL: http://de.statista.org/statistik/diagramm/studie/12740/umfrage/handybesitz/#info 1

[Sto] STOCKER, Markus:*Formale Grundlagen für Informatik - Zusammenfassung Skript SS2001*. URL: http://archiv.uniboard.ch/fgi/FGI_zusfass.pdf A

[Stö93] STÖCKER, Horst: *Taschenbuch mathematischer Formeln und moderner Verfahren*. Verlag Harri Deutsch Thun und Frankfurt/Main, 1993 (2 3-8171-1256-4) A, A.3

[Str90] STROUSTRUP, Bjarne: *Die C++ Programmiersprache*. Bonn : Addison-Wesley, 1990. – ISBN 3–925118–72–1 68, 2.3.2.3, 138, 156, 170, 3.6.3.1, F.1

[SV06] STÜTZLE, Thorsten ; VIERECK, Uwe: Adaptiver Gleitschutz für Schinenfahrzeuge. In: *AUTORAIL - Automatisierung für Schienenverkehrssysteme: Fahrzeugtechnik* VDE Kongress 2006 Aachen, 2006, 401-406 4

[Sys06] http://www.sysml.org/ 59, 3.4.4.1, 3.6

[Sys07] SYSTEMS, IAR:*IAR Embedded Workbench MISRA C*. URL: ftp://ftp.iar.se/WWWfiles/guides/MisraC.pdf 242

[Tea08] http://ucml.in.tum.de/index.php/Hauptseite 266, 3.8

[tec] TECHN., Peter T. Dipl. El.-Ing. Früh s. Dipl. El.-Ing. Früh F. Dipl. El.-Ing. Früh:*Software Entwicklung (SwE) im IT-Studiengang*. URL: http://home.zhwin.ch/~frp/SwE/Doku/Ood.pdf C.9

[Tec07] TECHNOLOGIE, Bayerisches S.:*Richtlinie über Elektromagnetische Verträglichkeit*. URL: http://www.stmwivt.bayern.de/pdf/europa/Elektromag_Vertraeglichkeit.pdf 1.4

[Tho02] THOMAS, Marco: Die Vielfalt der Modelle in der Informatik / Universität Potsdam - Didaktik der Informatik. 2002. – Forschungsbericht 16

[Tho08] http://www.thomalsky.com/zitate/zitate-2006-33.html 52

[TM06] http://www.lrt.mw.tu-muenchen.de/de/wissenschaft/veroeffentlichungen.phtml 148

[Tur08] TURSKI, Zygmond:*OBJECT-ORIENTED SYSTEMS ENGINEERING METHODOLOGY - An Overview of OOSEM*. URL: http://www.oei-edu.com/sr657.htm 57

[Ull07] ULLENBOOM, Christian: *978-3-8362-1146-8*. Bd. 7: *Java ist auch eine Insel - Programmieren mit der Java Standard Edition Version 6*. Galileo Computing, 2007 267

[VDI07] http://www.vdi.de/vdi/vrp/richtliniendetails/index.php?ID=9509528 2.2

[Ver06a] http://www.utzverlag.de/suche.php?criteria=systems+engineering&x=0&y=0 1.2, 148, 3.6.2.2

[Ver06b] http://www.smeal.psu.edu/misweb/systems/sycoipo.html 3.4.1

[W3C08] http://www.edition-w3c.de/TR/2000/REC-xml-20001006/ 268

[Wal08] WALTER, Ulrich Prof. Dr. rer. n.: Systems Engineering / Technische Universität München. 2008 (1). – Vorlesungsskript 1.1

[Web07] http://www.iso.org/iso/en/ISOOnline.frontpage 232

[Wei06] WEILKIENS, Tim: *Systems Engineering mit SysML/UML*. dpunkt.verlag, 2006 1.4, 59, 2, 2.6, 3.4.3.1.2, 3.4.3.1.2, 3.4.4.1, 3.4.4.1.1, 3.57, 3.4.4.1.3, 3.58, 3.4.4.1.3, B.4

[Wie06] http://www.phillex.de/axiom.htm 3.4.2.1

[Wik06a] http://de.wikipedia.org/wiki/Attribut 3.4.2.1

[Wik06b] http://de.wikipedia.org/wiki/Backus-Naur-Form 74, F.1

[Wik06c] http://de.wikipedia.org/wiki/Datenkapselung_Programmierung 138

[Wik06d] http://en.wikipedia.org/wiki/Endianness 107

[Wik06e] http://de.wikipedia.org/wiki/Hierarchie 3.4.2.1

[Wik06f] http://de.wikipedia.org/wiki/IEEE_754 285

[Wik06g] http://de.wikipedia.org/wiki/Klassifikation A

[Wik06h] http://de.wikipedia.org/wiki/Objekt_Programmierung 3.4.2.2, C.6

[Wik06i] http://de.wikipedia.org/wiki/Relation A

[Wik06j] http://de.wikipedia.org/wiki/Adjazenzmatrix_(Graphentheorie) 3.4.2.1, 307

[Wik06k] http://de.wikipedia.org/wiki/Schwarzes_Loch D

[Wik06l] http://de.wikipedia.org/wiki/SI-Einheitensystem 90, 2.2, F.1

[Wik06m] http://de.wikipedia.org/wiki/Stephen_Hawking 314

[Wik07a] http://de.wikipedia.org/wiki/Drehgestell 4

[Wik07b] http://de.wikipedia.org/wiki/Ethen 47

[Wik07c] http://de.wikipedia.org/wiki/Glitch_(Elektronik) 207

[Wik07d] http://de.wikipedia.org/wiki/Hardware_in_the_Loop 3.6.2.3.2

[Wik07e] http://de.wikipedia.org/wiki/Hauptprozessor 215

[Wik07f] http://de.wikipedia.org/wiki/IEC_61508 128, F.1

[Wik07g] http://en.wikipedia.org/wiki/Information_hiding 68

[Wik07h] http://en.wikipedia.org/wiki/IPO_Model 3.4.1

[Wik07i] http://de.wikipedia.org/wiki/Karnaugh-Veitch-Diagramm 208

[Wik07j] http://de.wikipedia.org/wiki/Komplexitaet_(Informatik) F.1

[Wik07k] http://de.wikipedia.org/wiki/Komplexitaetstheorie F.1

[Wik07l] http://de.wikipedia.org/wiki/Mathematisches_Objekt C.5

[Wik07m] http://de.wikipedia.org/wiki/Message_Sequence_Chart 76

[Wik07n] http://de.wikipedia.org/wiki/MISRA-C 231

[Wik07o] http://de.wikipedia.org/wiki/Nachbedingung F.1

[Wik07p] http://de.wikipedia.org/wiki/Objekt 2.5

[Wik07q] http://de.wikipedia.org/wiki/Programmablaufplan 111

[Wik07r] http://de.wikipedia.org/wiki/Random_Access_Memory 216

[Wik07s] http://de.wikipedia.org/wiki/Relation_(Mathematik) A, F.1

[Wik07t] http://de.wikipedia.org/wiki/Signatur_(Programmierung) F.1

[Wik07u] http://de.wikipedia.org/wiki/SCSI 241

[Wik07v] http://de.wikipedia.org/wiki/Systemtechnik C

[Wik07w] http://de.wikipedia.org/wiki/Tupel A.2

[Wik07x] http://de.wikipedia.org/wiki/Harmonisierte_Typenkurzzeichen_von_Leitungen 290

[Wik07y] http://de.wikipedia.org/wiki/VHDL 3.6.9

[Wik07z] http://de.wikipedia.org/wiki/Vorbedingung F.1

[Wik08a] http://de.wikipedia.org/wiki/Aussagenlogik 2.5.1

[Wik08b] http://de.wikipedia.org/wiki/Bitwertigkeit 107

[Wik08c] http://de.wikipedia.org/wiki/Byte-Reihenfolge 107

[Wik08d] http://de.wikipedia.org/wiki/Chomsky-Hierarchie 77

[Wik08e] http://de.wikipedia.org/wiki/Eigenschaft 2.9

[Wik08f] http://en.wikipedia.org/wiki/Embedded_system F.1

[Wik08g] http://embedded-system.net/lang/de/reference/what-is-embedded-system/ F.1

[Wik08h] http://de.wikipedia.org/wiki/Ex_falso_quodlibet 63, 2.5.1

[Wik08i] http://de.wikipedia.org/wiki/Extensible_Markup_Language 268

[Wik08j] http://de.wikipedia.org/wiki/Gantt-Diagramm 17

[Wik08k] http://de.wikipedia.org/wiki/Java_(Programmiersprache) 267

[Wik08l] http://de.wikipedia.org/wiki/Kelvin 93

[Wik08m] http://de.wikipedia.org/wiki/Projektphase 13

[Wik08n] http://de.wikipedia.org/wiki/Schmitt-Trigger 320

[Wik08o] http://de.wikipedia.org/wiki/Simultaneous_Engineering 6

[Wik08p] http://de.wikipedia.org/wiki/Syntax B.5

[Wik08q] http://de.wikipedia.org/wiki/Tooltip 305

[Win08] WINKLER, Johannes: *Modellierung von Verhaltenskompatibilität mit MSCs*. Boltzmannstraße 3, Technische Universität München, Diplomarbeit, 2008 37, 2.3.2.3, 2.5.2

[Wis07] http://www.itwissen.info/definition/lexikon///__arbitration_arbitrierung.html F.1

[Wis08a] http://www.itwissen.info/definition/lexikon/Semantik-semantics.html B.6

[Wis08b] http://www.itwissen.info/definition/lexikon/Syntax-syntax.html B.5

[WK06] http://www.klasse.nl/ocl/ocl-introduction.html 3.6.2.6.3

[Zim06] http://www.www-kurs.de/gloss_k.htm 2.3.2.2.1.2

[Zim07] ZIMMERMANN, Tobias: Regelparser für (U)CML-ed-Implementierung in JAVA / Technische Universität München. Version: 2007. http://www.lrt.mw.tum.de/bibliothekdb/anzeige.phtml?bID=6600. 2007. – Interdisziplinäres Projekt (IDP) 3.6.5, 3.6.5.2.1, 266

[ZSL] ZELLER, Andreas ; SNELTING, Gregor ; LINDING, Christian:*Software-Definition - Einführung in die Softwaretechnik*. URL: http://www.st.cs.uni-sb.de/edu/einst/02-definition.pdf 2.2.1