Institut für Informatik
der Technischen Universität München

Von separaten Interaktionsmustern zu konsistenten Spezifikationen reaktiver Systeme

Alexander Harhurin

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Francisco Javier Esparza Estaun

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Dr. h.c. Manfred Broy
2. Univ.-Prof. Dr. Martin Wirsing,
   Ludwig-Maximilians-Universität München

Die Dissertation wurde am 03.05.2010 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 23.09.2010 angenommen.

Kurzfassung

Durch den Bedarf für eine wachsende Anzahl softwarebasierter Funktionen bei einer konstanten Anzahl von Steuergeräten eines eingebetteten Systems ist die stärkere Interaktion von Funktionen verschiedener Steuergeräte unumgänglich. Seit einiger Zeit werden sie kombiniert, um komplexere Funktionen zu generieren. Funktionen erstrecken sich über mehrere Steuergeräte hinweg und können somit nicht mehr isoliert durch eine Software-Komponente implementiert werden. Mit zunehmender Vernetzung von Funktionen wird das Erfassen ihres Zusammenwirkens erschwert. „Welche Funktionen bietet das Gesamtsystem an?“ „Wie beeinflussen sie sich gegenseitig?“ Die Beantwortung dieser Fragen ist Voraussetzung für den Erfolg eines Software-Projekts.

In dieser Arbeit wird eine Modellierungstechnik zur Black-Box-Spezifikation der Funktionalität eines Systems eingeführt. Funktionen werden durch Interaktionsmuster zwischen dem System und seiner Umgebung beschrieben. Interaktionsmuster legen das externe Verhalten des Systems fest, ohne Vorentscheidungen bezüglich des Systemdesigns zu treffen. Durch die Kombination eigenständig spezifizierter Funktionen wird das Gesamtverhalten des Systems erlangt. Somit entsteht eine Spezifikation eines multifunktionalen Systems aus unterschiedlichen Benutzerperspektiven. Verschiedene Stakeholder beschreiben ihre Anforderungen in separaten Modellen, die anschließend unter Berücksichtigung ihrer Wechselwirkungen zu einer Gesamtspezifikation integriert werden.

Der Spezifikationstechnik liegt eine operationelle Semantik zugrunde. Interaktionsmuster werden mittels partieller I/O-Automaten modelliert. Die Integration der Interaktionsmuster zu einer Gesamtspezifikation ist in der formalen Fundierung durch Operatoren zur Kombination von Automaten definiert. Diese Operatoren können Automaten mit gemeinsamen Ausgabevariablen kombinieren. Die Wechselwirkungen zwischen Funktionen werden explizit in die Modellierung aufgenommen.

In dieser Arbeit werden automatische Analyseverfahren erarbeitet, die es erlauben, Widersprüche zwischen Anforderungen zu identifizieren und sie zu beseitigen. In einem Projekt, in dem Anforderungen von verschiedenen Stakeholdern formuliert werden, ist die Widerspruchsfreiheit der Spezifikation eine Voraussetzung für den Erfolg.

Die Struktur einer Architektur eines Systems ist prinzipiell unabhängig von der Struktur dessen Funktionen. In dieser Arbeit wird ein formaler Übergang von Anforderungen zu einer Architektur vorgestellt. Die eingeführte Transformation einer Hierarchie von Interaktionsmustern in ein Netzwerk logischer Komponenten stellt die Erhaltung aller spezifizierten Eigenschaften in der resultierenden Architektur sicher.

Auf Basis der eingeführten operationellen Semantik wurde ein prototypisches Werkzeug zur Modellierung von Interaktionsmustern implementiert, das die theoretischen Ergebnisse dieser Arbeit praktisch nutzbar macht. In diesem Werkzeug wurden zwei Fallstudien erarbeitet, welche die Anwendbarkeit der eingeführten Modellierungstechnik nachweisen.

Danksagung

An erster Stelle geht mein Dank für die hilfreiche Unterstützung bei der Erstellung meiner Dissertation an meinen Doktorvater Prof. Manfred Broy. Seltene aber sehr fruchtbare Diskussionen mit ihm haben meine Arbeit maßgeblich beeinflusst. Auch seine wertvollen Anmerkungen zur Ausarbeitung meiner Dissertation habe ich stets sehr geschätzt. Mein Dank gebührt auch Prof. Martin Wirsing für die Übernahme des Zweitgutachtens und für seine Unterstützung meines Promotionsverfahrens. Er hat mich beginnend mit der Vorlesung Informatik I im ersten Semester an der LMU über meine Diplomarbeit bis zum Rigorosum begleitet.

Großen Dank schulde ich meiner Kollegin Judith Thyssen. Die Zusammenarbeit mit ihr war eine wichtige Voraussetzung für das Entstehen dieser Arbeit. In zahlreichen – oft sehr kontrovers geführten – Diskussionen hat sie mir wertvolle Anregungen für meinen Denkprozess gegeben. Ohne ihre konstruktive Kritik wäre meine Forschungsarbeit niemals soweit gekommen.

Besonderer Dank gilt auch meinem langjährigen Zimmerkollegen und Mitautor Jewgenij Botaschanjan, der mich mit formalen Methoden der Informatik vertraut gemacht hat. Er verstand es stets meinen kreativen Ausschweifungen den nötigen mathematischen Unterbau zu verpassen.

Bei meinen Kollegen David Cruz und Christian Leuxner möchte ich mich für viele (auch fachübergreifende) Diskussionen und Anregungen bedanken. Ihre zahlreichen Korrekturen und Hinweise haben die Qualität meiner Dissertation deutlich verbessert. Für das aufmerksame Korrekturlesen und ihre Verbesserungsvorschläge möchte ich auch Felix Leis, Elena und Trixi Hotze danken. Für die Unterstützung im Umgang mit AutoFocus danke ich Florian Hölzl. Silke Müller danke ich insbesondere für ihre Hilfsbereitschaft und die fröhliche Atmosphäre am Lehrstuhl.

Besonderer Dank ergeht an meine Eltern Margarita und Vladimir sowie an meinen Bruder Anatoliy Kharkhurin für alles, was sie mir auf den Weg gegeben haben. Großen Dank schulde ich meiner Freundin Mascha Druker für die Geduld und den Rückhalt, den sie mir während der ganzen Arbeit immer wieder gegeben hat. Ihr Verständnis für ausgefallene Urlaube und einsame Wochenenden weiß ich sehr zu schätzen. Ich danke Olga Domnitsch für ihren Glauben an mich und ihre Zuversicht.

Inhaltsverzeichnis

Abbildungsverzeichnis

KAPITEL 1

Einführung

Eine angemessene Lösung für ein Problem zu finden, ist oft nicht trivial. Eine präzise Beschreibung des Problems ist der erste Schritt auf dem Weg zur Problemlösung. In einem Software-Projekt, in dem von verschiedenen Beteiligten formulierte Anforderungen von anderen Entwicklern realisiert werden, ist die Eindeutigkeit und Konsistenz der Anforderungsspezifikation eine Voraussetzung für den Erfolg des Projekts.

In der vorliegenden Arbeit wird eine Technik zur formalen Spezifikation funktionaler Anforderungen eingeführt. Das angestrebte Ziel ist eine Spezifikation des Gesamtverhaltens eines reaktiven Systems, die sich aus der Kombination separater, (zum Teil) unvollständiger Interaktionsmuster ergibt und verschiedene Aspekte des Verhaltens aus unterschiedlichen Benutzerperspektiven beschreibt. Eine anschließende Konsistenzanalyse stellt sicher, dass die funktionalen Anforderungen konsistent (d.h. widerspruchsfrei) sind. Die formale Spezifikation lässt sich anschließend automatisch in ein Modell der Lösungsdomäne transformieren.

Inhalt

Diese Einführung gibt einen Überblick über die vorliegende Arbeit. Die Bedeutung der modellbasierten Entwicklung im Allgemeinen und die der formalen Spezifikation im Besonderen werden in Abschnitt 1.1 verdeutlicht. Anschließend werden offene Fragestellungen und der Handlungsbedarf in diesem Bereich abgeleitet. Um die aufgezeigten Probleme anzugehen, wird in Abschnitt 1.2 ein dienstbasierter Ansatz zur Spezifikation der Systemfunktionalität vorgeschlagen. In Abschnitt 1.3 wird der Ansatz in den Entwicklungsprozess eingeordnet und dessen Eignung für verschiedene Systemtypen diskutiert. Schließlich gibt Abschnitt 1.4 einen Überblick über die Struktur und den Aufbau dieser Arbeit.

1.1. Motivation und Problemstellung

Die Entwicklung eingebetteter Software war bisher – geprägt durch den klassischen Zulieferprozess – auf eine Hardware-Topologie ausgerichtet. Eine Familie verwandter *Nutzerfunktionen*[1] entsprach vielfach eins-zu-eins einem Steuergerät. Es bestand nur wenig Interaktion zwischen den Funktionen unterschiedlicher Steuergeräte. Als Folge dessen ist der Stand der Technik in der Industrie der Einsatz komponentenbasierter Ansätze, die durch Kommunikationstechnologien und Netzwerkprotokolle zwischen Hardware-Komponenten geprägt sind [BKPS07, S. 370].

Durch den Bedarf für eine wachsende Anzahl softwarebasierter Nutzerfunktionen bei einer konstanten Anzahl von Steuergeräten ist die stärkere Interaktion von Funktionen verschiedener Steuergeräte unumgänglich – seit einiger Zeit werden Funktionen kombiniert, um komplexere Funktionen zu generieren. Dadurch kann eine ausschließlich steuergeräteorientierte Architektur der gestiegenen Komplexität hinsichtlich Entwicklungsqualität und -effizienz nicht mehr Rechnung tragen. Die Nutzerfunktionen beschränken sich nicht mehr nur auf ein Steuergerät und können somit auch nicht mehr isoliert durch einen Zulieferer bereitgestellt werden [BKPS07, S. 357].

Mit zunehmender Vernetzung von Funktionen wird es schwieriger, einen Überblick über folgende Fragestellungen zu bekommen [BHRS08]:

- Welche Funktionen bietet das Gesamtsystem an?
- Welche Funktionen beeinflussen sich gegenseitig und auf welche Weise?
- Durch welche Komponenten werden die Funktionen erbracht?

Da diese Informationen jedoch Voraussetzung für den Erfolg eines Software-Projekts sind, wird eine strukturierte und präzise Beschreibung von Nutzerfunktionen und ihrer Wechselwirkungen benötigt, welche von einer späteren Implementierung abstrahiert.

Die heutzutage eingesetzten Techniken zur Beschreibung von Nutzerfunktionen können den oben genannten Herausforderungen nicht Rechnung tragen.

[1] Eine Nutzerfunktion ist ein Dienst, den das System seiner Umgebung über eine Schnittstelle zur Verfügung stellt. Eine Nutzerfunktion charakterisiert die Nutzung des Systems zu einem bestimmten Zweck, d.h. seine Reaktion auf bestimmte Eingaben und sein Verhalten in bestimmten Situationen.

Hierfür gibt es mehrere Gründe:

- Die meisten Techniken zur Beschreibung von Nutzerfunktionen weisen Mängel in der Präzision und Eindeutigkeit auf, da sie keine formale Semantik besitzen.
- Formal fundierte Modellierungstechniken sind für die Beschreibung von Nutzerfunktionen nicht geeignet, da sie für den komponentenbasierten Entwurf des Systems entwickelt wurden.
- In den meisten formalen Ansätzen (siehe z.B. [Hoa85, Mil89, BS01]) ist es schwierig, das Systemverhalten aus verschiedenen Perspektiven zu beschreiben, da einzeln spezifizierte Nutzerfunktionen nicht ohne explizite Designschritte zu einer größeren Spezifikation kombiniert werden können.
- In den meisten Ansätzen kann eine von der Implementierung abstrahierte Wechselwirkung zwischen Nutzerfunktionen nicht modelliert werden.
- Der Übergang von der funktionalen Spezifikation zum Entwurf ist in den meisten Ansätzen mit Problemen behaftet, da die Modelle der Spezifikation und des Entwurfs auf unterschiedlichen formalen Grundlagen basieren.

Im Folgenden wird auf diese Mängel heutiger Ansätze im Detail eingegangen und daraus der Handlungsbedarf in diesem Bereich abgeleitet.

Unpräzise Spezifikationen Eine Spezifikation ist in einem (geographisch verteilten) Entwicklungsprozess das wichtigste Kommunikationsmittel zwischen den an der Entwicklung beteiligten Personen. Alle Design- und Implementierungsentscheidungen, alle Verifikations- und Validierungsschritte basieren auf der Spezifikation. Somit hängt die Qualität eines Systems unmittelbar von der Qualität einer Spezifikation ab. Die Anforderungen müssen so präzise festgehalten werden, dass sie von allen Beteiligten eindeutig interpretiert werden können. „Prinzipiell sollte die Spezifikation der funktionalen Anforderungen an ein System sowohl vollständig als auch konsistent sein. Vollständigkeit bedeutet, dass die Spezifikation alle vom Benutzer benötigten Dienste festlegt, während die Konsistenz verlangt, dass die Anforderungen keine widersprüchlichen Festlegungen enthalten.“ [Som01, S. 110]. Viele Probleme der Software-Entwicklung entstehen durch die Ungenauigkeit der Spezifikation. Diese Ungenauigkeit erschwert eine präzise Problemformulierung und damit eine angemessene Lösungsfindung.

Eine Voraussetzung für eine präzise Spezifikation ist eine formale Semantik, die jedoch vielen heute in der Praxis eingesetzten Ansätzen zur Erfassung von Anforderungen (wie z.B. Rational DOORS[2] oder UML[3] Use-Case-Diagrammen) fehlt. Dementsprechend sind die erfassten Modelle mehrdeutig und können nicht automatisch analysiert werden.

Demzufolge besteht noch Bedarf für eine formal fundierte und gleichzeitig für Anforderungsanalytiker intuitive Technik zur Spezifikation funktionaler Anforderungen.

[2] http://www.ibm.com/software/awdtools/doors/
[3] http://www.uml.org/

Komponenten statt Nutzerfunktionen Typischerweise dienen heutige Systeme nicht mehr nur einer spezifischen Funktion, sondern bieten eine umfangreiche Familie interagierender Funktionen an. Beispielsweise sind heutzutage in einem Auto der Luxusklasse über 2000 softwarebasierte Funktionen implementiert, die sich über ca. 70 Steuergeräte hinweg erstrecken [BKPS07].

Die in der Praxis etablierten komponentenbasierten Ansätze (wie z.B. UML Komponentendiagramme, MathWorks Simulink[4] oder ASCET-SE[5]) konzentrieren sich ausschließlich auf die innere Struktur des Systems und bieten keine Möglichkeiten, von der komponentenbasierten Lösungsdomäne zu abstrahieren und Nutzerfunktionen (d.h. die Problemdomäne) explizit darzustellen. Die meisten Funktionen eines Systems werden durch eine Interaktion mehrerer Komponenten erbracht. Es handelt sich um so genannte *crosscutting* (oder komponentenübergreifende) Funktionen. Für die Anforderungsanalyse ist es jedoch wichtig, sich auf einzelne Funktionen und nicht auf Interaktionen von Komponenten zu konzentrieren. Denn die Modellierung von Funktionen durch interagierende Komponenten stellt eine Vorentscheidung bezüglich einer möglichen Implementierung dar und sollte in der Anforderungsanalyse vermieden werden.

Um einer wachsenden Anzahl kombinierter Funktionen Rechnung zu tragen, ist eine funktionsorientierte Modellierungstechnik erforderlich, die von der inneren Struktur des Systems abstrahiert bzw. sich mit der Spezifikation der Systemfunktionalität aus der Black-Box-Sicht befasst.

Interaktionsmuster ohne Semantik Wenn ein Auftraggeber einen Auftrag an seine Zulieferer vergibt, sollte er seine Anforderungen in einem Lastenheft in einer so abstrakten Form definieren, dass der Lösung nicht vorgegriffen wird. Es ist wichtig, dem Entwickler die Möglichkeit zu geben, geeignete Lösungen zu erarbeiten, ohne ihn durch zu restriktive Anforderungen in seiner Lösungskompetenz einzuschränken. Anwendungsfälle sind eine geeignete Spezifikationstechnik, das Systemverhalten so allgemein wie möglich und so einschränkend wie nötig zu formulieren [JBR99, Kapitel 3]. *Stakeholder*[6] finden es gewöhnlich einfacher, einen Bezug zu realen Anwendungsfällen herzustellen als zu einer inneren Struktur des Systems. Sie können einen Anwendungsfall, welcher ihre Rolle im Zusammenhang mit dem System beschreibt, verstehen und hinterfragen [Som01, S. 142]. In der Domäne reaktiver Systeme ist ein Anwendungsfall ein *Interaktionsmuster*, d.h. ein zeitlich geordneter Austausch von Nachrichten zwischen dem System und seiner Umgebung.

Modellierungstechniken, die das reaktive Systemverhalten als eine Menge von Interaktionsmustern beschreiben, existieren bereits in der Praxis, allen voran die UML-Sequenzdiagramme. Sie sind allerdings „ohne die fundamental ausgearbeitete, verstandene und

[4] http://www.mathworks.com/products/simulink/

[5] http://www.etas.com/de/products/ascet_se_software_engineering.php

[6] Stakeholder repräsentieren eine Abstraktion, indem ein Stakeholder jeweils die Zusammenfassung aller Personen mit gleicher Interessenlage und gleicher Sicht auf das System repräsentiert. In der vorliegenden Arbeit sind Stakeholder die Informationslieferanten für Anforderungen an das System.

trotz vieler Einzelansätze fehlende allgemein anerkannte Semantik“ [BR07].

Es besteht also noch Handlungsbedarf, für diese Techniken einen theoretischen Kern zu erarbeiten, insbesondere einen Operator zur Kombination von Interaktionsmustern. Außerdem ist es für eine automatische Analyse von Spezifikationen wichtig, dass dieser Operator eine operationelle Semantik besitzt.

Einziges Modell des Systemverhaltens Typischerweise beschränken sich Nutzerfunktionen der heutigen Systeme nicht mehr auf einen einzelnen Benutzer und können somit nicht isoliert durch einen Stakeholder beschrieben werden. Bei größeren Systemen gibt es normalerweise verschiedene Typen von Benutzern, die unterschiedliche Anforderungen an die Systemfunktionalität haben [Som01, S. 135]. Mehrere Stakeholder spezifizieren das System aus unterschiedlichen Benutzerperspektiven, indem sie nur für sie relevante Anwendungsfälle beschreiben. In diesem Sinne kann eine Spezifikation aus mehreren Modellen verschiedener Benutzerperspektiven bestehen. Für die erfolgreiche Validierung einer Spezifikation ist es wichtig, dass die Modelle einzelner Perspektiven in der Spezifikation klar erkennbar und von einander trennbar sind.

Die in der Praxis etablierten Ansätze setzen ein einziges Modell des Systemverhaltens voraus, in das Anforderungen aus allen Informationsquellen zusammenfließen. Beschreibt man beispielsweise das Verhalten ein und desselben Systems mit mehreren UML-Sequenzdiagrammen, gibt es in der UML keinen Operator, durch den diese Modelle kombiniert werden könnten[7]. Mehrere Stakeholder wenden sich an einen Anforderungsanalytiker, der alle Anforderungen konsolidiert und in ein Modell zusammenfasst. Anforderungen mehrerer Stakeholder in einem Schritt in die Gesamtspezifikation zu integrieren, birgt die Gefahr, dass einige von ihnen fehlerhaft dargestellt oder unberücksichtigt bleiben. Die neu erhobenen Anforderungen können unter Einfluss der bereits existierenden Gesamtspezifikation verzerrt werden.

Um die wachsende Komplexität der heutigen Systeme zu beherrschen, sind Spezifikationstechniken erforderlich, welche die Modellierung unter Berücksichtigung mehrerer Benutzerperspektiven ermöglichen und sie sowohl zur Organisation der Anforderungserhebung als auch zur Strukturierung der Spezifikation benutzen [Som01, S. 136].

Feature Interaction Wie bereits erwähnt, bieten heutige eingebettete Systeme eine Vielfalt von Funktionen an, die meist nicht isoliert, sondern in Abhängigkeit von einander agieren. Die Komplexität und die Größe der Systeme führen dazu, dass sich die Funktionen manchmal in unvorhergesehener und unerwünschter Weise gegenseitig beeinflussen. Dieses Problem ist unter dem Begriff „feature interaction“ bekannt. Eine prominente Definition dieses Problems stammt von Zave: „A feature interaction is some way in which a feature or features modify or influence another feature in defining

[7] Die MSC-Operatoren `alt` und `par` (vgl. [IT99]) sind für die Konjunktion von Anforderungen nicht geeignet. Während in der `alt`-Kombination ein Verhalten aus mehreren gewählt wird, besitzt der `par`-Operator eine Interleaving-Semantik. Für die Konjunktion von Anforderungen aus unterschiedlichen Perspektiven ist jedoch ein synchron nebenläufiger Operator notwendig (vgl. [Bro05a, Kapitel 2.6]).

the overall system behavior“ [Zav03]. Die Nachvollziehbarkeit des Systemverhaltens wird in der Regel immer schwieriger. Sind bestimmte Funktionen unabhängig voneinander? Welche Auswirkung hat eine Funktion auf eine andere? Die Beantwortung dieser Fragen erfordert Wissen über die Zusammenhänge zwischen den Funktionen. Unerwünschte oder unberücksichtigte Wechselwirkungen zwischen Funktionen stellen einen erheblichen Qualitätsmangel einer Spezifikation dar.

„So far, the understanding of these feature interactions between the different functions ... is still in its infancy“ [BKPS07]. Die meisten Ansätzen zum *Feature Interaction* (wie z.B. [JZ98, Zav03, FN03]) überprüfen die Erfüllung von Eigenschaften durch sequentiell komponierte Komponenten. Den unerwünschten Wechselwirkungen zwischen Nutzerfunktionen innerhalb einer Spezifikation wurde bis jetzt nicht viel Aufmerksamkeit gewidmet.

In der Praxis besteht also Bedarf für Analyseverfahren, welche die unberücksichtigten interfunktionalen Wechselwirkungen in der Spezifikation (am besten automatisch) entdecken können.

Fehlende Integration formaler Modelle Die Rolle der formal fundierten Spezifikation im Software-Engineering wurde bereits hervorgehoben. Nicht weniger wichtig für den Erfolg eines Software-Projekts ist die modellbasierte Entwicklung, bei der das System auf einem hohen Abstraktionsgrad konzipiert und schrittweise um Implementierungsdetails erweitert wird. „An attractive vision is an integrated modeling approach that captures the relationship between all the models, and where parts of some models are generated from the models used in previous activities“ [BKPS07]. Die Evolution der Software wird durch die Trennung der technischen Realisierung und der Nutzerfunktionen wesentlich vereinfacht.

Gegenwärtig werden formale Modelle in einzelnen Phasen der Entwicklung erstellt und nicht weiter verwendet, da der semantische Zusammenhang zu anderen Modellen fehlt. Diese Insellösungen sind für den Entwicklungsprozess von geringem Nutzen. Präzise definierte Spezifikationen und Architekturen haben nur geringen Mehrwert für die Qualität des Software-Systems, falls sie auf unterschiedlichen Theorien basieren und demzufolge in keiner formalen Beziehung zueinander stehen. Ein klassisches Beispiel einer lückenhaften modellbasierten Entwicklung ist die UML. „Die Stärken einzelner in der UML enthaltener Theorien sind nicht integriert und daher auch nicht integriert nutzbar“ [BR07].

Es besteht also Bedarf für eine zusammenhängende Kette von Abstraktionsebenen entlang den klassischen Phasen des Entwicklungsprozesses. Die Anforderungs-, Entwurfs- und Implementierungsmodelle müssen auf denselben theoretischen Grundlagen basieren.

1.2. Beitrag der Arbeit

Der Beitrag dieser Arbeit ist ein formales Framework für die Spezifikation und Analyse der Gesamtfunktionalität eines Systems, dessen Verhalten sich aus der Kombination

einzelner Nutzerfunktionen ergibt. Die Funktionen werden dabei in Form von Interaktionsmustern aus der Black-Box-Sicht beschrieben, so dass sie für Benutzer verständlich sind, ohne dass Wissen über die Implementierung vorausgesetzt wird. Sie legen nur das externe Verhalten des Systems fest und vermeiden Vorentscheidungen bezüglich des Systemdesigns.

Modulare Spezifikation des Schnittstellenverhaltens In dieser Arbeit wird eine Spezifikationstechnik erarbeitet, die es erlaubt, das Systemverhalten aus verschiedenen Benutzerperspektiven zu beschreiben. Im ersten Schritt werden Teilverhalten des Systems aus unterschiedlichen Benutzerperspektiven von zuständigen Stakeholdern in separaten Modellen spezifiziert. Dabei können bestimmte Aspekte des Verhaltens widersprüchlich oder für eine Benutzergruppe irrelevant (und somit in der Perspektive dieser Benutzergruppe unterspezifiziert) sein. Erst in einem darauffolgenden Schritt werden diese Modelle zu einer Gesamtspezifikation integriert. Somit entsteht eine Spezifikation aus sich gegenseitig beeinflussenden Modellen unterschiedlicher Perspektiven.

Die separate Darstellung von Perspektiven ist für die gesamte Anforderungsanalyse von Nutzen:

- Die Erhebung von Anforderungen wird von deren Integration in die Gesamtspezifikation klar abgegrenzt.
- Widersprüchliche Anforderungen (verschiedener Stakeholder) werden nicht „on-the-fly“ während ihrer Erhebung, sondern in einem separaten Prozess explizit mittels geeigneter Verfahren behandelt.
- Bei einer Trennung von Perspektiven kann ein Stakeholder seine individuellen Anforderungen isoliert von anderen validieren, was zur Beherrschung der Komplexität der Spezifikation beiträgt.

Operationelle Semantik von Interaktionsmustern Die in dieser Arbeit vorgeschlagene Beschreibungstechnik basiert auf der FOCUS-Theorie von Broy [BS01] und der Automaten-Theorie [HU79].

Eine formal fundierte Spezifikation von Anforderungen bringt mehrere Vorteile mit sich:

- Die Spezifikation ist *eindeutig*. Nur eine eindeutig interpretierbare Spezifikation kann als Vertrag zwischen den Kunden und den Entwicklern eines Systems dienen.
- Die Spezifikation ist *automatisch analysierbar*. Die operationelle Semantik der Spezifikation ermöglicht ihre automatische Analyse auf Konsistenz und Korrektheit.
- Die Spezifikation dient einem nachweisbar korrekten *Übergang zum Design* des Systems als Basis.

Konsistenzanalyse Um die Eindeutigkeit einer Spezifikation sicherzustellen, werden in dieser Arbeit mehrere automatische Analyseverfahren erarbeitet, die es erlauben, Wi-

dersprüche zwischen Anforderungen zu identifizieren. Zusätzlich zur Analyse von Anforderungen, bilden algorithmische Verfahren zur Eliminierung von Widersprüchen einen weiteren Beitrag der vorliegenden Arbeit. In einem Projekt, in dem Anforderungen von verschiedenen Stakeholdern formuliert werden, ist die Widerspruchsfreiheit der Spezifikation eine Voraussetzung für den Erfolg.

Integration zweier Abstraktionsebenen Der in dieser Arbeit vorgeschlagene Ansatz ist in ein 3-Ebenen-Modell für das Engineering eingebetteter, Software-intensiver Systeme [BFG+08] integriert. Das 3-Ebenen-Modell definiert drei Abstraktionsebenen entlang den klassischen Phasen des Entwicklungsprozesses: die funktionale, logische und technische Architektur (vgl. Abbildung 1.1). Während die FOCUS-Theorie von Broy [BS01] die theoretischen Grundlagen und das Werkzeug AutoFocus[8] eine Tool-Unterstützung für die logische Architektur liefern, ist die funktionale Spezifikation immer noch eine der großen Herausforderungen in dem 3-Ebenen-Modell. Außerdem ist die konzeptuelle Lücke zwischen den beiden oberen Abstraktionsebenen bis heute ein ungelöstes Problem im Entwicklungsprozess.

Die vorliegende Arbeit trägt zur Lösung dieses Problems bei, indem der Übergang zwischen diesen Abstraktionsebenen definiert wird. Die Semantik der funktionalen Spezifikation wird in der Arbeit auf denselben theoretischen Grundlagen wie die der logischen Architektur definiert. Diese umfassende Modellierungstheorie ermöglicht einen formalen und lückenlosen Übergang von Anforderungen zum Design.

1.3. Einordnung und Abgrenzung

In diesem Abschnitt werden verschiedene Klassen von Systemen aufgezählt, für die die vorgeschlagenen Konzepte geeignet sind. Anschließend wird der Ansatz in den Entwicklungsprozess für eingebettete Systeme eingeordnet.

1.3.1. Für den Ansatz geeignete Systemklassen

Der vorgeschlagene Ansatz ist in erster Linie für die Spezifikation *multifunktionaler* Systeme geeignet, d.h. Systeme die mehrere einander beeinflussende Nutzerfunktionen anbieten. Wechselwirkungen zwischen den Funktionen können durch formale Analysen identifiziert und explizit dargestellt werden. Der Ansatz kann folglich auch zur Spezifikation eines monofunktionalen Systems verwendet werden. Allerdings liefert er hier keinen Mehrwert, da sein Nutzen insbesondere in der Analyse und Modellierung des Zusammenspiels mehrerer Funktionen liegt. Ein Beispiel eines multifunktionalen Systems ist ein Auto mit seiner Vielfalt von Funktionen.

Der Ansatz wurde zur Modellierung *reaktiver* und *digitaler* Systeme [MP95] entwickelt. Ein reaktives System befindet sich in ständiger Interaktion mit seiner Umgebung: auf

[8] http://af3.in.tum.de

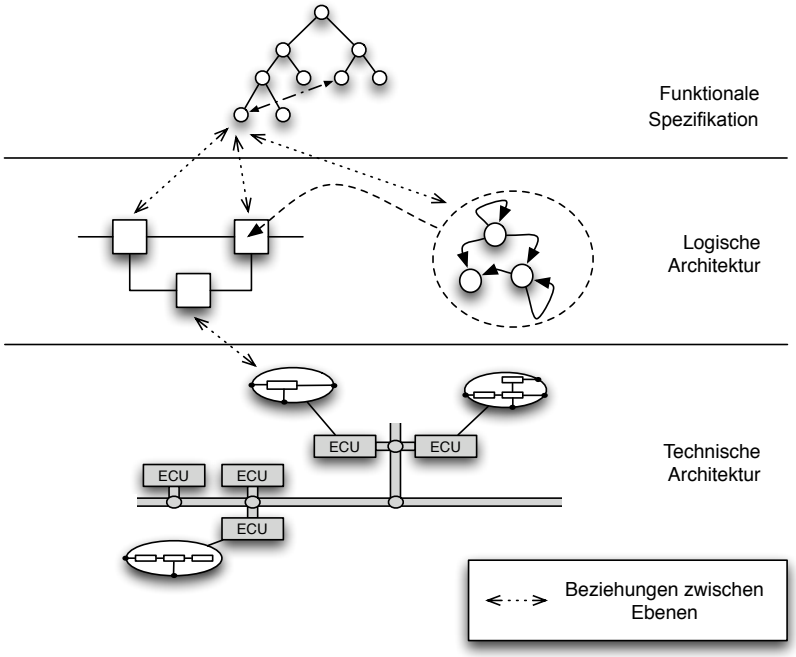


Abbildung 1.1.: 3-Ebenen-Modell [BFG+08]

Eingaben aus der Umgebung reagiert es mit Ausgaben, die innerhalb definierter Zeitgrenzen erfolgen müssen. Ein reaktives System terminiert in der Regel nie. Ein digitales System interagiert mit seiner Umgebung in diskreten Zeitintervallen mittels diskreter Nachrichten. Beispielsweise muss die Airbag-Steuerung im Auto permanent die Messwerte der Sensoren verarbeiten und entscheiden, ob der Airbag ausgelöst wird.

Eine Unterklasse der reaktiven Systeme sind *eingebettete Kontrollsysteme* mit ihren klar definierten Schnittstellen zur physikalischen Umgebung. Ein eingebettetes Kontrollsystem erhält Informationen über den aktuellen Zustand eines mechanischen Prozesses durch Sensoren und steuert den Prozess durch Aktuatoren.

„Die Kosten und Schwierigkeiten bei der Einführung formaler Methoden in den Softwareprozess sind sehr hoch“ [Som01, S. 204]. Deshalb werden formale Ansätze nur auf dem Gebiet *kritischer* Systeme angewendet, bei denen Eigenschaften wie Sicherheit oder Zuverlässigkeit besonders wichtig sind (z.B. Raumfahrt- oder medizinische Kontrollsysteme). Bei diesen Systemen können Fehler nicht toleriert werden.

1.3.2. Einbettung in den modellbasierten Entwicklungsprozess

In Abbildung 1.1 ist das 3-Ebenen-Modell für Software-intensive Systeme aus [BFG+08] im Überblick dargestellt. Es ist grundsätzlich unterteilt in folgende drei Abstraktions-

ebenen (der Abstraktionsgrad nimmt dabei von oben nach unten ab):

- Die *funktionale Spezifikation* befasst sich mit den Nutzerfunktionen aus der Black-Box-Sicht und abstrahiert von allen Implementierungsdetails. Sie strukturiert die Funktionen hierarchisch und definiert Wechselwirkungen zwischen ihnen.
- Die *logische Architektur* strukturiert das System in logische (d.h. von der zugrunde liegenden Hardware unabhängige), kommunizierende Einheiten, deren Gesamtverhalten das in der Spezifikation festgelegte Verhalten realisiert.
- Die *technische Architektur* beschreibt schließlich die Realisierung, welche aus Hardware und Software besteht.

Während die Spezifikation das Problem definiert, ist die logische Architektur das erste Modell der Lösungsdomäne. Die technische Architektur ist die Umsetzung der logischen Architektur in ein System aus Hardware und Software.

Mit der vorliegenden Arbeit wird eine operationelle Semantik für die funktionale Spezifikation des 3-Ebenen-Modells definiert. Darüber hinaus wird ein eigenschaftserhaltender Übergang von den Modellen der funktionalen Spezifikation zu den Modellen der logischen Architektur vorgeschlagen. Der Übergang profitiert von der Tatsache, dass die Modelle der beiden Abstraktionsebenen auf derselben Modellierungstheorie basieren.

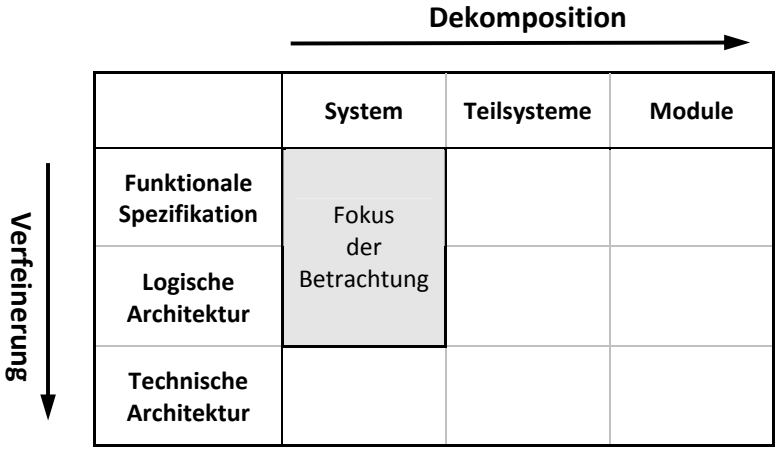


Abbildung 1.2.: Zwei Dimensionen der Systembeschreibung

„Idealerweise sollte eine Spezifikation keinerlei Festlegungen in Bezug auf den Entwurf enthalten. In der Praxis ist dies außer bei sehr kleinen Systemen unrealistisch. Eine Dekomposition (Zerlegung) der Architektur ist notwendig, um die Spezifikation zu strukturieren und zu organisieren. Das Architekturmodell ist häufig der Ausgangspunkt für die Spezifikation der verschiedenen Teile des Systems“ [Som01, S. 225]. Das liegt vor allem daran, dass es auf der notwendigen Detailstufe einer formalen Spezifikation praktisch unmöglich ist, das Systemverhalten aus der Black-Box-Sicht zu beschreiben. Eine anfängliche Architektur kann die Strukturierung der Spezifikation erleichtern, indem die Funktionen entsprechend den Subsystemen aufgeteilt werden. Dementsprechend entste-

hen die Modelle der Spezifikation und die der logischen Architektur parallel zueinander.

Obwohl der Fokus der vorliegenden Arbeit auf der funktionalen Spezifikation und dem Übergang zur logischen Architektur liegt, unterstützen die eingeführten Konzepte – angelehnt an Leveson [Lev00] – zwei Dimensionen der Systembeschreibung (vgl. Abbildung 1.2). Die horizontale Dimension bestimmt die Dekomposition des Systems in Teilsysteme und weiter in Module. Die vertikale Dimension spiegelt die bereits erwähnten Abstraktionsebenen wider. Die Modelle der unteren Ebenen verfeinern die der oberen. Mehr Details zur zweidimensionalen Systembeschreibung finden sich in [TRS+10].

1.4. Aufbau der Arbeit

Im Folgenden wird ein kurzer Abriss des Inhalts und des Aufbaus der Arbeit gegeben.

Kapitel 2 führt ein durchgängiges Fallbeispiel ein, anhand dessen die Konzepte der vorliegenden Arbeit veranschaulicht werden. Als Fallbeispiel wurde eine typische Automobilfunktion, das ACC-System (Adaptive Cruise Control), gewählt.

Kapitel 3 erklärt grundlegende Ideen des Ansatzes sowie seine wichtigsten Merkmale anschaulich, jedoch informell. Hier werden zwei orthogonale Arten der Systembeschreibung – die strukturelle Komposition und die funktionale Kombination – eingeführt und motiviert. Der Beitrag dieses Kapitels ist eine dienstbasierte Beschreibungstechnik zur Spezifikation von Nutzerfunktionen. Die Konzepte und Nutzen dieser Technik im Requirements Engineering werden erläutert.

Kapitel 4 befasst sich mit der operationellen Semantik der dienstbasierten Spezifikationstechnik. Dabei liegt der Fokus der Betrachtung auf einer zustandsbasierten Semantik, die eine automatische Anforderungsanalyse ermöglicht. Der wichtigste Beitrag dieses Kapitels ist die formale Definition der Operatoren zur Kombination von Nutzerfunktionen und der Nachweis ihrer algebraischen Eigenschaften.

Kapitel 5 geht auf mehrere automatische Analyseverfahren ein, um die Eindeutigkeit einer Spezifikation sicherzustellen. Zusätzlich zur Konsistenz- und Korrektheitsprüfung von Anforderungen werden außerdem verschiedene algorithmische Verfahren zur Herstellung der Konsistenz und Korrektheit der Spezifikation erarbeitet. Außerdem wird in diesem Kapitel die formale Integration des vorgestellten Ansatzes in die JANUS-Theorie [Bro05c] von Broy nachgewiesen.

Kapitel 6 erläutert Unterschiede zwischen der Spezifikation und der logischen Architektur und definiert einen formalen Übergang zwischen den beiden Abstraktionsebenen. Dieser Übergang stellt eine eigenschaftserhaltende Transformation der Modelle dar, welche die Erfüllung aller in der Spezifikation spezifizierten Eigenschaften in der resultierenden Architektur gewährleistet.

Kapitel 7 stellt ein prototypisches Werkzeug zur dienstbasierten Spezifikation vor, das die theoretischen Ergebnisse der vorliegenden Arbeit praktisch nutzbar macht.

Kapitel 8 stellt den eingeführten Ansatz den verwandten Arbeiten, allen voran aus den Bereichen des Requirements Engineering und der Automaten-Theorie, gegenüber.

Kapitel 9 fasst die Ergebnisse der Arbeit zusammen und gibt einen Ausblick auf weitere interessante Fragestellungen.

Anhang A beinhaltet einige Beweise der Sätze aus den vorangehenden Kapiteln.

Anhang B zeigt anhand eines Beispiels, wie dienstbasierte Modelle in einem Model Checker analysiert werden können.

Anhang C beschreibt zwei Fallstudien und illustriert dadurch die Anwendbarkeit der Ergebnisse.

Eigene Veröffentlichungen

Die vorliegende Arbeit basiert zum Teil auf den folgenden bereits veröffentlichen Publikationen [BGH+06, GHH07b, GHH07a, BHK08, HH08b, HH08a, BFG+08, BBG+08, BH09b, BH09c, HHR09, TRS+10].

KAPITEL 2

Fallbeispiel

Die in der vorliegenden Arbeit eingeführten Konzepte werden anhand eines durchgängigen Fallbeispiels veranschaulicht. Als Fallbeispiel wurde eine typische Automobilfunktion, das ACC-System (Adaptive Cruise Control), gewählt. Die Angaben zum Fallbeispiel sind größtenteils [PSE07] und [STT+05] entnommen. Für eine ausführliche Beschreibung des Systems sei auf diese Publikationen verwiesen. Alle relevanten Details werden jedoch an den entsprechenden Stellen im Dokument eingeführt. Die vollständige (dienstbasierte) Spezifikation des Systems ist in Anhang C.1 zu finden.

Inhalt

2.1. Informelle Beschreibung

Das elektronische Fahrzeugsystem ACC ermöglicht eine kombinierte Geschwindigkeits- und Abstandskontrolle. Wie ein herkömmlicher Fahrgeschwindigkeitsregeler lässt das ACC das Fahrzeug mit einer gewählten Geschwindigkeit fahren. Holt das eigene Fahrzeug ein vorausfahrendes Fahrzeug ein, bremst das ACC automatisch ab und hält einen vom Fahrer festgelegten Abstand. Sobald sich im Messbereich kein vorausfahrendes Fahrzeug mehr befindet, beschleunigt das ACC das Auto wieder auf die gewählte Geschwindigkeit.

Im Wesentlichen umfasst die ACC-Funktionalität drei Teilfunktionen, die im Folgenden beschrieben werden.

Tempomat Der Tempomat gleicht die Fahrzeuggeschwindigkeit an die vom Fahrer eingestellte Wunschgeschwindigkeit an. Die Erstere kann situationsabhängig mit Motor- oder Bremseneinsatz erhöht, reduziert oder gehalten werden. Bei einer erhöhten Querbeschleunigung in einer Kurve unterbindet das System die automatische Beschleunigung. Am Kurvenausgang beschleunigt das System auf die gewählte Geschwindigkeit.

Abstandskontrolle Sollte die Spur vor dem Fahrzeug nicht frei sein, wird gebremst oder beschleunigt, um entweder den Abstand zum Vordermann oder die eingestellte Geschwindigkeit einzuhalten. Diese Abstandskontrolle basiert auf einem Radarsensor, der aus den reflektierten Signalen vorausfahrender Fahrzeuge Abstand und Relativgeschwindigkeit zum eigenen Fahrzeug berechnet. Der Einsatzbereich der Abstandskontrolle liegt zwischen 30 und 180 km/h – in anderen Bereichen kann sie nicht eingeschaltet werden. Außerdem wird sie analog zum Tempomat auf eine maximale Querbeschleunigung in einer Kurve begrenzt.

Sind die beiden Funktionen aktiviert, stellt das ACC sicher, dass sowohl die Wunschgeschwindigkeit nicht überschritten als auch der minimale Abstand eingehalten werden.

Stop&Go Ist die Abstandskontrolle aktiv und hält das vorausfahrende Fahrzeug an, wird durch die Abstandskontrolle das Fahrzeug bis zum Stillstand abgebremst. Danach erfolgt ein automatisches Anfahren, sofern das Vorderfahrzeug innerhalb von zirka 3 s wieder beschleunigt und der Fahrer danach die Anfahrt über den Bedienhebel bestätigt. Das Fahrzeug folgt daraufhin wieder automatisch dem Vorderfahrzeug. Ab 30 km/h übernimmt die Funktion *Abstandskontrolle* wieder die Steuerung.

Abbildung 2.1.: Lenkstockhebel für ACC [PSE07]

Bedienung Die Bedienung des ACC-Systems erfolgt über einen Lenkstockhebel (siehe Abbildung 2.1). Der Fahrer kann ab einer Fahrgeschwindigkeit von 30 km/h durch Betätigung des Lenkstockhebels die aktuelle Geschwindigkeit als Wunschgeschwindigkeit übernehmen und diese dann im Bereich von 30 bis 250 km/h beliebig verändern. Nach einer Deaktivierung des Systems durch den Fahrer kann über die Resume-Taste das System mit der zuletzt gespeicherten Soll-Geschwindigkeit erneut aktiviert werden.

Der Fahrer kann zwischen vier Abstandsstufen zum vorausfahrenden Fahrzeug wählen, die Zeitintervallen zwischen 1 und 3 Sekunden bis zum Vordermann entsprechen.

Die Wunschgeschwindigkeit wird im Informationsdisplay des Kombiinstruments (siehe Abbildung 2.2) in Form eines Scheibenzeigers dargestellt. Ein leuchtendes Fahrzeugsymbol signalisiert, dass ein vorausfahrendes Objekt erkannt ist. Die Anzahl der beleuchteten Abstandsbalken zeigt die eingestellte Abstandsstufe an.

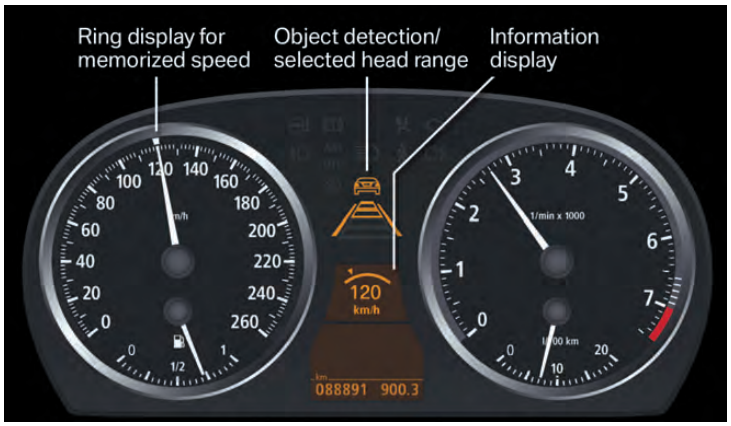


Abbildung 2.2.: Kombiinstriment für ACC [STT+05]

2.2. Informelle Strukturierung der Funktionalität

Man beachte, in Abbildungen 2.3 und 2.4 handelt es sich um intuitive Skizzen. Die Semantik der Diagramme wird in den nachfolgenden Kapiteln eingeführt.

Teilsysteme Das ACC-System ist in drei Teilsysteme unterteilt (vgl. Abbildung 2.3), die `Hebelkontrolle` zur Verwaltung von Eingaben, die durch den Lenkstockhebel eingegeben werden, die `Display-Steuerung` zur Darstellung der relevanten Daten im Informationsdisplay des Kombiinstruments und die `ACC-Steuerung` zur Berechnung der Steuerbefehle an die Motor- und Bremssteuerung. Sowohl zwischen einzelnen Teilsystemen als auch zwischen dem System und seiner Umgebung findet ein gerichteter Nachrichtenfluss statt. Mit der Systemumgebung sind die Benutzer des Systems (im Beispiel der Fahrer) sowie die umliegenden Systeme, mit denen das betrachtete System interagiert (Motor- und Bremssteuerung, Radarsensor und Informationsdisplay), gemeint.

Teilfunktionen Im Gegensatz zum primitiven Verhalten der beiden Teilsysteme `Hebelkontrolle` und `Display-Steuerung`, die Daten lediglich zwischenspeichern und weiterleiten, bietet die `ACC-Steuerung` eine Familie von komplexen und voneinander abhängigen Nutzerfunktionen an. Nutzerfunktionen sind an der Systemgrenze durch Benutzer

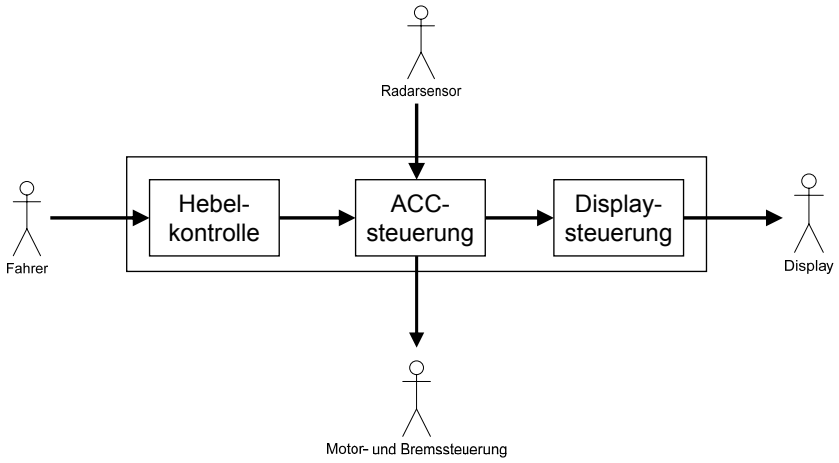


Abbildung 2.3.: Unterteilung des ACC-Systems in Teilsysteme

wahrgenommene Systemverhalten. Um das Zusammenspiel der einzelnen Funktionen besser zu verstehen, bietet es sich an, sie hierarchisch zu strukturieren und Wechselwirkungen zwischen ihnen explizit zu markieren (vgl. Abbildung 2.4).

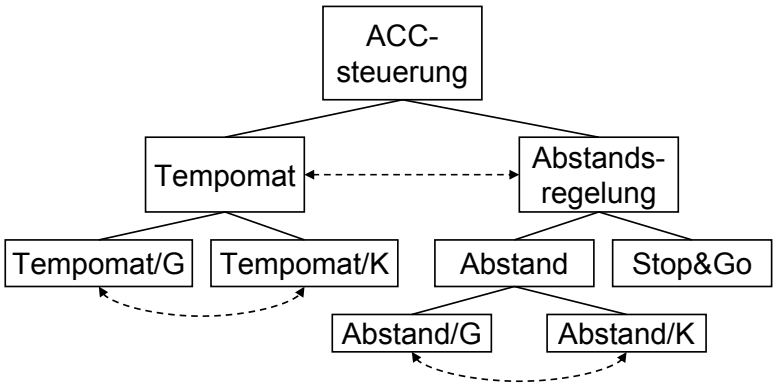


Abbildung 2.4.: Unterteilung der ACC-Funktion in Teilfunktionen

Die ACC-Steuerung ist hierarchisch in zwei Teilfunktionen unterteilt: `Tempomat` und `Abstandsregelung`. Die Funktion Tempomat ist wiederum unterteilt in Funktionen `Tempomat/G` (das Verhalten des Tempomats auf einer Gerade) und `Tempomat/K` (Tempomat in einer Kurve). Die Abstandsregelung besteht aus zwei Teilfunktionen `Abstand` (Regelung bei einer Geschwindigkeit über 30 km/h) und `Stop&Go` (Regelung unter 30 km/h). Die Funktion `Abstand` ist analog zur Funktion `Tempomat` in Teilfunktionen `Abstand/G` und `Abstand/K` unterteilt. Gestrichelte Pfeile weisen auf Wechselwirkungen zwischen Funktionen hin.

KAPITEL 3

Dienstbasierte Software-Entwicklung

Typischerweise bieten heutige eingebettete Systeme nicht mehr nur eine spezifische Funktion an, sondern stellen eine umfangreiche Familie von einander beeinflussenden Nutzerfunktionen bereit. Das vorliegende Kapitel führt eine funktionsorientierte Modellierungstechnik zur Spezifikation multifunktionaler Systeme ein. Einzelne Teilfunktionen der Gesamtfunktionalität werden in Form separater Interaktionsmuster aus unterschiedlichen Benutzerperspektiven spezifiziert. Anschließend werden diese Muster zu einer Gesamtspezifikation integriert. Der Fokus der Betrachtung dieser Technik liegt auf Nutzerfunktionen und Wechselwirkungen zwischen diesen.

In diesem Kapitel werden grundlegende Ideen des Ansatzes sowie seine wichtigsten Merkmale anschaulich jedoch informell erklärt. Die der Spezifikationstechnik zugrunde liegende operationelle Semantik wird in Kapitel 4 eingeführt.

Inhalt

Die Entwicklung komplexer reaktiver Softwaresysteme erfordert ein strukturiertes, modulares Vorgehen und angemessene Techniken zur präzisen Beschreibung der Systemfunktionalität [BR07]. Die Anzahl der von einem System angebotenen Nutzerfunktionen und insbesondere ihre komplexen Wechselwirkungen erschweren die Modellierung einer konsistenten und nachvollziehbaren Spezifikation des Systems. Traditionelle Modellierungstechniken des Designs (wie z.B. Komponenten), die eher für die Beschreibung der Implementierung konzipiert sind, sind für die Modellierung von Nutzerfunktionen nicht geeignet [BKPS07, S. 370]. Die Spezifikation multifunktionaler Systeme erfordert eine Technik, die sich der Modellierung der Nutzerfunktionen und nicht deren Implementierung widmet. Diese Technik muss sowohl die Spezifikation einzelner Funktionen als auch ihre Kombination zum Gesamtverhalten unterstützen.

Während die komponentenbasierte Entwicklung in den letzten dreißig Jahren umfassend erforscht wurde (z.B. in [Hoa85, Mil89, BS01]), ist die funktionsorientierte Spezifikation [Bro07b] immer noch eine der großen Herausforderungen bei der Entwicklung von Software-Systemen. Diese Art der Spezifikation ist der Schwerpunkt der vorliegenden Arbeit.

Eine funktionsorientierte Spezifikation definiert, wie das System sich zu verhalten hat (Problemdomäne), ohne dabei eine mögliche Implementierung (Lösungsdomäne) anzugeben. Sie gibt dem Entwickler die Möglichkeit, optimale Lösungen zu erarbeiten, ohne ihn durch zu restriktive Anforderungen in seiner Lösungskompetenz einzuschränken. Eine funktionsorientierte Spezifikation umfasst eine Hierarchie von Anwendungsfällen und eine Menge von Wechselwirkungen zwischen diesen. In der Domäne reaktiver Systeme ist ein Anwendungsfall ein *Interaktionsmuster* zwischen dem System und seiner Umgebung. Außerdem kann eine Spezifikation mehrere Systemmodi definieren, in denen Nutzerfunktionen aktiviert oder deaktiviert werden können.

Die in der vorliegenden Arbeit eingeführte Technik unterstützt die Spezifikation eines multifunktionalen Systems aus *unterschiedlichen Benutzerperspektiven*. Verschiedene *Stakeholder* (deutsch Interessenvertreter) beschreiben ihre individuellen Anforderungen in separaten Modellen, die anschließend unter Berücksichtigung ihrer Wechselwirkungen zu einer Gesamtspezifikation integriert werden.

Das System wird aus der *Black-Box-Sicht* beschrieben. Dabei folgt der Ansatz der Definition der Black-Box-Anforderungsspezifikation von Leveson [Lev95, Kapitel 15.4]. Nach dieser besteht ein softwaregesteuertes System aus vier Bestandteilen: dem mechanischen Prozess, Aktuatoren, Sensoren und dem zu spezifizierenden eingebetteten Software-System (vgl. Abbildung 3.1). Das Software-System erhält Informationen über den aktuellen Zustand des mechanischen Prozesses durch seine Sensoren und steuert den Prozess durch seine Aktuatoren. Leveson schlägt eine strikte Black-Box-Spezifikation vor. Jede an der Schnittstelle des Software-Modells sichtbare Ausgabe muss von einer an der Schnittstelle sichtbaren Eingabe verursacht werden.

> „A black-box statement of behavior allows statements and observations to be made only in terms of outputs and the externally observable conditions

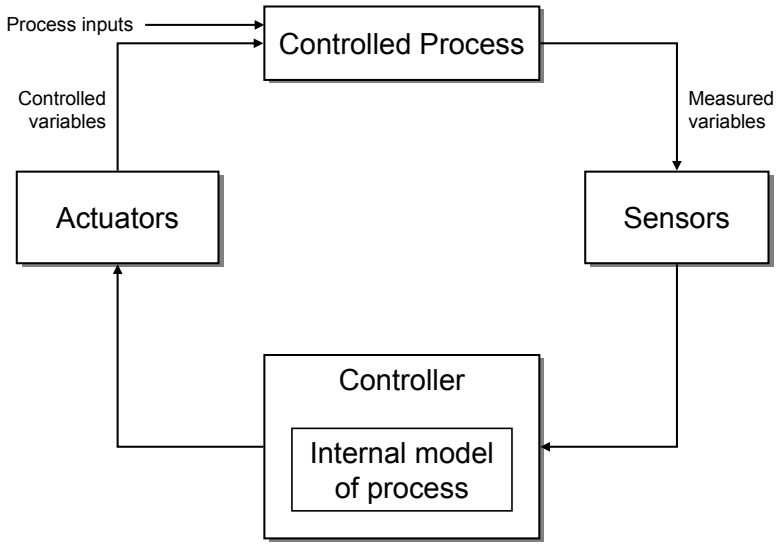


Abbildung 3.1.: Black-Box-Anforderungsspezifikation [Lev95, S. 365]

> or events that stimulate or trigger them. In terms of the state machine, this restriction means that both the states and the events on the transitions must be externally observable.“ [Lev95, S. 366].

Der in diesem Kapitel eingeführten Spezifikationstechnik liegt eine *operationelle Semantik* [Plo81] (vgl. Abschnitt 4.3) zugrunde, die der Simulation und Verifikation funktionaler Anforderungen als Basis dient.

Im folgenden Abschnitt werden Anforderungen an eine Modellierungstechnik zur Spezifikation funktionaler Anforderungen aufgezeigt. In Abschnitt 3.2 werden zwei orthogonale Arten der Systembeschreibung – die strukturelle Komposition und die funktionale Kombination – eingeführt. Abschnitt 3.3 ist der Schwerpunkt dieses Kapitels und führt eine dienstbasierte Spezifikationstechnik ein. In Anschnitt 3.4 wird der Nutzen dieser Spezifikation im Requirements Engineering dargestellt. In Abschnitt 3.5 wird der Ablauf der dienstbasierten Anforderungsanalyse kurz erläutert.

3.1. Anforderungen an die Spezifikationstechnik

Im Folgenden werden die wichtigsten Anforderungen an die zu erarbeitende Modellierungstechnik zur Beschreibung funktionaler Anforderungen aufgezählt, die größtenteils aus [Lev95, BCK98, JBR99, BKPS07] stammen.

Explizite Darstellung von Nutzerfunktionen Die Funktionalität eines multifunktionalen Systems besteht aus mehreren Nutzerfunktionen, die an der Systemgrenze durch einen *Benutzer* wahrgenommen werden. Eine einzelne Funktion stellt einen Ausschnitt

aus dem Gesamtverhalten dar. Die primäre Aufgabe der Anforderungsanalyse ist es, diese Nutzerfunktionen zu erheben und zu spezifizieren, ohne dabei eine Vorentscheidung über die Implementierung zu treffen.

Folglich muss die Spezifikationstechnik eine explizite und modulare Darstellung von Nutzerfunktionen unterstützen. Diese Darstellung muss von jeglicher Form des Entwurfs abstrahieren.

Beispielsweise bietet das ACC-System seiner Umgebung Nutzerfunktionen wie *Tempomat*, *Abstandsregelung* oder *Stop&Go* an (vgl. Abbildung 2.4 auf Seite 16).

Kombinationsoperator für Nutzerfunktionen Um die Komplexität der multifunktionalen Systeme bewältigen zu können, ist es einerseits wichtig, die Systemfunktionalität hierarchisch strukturieren zu können. Andererseits ist es wichtig, diese hierarchische Struktur für den Anforderungsanalytiker möglichst einfach und intuitiv zu gestalten. Er soll sich nur mit dem Block-Box-Verhalten befassen – interne Strukturen (wie z.B. Informationsflüsse zwischen Nutzerfunktionen oder interne Funktionen) dürfen für die Kombination von Nutzerfunktionen nicht benutzt werden. Die einzige Art der Interaktion in der Anforderungsspezifikation muss zwischen dem System und seiner Umgebung erfolgen [JBR99, S. 137]. Weiterhin soll der Analytiker die Systemfunktionalität in modularen Schritten spezifizieren können, ohne sich über die Wechselwirkungen zwischen Nutzerfunktionen Gedanken machen zu müssen.

Aus diesen Gründen muss die Spezifikationstechnik es ermöglichen, das Verhalten des Gesamtsystems aus der Kombination der Beschreibungen von Nutzerfunktionen zu gewinnen, ohne dass Informationen über die inneren Strukturen des Systems vorliegen.

Der geforderte Operator muss die beiden eigenständigen Nutzerfunktionen *Tempomat* und *Abstandsregelung* so kombinieren, dass die kombinierte Funktion *ACC-Steuerung* das in Kapitel 2.1 beschriebene Verhalten des ACC-Systems aufweist.

Explizite Darstellung interfunktionaler Wechselwirkungen Eine der schwierigsten und gleichzeitig wichtigsten Aufgaben bei der Spezifikation eingebetteter Systeme ist die Identifizierung und explizite Modellierung von Wechselwirkungen zwischen Nutzerfunktionen [Zav03]. Die Komplexität multifunktionaler Systeme verursacht das Problem unerwünschter Wechselwirkungen zwischen Funktionen. Wechselwirkungen beschreiben, wie die eigenständig definierten Funktionen zusammenspielen bzw. sich gegenseitig beeinflussen, um das gewünschte Gesamtverhalten zu erbringen. Für eine inkrementelle Erhebung und eine handhabbare Analyse von Anforderungen ist es erforderlich, Funktionen und ihre Wechselwirkungen separat von einander modellieren zu können.

Inkrementelle Erhebung Separate Modelle von Wechselwirkungen erlauben dem Anforderungsanalytiker, die Erhebung von Anforderungen in zwei Phasen durchzuführen. Im ersten Schritt kann er sich auf die Nutzerfunktionen an sich und anschließend auf ihre Wechselwirkungen konzentrieren. Ohne diese Trennung müssten alle Funktionen, zwischen denen eine Wechselwirkung existiert, in einem Schritt

erfasst werden.

Handhabbare Analyse Explizite Modelle der Wechselwirkungen tragen zur Vermeidung monolithischer Spezifikationen bei. Dadurch kann die Analyse isolierter Funktionen von deren Integration zur Gesamtspezifikation getrennt werden.

Im ACC-System existiert z.B. eine Wechselwirkung zwischen den beiden Funktionen *Tempomat* und *Abstandsregelung*, die sicherstellt, dass sowohl die Wunschgeschwindigkeit als auch der minimale Abstand ständig eingehalten werden. Eine explizite Darstellung dieser Wechselwirkung trägt zum besseren Verständnis des Gesamtverhaltens bei.

Systemmodi Die Unterteilung des Systemverhaltens in Modi kann die Erfassung von Anforderungen für reaktive Systeme in vielen Fällen vereinfachen. *Systemmodi* (engl. modes of operation [JLHM91]) definieren Mengen von Systemzuständen, in denen das System sich äquivalent verhält. Sie stellen ein wichtiges Mittel der Fein- und Grobdekomposition des Verhaltens eingebetteter Systeme dar [BBR+07].

Die ACC-Steuerung kann sich in den Modi „Ausgeschaltet“, „Eingeschaltet und Deaktiviert“, „Aktiviert und Objekt erkannt“, etc. befinden.

Die zu erarbeitende Spezifikationstechnik muss explizite Modellierung von Modi unterstützen. Dabei müssen (insbesondere bei sehr umfangreichen Spezifikationen) die folgenden zwei Entwicklungsszenarien berücksichtigt werden:

- *Verschiedene Funktionen in einem Modus.* Das Systemverhalten wird zuerst in eine Hierarchie von Systemmodi unterteilt. In jedem Modus wird eine Hierarchie von Funktionen spezifiziert.

  Beispielsweise wird im Modus „Aktiviert und Objekt erkannt“ der ACC-Steuerung eine Hierarchie der Teilfunktionen definiert.

- *Verschiedene Modi einer Funktion.* Zuerst wird eine Hierarchie von Funktionen aufgestellt, anschließend wird das Verhalten jeder Funktion in Modi unterteilt.

  Die ACC-Steuerung wird z.B. in die Funktionshierarchie aus Abbildung 2.4 unterteilt. Jede dieser Funktionen wird in den Modi „Ausgeschaltet“, „Eingeschaltet und Deaktiviert“, etc. separat definiert.

Die Entscheidung, ob eine Funktion zuerst in Modi oder in Teilfunktionen unterteilt wird, soll dem Anforderungsanalytiker überlassen werden. Das heißt, die Spezifikationstechnik muss beliebige Kombinationen der beiden Vorgehensweisen unterstützen.

***Bemerkung* 3.1** (Interne und externe Modi)**.** Es gibt zwei Arten, über Modi das Systemverhalten zu definieren. Die einen – auch interne Modi genannt – zeigen den Betriebszustand des Systems an (z.B. die von Jaffe et al. [JLHM91]). Das System geht – abhängig von dessen Eingaben – durch unterschiedliche Betriebszustände. Die sequentielle Kombination aller Betriebszustände ergibt das Gesamtverhalten des Systems.

Die anderen – auch externe Modi [Bro07a] genannt – stellen sicher, dass Betriebszustände, die außerhalb einer Nutzerfunktion liegen und die Funktion beeinflussen, berück-

sichtigt werden. Diese Modi sind neben den eigentlichen Funktionseingaben gewissermaßen Vorgaben an die Funktion. Zu berücksichtigen ist dabei, dass die Funktion selber ggf. für andere Funktionen externe Modi generiert.

In der vorliegenden Arbeit werden nur interne Modi betrachtet. □

3.2. Zwei Arten der Strukturierung

Angelehnt an Leveson [Lev00] unterstützt die in der vorliegenden Arbeit eingeführte Spezifikationstechnik zwei Arten der Strukturierung der Systemfunktionalität, die strukturelle (De-)Komposition und die funktionale (De-)Kombination. Die Grundidee des Ansatzes von Leveson ist, dass das betrachtete System immer aus der Black-Box-Sicht beschrieben wird, die Systemgrenze jedoch beliebig verschiebbar ist (vgl. Abbildung 3.2). Zuerst werden funktionale Anforderungen an das Gesamtsystem gestellt (vgl. 3. Zeile,

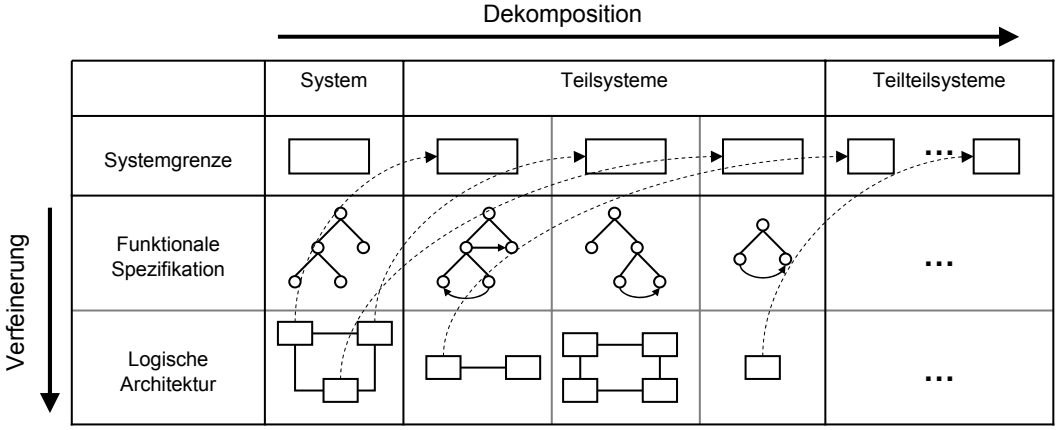


Abbildung 3.2.: Zwei Arten der Strukturierung

2. Spalte in der Matrix). Dabei werden nur diejenigen Anforderungen behandelt, welche die interne Struktur des Systems außen vor lassen. Nach der Spezifikation der Anforderungen an das Gesamtsystem erfolgt ein anfänglicher Entwurf der inneren Struktur des Systems. Die Struktur besteht aus einem Netz von interagierenden Teilsystemen, genannt *Logische Architektur*, das die Anforderungen realisiert (vgl. 4. Zeile, 2. Spalte). Der Übergang von der funktionalen Spezifikation zur logischen Architektur wird in Kapitel 6 erläutert. Wichtig ist an dieser Stelle die Verfeinerungsrelation zwischen den beiden Modellen. Durch den Entwurf einer logischen Architektur für das Gesamtsystem entstehen Anforderungen an Teilsysteme. Die Grenze des betrachteten Systems wird verschoben – spezifiziert wird nicht das Verhalten des Gesamtsystems, sondern die im Netzwerk verwendeten Teilsysteme (vgl. 3. Zeile, 3.-5. Spalten). Die anderen Teilsysteme befinden sich nun in der Umgebung des betrachteten Teilsystems. Genau wie vorher das Gesamtsystem wird ein Teilsystem aus der Black-Box-Sicht beschrieben, d.h. das an seiner Grenze wahrgenommene Verhalten wird spezifiziert. Die dadurch entstandenen

Spezifikationen werden jeweils durch ein weiteres Netzwerk von Teilsystemen realisiert. Diese Schritte werden solange wiederholt bis die einzelnen Teilsysteme nicht mehr dekomponiert werden müssen.

Bei der Zerlegung eines Systems in ein Netzwerk kollaborierender Teilsysteme (oder Komponenten) handelt es sich um die *strukturelle Dekomposition*. Diese Art der Strukturierung ist in den meisten Modellierungstechniken vorgesehen (z.B. [Hoa85, Mil89, BS01]). Die Zerlegung des Systemverhaltens in einzelne Interaktionsmuster wird in den meisten Ansätzen nicht behandelt. Ohne diese *funktionale Dekombination* ist es jedoch nicht möglich, aus einzelnen funktionalen Anforderungen die Gesamtspezifikation zu konstruieren. Die Dekombination bildet die Grundlage für die hierarchische Strukturierung von Nutzerfunktionen und liegt im Fokus der vorliegenden Arbeit.

Im Folgenden werden die beiden orthogonalen Arten der Strukturierung eingeführt. Anschließend wird erläutert, warum die etablierte strukturelle Komposition für die Anforderungsspezifikation nicht ausreicht. Die Vorteile der funktionalen Kombination werden in Abschnitt 3.4 erläutert.

Beispiel 3.1 (ACC-Teilsysteme)**.** Für die nachfolgende Erläuterung der beiden Dekompositionsarten wird das Diagramm aus Abbildung 2.3 auf Seite 16 verfeinert, indem der abstrakte Datenfluss zwischen den ACC-Teilsystemen durch konkrete Kommunikationskanäle ersetzt wird (vgl. Abbildung 3.3). Das Teilsystem `Hebelkontrolle` wandelt unterschiedliche Betätigungen des Lenkstockhebels in Geschwindigkeits- und Abstandswerte um. Dazu hat das Teilsystem drei Eingabekanäle aus seiner physikalischen Umgebung: `speed` zur Kontrolle der Hebelbetätigung, `distance` zur Kontrolle des kleinen Stufenschalters und `resume` zur Kontrolle der Resume-Taste (vgl. Abbildung 2.1 auf Seite 14). Diese Eingaben werden in eine Anfahrbestätigung sowie Wunschgeschwindigkeits- und Wunschabstandswerte umgewandelt und durch Ausgabekanäle `rSpeed`, `rDistance`, `rResume` an das Teilsystem `ACC-Steuerung` weitergeleitet. Zusätzlich zu diesen Kanälen hat die `ACC-Steuerung` drei weitere Eingabekanäle aus seiner physikalischen Umgebung: `cSpeed`, `cDistance` und `turn` zum Einlesen der momentanen Geschwindigkeit, des Abstands zum Vordermann sowie des Lenkradwinkels. Die `ACC-Steuerung` berechnet anhand dieser Eingabedaten die Steuerbefehle an die Motor- und Bremssteuerung und gibt sie durch die Ausgabekanäle `mInstr` und `bInstr` aus. Die Wunschgeschwindigkeit, der Wunschabstand sowie die Information, ob ein vorausfahrendes Objekt erkannt ist, werden an die `Displaysteuerung` entsprechend durch Kanäle `speed`, `distance` und `object` weitergeleitet. In diesem Teilsystem werden die Werte der Mensch-Maschine-Schnittstelle generiert und durch den Kanal `hmi` an das Informationsdisplay (siehe Abbildung 2.2) weitergeleitet. □

3.2.1. Strukturelle Komposition

Bei der strukturellen (De-)Komposition des Systemverhaltens wird die Funktionalität eines Systems als ein Netzwerk kollaborierender Komponenten modelliert. Eine *Kompo-*

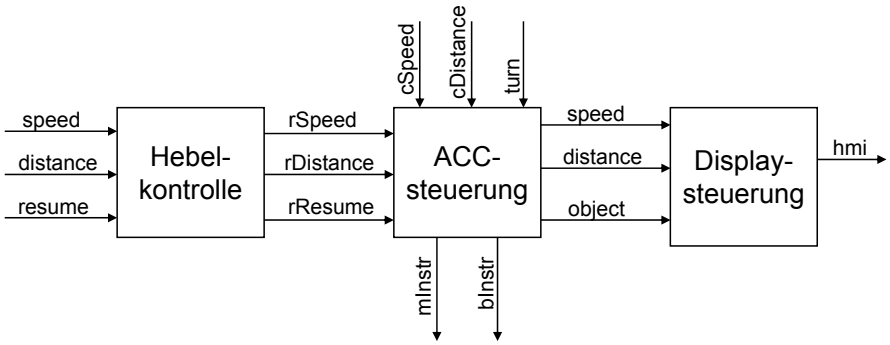


Abbildung 3.3.: Architektonische Strukturierung des ACC in Teilsysteme

nente hat eine syntaktische Schnittstelle und eine Verhaltensspezifikation. Die *syntaktische Schnittstelle* besteht aus einer Menge von Ein- und Ausgabekanälen (vgl. Abbildung 3.4). Die *Verhaltensspezifikation* einer Komponente ist eine totale Funktion, die Eingabewerte auf Ausgabewerte abbildet. Eine *totale* Funktion definiert Ausgaben für alle syntaktisch korrekten Eingaben. Ein *Netzwerk* von Komponenten hat externe Kanäle zu seiner Umgebung sowie interne Kanäle, durch die der Datenfluss zwischen Komponenten erfolgt. Weitere Details über die strukturelle Komposition finden sich in [BS01].

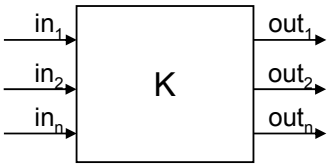


Abbildung 3.4.: Syntaktische Schnittstelle einer Komponente

Beispiel 3.2 (Strukturelle Dekomposition der ACC-Steuerung)**.** Eine mögliche Dekomposition des Teilsystems `ACC-Steuerung` ist in Abbildung 3.5 skizziert. Das Netzwerk besteht aus vier Komponenten, hat dieselben Ein- und Ausgabekanäle zu seiner Umgebung wie das Teilsystem `ACC-Steuerung` aus Abbildung 3.3 und interne Kanäle zwischen den Komponenten. Die Komponente `Splitter` vervielfältigt Eingaben aus der Umgebung der ACC-Steuerung und leitet die Werte an nachfolgende Komponenten weiter. Die Komponenten `Tempomat` und `Abstandsregelung` sind entsprechend zuständig für die Steuerung der Geschwindigkeit und des Abstandes. Die Komponente `Merge` entscheidet anhand der Werte an den Eingabekanälen `cSpeed` und `cDistance` wessen Steuerbefehle weitergeleitet werden. Die Kollaboration der vier Komponenten ergibt die Gesamtfunktionalität des Teilsystems `ACC-Steuerung`. □

Alle wichtigen Merkmale und Vorteile der strukturellen Komposition sind z.B. in [BCK98, BS01] ausführlich erläutert. Sie liegen jedoch nicht im Fokus der vorliegenden Arbeit.

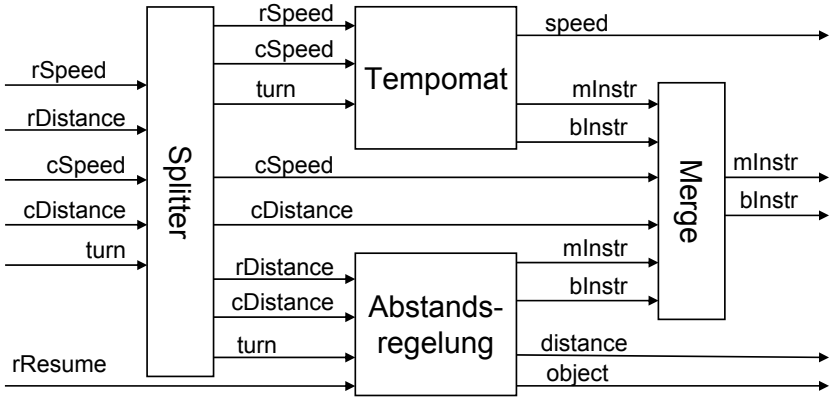
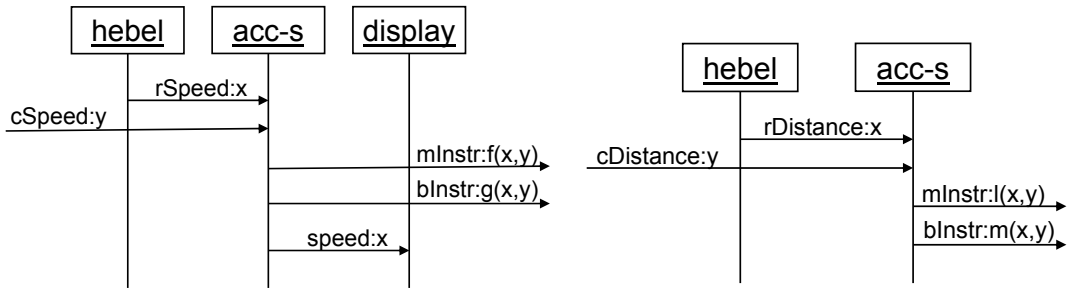


Abbildung 3.5.: Strukturelle Dekomposition der ACC-Steuerung

3.2.2. Funktionale Kombination

Als Ergänzung zur strukturellen Dekomposition wird in der vorliegenden Arbeit ein neuer Operator zur Kombination von Nutzerfunktionen eingeführt, welcher von der Struktur logischer Komponenten abstrahiert. Dabei wird eine Nutzerfunktion durch ein Interaktionsmuster dargestellt. Die Teilfunktionen werden zunächst eigenständig spezifiziert. Danach werden sie durch den definierten Operator zu größeren Funktionen und schließlich zur Gesamtfunktion kombiniert. Das Zusammenspiel der Teilfunktionen, insbesondere unter Berücksichtigung ihrer Wechselwirkungen, ergibt die Gesamtfunktionalität des Systems.

Beispiel 3.3 (Familie von Interaktionsmustern)**.** Die Funktionalität der ACC-Steuerung wird in eine Hierarchie von Teilfunktionen strukturiert, die in etwa der Hierarchie aus Abbildung 2.4 auf Seite 16 entspricht. Im Gegensatz zum Netzwerk von Komponenten aus Abbildung 3.5 wird hier eine Menge von Interaktionsmustern definiert, die an der Systemgrenze beobachtbar ist. Im Falle der ACC-Steuerung ist die Systemgrenze die Schnittstellen zu den Teilsystemen `Hebelkontrolle` und `Displaysteuerung`, zu den Sensoren der aktuellen Geschwindigkeit und des Abstandes sowie zu den Aktuatoren der Motor- und Bremssteuerung (vgl. Abbildungen 2.3 auf Seite 16 und 3.3 auf Seite 24). In Abbildung 3.6 sind zwei beispielhafte Interaktionsmuster der ACC-Steuerung mit ihrer Umgebung dargestellt. In der Abbildung wurden UML-Sequenzdiagramme verwendet. An dieser Stelle sind die Diagramme jedoch intuitiv und nicht mit einer der UML-Semantiken zu interpretieren. Das Diagramm aus Abbildung 3.6(a) beschreibt das folgende Interaktionsmuster. Die ACC-Steuerung (`acc-s`) erhält durch den Kanal `rSpeed` die Wunschgeschwindigkeit x und durch den Kanal `cSpeed` die momentane Geschwindigkeit y. Darauf verschickt das Teilsystem die beiden Steuerbefehle $f(x, y)$ durch den Kanal `mInstr` und $g(x, y)$ durch den Kanal `bInstr` sowie den Wert der Wunschgeschwindigkeit durch den Kanal `speed` an seine Umgebung. Abbildung 3.6(b) beschreibt ein ähnliches Interaktionsmuster, an dem aber nur zwei Teilsysteme beteiligt sind. Die Kombination der beiden Interaktionsmuster (zusammen mit den restlichen, an dieser



(a) Interaktionsmuster `Tempomat` (b) Interaktionsmuster `Abstandskontrolle`

Abbildung 3.6.: Familie von Interaktionsmustern

Stelle weggelassenen Mustern) ergibt die Gesamtfunktionalität der ACC-Steuerung. In Abbildung 3.7 wird die Kombination der beiden Interaktionsmuster durch ein kombiniertes Fragment mit einem fiktiven Interaktionsoperanden `merge` bezeichnet (der synchrone Nebenläufigkeit bezeichnet). □

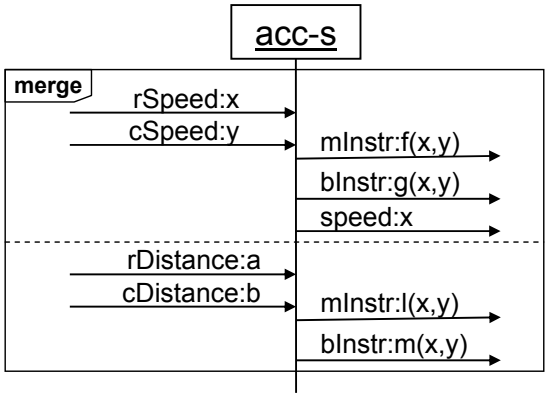


Abbildung 3.7.: Kombination von Interaktionsmustern

Die funktionale Kombination ist der Schwerpunkt der vorliegenden Arbeit und wird in Abschnitt 3.3 ausführlich erläutert. Ihre Vorteile gegenüber der strukturellen Komposition werden in Abschnitt 3.4 dargestellt.

***Bemerkung* 3.2** (Zusammenspiel der beiden Arten der Strukturierung)**.** Es ist wichtig hervorzuheben, dass sich die beiden Arten der Strukturierung nicht gegenseitig ersetzen sondern ergänzen. So wie in der Matrix aus Abbildung 3.2 dargestellt, soll eine gute Spezifikationstechnik sowohl die strukturelle Komposition als auch die funktionale Kombination beinhalten. □

3.2.3. Mängel der strukturellen Komposition

Obwohl die strukturelle Komposition für die Beschreibung der logischen Architektur gut geeignet ist, weist sie in Bezug auf das Requirements Engineering mehrere Schwächen auf. Im Requirements Engineering liegt der Fokus der Betrachtung auf Nutzerfunktionen, die das System erbringen muss, für deren Beschreibung ein komponentenbasiertes Modell nicht geeignet ist. Im Folgenden werden die wichtigsten Merkmale der strukturellen Komposition erläutert, die den Einsatz komponentenbasierter Modelle für die Spezifikation funktionaler Anforderungen erschweren. Dabei beziehen wir uns auf die Anforderungen aus Abschnitt 3.1.

Explizite Darstellung von Nutzerfunktionen Eine Funktion, wie sie aus der Black-Box-Sicht wahrgenommen wird, ist selten durch genau eine Komponente realisiert. Die meisten Funktionen eines multifunktionalen Systems werden durch eine Interaktion mehrerer Komponenten erbracht. Es handelt sich um so genannte *cross-cutting* (oder komponentenübergreifende) Funktionen. Für die Anforderungsanalyse ist es jedoch wichtig, über einzelne Funktionen und nicht über Interaktionen von Komponenten reden zu können.

Beispiel 3.4. Im Netzwerk von Komponenten aus Abbildung 3.5 sind die beiden Funktionen *Geschwindigkeits-* und *Abstandsregelung* lediglich implizit erfasst worden. Es ist nicht möglich, eine der beiden Funktionen separat zu analysieren. Um z.B. die Funktion *Geschwindigkeitsregelung* analysieren zu können, müssen alle vier Komponenten der `ACC-Steuerung` betrachtet werden. Vor allem kann die Komponente `Abstandsregelung` nicht weggelassen werden, obwohl ihr Verhalten in der Spezifikation der Geschwindigkeitsregelung eigentlich keine Rolle spielt. Sie ist jedoch ein fester Bestandteil des Netzwerkes, ohne den das Verhalten der Geschwindigkeitsregelung undefiniert wäre. Wenn die Komponente `Abstandsregelung` zusammen mit den entsprechenden Kanälen aus dem Modell entfernt wird, ist das Verhalten der Komponente `Merge` undefiniert – die Verhaltensspezifikation benötigt Werte an den beiden Kanälen `mInstr` und `bInstr`. □

Kombinationsoperator für Nutzerfunktionen Nutzerfunktionen lassen sich durch die strukturelle Komposition schwer kombinieren. Wegen des fehlenden Kombinationsoperators ist die inkrementelle Erhebung von Anforderungen nicht möglich. In einer komponentenbasierten Spezifikation wird deswegen angenommen, dass das Gesamtverhalten des Systems zum Zeitpunkt der Modellierung bereits vollständig bekannt ist. Der Ersteller des Netzwerks von Komponenten muss sich den Überblick über alle Funktionen und vor allem über alle ihre Wechselwirkungen verschaffen. Kommt eine weitere komponentenübergreifende Funktion zu einem bestehenden Netzwerk hinzu, muss häufig das ganze Modell umstrukturiert werden. Für die Kombination von Funktionen wird immer eine Designentscheidung benötigt.

Beispiel 3.5. Im laufenden Beispiel ist es nicht möglich, die Geschwindigkeits- und Abstandsregelung separat voneinander zu spezifizieren und anschließend mittels eines Operators zu kombinieren. Für ihre Kombination werden zwei weitere Komponenten

erstellt. Kommt eine weitere Funktion hinzu, z.B. *Stop&Go*, müssen die Komponenten `Splitter` und `Merge` entsprechend angepasst werden. □

Integration von Interaktionsmustern Sind verschiedene Interaktionsmuster ein und derselben Nutzerfunktion jeweils mit einer Komponente realisiert, können sie nur durch explizites Re-Design komponiert werden. In diesem Fall überlappen die Schnittstellen der Komponenten zwangsweise, d.h. sie haben gemeinsame Ein- und Ausgabekanäle. Allerdings sind in den meisten komponentenbasierten Ansätzen [LT89, HP98, Hen00, dAH01, BS01, Sch05] keine gemeinsamen Ausgabeports zugelassen. Dadurch ist es syntaktisch nicht möglich, überlappende Komponenten ohne explizites Re-Design zu komponieren. In den Ansätzen mit gemeinsamen Ausgabeports ist synchrone Nebenläufigkeit nicht definiert, die für die Integration von Interaktionsmustern notwendig ist (vgl. [Bro05a, Kapitel 2.7]).

Explizite Darstellung interfunktionaler Wechselwirkungen Eine Wechselwirkung zwischen zwei Nutzerfunktionen kann in einem komponentenbasierten Modell nur durch interne Kanäle realisiert werden. Diese Art der Realisierung ergibt folgende Probleme:

Modularitätsprinzip Die Komposition zweier Komponenten ergibt nicht das Gesamtverhalten der beiden. Für die Berücksichtigung von Wechselwirkungen müssen die ursprünglichen Spezifikationen modifiziert werden. Die syntaktischen Schnittstellen werden um interne Kanäle, die Verhaltensspezifikationen um das Senden/Empfangen von Interaktionsnachrichten erweitert. Nach dieser Modifizierung können die Komponenten nicht mehr unabhängig voneinander analysiert werden.

Im laufenden Beispiel müssen die Komponenten `Splitter` und `Merge` jedes Mal modifiziert werden, wenn eine weitere Funktion dem Modell hinzugefügt wird.

Black-Box-Sicht Sowohl die Nutzerfunktionen als auch ihre Wechselwirkungen sind im Modell nur implizit enthalten. Aus der Black-Box-Sicht sind für den Benutzer die ursprünglichen, separaten Funktionen nicht mehr erkennbar. Solch eine monolithische Spezifikation lässt sich aber schwer analysieren.

Im Netzwerk in Abbildung 3.5 sind die Funktionen *Tempomat* und *Abstandsregelung* nur implizit vorhanden – sie sind über mehrere Komponenten verteilt.

Komponentenübergreifende Modi Modi von Nutzerfunktionen können z.B. durch Modecharts [JM86] oder Statecharts [Har88] modelliert werden. Jeder Komponente ist dabei ein Mode-Transition-Diagramm zugeordnet. Diese Diagramme bestehen aus Modi und Transitionen zwischen Modi, wobei jedem Modus ein Verhalten in Form eines hierarchischen Automaten zugeordnet ist. Wenn das Systemverhalten zuerst in eine Hierarchie von Modi unterteilt wird, ist es danach nicht mehr möglich, in einem Modus eine Hierarchie von Komponenten zu spezifizieren. Modecharts, Statecharts und andere etablierte

automatenbasierte Ansätze lassen in einem Modus kein Netzwerk von Komponenten zu[1]. Dies bedeutet, dass in einem Systemmodus das Gesamtverhalten nicht mehr in Komponenten unterteilt werden kann – das Verhalten muss durch einen einzigen Automaten spezifiziert werden. In dieser Form sind komponentenübergreifende Modi bei umfangreichen multifunktionalen Systemen nicht anwendbar.

Beispiel 3.6. Das Verhalten der ACC-Steuerung im Modus „Aktiviert und Objekt erkannt“ kann durch Modecharts oder Statecharts nicht mehr in eine Hierarchie von Teilfunktionen `Tempomat` und `Abstandsregelung` unterteilt werden. □

3.3. Dienstbasierte Spezifikation

Dieser Abschnitt führt eine dienstbasierte Spezifikationstechnik zur Modellierung der Systemfunktionalität ein. Diese Technik beschreibt das Systemverhalten, wie es an der Systemgrenze durch Benutzer wahrgenommen wird. Dabei kann ein Benutzer ein Mensch, aber auch ein anderes System sein.

Bei der dienstbasierten Spezifikation wird für das zu spezifizierende Gesamtsystem die Systemgrenze festgelegt. Sie umfasst die Definition der Schnittstelle zur externen Umgebung (in eingebetteten Systemen überwiegend Sensoren und Aktoren) sowie der Schnittstelle zu umliegenden Systemen, mit denen das betrachtete System interagiert. Das Verhalten des Gesamtsystems wird dann aus der Black-Box-Sicht spezifiziert, d.h. es wird der mögliche Nachrichtenaustausch an der identifizierten Systemgrenze festgelegt. Dabei wird das Verhalten in Form eines Dienstmodells strukturiert erfasst. Dieses setzt sich zusammen aus einer Hierarchie von Diensten und Wechselwirkungen zwischen diesen. Ein *Dienst* liefert eine formale Spezifikation einer Nutzerfunktion des Systems. Wechselwirkungen beschreiben, wie die eigenständig definierten Dienste sich gegenseitig beeinflussen, um das gewünschte Gesamtverhalten zu erbringen. Besteht also eine Wechselwirkung zwischen Diensten, so ergibt sich das Gesamtverhalten nicht allein aus den Spezifikationen der einzelnen Dienste, sondern nur unter zusätzlicher Berücksichtigung der Wechselwirkung. Die Wechselwirkungen machen somit die Interaktion zwischen Diensten explizit und definieren das über die eigenständigen Spezifikationen hinausgehende Interaktionsverhalten.

Die dienstbasierte Spezifikation strukturiert die vom System angebotenen Nutzerfunktionen hierarchisch und abstrahiert von der logischen Verteilung und Implementierungsdetails. Ein Dienst stellt somit einen Ausschnitt (oder eine Projektion) aus dem Gesamtverhalten dar. Die Systemgrenze ändert sich bei dieser Zerlegung nicht: Teildienste spezifizieren nur das Verhalten, wie es an der Grenze des Gesamtsystems beobachtbar ist. Insbesondere stellt die syntaktische Schnittstelle eines Teildienstes eine Projektion aus der Gesamtschnittstelle dar.

Im Dienstmodell sind partielle Verhaltensdefinitionen zugelassen, damit auch unterspezifiziertes Verhalten modellierbar ist. Dienste legen das Verhalten nicht notwendigerweise

[1] Eine der wenigen Ausnahmen bildet der AutoMoDe-Ansatz von Bauer et al. [BBR+07].

für alle möglichen Eingaben fest.

Im nächsten Abschnitt wird die Notationstechnik zur Beschreibung einer Nutzerfunktion eingeführt. In Abschnitt 3.3.2 wird ein Operator zur Kombination von Nutzerfunktionen definiert. Abschnitt 3.3.3 definiert die allgemeine Form des Kombinationsoperators, die priorisierte Kombination. Das Metamodell der dienstbasierten Spezifikation wird in Abschnitt 3.3.4 präsentiert.

3.3.1. Einzelner Dienst

Typischerweise betrachten mehrere Stakeholder ein System aus unterschiedlichen Perspektiven und spezifizieren demzufolge jeweils unterschiedliche Aspekte des Systemverhaltens. So sind z.B. für einen Entwickler der Motorsteuerung ganz andere Aspekte der ACC-Steuerung von Interesse als für einen Entwickler des Kombiinstruments. Daher ist eine Spezifikation eine Sammlung von Anwendungsfällen, die eventuell unvollständige und beispielhafte Interaktionsmuster des Systems mit seiner Umgebung beschreiben.

Im Allgemeinen beschreibt ein Anwendungsfall (engl. use case) eine mögliche Anwendung des Systems. Dabei ist es wichtig, wie die Benutzer mit dem System interagieren und wie das System auf die Interaktionen seiner Umwelt reagiert. Der Schwerpunkt liegt dabei auf der Fragestellung *was* passiert, nicht *wie* es passiert.

> „A use case is a description of a set of sequence of actions, including variants, that a system performs that yields an observable result of value to a particular actor“ [JBR99, S. 432].

In der Domäne eingebetteter reaktiver Systeme liegt der Fokus der Betrachtung auf dem Nachrichtenaustausch zwischen dem System und seiner Umgebung. In dieser Domäne (und in der vorliegenden Arbeit) werden die Begriffe „Anwendungsfall“ und „Interaktionsmuster“ synonym verwendet.

Definition 3.1 (Anwendungsfall/Interaktionsmuster)**.** Ein Anwendungsfall (auch Interaktionsmuster genannt) ist eine Menge von zeitlich geordneten, an der Systemgrenze beobachtbaren Folgen von Nachrichten zwischen dem System und seiner Umgebung. □

***Bemerkung* 3.3** (Anwendungsfall vs. Use Case)**.** Es ist wichtig, den Unterschied zwischen der Definition eines Anwendungsfalls in der vorliegenden Arbeit und der eines Use Cases zu betonen. Ein Anwendungsfall ist im Kontext reaktiver Systeme ein Interaktionsmuster, das beschreibt, wie man mit dem System interagieren kann. Unter einem Use Case ist dagegen eine bestimmte Vorlage für die Beschreibung eines Anwendungsfalls gemeint (auch Use Case templates genannt). Mögliche Use-Case-Schablonen sind z.B. in [JBR99, Coc03] beschrieben. In der vorliegenden Arbeit bezieht sich der Begriff „Anwendungsfall“ ausschließlich auf Definition 3.1. □

Die Anforderungsspezifikation stellt einen Kontrakt zwischen dem Auftraggeber und dem Entwickler dar. Daher ist es wichtig, dass sie für beide Seiten verständlich ist. Um das zu

erreichen, müssen Anwendungsfälle mit den Begriffen des Benutzers, d.h. aus der Black-Box-Sicht definiert werden. Innere Strukturen des Systems, die für den Benutzer nicht sichtbar sind, dürfen nicht zur Beschreibung eines Anwendungsfalls verwendet werden. Weiterhin muss ein Anwendungsfall eine für den Benutzer erkennbare Funktionalität darstellen. Anwendungsfälle sollen nicht in sehr kleine und abstrakte Interaktionsmuster zerlegt werden (um z.B. Redundanz zu verringern) – ein Benutzer soll den Zweck eines Anwendungsfalls nachvollziehen können.

Für die Beschreibung von Anwendungsfällen existieren mehrere Notationen, allen voran die UML-Use-Case- und Sequenzdiagramme. Für eine formale und automatische Anforderungsanalyse wird in der vorliegenden Arbeit eine automatenbasierte Notation eingeführt.

Definition 3.2 (Dienst)**.** Ein Dienst ist die Formalisierung eines Interaktionsmusters, d.h. einer Relation zwischen Ein- und Ausgabenachrichten des Systems.

Ein Dienst hat eine syntaktische Schnittstelle und eine Verhaltensspezifikation. Die *syntaktische Schnittstelle* umfasst eine Menge getypter Ein- und Ausgabeports, durch die der Dienst mit seiner Umgebung kommuniziert. Die *Verhaltensspezifikation* definiert in Form eines partiellen I/O-Transitionssystems (im Weiteren auch Automat genannt) eine *partielle* Abbildung von Ein- auf Ausgabewerte an den Ports der Schnittstelle. □

Im Folgenden wird eine informelle Definition eines partiellen I/O-Transitionssystems eingeführt, das in etwa den I/O-Automaten von Lynch und Tuttle [LT89] entspricht. Eine formale Semantik des Transitionssystems wird in Abschnitt 4.1.2 definiert. Ein I/O-Automat besteht aus einer Menge von Input-, Output- und lokalen Variablen sowie einer Menge von Transitionen. Eine der lokalen Variablen speichert einen so genannten Kontrollzustand des Automaten. Mindestens einer der Kontrollzustände muss Initialzustand sein. Eine Transition definiert den Übergang von einem Kontrollzustand in einen weiteren Zustand. Eine Transition besteht aus vier Bestandteilen: der Vorbedingung, den Ein- und Ausgabemustern und der Nachbedingung. Die *Vorbedingung* enthält Bedingungen an die Eingaben und die aktuellen Werte der lokalen Variablen. Die *Ein-* und *Ausgabemuster* beschreiben entsprechend die Werte der Ein- und Ausgabevariablen. Der boolesche Ausdruck `i?v` bezeichnet ein Eingabemuster, das wahr ist, wenn die Eingabevariable i den Wert v enthält. Das Ausgabemuster `o!v` ist wahr, wenn die Ausgabevariable o den Wert v enthält. Atomare Ein- und Ausgabemuster werden durch die logische Konjunktion konkateniert. Die Nachbedingung definiert die Werte der lokalen Variablen nach der Ausführung der Transition. Eine Transition kann für eine Belegung ausgeführt werden, wenn diese Belegung der Eingabe- und lokalen Variablen die Vorbedingung sowie das Eingabemuster der Transition erfüllt. Das Ergebnis der Ausführung einer Transition ist eine Variablenbelegung, die das Ausgabemuster und die Nachbedingung erfüllen.

Der Automat kann nichtdeterministisch sein, d.h. für dieselben Eingaben und lokalen Variablen können mehrere ausführbare Transitionen existieren. Dem Automaten liegt ein diskretes Zeitmodell zugrunde: die Ausführung einer Transition benötigt genau ein konstantes Zeitintervall. Es gibt einen Zustand vor und einen nach der Ausführung der

Transition. Der Automat definiert die Reaktion nur auf eine Teilmenge aller möglichen Eingaben. Für die restlichen Eingaben wird keine Transition ausgeführt – der Automat modifiziert seine lokale Variablen nicht und beschränkt die Ausgaben nicht. In diesem Fall können die Ausgabevariablen beliebige Werte aus dem Wertebereich enthalten. Daraus folgt die Definition des Definitionsbereichs eines Dienstes.

Definition 3.3 (Definitionsbereich)**.** Der Definitionsbereich eines Dienstes ist die Menge aller syntaktisch korrekten Eingaben, die vom Dienst verarbeitet werden können. Ein Dienst kann eine Eingabe verarbeiten, falls für diese Eingabe eine definierte, ausführbare Transition im Transitionssystem des Dienstes existiert. □

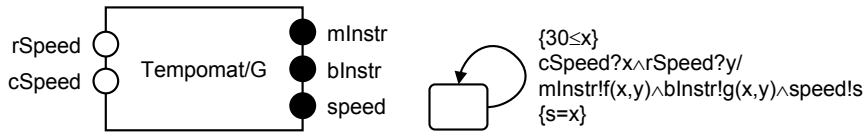
***Bemerkung* 3.4** (Partialität des Dienstes)**.** Der Grund für beliebige Ausgaben auf undefinierte Eingaben liegt darin, dass ein Anwendungsfall nur eine von vielen funktionalen Anforderungen an das System darstellt. Eine einzige Anforderung beschäftigt sich mit einer Teilmenge der Eingaben. Für die restlichen Eingaben beschränkt sie die Ausgaben nicht – alle möglichen Ausgaben sind gültig. Diese partielle Beschreibung macht es möglich, die Spezifikation eines Systems über mehrere Dienste zu verteilen bzw. die Reaktion auf bestimmte Eingaben unspezifiziert zu lassen. □

Beispiel 3.7 (Dienst `Tempomat/G`)**.** Der Dienst `Tempomat/G` aus Abbildung 3.8 formalisiert den Anwendungsfall, in dem die ACC-Steuerung die momentane Fahrgeschwindigkeit an die Wunschgeschwindigkeit angleicht. Das Fahrzeug befindet sich auf einer Gerade und die momentane Geschwindigkeit ist über 30 km/h. Die syntaktische Schnittstelle des Dienstes ist in Abbildung 3.8(a) graphisch dargestellt. Sie umfasst zwei Eingabeports `rSpeed` und `cSpeed` und drei Ausgabeports `mInstr`, `bInstr` und `speed`. Die Verhaltensspezifikation ist durch den Automaten aus Abbildung 3.8(b) gegeben. Der Automat besteht aus einem Zustand und einer Transition. Immer wenn die Eingabevariablen *cSpeed* und *rSpeed* entsprechend die Werte x und y erhalten (Eingabemuster `cSpeed?x`$\wedge$`rSpeed?y`) und der Wert x größer gleich 30 ist (Vorbedingung {`30`$\leq$`x`}, der Wert y ist unbeschränkt), wird die Transition ausgeführt. Der Nachfolgezustand der Transition ist die Variablenbelegung $mInstr = f(x, y)$, $bInstr = g(x, y)$, $speed = s$ und $s = x$. $f(x, y)$ und $g(x, y)$ sind zwei Funktionen, deren Definition an dieser Stelle nicht angegeben wird, die Variable s ist eine lokale Variable mit der Initialbelegung $s = 0$. Diese Variablenbelegung erfüllt das Ausgabemuster `mInstr!f(x,y)`$\wedge$`bInstr!g(x,y)`$\wedge$`speed!s` und die Nachbedingung {`s=x`}.

Ist die momentane Geschwindigkeit unter 30 km/h gibt es keine ausführbare Transition – der Dienst ist partiell definiert. Das heisst, in seinem Definitionsbereich liegen alle Tupel von Wunschgeschwindigkeit und momentaner Geschwindigkeit für die gilt, dass die momentane Geschwindigkeit größer oder gleich 30 km/h ist. □

3.3.2. Dienstkombination

Die Spezifikation eines multifunktionalen Systems umfasst mehrere Anwendungsfälle, welche verschiedene Verhalten eines Systems definieren. Anwendungsfälle können „über-



(a) Schnittstelle (b) Verhaltensspezifikation

Abbildung 3.8.: Spezifikation des Dienstes `Tempomat/G`

lappen“, indem sie das Verhalten an denselben Ein- und/oder Ausgabeports spezifizieren. Anforderungen werden von verschiedenen Stakeholdern aufgestellt und können daher zunächst widersprüchlich sein. Das heißt, sie fordern zum selben Zeitpunkt unterschiedliche Reaktionen auf dieselben Eingaben an denselben Ausgabeports.

Demzufolge wird im Folgenden ein Operator zur Kombination von Diensten definiert. Die Idee, die Gesamtspezifikation aus mehreren Teilmodellen zu bilden, ist nicht neu und geht auf die Arbeiten von Dijkstra [Dij72] und Parnas [Par72] zurück. Die meisten Ansätze, die dem von Dijkstra und Parnas vorgeschlagenen Prinzip der Modularität folgen, komponieren Teilmodelle durch die strukturelle Komposition aus Abschnitt 3.2.1. Dabei wird definiert, wie *unterschiedliche* Teilsysteme (Module, Komponenten) zusammen agieren. Jedes Teilsystem ist in Form eines *einzigen* Modells dargestellt – überlappende Interaktionsmuster sind innerhalb eines Systems nicht modellierbar.

Mit dem Prinzip der funktionalen Kombination aus Abschnitt 3.2.2 können eigenständig modellierte Dienste zu größeren Diensten kombiniert werden. Das Ziel dabei ist, verschiedene Interaktionsmuster ein und desselben Systems zu integrieren. Intuitiv ausgedrückt, ist ein kombinierter Dienst ein Container, der mehrere parallel agierende Teildienste enthält. Während jeder der Teildienste nur einen Aspekt des Verhaltens definiert, ergibt ihre Kombination das Gesamtverhalten. Dienste können dabei dieselben Ein- wie auch Ausgabeports haben. Allerdings darf keine direkte Kommunikation zwischen Diensten stattfinden – Dienste empfangen ihre Eingaben aus und senden ihre Ausgaben ausschließlich an die Systemumgebung. Dies entspricht genau der Definition von Anwendungsfällen von Jacobson et al. [JBR99, S. 137]:

> „Use-case instances do not interact with other use-case instances. The only kind of interactions in the use-case model occur between actor instances and use-case instances. The reason for this is that we want the use-case model to be simple and intuitive to allow fruitful discussions with end-users and other stakeholders without getting us bogged down in details“.

Laut dieser Definition darf ein Anwendungsfall dem anderen keine Nachrichten senden. Die einzige Art der Interaktion in einer anwendungsfallorientierten Spezifikation findet zwischen dem System und seiner Umgebung statt. Das liegt daran, dass Benutzer sich nicht mit der internen Struktur des Systems (z.B. Schnittstellen zwischen Anwendungsfällen) auseinander setzen sollen.

Die Semantik des Kombinationsoperators Anders als bei der klassischen Auffassung der Komposition [LT89, dAH01], welche die Anzahl möglicher Verhalten einzelner Modelle *reduziert*, ist an dieser Stelle ein Operator zur *Erweiterung* der Gesamtfunktionalität von Interesse. Das folgt aus der Beobachtung, dass in einer Spezifikation jeder weitere Anwendungsfall das definierte Systemverhalten um ein weiteres Stück Funktionalität erweitert. In der strukturellen Komposition jedoch schränkt jede weitere Komponente das Gesamtverhalten ein – das komponierte System kann einen Schritt genau dann machen, wenn *alle* seine Komponenten einen Schritt machen können. In der funktionalen Kombination fügt ein Dienst der Spezifikation Verhalten hinzu – der Definitionsbereich des Systems wird erweitert und das Verhalten des Systems wird deterministischer.

Definition 3.4 (Dienstkombination)**.** Die syntaktische Schnittstelle des kombinierten Dienstes, der aus mehreren Teildiensten besteht, ist die Vereinigung der Schnittstellen aller seiner Teildienste. In der Kombination führen all diejenigen Dienste je eine Transition aus, welche die aktuelle Eingabe verarbeiten können und deren auszuführende Transitionen untereinander nicht widersprüchlich sind. Ausschließlich diese Dienste bestimmen die Ausgaben des kombinierten Dienstes. Die restlichen Dienste modifizieren ihre lokalen Variablen nicht und schränken die Werte an den Ausgabeports nicht ein. Transitionen sind für eine gegebene Eingabe widersprüchlich, falls ihre Ausgabemuster für diese Eingabe kontradiktorische logische Aussagen sind. □

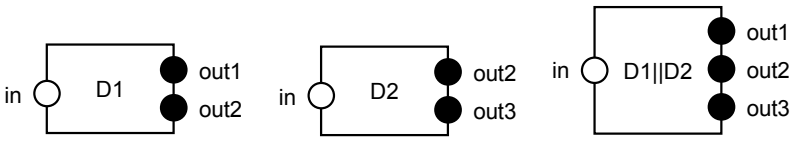
***Bemerkung* 3.5** (Synchrone Nebenläufigkeit mit gemeinsamen Variablen)**.** Broy definiert in [Bro05a, Kapitel 2.6] mehrere Arten der Nebenläufigkeit. Beim *Interleaving* (asynchroner Nebenläufigkeit) von zwei Transitionssystemen führt jeweils nur eines der beiden Systeme eine Transition aus. Ein besonderer Fall vom Interleaving ist *Stuttering*, bei dem auch möglich ist, dass keines der beiden Systeme eine Transition ausführt. Bei *synchroner Nebenläufigkeit* werden jeweils zwei Transitionen der gegebenen Transitionssysteme gleichzeitig in einer Transition des zusammengesetzten Systems ausgeführt. Broy lässt auch Mischungen der beiden Konzepte der Komposition zu, wobei einige der Transitionen synchron und andere im Interleavingmodus ausgeführt werden. Die Voraussetzung für diese Mischform ist die disjunkten Variablenmengen der beiden Transitionssysteme. Im Gegensatz zu [Bro05a] definiert die Dienstkombination aus Definition 3.4 eine Mischung aus dem Stuttering und synchroner Nebenläufigkeit mit gemeinsamen Variablen. □

***Bemerkung* 3.6** (Definitionsbereich und Bild des kombinierten Dienstes)**.** Nach dieser Definition und unter der Annahme der konfliktfreien Dienstkombination (mehr dazu in Abschnitt 5.2.2) kann der kombinierte Dienst die *Vereinigung* der Eingaben aller Teildienste verarbeiten. Die Reaktion des kombinierten Dienstes auf eine Eingabe entspricht dem *Durchschnitt* der Ausgaben all derjenigen Teildienste, die diese Eingabe in der Kombination verarbeiten können. □

***Bemerkung* 3.7** (Sammlung von Anforderungen)**.** Es ist wichtig zu betonen, dass es sich bei der Spezifikation um eine Sammlung von Anforderungen an das Systemverhalten handelt. Eine gegebene Anforderung definiert die Reaktion des Systems nur auf

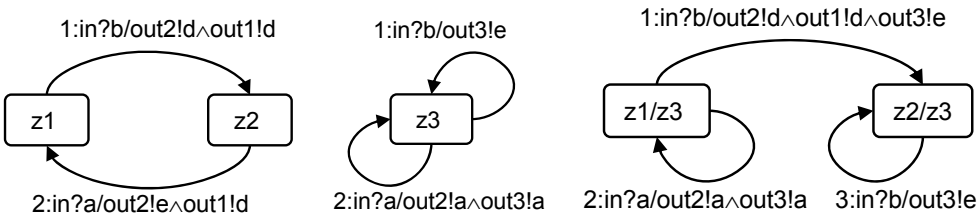
bestimmte Eingaben. Für alle anderen Eingaben schränkt diese Anforderung das Systemverhalten nicht ein. □

Alle wichtigen Merkmale der Dienstkombination werden zuerst anhand eines einfachen Beispiels erläutert. Anschließend wird ein Ausschnitt aus der Spezifikation der laufenden Fallstudie gezeigt.



(a) Dienst `D1` (b) Dienst `D2` (c) Kombination

Abbildung 3.9.: Schnittstellen der Dienste aus Beispiel 3.8



(a) Dienst `D1` (b) Dienst `D2` (c) Dienstkombination `D1` und `D2`

Abbildung 3.10.: Verhalten der Dienste aus Beispiel 3.8

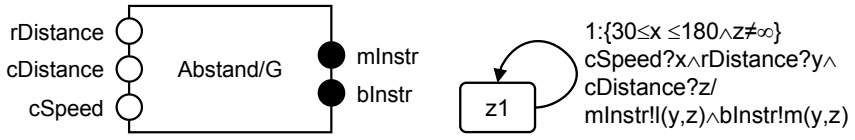
Beispiel 3.8 (Funktionale Dienstkombination)**.** Seien zwei Dienste `D1` und `D2` gegeben. Der Dienst `D1` hat den Eingabeport `in` und zwei Ausgabeports `out1` und `out2`, der Dienst `D2` hat den Eingabeport `in` und die Ausgabeports `out2` und `out3` (vgl. Abbildungen 3.9(a) und 3.9(b)). Ihre Schnittstellen überlappen an zwei gemeinsamen Ports `in` und `out2`. Das Verhalten der beiden Dienste ist durch die entsprechenden Automaten aus Abbildungen 3.10(a) und 3.10(b) definiert. Man beachte, die beiden Dienste haben disjunkte Mengen von Ausgaben am gemeinsamen Port `out2` für die Eingabe `a`. Die funktionale Kombination der beiden Dienste (bezeichnet durch `D1`‖`D2`) ergibt den Dienst aus Abbildungen 3.9(c) und 3.10(c). Der Zustand `z1/z3` (bzw. `z2/z3`) im kombinierten Automaten bedeutet, dass `D1` sich im Zustand `z1` (bzw. `z2`) und `D2` im Zustand `z3` befinden.

Nach Definition 3.4 kann der kombinierte Dienst folgende Schritte machen. Empfängt der Dienst die Nachricht `b` durch den Port `in` und befindet sich der Teildienst `D1` im Zustand `z1`, können die beiden Teildienste jeweils Transitionen 1 gleichzeitig ausführen. Dies ergibt Transition 1 im kombinierten Automaten aus Abbildung 3.10(c). Diese Transition bestimmt Werte an allen drei Ausgabeports. Trifft die Nachricht `a` ein, kann der

Teildienst `D1` keine Transition im Zustand `z1` ausführen – D2 führt Transition 2 aus. Dies ergibt Transition 2 im kombinierten Automaten, welche die Ausgabe durch den Port `out1` nicht spezifiziert – ein beliebiger Wert kann durch den Port gesendet werden. Auch für die Nachricht `b` im Zustand `z2` hat der Dienst `D1` keine ausführbare Transition – D2 führt Transition 1 aus, was Transition 3 im kombinierten Automaten ergibt. Dabei sind die Ports `out1` und `out2` nicht spezifiziert. Im Zustand `z2/z3` hat der kombinierte Automat keine ausführbare Transition für die Eingabe `a`. Das liegt daran, dass Transitionen 2 der beiden Teildienste nicht kombinierbar sind – für dieselbe Eingabe `a` fordern sie widersprüchliche Werte an demselben Port `out2`. Daher sind alle Ausgabewerte im Zustand `z2/z3` für die Eingabe `a` unspezifiziert.

Welche formalen Beziehungen zwischen den Diensten `D1`, `D2` und dem kombinierten Dienst bestehen, wird in Abschnitt 4.4.1 erläutert. □

Das Beispiel aus der Fallstudie ist wesentlich einfacherer, da die Definitionsbereiche der beiden Teildienste disjunkt sind.



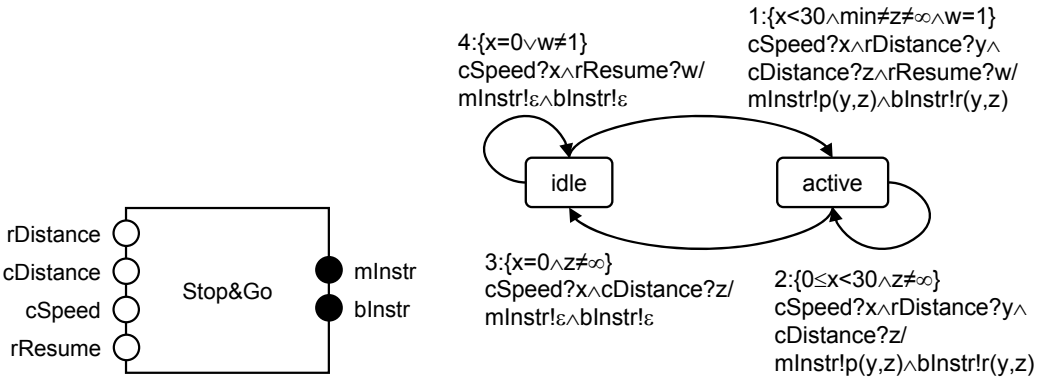
(a) Schnittstelle (b) Verhalten

Abbildung 3.11.: Dienst `Abstand/G`

Beispiel 3.9 (Dienstkombination `Abstandsregelung/G`)**.** Es wird die Kombination der beiden Dienste `Abstand/G` und `Stop&Go` betrachtet. Der Dienst `Abstand/G` aus Abbildung 3.11 spezifiziert das Verhalten der ACC-Steuerung bei einer Geschwindigkeit zwischen 30 und 180 km/h, wenn ein vorausfahrendes Fahrzeug erkannt ist. Die einzige Transition des Dienstautomaten ist ausführbar, wenn der Wert am Eingabeport `cSpeed` zwischen 30 und 180 liegt und der Wert am Port `cDistance` ungleich ∞ ist. Die Nachricht ∞ bedeutet, dass kein Objekt erkannt ist. Anhand der Werte an den Ports `cDistance` und `rDistance` werden die Steuerbefehle an den Motor und die Bremse berechnet. Die Funktionen $l(x, y)$ und $m(x, y)$ für die Berechnung der Steuerbefehle spielen in der Erläuterung des Ansatzes keine Rolle und werden an dieser Stelle nicht definiert.

Der Dienst Stop&Go aus Abbildung 3.12 spezifiziert das Verhalten der ACC-Steuerung bei einer Geschwindigkeit unter 30 km/h. Ist das eigene Fahrzeug durch ein anhaltendes Vorderfahrzeug bis in den Stillstand abgebremst worden (Zustand `idle` im Dienstautomaten), erfolgt ein automatisches Anfahren, sofern das Vorderfahrzeug wieder beschleunigt und der Fahrer die Anfahrt über den Bedienhebel bestätigt. Das eigene Fahrzeug folgt daraufhin wieder automatisch dem Vorderfahrzeug (Zustand `active`). Transition 1 ist ausführbar, wenn ein Objekt erkannt, der Abstand zum Objekt größer als der minimale Abstand (das Objekt ist vorgefahren) und die `resume`-Nachricht eingetroffen ist.

Die Berechnung der Steuerbefehle erfolgt (genau wie bei Transition 2) anhand der Funktionen $p(x, y)$ und $r(x, y)$, die an dieser Stelle nicht definiert werden. Ist die momentane Geschwindigkeit gleich null, wird Transition 3 ausgeführt – der Automat geht in den Zustand `idle` über. An die Brems- und Motorsteuerung werden dabei keine Steuerbefehle versendet (ε bedeutet eine leere Nachricht). Dasselbe gilt für Transition 4, die ausgeführt wird, solange die momentane Geschwindigkeit gleich null oder keine resume-Nachricht eingetroffen ist.



(a) Schnittstelle (b) Verhalten

Abbildung 3.12.: Dienst `Stop&Go`

Die Schnittstelle des kombinierten Dienstes ist mit der des Dienstes `Stop&Go` identisch. Die Dienstkombination ergibt das Verhalten, das durch den hierarchischen Automaten [HN96] aus Abbildung 3.13 schematisch dargestellt ist. Wegen disjunkter Definitionsbereiche der beiden Teildienste kann zu jedem Zeitpunkt entweder der eine oder der andere Dienst eine Transition ausführen. Bei einer momentanen Geschwindigkeit von unter 30 km/h kann nur der Dienst `Stop&Go`, ansonsten nur der Dienst `Abstand/G` einen Schritt machen. □

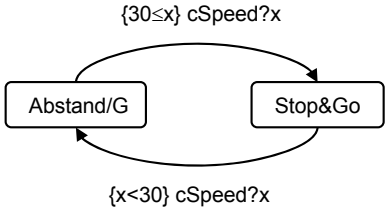


Abbildung 3.13.: Kombination der Dienste `Abstand/G` und `Stop&Go`

3.3.3. Priorisierte Dienstkombination

Bis jetzt wurde nur ein Sonderfall der Dienstkombination betrachtet, in dem zwischen Diensten keine Wechselwirkung existiert. Im Allgemeinen können Anwendungsfälle eine höhere Priorität haben als andere Anwendungsfälle. Hat ein Anwendungsfall den Vorrang vor den anderen, muss das System nur das von ihm beschriebene Interaktionsmuster aufweisen – die anderen beschränken das Systemverhalten in keiner Weise. Dabei erfolgt die Priorisierung situationsabhängig, d.h. ein Anwendungsfall ist nicht immer, sondern nur in bestimmten Fällen priorisiert. Beispielsweise muss das Systemverhalten im Notfall Vorrang vor dem regulären Verhalten haben. Ist das Verhalten mit den Anwendungsfällen „Reguläres Verhalten“ und „Notfallverhalten“ beschrieben, so muss das System in einem Normalfall nur die Anforderungen des Ersten und in einem Notfall nur die des Letzteren erfüllen. Ein ähnlicher Mechanismus zur Strukturierung von Anforderungen ist die Strukturierung über Systemmodi: das System wechselt sein Verhalten bei einem Moduswechsel. Ein klassisches Beispiel für den Moduswechsel ist „Fliegen“, „Landen“und „Rollen“eines Flugzeuges. Für jeden der drei Modi gibt es je eine Menge relevanter Anwendungsfälle, für die sich das Flugzeug bei gleichen Eingaben unterschiedlich verhält.

Um diesem Sachverhalt Rechnung zu tragen, wird das Konzept der priorisierten Kombination eingeführt. Diese Art der Kombination erlaubt es, abhängig von den aktuellen Eingaben einen Anwendungsfall in Bezug auf andere zu priorisieren. Mit anderen Worten handelt es sich dabei um eine *dynamische Priorisierung* von Anwendungsfällen. Bei einer dynamischen Priorisierung muss das System nur das Verhalten für eine Eingabe aufweisen, das von dem für diese Eingabe priorisierten Anwendungsfall spezifiziert ist – die anderen, so genannten unpriorisierten Anwendungsfälle bestimmen das Systemverhalten nicht. Im Falle einer statischen Priorisierung wäre immer derselbe Anwendungsfall priorisiert, unabhängig von den Eingaben.

Definition 3.5 (Priorisierte Dienstkombination)**.** Sei S eine priorisierte Dienstkombination, die aus zwei Teildiensten S_1 und S_2 besteht[2]. Ihre syntaktische Schnittstelle umfasst alle Ports der beiden Teildienste. In dieser Kombination kann einer der beiden Teildienste abhängig von den Eingaben eine höhere Priorität haben als der andere. Die Priorisierung wird von einem *Priorisierungsdienst* S_P gesteuert, dessen Schnittstelle alle Eingabeports der beiden Teildienste und keine Ausgabeports umfasst. Die Transitionen des Dienstes S_P können einen der beiden Dienste priorisieren. Wenn für die aktuelle Eingabe eine Transition von S_P ausgeführt wird und diese Transition den Dienst S_2 priorisiert, ist nur S_2 aktiviert – S_1 ändert seine lokalen Variablen nicht und beschränkt das Verhalten an seinen Ausgabeports nicht. Kann der Dienst S_P keine Transition ausführen oder die ausgeführte Transition priorisiert keinen der beiden Dienste, verhält sich die priorisierte Kombination wie die normale Kombination aus Definition 3.4. □

Aus der Definition folgt, dass der Prioritätsdienst S_P alle Eingaben in drei Mengen aufteilt. Für eine Menge von Eingaben stimmt das Verhalten des kombinierten Dienstes

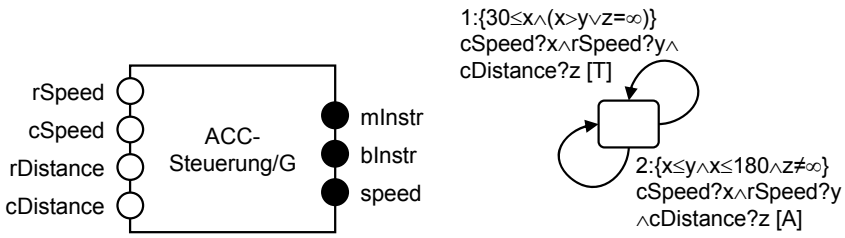
[2] In Abschnitt 4.3.3 wird die Priorisierung auf n Teildienste verallgemeinert.

nur mit dem Verhalten eines der beiden Teildienste, für die restlichen Eingaben mit dem Verhalten der beiden überein.

Es ist hervorzuheben, dass Anwendungsfälle für die Kombination nicht modifiziert werden müssen. Weder ihre syntaktischen Schnittstellen noch die Automaten müssen angepasst werden, um die Priorisierung zu realisieren. Dadurch bleibt die Modularität einzelner Anwendungsfälle erhalten.

Beispiel 3.10 (Priorisierte Kombination `ACC-Steuerung/G`)**.** Es werden die priorisierte Kombination der Dienste `Tempomat/G` und `Abstandsregelung/G` aus Beispielen 3.7 und 3.9 betrachtet. Die ACC-Steuerung muss sich wie der Dienst `Tempomat/G` verhalten, falls kein vorausfahrendes Fahrzeug erkannt und die momentane Geschwindigkeit größer als die Wunschgeschwindigkeit oder als 180 km/h ist. In allen anderen Fällen verhält sich das System wie der Dienst `Abstandsregelung/G`. Der Einfachheit halber wird das Verhalten in einer Kurve (`ACC-Steuerung/K`) an dieser Stelle nicht betrachtet.

Die Schnittstelle des kombinierten Dienstes ist in Abbildung 3.14(a) dargestellt. Das Verhalten des Prioritätsdienstes ist durch den Automaten aus Abbildung 3.14(b) gegeben. Der Automat besteht aus zwei Transitionen, die jeweils einen der beiden Teildienste priorisieren (`[T]` oder `[A]`). Ist der Wert am Port `cSpeed` größer oder gleich 30 und größer als der Wert am Port `rSpeed` oder der Wert am Port `cDistance` gleich ∞, wird Transition 1 ausgeführt und dadurch der Dienst `Tempomat/G` priorisiert – die `ACC-Steuerung/G` verhält sich wie der `Tempomat/G`. Ist der Wert am Port `cSpeed` kleiner als der Wert am Port `rSpeed` und kleiner gleich 180 und der Wert am Port `cDistance` ungleich ∞, wird Transition 2 ausgeführt – das System verhält sich wie der Dienst `Abstandsregelung/G`. Kann keine Transition ausgeführt werden (z.B. bei einer Geschwindigkeit unter 30 km/h und ohne ein vorausfahrendes Fahrzeug), verhält sich der kombinierte Dienst wie bei einer normalen Kombination aus Definition 3.4. □



(a) Schnittstelle (b) Priorisierungsdienst

Abbildung 3.14.: Dienst `ACC-Steuerung/G`

***Bemerkung* 3.8** (Beliebige Werte an den Ausgabeports)**.** Man beachte, dass die priorisierte Kombination in bestimmten Fällen beliebige Werte an den Ausgabeports zulässt. Das ist immer der Fall, wenn der gerade priorisierte Dienst für einen der Ausgabeports keine Werte spezifiziert, d.h. der Port ausschließlich aus der Schnittstelle des aktuell nicht priorisierten Dienstes ist. In Abbildung 3.9 auf Seite 35 wären das die Ports `out1`

bzw. `out2`, wenn jeweils die Dienste `D2` bzw. `D1` priorisiert würden. Im Zeitintervall der Priorisierung des Dienstes `D1` (bzw. `D2`) beschränkt die Dienstkombination die Werte am Port `out3` (bzw. `out1`) nicht. Will man diese Art der Unterspezifikation vermeiden, müssen die Dienste entsprechend erweitert werden, so dass alle Ausgabeports zu jeder Zeit von dem aktuell priorisierten Dienst kontrolliert werden. □

Es ist wichtig zu betonen, dass der eingeführte Ansatz keine weiteren funktionalen Beziehungen (vor allem keinen internen Informationsfluss) zwischen Diensten unterstützt. Einerseits liegt es daran, dass alle Beziehungen zwischen Anwendungsfällen dem Benutzer in einer verständlichen Form dargestellt werden müssen. Interne Informationsflüsse sind aus Black-Box-Sicht nicht sichtbar und für den Benutzer daher nicht nachvollziehbar. Im Gegensatz dazu sind sowohl die Eingaben, die eine Priorisierung verursachen, als auch deren Auswirkung direkt an der Systemgrenze sichtbar. Zudem war die priorisierte Kombination für die Black-Box-Spezifikation der beiden multifunktionalen Systeme aus den Fallstudien (siehe Anhang C) ausreichend.

3.3.4. Dienstmodell

Das Metamodell der dienstbasierten Spezifikation ist durch das UML-Klassendiagramm in Abbildung 3.15 gegeben. Die Systemspezifikation besteht aus einer Menge von Diensten, die entweder atomar oder kombiniert sind. Während das Verhalten eines atomaren Dienstes durch einen I/O-Automaten definiert ist, ergibt die Kombination der Teildienste das Verhalten des kombinierten Dienstes. Die syntaktische Schnittstelle eines Dienstes besteht aus einer Menge getypter Ports, die entweder Ein- oder Ausgabeports sein können. Ein kombinierter Dienst kann mehrere Priorisierungen enthalten, die Wechselwirkungen zwischen mehreren Teildiensten spezifizieren.

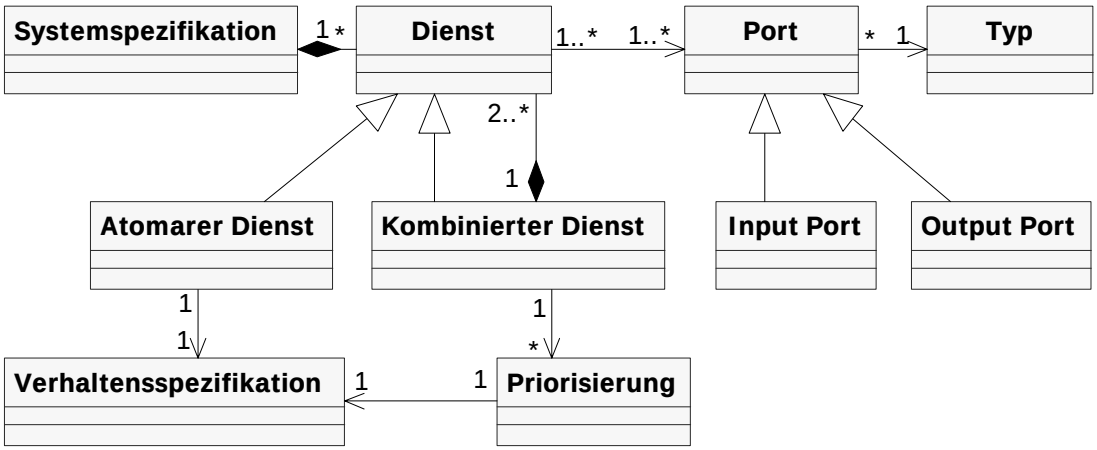


Abbildung 3.15.: Metamodell der dienstbasierten Spezifikation

Beispiel 3.11 (Dienstbasierte Spezifikation der ACC-Steuerung)**.** Die vollständige dienstbasierte Spezifikation der ACC-Steuerung ist in Abbildung 3.16 dargestellt. Der oberste Dienst ist aus zwei Teildiensten `Tempomat` und `Abstandsregelung` kombiniert, zwischen denen eine Priorisierung `TvsA` existiert. Die beiden Teildienste des kombinierten Dienstes `Abstandsregelung` spezifizieren das Verhalten, wenn ein vorausfahrendes Fahrzeug erkannt ist, `Abstand` bei einer Geschwindigkeit über und `Stop&Go` bei einer Geschwindigkeit unter 30 km/h. Das Verhalten des Tempomats auf einer Gerade unterscheidet sich von jenem in einer Kurve. Deswegen ist der Dienst `Tempomat` in zwei Teildienste unterteilt: während `Tempomat/G` das Verhalten auf einer Gerade spezifiziert, ist das Verhalten in einer Kurve vom `Tempomat/K` bestimmt. Zwischen den beiden Diensten existiert eine Priorisierung `GvsK/T`. Dasselbe gilt für den kombinierten Dienst `Abstand`. □

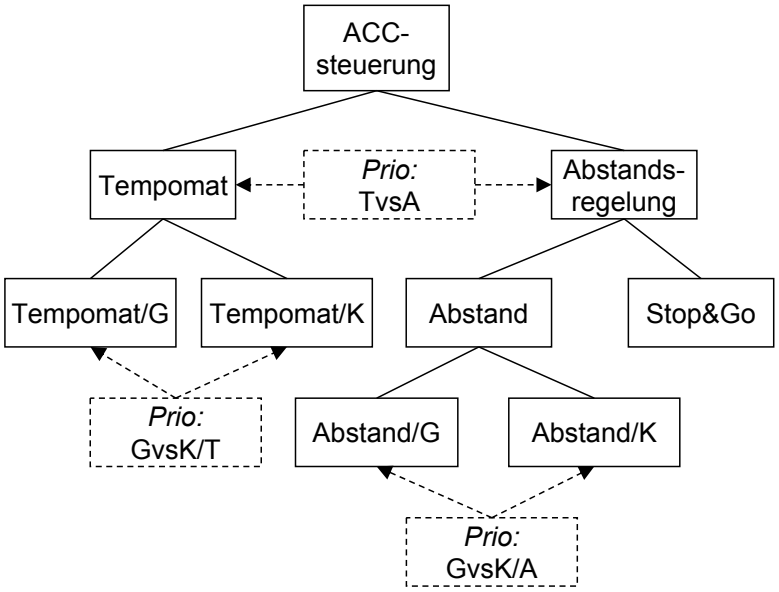


Abbildung 3.16.: Dienstbasierte Spezifikation der ACC-Steuerung

3.4. Nutzen der dienstbasierten Software-Entwicklung

Die vorgestellte Art der Modellierung hat folgende charakteristische Merkmale:

- Sie beschreibt funktionale Anforderungen in Form von Anwendungsfällen und abstrahiert von einer logischen Struktur.
- Sie ermöglicht, ein System aus unterschiedlichen Perspektiven zu beschreiben.
- Sie ermöglicht, das Verhalten sowohl in Modi als auch in Teilfunktionen zu unterteilen.
- Ihr liegt eine formale operationelle Semantik zugrunde.

Im Folgenden wird der Nutzen der dienstbasierten Spezifikation im Requirements Engineering, strukturiert nach diesen Merkmalen, dargestellt.

3.4.1. Anwendungsfallorientierte Entwicklung

Nach DIN 69901-5 beschreibt eine funktionale Anforderungsspezifikation (auch Lastenheft oder Kundenspezifikation genannt) die „vom Auftraggeber festgelegte Gesamtheit der Forderungen an die Lieferungen und Leistungen eines Auftragnehmers innerhalb eines Auftrages“. Dabei ist es wichtig, dem Entwickler die Möglichkeit zu geben, optimale Lösungen zu erarbeiten, ohne seine Lösungsmöglichkeiten durch zu restriktive Anforderungen zu sehr einzuschränken. Anwendungsfälle erweisen sich als eine geeignete Spezifikationstechnik, funktionale Anforderungen so allgemein wie möglich und so einschränkend wie nötig zu formulieren.

Tatsächliche Anforderungen in verständlicher Form Im Requirements Engineering werden unter anderem zwei Ziele verfolgt: „tatsächliche“ Benutzeranforderungen zu finden und sie in einer sowohl für den Benutzer als auch für den Entwickler verständlichen Form darzustellen.

Unter tatsächlichen Anforderungen sind Anforderungen gemeint, deren Implementierung dem Benutzer Mehrwert bringt. Bei der Spezifikation reaktiver Systeme soll die Interaktion zwischen dem System und dem Benutzer im Mittelpunkt der Betrachtung stehen. Wenn eine Funktion ohne Einbeziehung des Benutzers beschrieben wird (z.B. eine für den Benutzer nicht sichtbare Funktion), ist es schwer zu beurteilen, ob sie nötig oder wertvoll ist. Die übliche Frage im Requirements Engineering, was das System tun soll, ist öfters irreführend [JBR99, S. 38]. Als Antwort auf diese Frage wird häufig eine Liste komplexer Funktionen erstellt, die jedoch für keinen Benutzer nützlich sind. Ein Anwendungsfall beantwortet die Frage: „Was soll das System für *Benutzer* tun“. Dadurch hilft die anwendungsfallorientierte Entwicklung dem Anforderungsanalytiker, Funktionen zu identifizieren, die Benutzerwünsche erfüllen. Somit entsteht eine funktionale Spezifikation aus einer Menge von Anwendungsfällen, die jeweils ein mögliches Nutzungsszenario für eine Gruppe von Benutzern spezifizieren.

Darüber hinaus sind Anwendungsfälle ein intuitiver Ansatz, Funktionen eines eingebetteten Systems aus der Benutzersicht zu beschreiben. Ein Anwendungsfall beschreibt eine mögliche Interaktion zwischen dem Benutzer und dem System in Form einer Sequenz von Ein- und Ausgaben. Diese Abstraktion von den Implementierungsdetails erlaubt dem Benutzer, das Systemverhalten mit ihm vertrauten Begriffen – Beobachtungen an der Systemgrenze – zu beschreiben, ohne sich mit der Implementierung befassen zu müssen.

Inkrementelle Anforderungsanalyse Typischerweise entsteht eine Spezifikation in mehreren Iterationen. Ihre initiale Version umfasst lediglich die wichtigsten Anwendungsfälle, die iterativ um weitere Fälle ergänzt werden. Beispielsweise wird die Fehlerbehandlung

üblicherweise viel später als das Verhalten im Normalfall behandelt. In den traditionellen (komponentenbasierten) Ansätzen müssen alle möglichen Fälle in einem Schritt modelliert oder durch explizites Re-Design aller Modelle kombiniert werden. Die Partialität von Diensten macht es möglich, einzelne Anwendungsfälle in separaten Modellen zu spezifizieren, um sie dann inkrementell und ohne Re-Design zu kombinieren. Die Anforderungsanalyse kann dabei zu jeder Zeit beginnen – die Vollständigkeit der dienstbasierten Spezifikation ist keine Voraussetzung für die Analyse auf Konsistenz.

Funktionsorientierte Spezifikation Die Entwicklung eingebetteter Software-Systeme war bisher auf eine technische Architektur ausgerichtet: eine Familie von Funktionen entsprach vielfach eins-zu-eins einem Steuergerät. Heutige Systeme umfassen Funktionen, die sich über Steuergeräte hinweg erstrecken, miteinander interagieren und letztendlich in die Zuständigkeit mehrerer Steuergeräteverantwortlicher fallen [BHRS08].

Die dienstbasierte Spezifikation stellt die Basis für die Abkehr von der Implementierungssicht dar und bietet damit die notwendige funktionsorientierte Sicht, um komponentenübergreifende Funktionen spezifizieren und analysieren zu können. Darüber hinaus ist eine funktionsorientierte Spezifikation eine notwendige Voraussetzung, um Verantwortlichkeiten für einzelne Funktionen im Entwicklungsprozess explizit zu verankern.

Entwicklungsprozess Im Folgenden wird kurz aufgezeigt, inwiefern der Entwicklungsprozess in den jeweiligen Phasen von einer anwendungsfallorientierten Spezifikation profitiert.

Systementwurf Durch die Spezifikation von Anwendungsfällen und Wechselwirkungen zwischen diesen können Designentscheidungen motiviert und unterstützt werden. Wegen der Vermeidung von Redundanz sollen gleiche Anwendungsfälle, die sich in unterschiedlichen Teilhierarchien befinden, von derselben Komponente implementiert werden. Anwendungsfälle, die sich in derselben Teilhierarchie befinden, haben oft dieselbe syntaktische Schnittstelle und beschreiben unterschiedliche Aspekte desselben Verhaltens – auch sie werden oft von einer Komponente implementiert. Priorisierungen zwischen Diensten weisen auf einen Datenfluss zwischen den Komponenten hin, welche an der Implementierung dieser Dienste beteiligt sind. Der Übergang von der Spezifikation zur Architektur wird in Kapitel 6 erläutert.

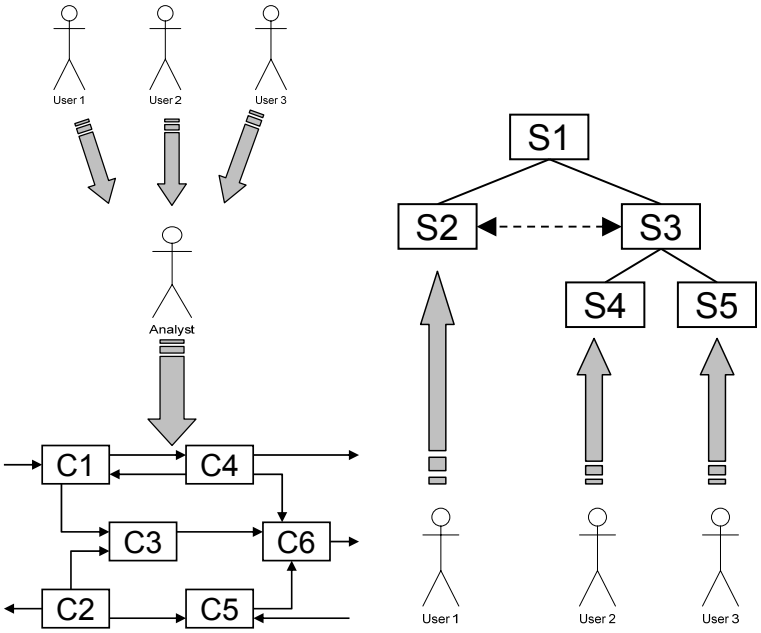
Testen Typischerweise definiert ein Testfall eine Menge von Eingaben, Vorbedingungen und zu erwartenden Ausgaben. Diese Testfälle können direkt aus dienstbasierten Spezifikationen abgeleitet werden. Der wesentliche Vorteil solcher Tests ist der, dass sie die zu erwartenden Anwendungsfälle des Systems abbilden, jedoch nicht durch die Implementierung geprägt sind. Fehler in der Testphase können exakt Anwendungsfällen und über diese den beteiligten Komponenten zugeordnet werden.

Wartung Eine eindeutige n:m-Beziehung zwischen Anwendungsfällen und Komponenten trägt zur Verbesserung der Wartbarkeit des Systems bei. Ändern sich im Laufe der Entwicklung Anforderungen, ist es einfach, den Kreis der betroffenen Komponenten

zu identifizieren und umgekehrt.

3.4.2. Mehrere Benutzerperspektiven auf Systemfunktionalität

Typischerweise beschränken sich Funktionen heutiger Software-intensiver Systeme nicht mehr auf einen einzelnen Benutzer und können somit nicht isoliert durch einen Stakeholder beschrieben werden. Vielmehr umfassen heutige Systeme tausende verschiedene Funktionen, die miteinander interagieren und in die Zuständigkeit mehrerer Stakeholder fallen. Mehrere Benutzer spezifizieren das System aus unterschiedlichen Perspektiven, indem sie nur für sie relevante Anwendungsfälle beschreiben. Unter einer Benutzerperspektive ist in diesem Kontext die Sicht eines Benutzers auf das zu spezifizierende System gemeint. Die Perspektiven unterschiedlicher Benutzergruppen passen selten zusammen – bestimmte Aspekte des Verhaltens können widersprüchlich oder für eine Benutzergruppe irrelevant (und somit in der Perspektive dieser Benutzergruppe unterspezifiziert) sein.



(a) Eine einzige Perspektive (b) Mehrere Perspektiven

Abbildung 3.17.: Benutzerperspektiven auf Systemfunktionalität

Traditionelle Ansätze wie z.B. [Hoa85, Mil89] setzen ein einziges Modell der Systemfunktionalität voraus, in das Anforderungen aus allen Informationsquellen zusammenfließen. Mehrere Benutzer wenden sich an *einen* Analytiker, der alle Anforderungen konsolidieren und in *ein* Modell zusammenfassen muss (vgl. Abbildung 3.17(a)). Anforderungen mehrerer Stakeholder in einem Schritt in die Gesamtspezifikation zu integrieren, birgt die Gefahr, dass einige von ihnen falsch dargestellt oder ganz vernachlässigt werden. Die neu erhobenen Anforderungen können unter Einfluss der bereits existierenden Gesamt-

spezifikation verzerrt werden – der Integrationsprozess ist implizit und erfolgt meistens im Kopf des Analytikers.

Der dienstbasierte Ansatz macht es dagegen möglich, die Anforderungsanalyse in zwei Phasen durchzuführen. Im ersten Schritt werden Verhalten eines Systems aus unterschiedlichen Benutzerperspektiven von zuständigen Stakeholdern in separaten Modellen spezifiziert (vgl. Abbildung 3.17(b)). Erst in einem darauffolgenden Schritt werden diese Modelle zur Gesamtspezifikation integriert. Somit entsteht eine Spezifikation aus einer Menge individueller Perspektiven und Wechselwirkungen zwischen diesen.

Die separate Darstellung von Perspektiven ist für die gesamte Anforderungsanalyse von Nutzen. Im Folgenden wird aufgezeigt, inwiefern die Phasen des Requirements Engineering von der perspektivenorientierten Spezifikation der Funktionalität profitieren können.

Erhebung von Anforderungen Die Erhebung von Anforderungen wird von ihrer Integration in die Gesamtspezifikation klar abgegrenzt. Jeder Stakeholder beschreibt die Funktionalität in einem separaten Modell so wie er sie vom System erwartet, anstatt neue Anwendungsfälle einem zentralen Modell hinzufügen. Ein Benutzer beschreibt für ihn relevante Anwendungsfälle ohne dabei andere Anwendungsfälle berücksichtigen zu müssen. Dies hilft dem Benutzer, sich mit seinem Beitrag zur Gesamtspezifikation zu identifizieren – er sieht, wie seine Sicht auf das System separat von den anderen Beteiligten dargestellt wird.

In der Anfangsphase der Analyse können Widersprüche zwischen Perspektiven ignoriert werden. Dies erlaubt dem Stakeholder, seine Anforderungen erst vollständig zu definieren, um sie später mit den Erwartungen anderer Benutzer zu konfrontieren. Dies entspricht dem Vorgehen eines Menschen beim Erforschen einer Idee, der seine Szenarien bis zum Ende durchspielen will, ohne sich gleich in lästigen und öfters trivialen Details zu verlieren. Erst wenn die einzelnen Perspektiven modelliert und alle Beteiligten bereit sind, darüber zu diskutieren, werden sie in die Gesamtspezifikation integriert.

Integration von Anforderungen Die Integration von Anforderungen aus unterschiedlichen Perspektiven ist im dienstbasierten Ansatz ein expliziter Entscheidungsprozess. Die explizite Integration bedeutet, dass Widersprüche zwischen Anforderungen nicht „on-the-fly“ während ihrer Erhebung, sondern in einem separaten Prozess mittels eines geeigneten Verfahrens gelöst werden (mehr dazu in Kapitel 5). Dies garantiert, dass die Reihenfolge, in der Anforderungen erhoben werden, keinen Einfluss auf die endgültige Spezifikation hat.

Im Gegensatz zu den traditionellen Ansätzen, die nur widerspruchsfreie Modelle zulassen, ist die Widerspruchsfreiheit keine notwendige Voraussetzung für die Kombinierbarkeit von Diensten. Dadurch können bei der Integration von Diensten Widersprüche zu einem späteren Zeitpunkt gelöst werden. Ein Widerspruch zwischen Diensten weist auf mehrere Alternativen zwischen Anforderungen hin. Bei der Lösung eines Widerspruchs wird eine Entscheidung getroffen, und ab diesem Zeitpunkt wird nur eine Alternative

weiter verfolgt. Eine voreilige Entscheidung bricht die Erforschung weiterer Ideen ab und impliziert eine eventuell unnötige Einschränkung der Weiterentwicklung.

Ein weiteres wichtiges Merkmal des dienstbasierten Ansatzes ist die Nachvollziehbarkeit getroffener Entscheidungen während der Anforderungsanalyse. Eine Widerspruchslösung wird in einem separaten Modell (in Form einer Priorisierung) festgehalten, die sich mit den ursprünglichen Anforderungsmodellen nicht vermischen. Sowohl ursprüngliche Anforderungen als auch Priorisierungen zwischen diesen bleiben identifizierbar, so dass die Rückverfolgung von Entscheidungen zwischen alternativen Anforderungen einfacher ist als in einem komponentenbasierten Modell. Außerdem ist es möglich, dass Entscheidungen rückgängig gemacht werden müssen, wenn sie sich im Laufe der Entwicklung als falsch herausstellen. In diesem Fall ist es kompliziert, eine andere Alternative zu finden, wenn die ursprünglichen Anforderungen nicht mehr separat vorhanden sind. In einer dienstbasierten Spezifikation muss man nur eine Priorisierung eliminieren oder modifizieren, um eine andere Entscheidung zu treffen.

Validierung und Verifikation Der dienstbasierte Ansatz trägt zur Validierung von Anforderungen bei. Die Validierung der Spezifikation eines komplexen multifunktionalen Systems ist eine mühsame Aufgabe, die am besten in mehrere handhabbare Teile aufgeteilt und nicht von einer Person durchgeführt werden soll. In einem komponentenbasierten Modell, in dem einzelne Anforderungen nicht separat identifizierbar sind, ist die Zuordnung von Verantwortlichkeiten kaum möglich – der Urheber einer Anforderung (oder Funktionsverantwortlicher) verliert jeden Bezug zu seiner Funktion. Bei der Trennung von Perspektiven kann ein Benutzer seine Anforderungen isoliert von den anderen validieren. Sein Modell kann schnell modifiziert werden, wenn er feststellt, dass sein Beitrag zur Gesamtspezifikation falsch dargestellt ist. Erst im darauffolgenden Schritt wird die Auswirkung seines Modells auf die Gesamtspezifikation validiert – ob das Modell den Anforderungen aus anderen Modellen widerspricht, welche Wechselwirkungen zwischen Perspektiven bestehen. Jede Funktion behält ihren Funktionsverantwortlichen, der sich an den entsprechenden Konfliktlösungen beteiligt.

Die eindeutigen Beziehungen zwischen Funktionsverantwortlichen und ihren Diensten sowie zwischen Diensten und ihren Implementierungen (mehr dazu in Kapitel 6) sind auch für die Verifikation der Implementierung von Nutzen. In einem modellbasierten Ansatz (wie z.B. in [BFG+08]) kann ein Funktionsverantwortlicher über eine explizite Kette von Modellen (Dienste – logische Komponenten des Designs) überprüfen, ob seine Anforderungen richtig implementiert wurden. Dabei wird nur ein Ausschnitt der Implementierung gegen einen Ausschnitt der Spezifikation verifiziert. Somit ist die dienstbasierte Spezifikation eine Voraussetzung für eine *kompositionale Verifikation* [dRdBH+01], welche eine vollständige Analyse größerer Systeme überhaupt erst möglich macht.

3.4.3. Modusbasierte Spezifikation

Die dynamische Priorisierung macht es möglich, die funktions- und modusbasierten Vorgehensweisen zu kombinieren. Im eingeführten Ansatz gibt es keinen formalen Unterschied zwischen Nutzerfunktionen und Systemmodi – Modi sind lediglich ein methodologisches Mittel zur Strukturierung von Nutzerfunktionen. Sie erlauben eine übergeordnete Sicht auf die Systemfunktionalität. Laut Jahanian und Mok [JM86], „modes may be viewed as control information that impose structure on the operation of a system“. Wie bereits erwähnt, sind ein klassisches Beispiel für Systemmodi Zustände wie „Fliegen“, „Landen“ und „Rollen“ eines Flugzeuges. Dieselbe Nutzerfunktion weist abhängig vom aktuellen Modus unterschiedliches Verhalten auf und wird dementsprechend durch unterschiedliche Dienste spezifiziert. Systemmodi erleichtern die Unterteilung der Gesamtmenge von Nutzerfunktionen in Teilmengen mit äquivalentem Verhalten. Modi werden (genau wie Nutzerfunktionen) durch Dienste und Moduswechsel durch Priorisierungsdienste formalisiert.

Es ist dem Benutzer überlassen, ob er mit der Unterteilung des Gesamtverhaltens in Teilfunktionen oder Systemmodi beginnt. Für eine Teilfunktion können deren Modi oder umgekehrt in einem Systemmodus Teilfunktionen spezifiziert werden. Je nach Spezifika der Nutzerfunktionen kann eines der beiden Paradigmen eingesetzt werden.

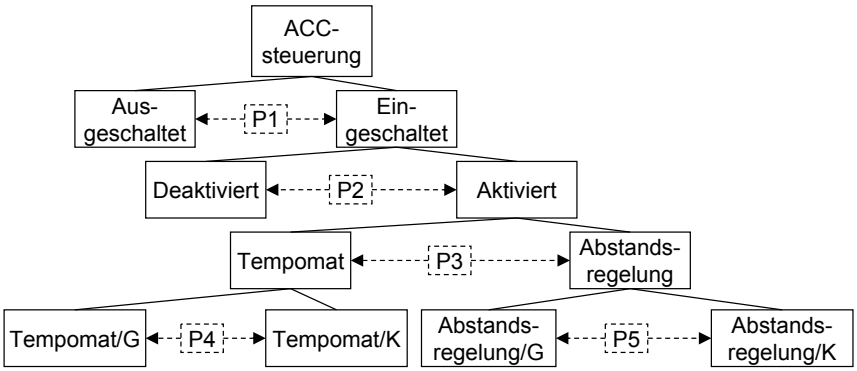


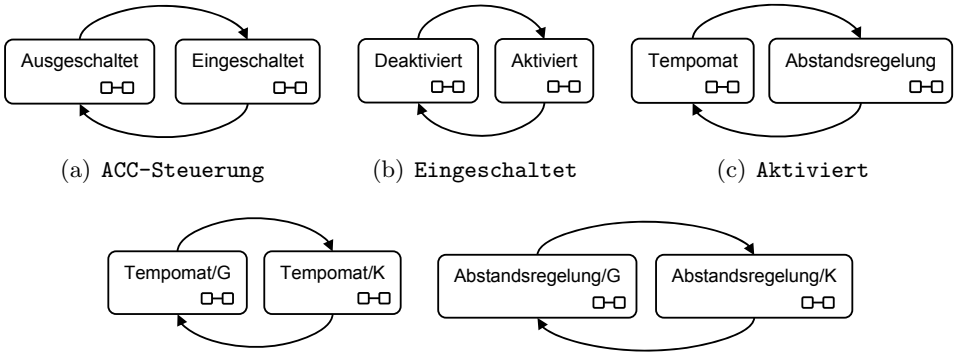
Abbildung 3.18.: ACC-Steuerung: Modus- und Funktionshierarchie

Beispiel 3.12 (Modus- und Funktionshierarchie)**.** Das Verhalten der ACC-Steuerung kann in drei Systemmodi unterteilt werden (die Nutzerfunktion *Stop&Go* wurde in diesem Beispiel weggelassen). Die vollständig ausgearbeitete Diensthierarchie der ACC-Steuerung ist in Anhang C.1 auf Seite 160 zu finden.

- *Ausgeschaltet*: Die ACC-Steuerung ist vom Fahrer ausgeschaltet.
- *Deaktiviert*: Die Steuerung ist eingeschaltet, kein vorausfahrendes Fahrzeug ist erkannt und die momentane Geschwindigkeit liegt unter 30 km/h.
- *Aktiviert*: Die Steuerung ist eingeschaltet und ein vorausfahrendes Fahrzeug ist erkannt bzw. die momentane Geschwindigkeit liegt über 30 km/h.

Diese Modi spiegeln sich in den obersten drei Ebenen der Diensthierarchie aus Abbildung 3.18 wider. In den Modi *Ausgeschaltet* und *Deaktiviert* verhalten sich die beiden Teilfunktionen *Tempomat* und *Abstandsregelung* identisch – eine weitere Zerlegung der beiden Dienste `Ausgeschaltet` und `Deaktiviert` ist nicht notwendig. Im Modus *Aktiviert* ist es sinnvoll, die beiden Teilfunktionen separat zu spezifizieren – der Dienst `Aktiviert` wird in weitere Teildienste unterteilt.

Dienste der untersten zwei Ebenen spezifizieren das „aktive“ Verhalten des Systems. Die Bedingungen wann das System aktiviert wird (Moduswechsel) und das Systemverhalten in einem nicht aktiven Modus, liegen nicht in der Verantwortung dieser Dienste. Durch diese modusbasierte Umstrukturierung ist die Spezifikation der Dienste einfacher als jene der ähnlichen Dienste aus Abbildung 3.16, die keine Modi berücksichtigen. Formal sind die Verhalten der beiden Spezifikationen identisch. □



(a) `ACC-Steuerung` (b) `Eingeschaltet` (c) `Aktiviert`

(d) `Tempomat` (e) `Abstandsregelung`

Abbildung 3.19.: Modi als interne Betriebszustände

***Bemerkung* 3.9** (Modi als interne Betriebszustände)**.** An Beispiel 3.12 ist zu sehen, dass eine Diensthierarchie als eine sequentielle Komposition interner Betriebszustände des Systems interpretiert werden kann. Das System geht – abhängig von seinen Eingaben – durch unterschiedliche Betriebszustände. In Abbildung 3.19 werden die Betriebszustände der ACC-Steuerung mit den herkömmlichen hierarchischen Zustandsdiagrammen skizziert. Die ACC-Steuerung wechselt ihren Betriebszustand von „Ausgeschaltet“ zu „Eingeschaltet“ (vgl. Abbildung 3.19(a)). Der Betriebszustand „Eingeschaltet“ ist weiter unterteilt in Betriebszustände „Deaktiviert“ und „Aktiviert“ (vgl. Abbildung 3.19(b)) und so weiter. □

3.4.4. Formale Anforderungsanalyse

Im Folgenden wird aufgezeigt, inwiefern der Entwicklungsprozess von einer formalen Anforderungsspezifikation profitieren kann.

Spezifikation als Kontrakt Eine der Aufgaben einer Anforderungsspezifikation liegt darin, Merkmale eines Systems zu definieren, mit denen die Dienstleistung des Auftragnehmers bei der Übergabe durch den Auftraggeber geprüft werden kann. Daher ist es wichtig, dass die Spezifikation sowohl vom Auftraggeber als auch vom Auftragnehmer gleich interpretiert wird. Das mathematische Modell aus Kapitel 4.3, das der dienstbasierten Spezifikation zugrunde liegt, macht die Spezifikation *eindeutig*.

Validierbarkeit Dank einer operationellen Semantik ist die dienstbasierte Spezifikation *simulierbar*. Noch bevor das System implementiert wird, kann es in einer Simulationsumgebung von Benutzern validiert werden. Die Simulation wird in Kapitel 7 erläutert.

Konsistenzüberprüfung Die Konsistenz einer dienstbasierten Spezifikation kann in vielen Fällen *automatisch* überprüft werden. Widersprüche sowie unerwünschte Wechselwirkungen (bekannt als *feature interactions*) zwischen Anforderungen können bereits in einer sehr frühen Phase des Entwicklungsprozesses entdeckt werden. Die Konsistenz einer Spezifikation wird in Kapitel 5 behandelt.

Modellbasierte Entwicklung Die dienstbasierte Spezifikation dient dem Übergang von funktionalen Anforderungen zum Design als Basis. Dieser Übergang ist ein Bestandteil einer formalen modellbasierten Software-Entwicklung. Dabei wird zwischen zwei Vorgehensweisen unterschieden. Bei einem *manuellen Übergang* erstellt der Entwickler eine Architektur auf Basis einer Spezifikation. Ein Model Checker kann überprüfen, ob das Systemmodell die formale Spezifikation erfüllt. Bei einem *automatischen Übergang* wird aus einer dienstbasierten Spezifikation automatisch eine komponentenbasierte Architektur generiert. Die Überprüfung der Korrektheit ist dabei überflüssig, wenn die Korrektheit der Transformation bewiesen ist. Der automatische Übergang wird in Kapitel 6 erläutert.

3.5. Workflow

In diesem Abschnitt werden einzelne Schritte der Anforderungsanalyse kurz erläutert. Dabei soll der im Folgenden aufgezeigte Arbeitsablauf nicht als eine verbindliche Vorschrift interpretiert werden. Dieser liegt nicht im Fokus der vorliegenden Arbeit. Es handelt sich eher um eine Erfolgsmethode (engl. *best practice*), die sich bei der Modellierung der beiden Fallstudien aus Anhang C bewährt hat.

1. *Systemgrenze.* Im ersten Schritt wird die Systemgrenze festgelegt. Stakeholder überlegen sich, was genau zu spezifizieren ist, welche Benutzer und benachbarten Systeme mit dem System interagieren können. Besonders wichtig ist in diesem Schritt die Entscheidung, ob das zu spezifizierende System als Ganzes aus der Black-Box-Sicht beschrieben oder zuerst in mehrere interagierende Teilsysteme zerlegt wird, die anschließend separat zu spezifizieren sind (vgl. die Matrix aus Abbildung 3.2).

2. *Syntaktische Schnittstelle.* Stakeholder definieren gemeinsam die syntaktische Schnittstelle des Systems – seine Ein- und Ausgabeports zusammen mit ihren Typen.
3. *Systemmodi.* Stakeholder überlegen sich Systemmodi und Moduswechsel. Die Beschreibung der Modi bleibt zuerst informell. Nur die Moduswechsel werden durch Priorisierungen formal spezifiziert. Das Ergebnis dieses Schrittes ist eine informelle Hierarchie von Systemmodi und eine Menge formal beschriebener Moduswechsel.
4. *Anwendungsfälle.* Jeder Stakeholder überlegt sich für jeden Systemmodus seine Anwendungsfälle. Bei der Beschreibung eines Anwendungsfalls müssen die anderen Fälle nicht berücksichtigt werden. Die Anwendungsfälle werden durch Dienste formalisiert. Anwendungsfälle, die sich geringfügig voneinander unterscheiden, können in einem Dienst zusammengefasst werden. Komplexe Anwendungsfälle können zuerst in kleinere Anwendungsfälle unterteilt und dann mittels Diensten formalisiert werden. Das Ergebnis ist eine Diensthierarchie für jeden einzelnen Systemmodus.
5. *Priorisierungen.* Die den Stakeholdern bereits bekannten Wechselwirkungen zwischen Anwendungsfällen werden durch Priorisierungsdienste formalisiert. Das Ergebnis dieses Schrittes ist eine dienstbasierte Spezifikation, die das komplette Systemverhalten auf den oberen Ebenen der Diensthierarchie in Systemmodi und auf den unteren Ebenen in Anwendungsfälle unterteilt.
6. *Konsistenz.* Die Konsistenz der Spezifikation wird automatisch überprüft, um Widersprüche zwischen Teilspezifikationen zu identifizieren. Der Anforderungsanalytiker versucht, die Widersprüche mittels zusätzlichen Priorisierungen und ohne Modifizierungen der ursprünglichen Dienste zu eliminieren. Das Ergebnis ist eine konsistente Spezifikation.
7. *Iterationen.* Schritte 3 bis 6 können beliebig oft wiederholt werden bis das Systemverhalten vollständig spezifiziert ist. Kommen weitere Systemmodi oder Anwendungsfälle hinzu, muss das bereits existierende Dienstmodell nicht modifiziert werden – neue Wechselwirkungen werden mittels neuer Priorisierungen definiert.

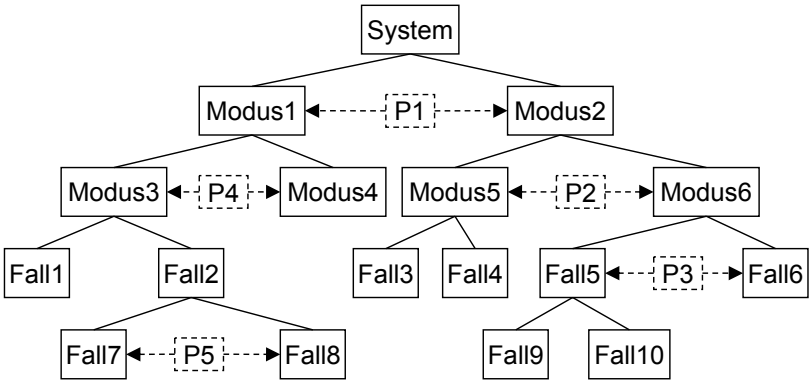


Abbildung 3.20.: Modi und Anwendungsfälle in einer Diensthierarchie

Das Ergebnis der Anforderungsanalyse ist eine vollständige und konsistente dienstbasierte Spezifikation wie in Abbildung 3.20 schematisch dargestellt. Dabei wird das Systemverhalten zuerst in Systemmodi und dann (falls nötig) in Anwendungsfälle unterteilt. Nur die Blattdienste und Priorisierungen werden mit Verhaltensspezifikationen ausformuliert, das Verhalten der kombinierten Dienste ergibt sich automatisch.

***Bemerkung* 3.10** (Nutzerfunktionen vs. Systemmodi)**.** Wie bereits in Abschnitt 3.4.3 erwähnt, gibt es keinen formalen Unterschied zwischen Nutzerfunktionen und Systemmodi. Modi sind lediglich eine in den beiden Fallstudien aus Anhang C bewährte Vorgehensweise zur Strukturierung von Nutzerfunktionen. Sowohl Funktionen als auch Modi werden durch Dienste modelliert. Moduswechsel werden durch Priorisierungsdienste formalisiert. □

3.6. Zusammenfassung

In diesem Kapitel wurden zwei orthogonale Arten der Systembeschreibung eingeführt sowie ihre Zusammenhänge und Unterschiede erläutert. Die strukturelle Dekomposition unterteilt das Verhalten in ein Netzwerk kollaborierender Komponenten und ist für die Beschreibung von Software-Architekturen gut geeignet. Die funktionale Dekombination strukturiert das Verhalten in eine Hierarchie von Interaktionsmustern und abstrahiert von jeder Implementierungsstruktur. Diese Art der Beschreibung ist besser für die Anforderungsanalyse geeignet.

Das Hauptaugenmerk dieses Kapitels lag auf der dienstbasierten Spezifikation funktionaler Anforderungen. Sie besteht aus folgenden Bestandteilen:

- Atomare Dienste, die einzelne Interaktionsmuster formalisieren,
- Kombinierte Dienste, die aus mehreren Teildiensten bestehen, und
- Kombinierte Dienste mit Priorisierung, die aus mehreren Teildiensten bestehen und Teildienste priorisieren können.

Die dienstbasierte Spezifikation hat folgende Merkmale. Sie beschreibt das Systemverhalten aus unterschiedlichen Benutzerperspektiven in Form von Interaktionsmustern. Das Verhalten kann sowohl modus- als auch funktionsorientiert hierarchisch strukturiert werden. Zwischen den Begriffen „Modus“ und „Nutzerfunktion“ gibt es keinen formalen Unterschied – Modi sind lediglich ein methodologisches Mittel zur übergeordneten Strukturierung von Funktionen. Der Spezifikation liegt eine formale operationelle Semantik zugrunde.

KAPITEL 4

Formales Framework

In diesem Kapitel wird die operationelle Semantik der dienstbasierten Spezifikationstechnik eingeführt. Dabei liegt der Fokus der Betrachtung auf einer zustandsbasierten Semantik, die eine automatische Anforderungsanalyse ermöglicht. Der wichtigste Beitrag dieses Kapitels sind die formalen Definitionen der beiden in Abschnitt 3.3 eingeführten Kombinationsoperatoren und der Nachweis ihrer algebraischen Eigenschaften.

Für einen formalen und lückenlosen Übergang von Anforderungen zum Design ist es außerdem wichtig, den vorgestellten Ansatz in eine umfassende Modellierungstheorie zur Beschreibung von Spezifikationen und Architekturen zu integrieren. Zu diesem Zweck wurde die FOCUS-Theorie von Broy [Bro07a, HHR09] gewählt. In diesem Kapitel wird die FOCUS-basierte Semantik der dienstbasierten Spezifikation eingeführt, während die Integration des Ansatzes in die FOCUS-Theorie in Kapitel 5 formal nachgewiesen wird.

Inhalt

Der dienstbasierte Ansatz wurde bisher informell eingeführt. Für unser Ziel – eine formale Anforderungsanalyse – ist es jedoch notwendig, die der Spezifikationstechnik zugrunde liegende Semantik formal zu definieren. Sowohl das Verhalten eines Dienstes als auch die beiden Kombinationsoperatoren müssen mathematisch fundiert sein.

Die operationelle Semantik des dienstbasierten Ansatzes wird in diesem Kapitel vorgestellt. Das Verhalten eines Dienstes wird durch ein Transitionssystem definiert. Anschließend wird gezeigt, wie zwei Transitionssysteme zu einem größeren kombiniert werden (mit und ohne Berücksichtigung einer Priorisierung zwischen diesen). Weiterhin werden die wichtigsten algebraischen Eigenschaften der beiden Operatoren nachgewiesen. Diese Eigenschaften, wie z.B. Kommutativität oder Assoziativität, sind bei der Modellierung einer Spezifikation von Bedeutung. Beispielsweise darf sich die Reihenfolge der Kombination von Diensten nicht auf das resultierende Gesamtverhalten auswirken. Gleichermaßen muss untersucht werden, ob die hierarchische Strukturierung eine semantische Bedeutung hat oder nur zur besseren Darstellung beiträgt.

Die Idee von Systemmodi wurde in Abschnitt 3.4.3 bereits erläutert. In der vorliegenden Arbeit werden Systemmodi vor allem bei der Korrektheitsprüfung verwendet, bei der die Erfüllung von Systemeigenschaften überprüft wird. Dabei kann es vorkommen, dass Eigenschaften nicht in allen Systemabläufen, sondern nur in ausgewählten Modi gelten müssen. Zu diesem Zweck werden in diesem Kapitel Systemmodi formal definiert.

Der vorgestellte Ansatz steht in der Vielzahl von Modellierungstechniken nicht isoliert da. Er beruht auf der umfassenden Modellierungstheorie für Anforderungsspezifikationen und Architekturentwurf multifunktionaler Software-Systeme, eingeführt von Broy [Bro07a, HHR09]. In dieser Theorie ist ein Dienst durch eine Relation zwischen Ein- und Ausgabeströmen gegeben. Beziehungen zwischen Diensten, wie z.B. Verfeinerung und Teildienst-Relation, sind auf Strömen definiert. Das Gesamtverhalten eines Systems wird durch eine formal fundierte Diensthierarchie (engl. Annotated Service Hierarchy) gegeben. Die beiden Ansätze unterscheiden sich im Wesentlichen in zwei Punkten:

- Während Broy eine denotationale Semantik definiert, wird mit der vorliegenden Arbeit eine operationelle Semantik der Diensthierarchie eingeführt. Die Letztere ist für eine automatische Analyse von Anforderungen notwendig.
- Broy schlägt einen deskriptiven Ansatz vor. Sein Ziel ist der Nachweis einer Beziehung zwischen zwei gegebenen Diensten. Im Gegensatz dazu ist das Ziel des vorgestellten Ansatzes, aus zwei gegebenen einen kombinierten Dienst zu konstruieren – der Ansatz ist konstruktiv.

Im Gegensatz zur *Systemtheorie* von Broy, die unterschiedliche Phänomene des Systems erklärt, wird in diesem Kapitel ein *Engineeringmodell* vorgeschlagen, das den Entwickler bei der Erstellung einer Anforderungsspezifikation unterstützt. In Kapitel 5 wird nachgewiesen, dass der vorgestellte Ansatz in der Broyschen Theorie vollständig integriert ist, d.h. dass die dienstbasierte Spezifikation aus Abschnitt 3.3 eine Instanz der annotierten Diensthierarchie von Broy ist.

Der folgende Abschnitt definiert die mathematischen Grundlagen für den Rest des Kapitels. In Abschnitten 4.2 und 4.3 werden die denotationale und operationelle Semantik des dienstbasierten Ansatzes eingeführt. In Abschnitt 4.4 werden die wichtigsten algebraischen Eigenschaften der beiden Kombinationsoperatoren nachgewiesen. In Abschnitt 4.5 werden Systemmodi formal definiert.

4.1. Grundlagen

Dieser Abschnitt definiert die mathematischen Grundlagen für den Rest des vorliegenden Kapitels. Die operationelle Semantik wird durch Transitionssysteme definiert. Die Zustände dieser Systeme sind durch Belegungen von Variablen gegeben. Die Mengen dieser Zustände werden durch prädikatenlogische Ausdrücke über diesen Variablen definiert. Die Definitionen von Belegungsfunktionen und Transitionssystemen sind an [BP99, PP07, Bot08] angelehnt.

4.1.1. Belegungsfunktionen

Zuerst wird der Begriff einer Variable eingeführt. Dafür werden der Typ einer Variable sowie die *subtype*-Relation zwischen Variablen definiert.

Definition 4.1 (Name und Typ von Variablen)**.** Seien VAR das Universum aller ungestrichelten und VAR' aller gestrichelten Variablenamen. Dabei gilt $\mathsf{VAR} \cap \mathsf{VAR}' = \emptyset$.

Seien M das Universum aller Nachrichten (oder Werte) und M^* die Menge aller endlichen Nachrichtensequenzen. Der *Typ* einer Variable (oder Variablendomäne) ist als eine Teilmenge von M^* definiert. Die Funktion $type(v) \in \wp(M^*)$ bildet jede Variable $v \in \mathsf{VAR} \cup \mathsf{VAR}'$ auf ihren Typ ab.

Für zwei Mengen von Variablen V_1, V_2 ist die *subtype*-Relation wie folgt definiert:

$$V_1 \ \texttt{subtype} \ V_2 \stackrel{\text{def}}{\Leftrightarrow} \forall v_1 \in V_1 : \exists v_2 \in V_2 : type(v_1) \subseteq type(v_2).$$

□

Im Folgenden wird eine Funktion definiert, die Variablen auf ihre Belegungen abbildet.

Definition 4.2 (Belegungsfunktion)**.** Für eine gegebene Menge von Variablen $V \subseteq \mathsf{VAR} \cup \mathsf{VAR}'$ ordnet eine *Belegungsfunktion* $\alpha \colon V \to M^*$ jeder Variable aus V eine *Belegung* vom Typ der jeweiligen Variable zu.

Eine Belegungsfunktion α für eine Variablemenge V ist *Typ-korrekt*, wenn folgende Aussage gilt:

$$\forall v \in V : \alpha(v) \in type(v).$$

Die Menge $\Lambda(V)$ umfasst alle Typ-korrekten Belegungsfunktionen für die Variablemenge V. □

Folgende Definition führt ein Prädikat auf Belegungsfunktionen ein, das in etwa dem Gleichheitsprädikat entspricht.

Definition 4.3 (Übereinstimmung zwischen Belegungsfunktionen)**.** Für zwei Mengen von Variablen $V, W \subseteq \mathsf{VAR} \cup \mathsf{VAR}'$ *stimmen* zwei Belegungsfunktionen $\alpha \in \Lambda(V)$ und $\beta \in \Lambda(W)$ in einer Schnittmenge U `subtype` $(V \cap W)$ *überein*, falls die beiden Funktionen α und β jeder Variable aus U dieselbe Belegung zuordnen:

$$\alpha \overset{U}{=} \beta \overset{\text{def}}{\Leftrightarrow} \forall v \in U : \alpha(v) = \beta(v).$$

Ein ähnliches Prädikat ist für zwei Mengen von Belegungsfunktionen definiert:

$$\mathcal{A} \overset{U}{=} \mathcal{B} \overset{\text{def}}{\Leftrightarrow} (\forall \alpha \in \mathcal{A} : \exists \beta \in \mathcal{B} : \alpha \overset{U}{=} \beta) \wedge (\forall \beta \in \mathcal{B} : \exists \alpha \in \mathcal{A} : \alpha \overset{U}{=} \beta).$$

□

Um bei der Definition eines Transitionssystems die Belegungen vor bzw. nach der Ausführung einer Transition differenzieren zu können, wird eine Funktion eingeführt, welche ungestrichelte auf gestrichelte Variablen abbildet.

Definition 4.4 (Priming)**.** Die *Priming*-Funktion bildet eine Variable $v \in \mathsf{VAR}$ auf eine typgleiche Variable $v' \in \mathsf{VAR}'$ ab. Dasselbe gilt für eine Menge von Variablen $V \subseteq \mathsf{VAR}$, die auf die Menge $V' \subseteq \mathsf{VAR}'$ mit $V' \overset{\text{def}}{=} \{v' \mid v \in V\}$ abgebildet wird.

Für eine Belegungsfunktion α über V steht α' für die entsprechende Belegungsfunktion über V', so dass folgende Aussage gilt:

$$\forall v \in V : \alpha(v) = \alpha'(v').$$

□

Wie bereits erwähnt, werden in einem Transitionssystem Mengen von Transitionen durch prädikatenlogische Ausdrücke auf seinen Variablen definiert. Im Folgenden wird eine logische Aussage auf einer Menge von Variablen sowie die Bedingung für die Erfüllung einer Aussage von einer Belegungsfunktion definiert.

Definition 4.5 (Logische Aussage)**.** Eine *logische Aussage* Φ ist ein Prädikat auf einer Menge freier Variablen: $\text{free.}\Phi \subseteq \mathsf{VAR} \cup \mathsf{VAR}'$.

Eine Belegungsfunktion α *erfüllt* eine logische Aussage Φ (bezeichnet durch $\alpha \vdash \Phi$) genau dann, wenn Φ nach dem Ersetzen aller freien Variablen $v \in \text{free.}\Phi$ durch ihre Belegungen $\alpha(v)$ zu wahr evaluiert:

$$\alpha \vdash \Phi \overset{\text{def}}{\Leftrightarrow} \Phi[\alpha(v)/v]_{v \in \text{free.}\Phi}.$$

Für eine logische Aussage Φ mit freien Variablen aus $V \cup V'$ und zwei Belegungsfunktionen $\alpha \in \Lambda(V)$ und $\beta' \in \Lambda(V')$ ist die Erfüllung der Aussage wie folgt definiert:

$$\alpha, \beta' \vdash \Phi \overset{\text{def}}{\Leftrightarrow} \Phi[\alpha(v)/v, \beta'(v')/v']_{v,v' \in \text{free.}\Phi}.$$

Alternativ schreibt man $\delta \vdash \Phi$ für $\delta \in \Lambda(V \cup V')$. □

4.1.2. Transitionssysteme

Ein Transitionssystem (engl. state transition system) ist gegeben durch ein Tupel

$$\mathcal{S} = (V, \mathcal{I}, \mathcal{T}). \tag{4.1}$$

Die Variablenmenge eines Transitionssystems V ist die Vereinigung der paarweise disjunkten Mengen typisierter Input-, Output- und lokaler Variablen: $V \stackrel{\text{def}}{=} I \uplus O \uplus L$ mit $V \subseteq \mathsf{VAR}$. Ein *Zustand* von $\mathcal{S}$ ist gegeben durch eine Belegungsfunktion $\alpha \in \Lambda(V)$. $\mathcal{I}$ definiert die Menge der *Initialzustände* des Systems. $\mathcal{T}$ beschreibt die Menge von *Zustandsübergängen* (oder Transitionen) von $\mathcal{S}$. Im Folgenden werden einzelne Elemente des Tupels $\mathcal{S}$ sowie Funktionen und Prädikate über $\mathcal{S}$ im Detail erläutert.

$\mathcal{I}$ ist eine logische Aussage über $L \cup O$, d.h. die Variablen aus I sind vom Prädikat nicht eingeschränkt – sie dürfen in $\mathcal{I}$ nicht vorkommen:

$$\forall \alpha, \beta \in \Lambda(V) : \alpha \overset{L \cup O}{=} \beta \Rightarrow \big(\alpha \vdash \mathcal{I} \Leftrightarrow \beta \vdash \mathcal{I}\big). \tag{4.2}$$

$\mathcal{T}$ ist eine Menge logischer Aussagen (in diesem Kontext auch Transitionen genannt), die auf einer Teilmenge von $V \cup V'$ definiert sind, d.h. für jede Transition $t \in \mathcal{T}$ gilt: $\text{free}.t \subseteq V \cup V'$. In einer Transition beschreiben ungestrichelte Variablen die Belegung vor und gestrichelte Variablen die Belegung nach der Ausführung der Transition. Eine Transition darf keine gestrichelten Variablen aus I und keine ungestrichelten Variablen aus O einschränken. Mit anderen Worten, eine Transition kann nur ihre Output- und lokale Variablen modifizieren. Für alle $t \in \mathcal{T}$ gilt:

$$\forall \alpha, \beta \in \Lambda(V \cup V') : \big(\alpha \overset{I \cup L}{=} \beta \wedge \alpha \overset{L' \cup O'}{=} \beta\big) \Rightarrow \big((\alpha \vdash t) \Leftrightarrow (\beta \vdash t)\big). \tag{4.3}$$

Dadurch verbieten wir einem System seine zukünftige Eingaben zu beeinflussen und erzwingen eine klare Trennung zwischen Eingaben, lokalen Variablen und Ausgaben. In diesem Kontext heißen Elemente aus I *externe* und Elemente aus $L \cup O$ *kontrollierte* Variablen (vgl. Definitionen von Parnas et al. [PM95]).

Eine Transition t kann mehrere Nachfolgezustände definieren, d.h. mehrere Belegungsfunktionen erfüllen die logische Formel t. Dadurch können *nichtdeterministische* Systeme modelliert werden.

Beispiel 4.1 (Transitionssystem des Dienstes `Abstand/G`)**.** Das Verhalten des Dienstes `Abstand/G` aus Abbildung 3.11(b) auf Seite 36 wird durch das Transitionssystem $\mathcal{S} = (V, \mathcal{I}, \mathcal{T})$ wie folgt definiert.

$V = I \uplus O \uplus L$ mit $I = \{rDist, cDist, cSpeed\}$, $L = \{z\}$ und $O = \{mInstr, bInstr\}$, wobei $type(rDist) = type(cSpeed) = type(mInstr) = type(bInstr) = type(z) = \mathbb{N} \cup \{\varepsilon\}$ und $type(cDist) = \mathbb{N} \cup \{\infty, \varepsilon\}$.

$\mathcal{I} = (z = 1 \wedge mInstr = \varepsilon \wedge bInstr = \varepsilon)$ definiert den Initialzustand von $\mathcal{S}$. Jede Belegung $\alpha \in \Lambda(V)$, die diese logische Aussage erfüllt, d.h. $\alpha(z) = 1 \wedge \alpha(mInstr) = \varepsilon \wedge \alpha(bInstr) = \varepsilon$, ist eine Initialbelegung von $\mathcal{S}$.

$\mathcal{T} = \{(z = 1 \wedge 30 \leq cSpeed \leq 180 \wedge cDistance \neq \infty \wedge z' = 1 \wedge mInstr' = l(rDist, cDist) \wedge bInstr' = m(rDist, cDist))\}$ definiert die einzige Transition von $\mathcal{S}$. Intuitiv gesprochen definiert diese Transition den Übergang von einer Belegung $\alpha \in \Lambda(V)$ mit $\alpha(z) = 1 \wedge 30 \leq \alpha(cSpeed) \leq 180 \wedge \alpha(cDistance) \neq \infty$ zu einer Belegung $\beta \in \Lambda(V)$ mit $\beta(z) = 1 \wedge \beta(mInstr) = l(\alpha(rDist), \alpha(cDist)) \wedge \beta(bInstr) = m(\alpha(rDist), \alpha(cDist))$. Angelehnt an die formale Definition von Transitionssystemen ist das Prädikat wie folgt zu interpretieren: Die Transition evaluiert für solche Belegungen α und β zu wahr, für die gilt, dass die Belegung α der Variable z einen Wert von 1, der Variable *cSpeed* einen Wert zwischen 30 und 180 und der Variable *cDistance* einen Wert ungleich unendlich zuweist (β ist analog definiert).

Das Diagramm für dieses Transitionssystem ist in Abbildung 3.11(b) auf Seite 36 gegeben. □

Um den Zusammenhang zwischen Transitionen und Belegungsfunktionen herzustellen, wird die Menge von Belegungsfunktionen $\mathrm{Succ}(\alpha)$ definiert, welche von einer Belegungsfunktion $\alpha \in \Lambda(V)$ aus durch die Transitionen aus $\mathcal{T}$ erreichbar sind:

$$\mathrm{Succ}(\alpha) \stackrel{\text{def}}{=} \{\beta \mid \exists t \in \mathcal{T} : \alpha, \beta' \vdash t\}. \tag{4.4}$$

Für eine Menge von Belegungsfunktionen $\mathcal{A} \subseteq \Lambda(V)$ ist die Menge ihrer Nachfolger wie folgt definiert:

$$\mathrm{Succ}(\mathcal{A}) \stackrel{\text{def}}{=} \bigcup_{\alpha \in \mathcal{A}} \mathrm{Succ}(\alpha).$$

Das System ist für eine Belegung α *ausführbar* (bezeichnet durch $\mathrm{En}(\alpha)$), wenn es für diese Belegung mindestens eine nachfolgende Belegung gibt:

$$\mathrm{En}(\alpha) \stackrel{\text{def}}{\Leftrightarrow} \mathrm{Succ}(\alpha) \neq \emptyset. \tag{4.5}$$

Aus dieser Definition folgt, dass die Transitionssysteme *input-disabled* definiert sind. Es ist also nicht gefordert, dass alle Belegungen eines Transitionssystems eine nachfolgende Belegung haben müssen.

Ein Transitionssystem erzeugt eine Menge von Berechnungssequenzen. Jede Berechnungssequenz ist eine endliche oder unendliche Folge von Zuständen. Sie werden auch Spuren (engl. traces) oder Abläufen (engl. runs) genannt. Die Sprache eines Transitionssystems $\mathcal{S}$ ist durch die Menge aller seiner Abläufe gegeben.

Definition 4.6 (Ablauf)**.** Ein *Ablauf* eines Systems ist eine endliche oder unendliche Sequenz von Belegungsfunktionen $\rho \stackrel{\text{def}}{=} \langle \alpha_0 \alpha_1 \alpha_2 \ldots \rangle$, wobei folgende Aussagen gelten:

$$\alpha_0 \vdash \mathcal{I} \qquad \text{und} \qquad \forall i \in \mathbb{N} : \alpha_{i+1} \in \mathrm{Succ}(\alpha_i).$$

$\langle\langle \mathcal{S} \rangle\rangle$ bezeichnet die Menge aller Systemabläufe oder die Sprache über $\mathcal{S}$. Die Menge $\langle\langle \mathcal{S} \rangle\rangle$ ist *prefix-closed*, d.h. sie umfasst nicht nur unendliche Abläufe, sondern auch ihre endlichen Präfixe.

Durch $\rho.i$ wird das i-te Element eines Systemablaufs ρ bezeichnet, d.h. $\rho.i \stackrel{\text{def}}{=} \alpha_i$. Durch $\rho(v)$ wird die Projektion eines Systemablaufs auf eine Variable $v \in V$ bezeichnet. □

Man beachte, die Sprache über $\mathcal{S}$ kann leer sein, wenn z.B. keine Belegungsfunktion das Prädikat $\mathcal{I}$ erfüllt.

Im Folgenden werden zwei Relationen auf Systemabläufen definiert. Seien zwei Systemabläufe ρ_1 und ρ_2 gegeben, welche auf Belegungsfunktionen aus $\Lambda(V_1)$ und $\Lambda(V_2)$ entsprechend definiert sind. Die Gleichheit der beiden Abläufe in einer Variablenmenge U `subtype` $(V_1 \cap V_2)$ ist wie folgt definiert:

$$\rho_1 \overset{U}{=} \rho_2 \overset{\text{def}}{\Leftrightarrow} \forall t \in \mathbb{N} : \rho_1.t \overset{U}{=} \rho_2.t. \tag{4.6}$$

Die Teilmenge-Relation zwischen zwei Mengen von Systemabläufen $\mathcal{R}_1$ und $\mathcal{R}_2$ in einer Variablemenge U ist wie folgt definiert:

$$\mathcal{R}_1 \overset{U}{\subseteq} \mathcal{R}_2 \overset{\text{def}}{\Leftrightarrow} \forall \rho_1 \in \mathcal{R}_1 : \exists \rho_2 \in \mathcal{R}_2 : \rho_1 \overset{U}{=} \rho_2. \tag{4.7}$$

4.1.3. Zustandsdiagramme

Die Zustandsdiagramme, die sowohl in den begleitenden Beispielen in Kapitel 3 als auch im CASE Tool (vgl. Kapitel 7) verwendet werden, sind graphische Darstellungen der oben eingeführten Transitionssysteme. Diese Notationstechnik ist bereits in Abschnitt 3.3.1 auf Seite 30 erläutert worden (vgl. z.B. Abbildung 3.8(b)). Die beiden Notationstechniken unterschieden sich in zwei Punkten:

- Im Gegensatz zu Transitionssystemen haben Zustandsdiagramme eine lokale Variable, die den Kontrollzustand des Zustandsdiagramms speichert;
- Ein Schritt eines Transitionssystems wird durch eine Succ-Relation über Belegungsfunktionen definiert, ohne dabei die konkrete Syntax von Transitionen anzugeben. Ein Zustandsdiagramm besteht aus Kontrollzuständen und konkreten Transitionen zwischen diesen.

Für eine ausführliche Beschreibung dieser Version von Zustandsdiagrammen sei auf [SPHP02] verwiesen.

4.2. Denotationale Semantik der dienstbasierten Spezifikation

Nachdem alle notwendigen mathematischen Grundlagen eingeführt sind, wird in diesem Abschnitt die denotationale Semantik des dienstbasierten Ansatzes definiert. Bei dieser Semantik handelt es sich um die FOCUS-Theorie für Anforderungsspezifikationen multifunktionaler Software-Systeme. In FOCUS wird das Schnittstellenverhalten eines Systems durch Relationen zwischen Ein- und Ausgabeströmen definiert. Da diese Semantik nicht im Fokus der vorliegenden Arbeit steht, wird sie im Folgenden nur kurz eingeführt. Für eine umfangreiche Erläuterung dieser Theorie sei der Leser auf [BS01, Bro05c, HHR09] verwiesen.

Der folgende Abschnitt führt das Basiselement der Theorie, den Nachrichtenstrom ein. Anschließend werden die Begriffe Dienst und Diensthierarchie definiert.

4.2.1. Nachrichtenströme

Der FOCUS-Theorie liegt die Idee gezeiteter Nachrichtenströme zugrunde. Durch diese Ströme wird die asynchrone Interaktion des Systems mit seiner Umgebung beschrieben. Intuitiv gesprochen, versteht man unter einem gezeiteten Nachrichtenstrom eine chronologisch geordnete Sequenz von Nachrichten.

Definition 4.7 (Gezeiteter Nachrichtenstrom)**.** Ein Strom von Nachrichten aus der Menge M ist eine unendliche Folge von Nachrichtensequenzen:

$$s : \mathbb{N}_+ \to M^*.$$

$\mathbb{N}_+$ bezeichnet hier die positiven ganzen Zahlen und M^* die Menge endlicher Sequenzen über M. Für jedes Zeitintervall $t \in \mathbb{N}_+$ bezeichnet $s(t)$ diejenige Sequenz von Nachrichten, die im Zeitintervall t übertragen wird[1].

Ein *Präfix* eines Nachrichtenstroms s wird durch $s \downarrow t$ bezeichnet und umfasst die ersten t Nachrichtensequenzen des Stroms.

Der *Filteroperator* $S©s$ filtert all diejenigen Nachrichten aus dem Nachrichtenstrom s aus, welche nicht in der Menge S sind. □

Aus der obigen Definition ist offensichtlich, dass der FOCUS-Theorie ein diskretes Zeitmodell zugrunde liegt. Die Zeit ist in eine unendliche Folge von Zeitintervallen gleicher Länge unterteilt.

4.2.2. Formale Darstellung eines Dienstes

Ein Dienst wird durch eine syntaktische Schnittstelle und ein Schnittstellenverhalten spezifiziert.

Syntaktische Schnittstelle Die Kommunikation eines Dienstes mit seiner Umgebung erfolgt ausschließlich durch seine syntaktische Schnittstelle. Die Schnittstelle besteht aus einer Menge I von Eingabeports und einer Menge O von Ausgabeports und wird durch den Ausdruck $(I \blacktriangleright O)$ bezeichnet. Für jeden Port $p \in I \cup O$ ist sein Datentyp $type(p)$ festgelegt, der definiert, welche Nachrichten durch den Port gesendet/empfangen werden können (vgl. Definition 4.1 auf Seite 55).

Das Verhalten eines Dienstes wird durch Kommunikationshistorien seiner Ports definiert. Diese werden wiederum durch Nachrichtenströme ausgedrückt.

[1] In der vorliegenden Arbeit werden nur Nachrichtensequenzen der Länge 1 betrachtet, d.h. $s : \mathbb{N}_+ \to M$.

Definition 4.8 (Porthistorie)**.** Sei P eine Menge typisierter Ports. Dann ordnet eine *Porthistorie* h von der Form

$$h: \ P \to (\mathbb{N}_+ \to M^*)$$

jedem Port $p \in P$ einen Nachrichtenstrom $h(p)$ zu.

$h(p)$ bezeichnet einen Nachrichtenstrom für den Port $p \in P$ mit $type(p) = M$.

Die Menge aller Historien für die Portmenge P wird durch $\mathbb{H}(P)$ bezeichnet.

$\mathbb{H}(I) \times \mathbb{H}(O)$ bezeichnet die Menge aller *syntaktisch* korrekten Historienpaare (x, y) für einen Dienst mit der Schnittstelle $(I \blacktriangleright O)$. □

Schnittstellenverhalten Das Schnittstellenverhalten eines Dienstes mit der Schnittstelle $(I \blacktriangleright O)$ ist durch eine Funktion der folgenden Form gegeben:

$$F: \ \mathbb{H}(I) \to \mathcal{P}(\mathbb{H}(O)).$$

Bei der Funktion F wird gefordert, dass sie folgende Eigenschaft erfüllt, die ein angemessenes Verhalten in Hinblick auf den Zeitfluss sicherstellt. Für alle Historien $x, z \in \mathbb{H}(I)$ und Zeitintervalle $t \in \mathbb{N}_+$ gilt:

$$F(x) \neq \emptyset \neq F(z) \wedge x \downarrow t = z \downarrow t \Rightarrow \{y \downarrow (t+1) : y \in F(x)\} = \{y \downarrow (t+1) : y \in F(z)\}.$$

Diese Eigenschaft heißt *starke Kausalität* und stellt sicher, dass im Modell ein konsistenter Zeitfluss gegeben ist. Die möglichen Ausgabehistorien von F hängen in den ersten $(t+1)$ Zeitintervallen von den Eingabehistorien der ersten t Intervalle ab. Das heißt, eine Funktion braucht für die Bearbeitung von Eingaben mindestens ein Zeitintervall. Diese Eigenschaft gilt nur für die Eingabeströme, für die die Ausgabemenge nicht leer ist. Gemäß der obigen Definition ist ein Dienst *partiell*, d.h. die Ausgaben eines Dienstes sind nur für eine Teilmenge von Eingaben definiert. Diese Teilmenge heißt der *Definitionsbereich* eines Dienstes und ist wie folgt definiert:

$$dom(F) \stackrel{\text{def}}{=} \{x \in \mathbb{H}(I) \mid F(x) \neq \emptyset\}.$$

Für eine Teilmenge von Historien $x \in \mathbb{H}(I)$ der Eingabeports werden die möglichen Historien $y \in \mathbb{H}(O)$ der Ausgabeports durch die Funktion $F(x)$ festgelegt. Ist $F(x)$ für alle $x \in dom(F)$ stets einelementig, so heißt F *deterministisch.*

4.2.3. Diensthierarchie

In diesem Abschnitt wird das Gesamtverhalten eines Systems, gegeben durch den Dienst F mit der Schnittstelle $(I \blacktriangleright O)$, in eine Hierarchie von Teildiensten strukturiert. Dabei wird angenommen, dass die einzelnen Dienste dieser Hierarchie nur auf eine Teilmenge von Ports und Nachrichten aus I und O zugreifen.

Für die Definition von Beziehungen zwischen Diensten werden folgende Hilfsrelationen benötigt.

Definition 4.9 (Historienprojektion)**.** Für eine Porthistorie $h: P \to (\mathbb{N}_+ \to M^*)$ ist ihre *Historienprojektion* $h|R$ auf eine Menge R mit R `subtype` P wie folgt definiert:

$$h|R : R \to (\mathbb{N}_+ \to M^*) \text{ mit}$$
$$\forall p \in R : h|R\ (p) \stackrel{\text{def}}{=} type(p) \copyright h(p),$$

wobei sich $type(p)$ auf den Typ der Variable p aus R bezieht.

Für eine Funktion $F: \mathbb{H}(I) \to \mathcal{P}(\mathbb{H}(O))$ ist ihre *Historienprojektion* $F|R$ auf die Menge R wie folgt definiert:

$$F|R: \{h|R \mid h \in \mathbb{H}(I)\} \to \mathcal{P}(\{h|R \mid h \in \mathbb{H}(O)\})$$

□

Bei der Projektion einer Porthistorie auf eine Portmenge R werden aus der Historie all diejenigen Ports gelöscht, die nicht in R sind. Außerdem werden aus den Strömen all diejenigen Nachrichten gelöscht, die nicht zu den verbleibenden Porttypen gehören.

Definition 4.10 (Verhaltensprojektion)**.** Für einen Dienst F mit der Schnittstelle $(I \blacktriangleright O)$ ist seine *Verhaltensprojektion* auf eine Schnittstelle $(I_1 \blacktriangleright O_1)$ mit I_1 `subtype` I und O_1 `subtype` O wie folgt definiert:

$$F\dagger(I_1 \blacktriangleright O_1)(x) \stackrel{\text{def}}{=} \{y|O_1 \mid \exists x' \in \mathbb{H}(I) : x = x'|I_1 \wedge y \in F(x')\}.$$

□

In einer Verhaltensprojektion werden bestimmte Ein- und Ausgabenachrichten eliminiert. Dabei kann es vorkommen, dass in einer Projektion die verbliebenen Ausgabenachrichten von einigen eliminierten Eingabenachrichten abhängen. In diesem Fall definiert die Projektion $F\dagger(I_1 \blacktriangleright O_1)$ mehr Verhalten als F.

Nun werden zwei für die Definition der Diensthierarchie wichtige Beziehungen zwischen Diensten eingeführt, die Verfeinerung und die Teildienst-Relation.

Ein Dienst kann durch Erweiterung seines Definitionsbereichs (angenommen das Verhalten ist nicht total) oder durch Einschränkung seines Nichtdeterminismus verfeinert werden.

Definition 4.11 (Dienstverfeinerung)**.** Ein Dienst F_2 mit der Schnittstelle $(I_2 \blacktriangleright O_2)$ ist eine *Verfeinerung* des Dienstes F_1 mit der Schnittstelle $(I_1 \blacktriangleright O_1)$ genau dann, wenn folgende Aussagen gelten: I_1 `subtype` I_2, O_1 `subtype` O_2 und

$$\forall x \in \mathbb{H}(I_1) : F_2\dagger(I_1 \blacktriangleright O_1)(x) \subseteq F_1(x).$$

In diesem Fall schreibt man $F_1 \succeq F_2$ (gelesen F_1 wird durch F_2 verfeinert). □

Ein Dienst kann aus mehreren Teildiensten bestehen. Die Beziehung zwischen einem gegebenen Dienst und einem seiner Teildienste ist wie folgt definiert.

Definition 4.12 (Teildienst-Relation)**.** Ein Dienst F_1 mit der Schnittstelle $(I_1 \blacktriangleright O_1)$ ist ein *Teildienst* von F_2 mit der Schnittstelle $(I_2 \blacktriangleright O_2)$ genau dann, wenn F_2 eine Verfeinerung von F_1 ist und folgende Aussage gilt:

$$dom(F_1) \subseteq dom(F_2 \dagger (I_1 \blacktriangleright O_1)).$$

Man schreibt $F_1 \sqsubseteq_{sub} F_2$.

Der Dienst F_1 ist ein *eingeschränkter* Teildienst von F_2, genau dann, wenn folgende Aussage gilt:

$$\exists R \subseteq dom(F_2) : F_1 \sqsubseteq_{sub} F_2|(dom(F_2) \setminus R).$$

□

Die Definition des eingeschränkten Teildienstes besagt, dass F_1 ein Teildienst von F_2 ist unter der Voraussetzung, dass ein geeigneter Teildefinitionsbereich für F_2 gewählt ist, d.h. bestimmte Eingabehistorien werden nicht berücksichtigt.

Mit den eingeführten Relationen kann nun die Diensthierarchie definiert werden.

Definition 4.13 (Diensthierarchie)**.** Eine Diensthierarchie ist ein azyklischer gerichteter Graph (D, w, V, H), der jeweils aus einer Menge von Diensten D, vertikalen und horizontalen Kanten $V, H \subseteq D \times D$ besteht.

Der Dienst $w \in D$ bezeichnet die Wurzel des Graphen, d.h. zu jedem Knoten aus D gibt es einen Pfad von w über Kanten aus V.

Der Graph ohne horizontale Kanten (D, w, V) ist ein Baum, d.h. der Pfad von der Wurzel zu jedem Knoten ist eindeutig.

In der Diensthierarchie gibt es eine vertikale Kante zwischen zwei Diensten genau dann, wenn einer ein (eingeschränkter) Teildienst des anderen ist:

$$(F_1, F_2) \in V \overset{\text{def}}{\Leftrightarrow} \exists R \subseteq dom(F_1) : F_2 \sqsubseteq_{sub} F_1|(dom(F_1) \setminus R),$$

wobei R eine leere Menge sein kann, d.h. $F_2 \sqsubseteq_{sub} F_1$.

Eine horizontale Kante $(F_1, F_2) \in H$ weist auf eine Abhängigkeit zwischen den Diensten F_1 und F_2 hin. Das hat zur Folge, dass ihr unmittelbarer gemeinsamer Vater F und mindestens einer der beiden Dienste F_1 und F_2 in einer echten eingeschränkten Teildienst-Relation stehen:

$$F_1 \sqsubseteq_{sub} F|(dom(F) \setminus R_1) \wedge F_2 \sqsubseteq_{sub} F|(dom(F) \setminus R_2), \text{ wobei}$$
$$R_1, R_2 \subseteq \mathbb{H}(I) \wedge (R_1 \neq \emptyset \vee R_2 \neq \emptyset).$$

Eine horizontale Kante kann nur zwischen zwei Diensten existieren, die in keiner Teildienst-Relation zueinander stehen. □

***Bemerkung* 4.1** (Unterschied zur Broyschen Definition)**.** Definition 4.13 weicht von der Broyschen Definition der *Annotated Service Hierarchy* in [Bro07a] ab. Broy definiert

eine horizontale Kante als ein Prädikat über zwei Diensten: es gibt eine horizontale Kante zwischen zwei Diensten genau dann, wenn einer der Dienste von Nachrichten abhängt, die außerhalb seines Definitionsbereichs, jedoch im Definitionsbereich des anderen Dienstes liegen. In Definition 4.13 gibt es diese Bedingung nicht. Im Gegensatz dazu wird die Auswirkung einer gegebenen Kante (Wechselwirkung) zwischen zwei beliebigen Diensten auf das Verhalten ihres gemeinsamen Vaters definiert: Falls es zwischen zwei Diensten eine horizontale Kante gibt, dann ist einer der beiden ein echter eingeschränkter Teildienst des gemeinsamen Vaters. □

4.3. Operationelle Semantik der dienstbasierten Spezifikation

Der vorangehende Abschnitt führte die denotationale Semantik der dienstbasierten Spezifikation ein, mit der Verhalten und Strukturen von Diensten ausgedrückt werden können. Vor allem wurden formale Kriterien für Teildienst- und Abhängigkeitsrelationen zwischen Diensten definiert. Allerdings handelt es sich bei der FOCUS-Theorie um einen *deskriptiven* Ansatz. In diesem Ansatz werden Dienste, zwischen denen eine Beziehung nachzuweisen ist, als gegeben angenommen. Im Gegensatz dazu streben wir einen *konstruktiven* Ansatz an, der die Kombination von Diensten explizit unterstützt. Seien zwei Dienste gegeben; unser Ziel ist, aus ihnen einen kombinierten Dienst zu konstruieren (vgl. die Definition der konstruktiven Mathematik von Bridges in [Bri97]). Dasselbe gilt für Abhängigkeiten zwischen Diensten. Von Interesse ist nicht nur, ob solch eine Abhängigkeit existiert, sondern auch wie ein kombinierter Dienst aus zwei gegebenen Diensten unter Berücksichtigung einer Abhängigkeit zwischen diesen zu konstruieren ist.

Im Folgenden wird eine operationelle Semantik der in Kapitel 3 vorgestellten Spezifikationstechnik eingeführt. Eine operationelle Semantik bedeutet nicht, dass in der Requirements Engineering Phase Vorentscheidungen über eine mögliche Implementierung der Systemfunktionalität getroffen werden. Das entsprechende Black-Box-Verhalten kann aus dieser Art der Beschreibung schematisch extrahiert werden [BP99]. Infolgedessen kann sie durch unterschiedliche Implementierungen realisiert werden.

Im folgenden Abschnitt wird eine zustandsbasierte Notation zur Spezifikation eines Dienstes eingeführt. Anschließend werden der Operator zur Kombination von Diensten sowie seine Verallgemeinerung, die priorisierte Kombination, eingeführt. Dies ergibt eine formale Semantik, die der Spezifikationstechnik aus Abschnitt 3.3 zugrunde liegt.

4.3.1. Atomarer Dienst

Ein Dienst ist durch eine syntaktische Schnittstelle und eine Verhaltensspezifikation gegeben. Die Schnittstelle $(I \blacktriangleright O)$ des Dienstes ist genauso definiert wie in Abschnitt 4.2.2. Die Verhaltensspezifikation eines Dienstes ist durch die Schnittstellenabstraktion eines Transitionssystems aus Abschnitt 4.1.2 gegeben, wobei die Variablen aus den Mengen I und O jeweils den Ein- und Ausgabeports der syntaktischen Schnittstelle entsprechen.

Es wird hier auf Beispiel 3.7 auf Seite 32 verwiesen.

***Bemerkung* 4.2** (Gleichheit zwischen Diensten)**.** Die Schnittstellenabstraktion eines Transitionssystems ist durch die Menge von Abläufen gegeben, die von den lokalen Variablen abstrahieren. Unter anderem bedeutet dies, dass zwei Dienste genau dann gleich sind, wenn sie dieselbe syntaktische Schnittstelle ($I \blacktriangleright O$) haben und dasselbe Verhalten an den Ein- und Ausgabeports aufweisen (vgl. die Definition von Abläufen auf Seite 58):

$$S_1 = S_2 \overset{\text{def}}{\Leftrightarrow} \langle\!\langle S_1 \rangle\!\rangle \overset{I \cup O}{=} \langle\!\langle S_2 \rangle\!\rangle.$$

Die Gleichheit zwischen zwei Mengen von Abläufen ist in Gleichung (4.7) definiert. □

Ein Zustand des Transitionssystems ist durch die Belegungen der Ein- und Ausgabeports und der lokalen Variablen des Dienstes gegeben. Gemäß Gleichung (4.4) auf Seite 58 können Eingaben (Belegungen der Eingabeports) sich frühestens im darauf folgenden Schritt auf Ausgaben (Belegungen der Ausgabeports) des Transitionssystems auswirken, d.h. der Dienst ist *stark kausal*.

***Bemerkung* 4.3** (Partielle Dienste)**.** Die *input-disabled*-Eigenschaft der Transitionssysteme macht es möglich, partielle Dienste zu spezifizieren. Ein Dienst ist partiell, falls es eine Belegung α gibt, für die keine ausführbaren Transitionen existieren, d.h. $\text{Succ}(\alpha) = \emptyset$. □

Verfeinerung Angelehnt an die denotationale Definition 4.11 der Dienstverfeinerung wird die Verfeinerungsrelation auf zwei Diensten durch Systemabläufe definiert.

Definition 4.14 (Dienstverfeinerung)**.** Ein Dienst $S_2 = (V_2, \mathcal{I}_2, \mathcal{T}_2)$ mit der Schnittstelle $(I_2 \blacktriangleright O_2)$ ist eine *Verfeinerung* des Dienstes $S_1 = (V_1, \mathcal{I}_1, \mathcal{T}_1)$ mit der Schnittstelle $(I_1 \blacktriangleright O_1)$ genau dann, wenn folgende Eigenschaften erfüllt sind: I_1 `subtype` I_2, O_1 `subtype` O_2 und

$$\{\rho_2 \in \langle\!\langle S_2 \rangle\!\rangle \mid \exists \rho_1 \in \langle\!\langle S_1 \rangle\!\rangle : \rho_1 \overset{I_1}{=} \rho_2\} \overset{I_1 \cup O_1}{\subseteq} \langle\!\langle S_1 \rangle\!\rangle.$$

In diesem Fall schreibt man $S_1 \succeq S_2$. □

Nach dieser Definition ist der Dienst S_2 eine Verfeinerung des Dienstes S_1, falls für dieselben Eingaben die Anzahl von Abläufen von S_2 auf V_1 kleiner gleich der Anzahl von Abläufen von S_1 auf V_1 ist.

Man beachte, gemäß Gleichung (4.6) auf Seite 59 kann die Variablenmenge V_2 nicht nur mehr Variablen enthalten, sondern die Typen dieser Variablen können größer sein als die Typen der entsprechenden Variablen aus V_1.

Definition 4.15 (Eingeschränkte Dienstverfeinerung)**.** Ein Dienst $S_2 = (V_2, \mathcal{I}_2, \mathcal{T}_2)$ mit der Schnittstelle $(I_2 \blacktriangleright O_2)$ ist eine *eingeschränkte Verfeinerung* des Dienstes $S_1 = (V_1, \mathcal{I}_1, \mathcal{T}_1)$ mit der Schnittstelle $(I_1 \blacktriangleright O_1)$ genau dann, wenn die Verfeinerungsrelation

aus Definition 4.14 unter Ausschluss aller Systemabläufe aus einer gegebenen Menge $R \subseteq \langle\langle S_2 \rangle\rangle$ gilt. D.h. folgende Eigenschaften müssen erfüllt sein: I_1 `subtype` I_2, L_1 `subtype` L_2, O_1 `subtype` O_2 und

$$\{\rho_2 \in \langle\langle S_2 \rangle\rangle \setminus R \mid \exists \rho_1 \in \langle\langle S_1 \rangle\rangle : \rho_1 \overset{I_1}{=} \rho_2\} \overset{I_1 \cup O_1}{\subseteq} \langle\langle S_1 \rangle\rangle.$$

In diesem Fall schreibt man $S_1 \succeq_R S_2$. □

Schnittstellenabstraktion In Definition 4.6 auf Seite 58 wird die Sprache eines Systems durch eine Menge von Systemabläufen definiert. Bei der Abstraktion des Systemverhaltens von lokalen Variablen wird die Sprache durch die Menge aller syntaktisch korrekten Historienpaare aus $\mathbb{H}(I) \times \mathbb{H}(O)$ definiert. Die Abbildung von Systemabläufen auf Porthistorien ist offensichtlich. Für einen gegebenen Dienst S mit der Schnittstelle $(I \blacktriangleright O)$ ist seine Schnittstellenabstraktion durch die Funktion F gegeben:

$$F(x) \overset{\text{def}}{=} \{y \in \mathbb{H}(O) \mid \exists \rho \in \langle\langle S \rangle\rangle : y \overset{O}{=} \rho \wedge x \overset{I}{=} \rho\}, \tag{4.8}$$

wobei die Gleichheit zwischen einer Historie x und einem Systemablauf ρ in einer Variablemenge V wie folgt definiert ist: $x \overset{V}{=} \rho \overset{\text{def}}{\Leftrightarrow} \forall t \in \mathbb{N}, v \in V : x(v)(t) = \rho(v).t$.

Korollar 4.1 (Verfeinerung zwischen Diensten und ihren Schnittstellenabstraktionen)**.** *Sei der Dienst S_2 eine (eingeschränkte) Verfeinerung des Dienstes S_1 nach Definition 4.14 (4.15), dann ist die Schnittstellenabstraktion F_2 von S_2 eine (eingeschränkte) Verfeinerung der Schnittstellenabstraktion F_1 von S_1 nach Definition 4.11:*

$$S_1 \succeq S_2 \Rightarrow F_1 \succeq F_2 \text{ und}$$
$$S_1 \succeq_R S_2 \Rightarrow F_1 \succeq_R F_2.$$

Beweis. Der Beweis folgt direkt aus Definitionen 4.11, 4.14, 4.15 und Gleichung (4.8). □

4.3.2. Dienstkombination

Die Dienstkombination wurde bereits in Definition 3.4 auf Seite 34 informell eingeführt. Zur Erinnerung sind im Folgenden ihre wichtigsten Eigenschaften aufgezählt:

- Die syntaktischen Schnittstellen der Teildienste können überlappen, d.h. verschiedene Dienste können dieselben Ports teilen.
- In der Kombination führen all diejenigen Dienste je eine Transition aus, welche die aktuelle Eingabe verarbeiten können und deren auszuführende Transitionen untereinander nicht widersprüchlich sind.
- Ausschließlich diese Teildienste bestimmen die Ausgaben des kombinierten Dienstes. Die restlichen Dienste modifizieren ihre lokalen Variablen nicht und schränken die Werte an den Ausgabeports nicht ein.

Nun folgt die formale Definition der Dienstkombination.

Zwei Dienste sind *kombinierbar*, falls keiner der beiden einen Ausgabeport hat, der gleichzeitig für den anderen ein Eingabeport ist, die Mengen ihrer Ports und lokalen Variablen jeweils disjunkt und ihre gemeinsamen Variablen ($V_1 \cap V_2$) vom selben Typ sind:

$$(I_1 \cup L_1) \cap (O_2 \cup L_2) = (O_1 \cup L_1) \cap (I_2 \cup L_2) = \emptyset. \tag{4.9}$$

Die Kombination $C = S_1||S_2$ von zwei kombinierbaren Diensten S_1 und S_2 ist wie folgt definiert:

$$C \stackrel{\text{def}}{=} (V_C, \mathcal{I}_C, \mathcal{T}_C), \text{ wobei}$$

- $I_C \stackrel{\text{def}}{=} I_1 \cup I_2$, $O_C \stackrel{\text{def}}{=} O_1 \cup O_2$, $L_C \stackrel{\text{def}}{=} L_1 \cup L_2$,
- $V_C \stackrel{\text{def}}{=} I_C \cup L_C \cup O_C$,
- $\mathcal{I}_C \stackrel{\text{def}}{=} \mathcal{I}_1 \wedge \mathcal{I}_2$,
- $\mathcal{T}_C$ ist durch die *Successor*-Funktion aus Gleichung (4.10) unten gegeben.

Der kombinierte Dienst kann für die Belegung α in einem der folgenden zwei Fälle einen Schritt machen (vgl. auch Bemerkung 3.5 auf Seite 34 zur Mischung aus asynchroner und synchroner Nebenläufigkeit mit gemeinsamen Variablen):

- Die beiden Dienste S_1 und S_2 sind mit der aktuellen Eingabe ausführbar (d.h. $\text{Succ}_1(\alpha) \neq \emptyset \wedge \text{Succ}_2(\alpha) \neq \emptyset$) und ihre ausführenden Transitionen sind nicht widersprüchlich (d.h. sie sind nicht kontradiktorische logische Aussagen);
- Nur einer der beiden Dienste ist für die aktuelle Belegung ausführbar ($\text{Succ}_1(\alpha) = \emptyset \vee \text{Succ}_2(\alpha) = \emptyset$). In diesem Fall werden lokale Variablen des unausführbaren Dienstes S_1 nicht modifiziert und seine Output-Ports $O_1 \setminus O_2$ unterliegen keinen Einschränkungen, d.h. diese Ports können beliebige Werte vom Porttyp enthalten.

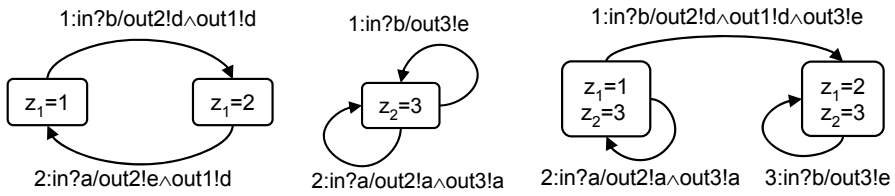
Formal ist die Succ-Funktion des kombinierten Dienstes wie folgt definiert:

$$\begin{aligned} \text{Succ}(\alpha) \stackrel{\text{def}}{=} & \{\beta \mid \exists t_1 \in \mathcal{T}_1, t_2 \in \mathcal{T}_2 : \alpha, \beta' \vdash t_1 \wedge t_2\} \\ & \bigcup_{i,j \in \{1,2\} \wedge i \neq j} \{\beta \mid (\exists t_i \in \mathcal{T}_i : \alpha, \beta' \vdash t_i) \wedge (\text{Succ}_j(\alpha) = \emptyset \wedge \alpha \stackrel{L_j}{=} \beta)\}, \end{aligned} \tag{4.10}$$

wobei $\text{Succ}_i(\alpha) = \emptyset$ zu wahr evaluiert genau dann, wenn S_i für die Belegung α nicht ausführbar ist.

Beispiel 4.2 (Dienstkombination)**.** Zur Illustration der Dienstkombination werden dieselben Transitionssysteme genommen wie in Beispiel 3.8 auf Seite 35, die in Abbildung 4.1 der Einfachheit halber erneut dargestellt und um Variablen, die den Zustand der Transitionssysteme repräsentieren, ergänzt werden.

Die Belegung β mit $\beta(z_1) = 2 \wedge \beta(z_2) = 3 \wedge \beta(out2) = d \wedge \beta(out1) = d \wedge \beta(out3) = e$ ist gemäß der ersten Zeile aus Gleichung (4.10) eine Nachfolgerin der Belegung α mit $\alpha(z_1) = 1 \wedge \alpha(z_2) = 3 \wedge \alpha(in) = b$, da die beiden Belegungen sowohl Transition 1 des



(a) Dienst D1 (b) Dienst D2 (c) Kombination D1‖D2

Abbildung 4.1.: Transitionssysteme aus Beispiel 4.2

Systems D1 als auch Transition 1 des Systems D2 erfüllen: $\alpha, \beta' \vdash (z_1 = 1 \wedge in = b \wedge z_1' = 2 \wedge out2' = d \wedge out1' = d) \wedge (z_2 = 3 \wedge in = b \wedge z_2' = 3 \wedge out3' = e)$. Dies ergibt Transition 1 des kombinierten Systems aus Abbildung 4.1(c).

Die Belegung β mit $\beta(z_1) = 1 \wedge \beta(z_3) = 3 \wedge \beta(out2) = a \wedge \beta(out3) = a$ ist gemäß der zweiten Zeile aus Gleichung (4.10) eine Nachfolgerin der Belegung α mit $\alpha(z_1) = 1 \wedge \alpha(z_2) = 3 \wedge \alpha(in) = a$, da die beiden Belegungen Transition 2 des Systems D2 erfüllen, das System D1 für die Belegung α nicht ausführbar ist (d.h. $\mathrm{Succ}_1(\alpha) = \emptyset$) und die lokale Variable von D1 nicht modifiziert wird (d.h. $\alpha(z_1) = \beta(z_1)$). Dies ergibt Transition 2 des kombinierten Systems. Man beachte, gemäß Gleichung (4.10) kann die Belegung β die Variable $out1$ auf einen beliebigen Wert aus dem Variabletyp abbilden und Transition 2 beschränkt folglich diese Variable nicht. Transition 3 des kombinierten Systems ist analog konstruiert. □

Die Dienstkombination ergibt wieder einen Dienst. Sie ist kommutativ und assoziativ. Auf diese algebraischen Eigenschaften wird in Abschnitt 4.4.1 eingegangen.

4.3.3. Priorisierte Kombination

Die priorisierte Dienstkombination wurde informell in Definition 3.5 auf Seite 38 eingeführt. Zur Erinnerung:

- Die Kombination wird von einem Priorisierungsdienst S_P gesteuert, dessen Schnittstelle die Eingabeports aller Teildienste und keine Ausgabeports umfasst.
- Wenn für eine Eingabe eine Transition von S_P ausgeführt wird und diese Transition den Dienst S_2 priorisiert, ist für diese Eingabe nur S_2 aktiviert – der andere Teildienst modifiziert seine lokalen Variablen nicht und beschränkt das Verhalten an seinen Ausgabeports in keiner Weise. Das bedeutet, dass die von S_2 unkontrollierten Ausgabeports uneingeschränkt bleiben und beliebige Werte vom Porttyp enthalten dürfen.
- Kann der Dienst S_P keine Transition ausführen oder die ausgeführte Transition priorisiert keinen Dienst, verhält sich die priorisierte wie die einfache Kombination.

Nun folgt die formale Definition der priorisierten Dienstkombination.

Die priorisierte Kombination $PC = S_1 \|^{S_P} S_2$ ist für zwei kombinierbare Dienste S_1, S_2 und einen Priorisierungsdienst S_P definiert. S_P ist wie folgt definiert:

$$S_P \stackrel{\text{def}}{=} (I_P \uplus L_P, \mathcal{I}_P, \mathcal{T}_P, p), \text{ wobei}$$

- $I_P \stackrel{\text{def}}{=} I_1 \cup I_2$,
- $L_P \cap (L_1 \cup L_2) = \emptyset$,
- Die Funktion $p\colon \mathcal{T}_P \to \{0, 1, 2\}$ legt fest, ob eine Transition aus $\mathcal{T}_P$ den Dienst S_1, S_2 oder keinen der beiden priorisiert.

Die priorisierte Kombination ist definiert wie folgt:

$$PC \stackrel{\text{def}}{=} (V_{PC}, \mathcal{I}_{PC}, \mathcal{T}_{PC}), \text{ wobei}$$

- $I_{PC} \stackrel{\text{def}}{=} I_1 \cup I_2$, $O_{PC} \stackrel{\text{def}}{=} O_1 \cup O_2$, $L_{PC} \stackrel{\text{def}}{=} L_1 \cup L_2 \cup L_P$,
- $V_{PC} \stackrel{\text{def}}{=} I_{PC} \cup L_{PC} \cup O_{PC}$,
- $\mathcal{I}_{PC} \stackrel{\text{def}}{=} \mathcal{I}_P \wedge \mathcal{I}_1 \wedge \mathcal{I}_2$,
- $\mathcal{T}_{PC}$ ist durch die Succ-Funktion aus Gleichung (4.11) unten definiert.

Die priorisierte Kombination kann für die Belegung α in einem der folgenden Fälle einen Schritt machen:

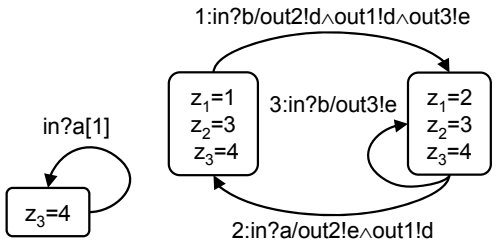
- Die unpriorisierte Kombination der beiden Dienste S_1 und S_2 ist ausführbar, der Priorisierungsdienst jedoch nicht.
- Es gibt eine ausführbare Transition des Priorisierungsdienstes, diese priorisiert jedoch keinen der beiden Dienste und die unpriorisierte Kombination ist ausführbar;
- Es gibt eine ausführbare Transition des Priorisierungsdienstes, diese priorisiert einen der beiden Dienste und der priorisierte Dienst ist ausführbar.

Die Succ-Funktion der priorisierten Kombination von S_1 und S_2 ist wie folgt definiert ($\mathcal{T}_C$ bezeichnet die Transitionsmenge der unpriorisierten Kombination $C = S_1 \| S_2$):

$$\begin{aligned} \text{Succ}(\alpha) \stackrel{\text{def}}{=} & \{\beta \mid \exists t \in \mathcal{T}_C : \forall t_P \in \mathcal{T}_P : (\alpha, \beta' \vdash t \wedge \neg t_P) \wedge \alpha \stackrel{L_P}{=} \beta\} \\ & \cup \{\beta \mid \exists t \in \mathcal{T}_C, t_P \in \mathcal{T}_P : (\alpha, \beta' \vdash t \wedge t_P) \wedge p(t_P) = 0\} \\ \bigcup_{i,j \in \{1,2\} \wedge i \neq j} & \{\beta \mid \exists t_i \in \mathcal{T}_i, t_P \in \mathcal{T}_P : (\alpha, \beta' \vdash t_i \wedge t_P) \wedge p(t_P) = i \wedge \alpha \stackrel{L_j}{=} \beta\}. \end{aligned} \tag{4.11}$$

Die erste Teilmenge definiert den Fall, wenn keine Transition von S_P ausführbar ist (dann gilt $\alpha \stackrel{L_P}{=} \beta$) , die zweite – wenn die ausführbare Transition keinen der beiden Dienste priorisiert (d.h. $p(t_P) = 0$). In beiden Fällen stimmt das Verhalten von $S_1 \|^{S_P} S_2$ mit dem Verhalten von $S_1 \| S_2$ überein (d.h. die ausführende Transition ist aus der Transitionsmenge der unpriorisierten Kombination: $t \in \mathcal{T}_C$). Die letzte Teilmenge beschreibt das gemeinsame Verhalten von S_i und S_P, d.h. das Verhalten vom Dienst S_1

oder S_2, welcher für die aktuelle Belegung α priorisiert ist. Die lokalen Variablen des unpriorisierten Dienstes werden nicht modifiziert (d.h. $\alpha \overset{L_j}{=} \beta$). Die vom priorisierten Dienst unkontrollierten Ausgabevariablen unterliegen keinen Einschränkungen.



(a) System P (b) Kombination D1‖^{P}D2

Abbildung 4.2.: Transitionssysteme aus Beispiel 4.3

Beispiel 4.3 (Priorisierte Dienstkombination)**.** Zur Illustration der priorisierten Dienstkombination werden die beiden Transitionssysteme aus Abbildungen 4.1(a) und 4.1(b) auf Seite 68 sowie der Priorisierungsdienst aus Abbildung 4.2(a) genommen. Bei der Ausführung seiner einzigen Transition wird der Dienst D1 priorisiert. Das kombinierte System D1‖^{P}D2 ist in Abbildung 4.2(b) dargestellt.

Die Belegung α mit $\alpha(z_1) = 1 \wedge \alpha(z_2) = 3 \wedge \alpha(z_3) = 4 \wedge \alpha(in) = b$ hat gemäß der ersten Zeile aus Gleichung (4.11) die Belegung β mit $\beta(z_1) = 2 \wedge \beta(z_2) = 3 \wedge \beta(z_3) = 4 \wedge \beta(out2) = d \wedge \beta(out1) = d \wedge \beta(out3) = e$ als eine Nachfolgerin, da die beiden Belegungen eine Transition der einfachen Kombination erfüllen (vgl. Beispiel 4.2) und der Priorisierungsdienst für die Belegung α nicht ausführbar ist. Dies ergibt Transition 1 des kombinierten Systems. Transition 3 ist analog konstruiert – sie gehört auch der Transitionsmenge der einfachen Kombination.

Die Belegung α mit $\alpha(z_1) = 2 \wedge \alpha(z_2) = 3 \wedge \alpha(z_3) = 4 \wedge \alpha(in) = a$ hat gemäß der dritten Zeile aus Gleichung (4.11) die Belegung β mit $\beta(z_1) = 1 \wedge \beta(z_2) = 3 \wedge \beta(z_3) = 4 \wedge \beta(out2) = e \wedge \beta(out1) = d$ als eine Nachfolgerin, da die beiden Belegungen Transition 2 des Systems D1 und die Transition des Priorisierungsdienstes erfüllen. Man beachte, die Belegung β weist der Variable $out3$ einen beliebigen Wert aus dem Variabletyp zu, da der unpriorisierte Dienst D2 seine Ausgabevariablen nicht beschränkt. Dies ergibt Transition 2 des kombinierten Systems. □

***Bemerkung* 4.4** (Priorisierung zwischen n Diensten)**.** Der Einfachheit halber wurde die priorisierte Kombination für zwei Dienste definiert. Es ist offensichtlich, dass sie sich auf n Dienste verallgemeinern lässt. Lediglich die Funktion p des Priorisierungsdienstes muss zu $p\colon \mathcal{T}_P \to 2^{[0..n]}$ verallgemeinert werden. Dadurch können mehrere Dienste gleichzeitig priorisiert werden. □

Es ist offensichtlich, dass die unpriorisierte Kombination aus Abschnitt 4.3.2 ein Sonderfall der priorisierten Kombination ist. Ihre Verhalten stimmen überein, falls der Priorisierungsdienst für keine Belegung ausführbar ist.

Die priorisierte Kombination ist kommutativ und distributiv jedoch nicht assoziativ. Diese Eigenschaften werden in Abschnitt 4.4.2 nachgewiesen.

4.3.4. Diensthierarchie

Aufbauend auf den beiden Kombinationsoperatoren aus den vorangehenden Abschnitten wird im Folgenden die Diensthierarchie definiert (vgl. Abbildung 4.3).

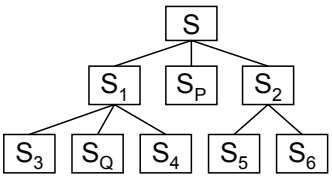


Abbildung 4.3.: Diensthierarchie: $S = (S_3 \parallel^{S_Q} S_4) \parallel^{S_P} (S_5 \parallel S_6)$

Definition 4.16 (Diensthierarchie)**.** Die Diensthierarchie ist durch einen azyklischen gerichteten Graphen $(D \cup P, w, V)$ gegeben. $D \cup P$ ist die Menge von Knoten, die aus einer Menge von Diensten D und einer Menge von Priorisierungsdiensten P besteht. $V \subseteq D \times (D \cup P)$ ist die Menge von Kanten, die einen Dienst mit einem anderen Dienst oder einem Priorisierungsdienst verbinden. Der Dienst $w \in D$ bezeichnet die Wurzel des Graphen, d.h. zu jedem Knoten aus $D \cup P$ gibt es einen Pfad von w.

Der Graph ist ein Baum, d.h. der Pfad von der Wurzel zu jedem Knoten ist eindeutig. Der Baum definiert eine Halbordnung auf der Menge $D \cup P$. Gemäß der Definition der Kanten ist jeder Priorisierungsdienst ein kleinstes Element der Menge $D \cup P$, d.h. ein Priorisierungsdienst kann nicht in weitere Dienste geteilt werden.

Die kleinsten Elemente aus D gemäß der Halbordnung sind atomare Dienste, die restlichen Dienste sind kombiniert. Ein kombinierter Dienst umfasst alle seine gemäß der Halbordnung unmittelbaren Nachfolger. Gibt es keinen Priorisierungsdienst unter seinen Nachfolgern, werden die Nachfolger anhand der einfachen Kombination aus Abschnitt 4.3.2 kombiniert. Andererseits wird die priorisierte Dienstkombination aus Abschnitt 4.3.3 angewendet. Für jeden Priorisierungsdienst $S_P \in P$ gilt Folgendes: ist S_P zwischen zwei Diensten $S_1, S_2 \in D$ definiert (d.h. $\exists S \in D : S = S_1 \parallel^{S_P} S_2$), dann hängen alle drei unter dem Dienst S: $(S, S_1), (S, S_2), (S, S_P) \in V$.

Für die beiden Kombinationsoperatoren gilt folgende Rangfolge: $S_1 \parallel S_2 \parallel^{S_p} S_3 = S_1 \parallel (S_2 \parallel^{S_p} S_3)$. □

***Bemerkung* 4.5** (Syntaktisch inkonsistente Diensthierarchie)**.** Wie bereits erwähnt, ist die priorisierte Kombination nicht assoziativ. Dies hat zur Folge, dass die Kombination $S_1 \parallel^{S_p} S_2 \parallel^{S_q} S_3$ mehrdeutig ist. Eine Diensthierarchie ist syntaktisch inkonsistent, falls mehrere Priorisierungsdienste aus P auf denselben Dienst aus D verweisen. Um diese Inkonsistenz zu vermeiden, müssen die betroffenen Dienste hierarchisch umstrukturiert werden, z.B. so $S_1 \parallel^{S_p} (S_2 \parallel^{S_q} S_3)$. □

4.3.5. Komposition

Wie bereits in Abschnitt 3.2 erwähnt, unterstützt die eingeführte Spezifikationstechnik sowohl die strukturelle als auch die funktionale Dekomposition. Obwohl die strukturelle Dekomposition nicht im Fokus der vorliegenden Arbeit liegt, wird sie im Folgenden formal definiert.

In der strukturellen Komposition können Dienste durch gleichnamige Ein- und Ausgabeports direkt miteinander kommunizieren. Zwei Dienste mit $V_1 = I_1 \cup L_1 \cup O_1$ und $V_2 = I_2 \cup L_2 \cup O_2$ sind komponierbar, wenn folgende Eigenschaften erfüllt sind $L_1 \cap V_2 = L_2 \cap V_1 = O_1 \cap O_2 = \emptyset$. Man beachte, im Gegensatz zur funktionalen Kombination dürfen zwei Dienste bei der strukturellen Komposition keine gemeinsamen Ausgabeports haben.

Für zwei komponierbare Dienste S_1 und S_2 ist die Komposition $C \stackrel{\text{def}}{=} S_1 \oplus S_2$ wie folgt definiert:

$$C \stackrel{\text{def}}{=} (V_C, \mathcal{I}_C, T_C) \text{ mit}$$

- $I_C \stackrel{\text{def}}{=} (I_1 \cup I_2) \setminus (O_1 \cup O_2)$, $O_C \stackrel{\text{def}}{=} (O_1 \cup O_2) \setminus (I_1 \cup I_2)$, $L_C \stackrel{\text{def}}{=} L_1 \cup L_2 \cup (V_1 \cap V_2)$,
- $V_C \stackrel{\text{def}}{=} I_C \cup L_C \cup O_C$,
- $\mathcal{I}_C \stackrel{\text{def}}{=} \mathcal{I}_1 \wedge \mathcal{I}_2$.

Die Transitionsmenge $\mathcal{T}_C$ ist durch die Succ-Function aus Gleichung (4.12) unten gegeben. Die Komposition macht einen Schritt genau dann, wenn seine beiden Teildienste ausführbar sind:

$$\text{Succ}(\alpha) \stackrel{\text{def}}{=} \{\beta \mid \exists t_1 \in \mathcal{T}_1, t_2 \in \mathcal{T}_2 \wedge \alpha, \beta' \vdash t_1 \wedge t_2\}. \tag{4.12}$$

Daraus folgt $\text{En}(\alpha) = \text{En}_1(\alpha) \wedge \text{En}_2(\alpha)$ (vgl. Gleichung (4.5) auf Seite 58).

Zur Illustration der strukturellen Komposition sei auf Beispiel 3.2 auf Seite 24 verwiesen.

Die Definition der Komposition entspricht der Definition aus [dAH01] und wurde deswegen sehr kurz erläutert. Wie in [Bot08] gezeigt, ist die strukturelle Komposition stark kausal, assoziativ und kommutativ.

4.4. Algebraische Eigenschaften

Um eine sinnvolle Aussage über die Gesamtfunktionalität eines Systems treffen zu können, ist es essentiell wichtig, algebraische Eigenschaften der Dienstkombination zu untersuchen. Beispielsweise ist in diesem Kontext eine relevante Frage, ob das resultierende Gesamtverhalten des Systems von der Reihenfolge der Kombination der einzelnen Teildienste abhängt. Folglich werden in den folgenden zwei Abschnitten die Eigenschaften wie Kommutativität, Assoziativität oder Distributivität der beiden Kombinationsoperatoren untersucht.

4.4.1. Kombination

In diesem Abschnitt stehen die Eigenschaften der unpriorisierten Dienstkombination aus Abschnitt 4.3.2 im Mittelpunkt.

Es wird zunächst gezeigt, dass die Kombination von zwei Diensten einen Dienst ergibt.

Satz 4.2 (Wohldefinierte Kombination)**.** *Seien zwei Dienste S_1 mit der Schnittstelle $(I_1 \blacktriangleright O_1)$ und S_2 mit der Schnittstelle $(I_2 \blacktriangleright O_2)$ jeweils durch ein Transitionssystem $\mathcal{S}_1 = (V_1, \mathcal{I}_1, \mathcal{T}_1)$ und $\mathcal{S}_2 = (V_2, \mathcal{I}_2, \mathcal{T}_2)$ definiert, dann ist ihre Kombination $S = S_1 \parallel S_2$ wiederum ein Dienst. Das heißt, die Dienstkombination hat disjunkte Variablenmengen I, L und O und erfüllt Eigenschaften (4.2) und (4.3) auf Seite 57.*

Beweis. Seien $V \stackrel{\text{def}}{=} I \cup L \cup O$ und $V_i \stackrel{\text{def}}{=} I_i \cup L_i \cup O_i$ mit $i \in \{1, 2\}$ gegeben.

- Die Disjunktheit von I, L und O folgt direkt aus der Definition der Kombinierbarkeit von Diensten in Gleichung (4.9) auf Seite 67.
- Eigenschaft (4.2) besagt, dass die Variablen aus I durch das Prädikat $\mathcal{I}$ nicht eingeschränkt sein dürfen: $\forall \alpha, \beta \in \Lambda(V) : \alpha \overset{L \cup O}{=} \beta \Rightarrow \big(\alpha \vdash \mathcal{I} \Leftrightarrow \beta \vdash \mathcal{I}\big)$. Hier werden zwei Fälle betrachtet. (1) Es gibt keine Belegung aus $\Lambda(V_1 \cup V_2)$, die sowohl $\mathcal{I}_1$ als auch $\mathcal{I}_2$ erfüllt: $\forall \alpha \in \Lambda(V) : \alpha \nvdash \mathcal{I}$. Dann gilt die Konklusion der Formel trivialerweise. (2) Es existiert eine Belegung, welche die beiden Prädikate erfüllt. O.B.d.A. wird α als eine Belegung mit $\alpha \overset{L_1 \cup O_1}{=} \alpha_1 \wedge \alpha \overset{L_2 \cup O_2}{=} \alpha_2$ definiert, die wegen Eigenschaft (4.9) weiterhin keine Eingabeports einschränkt. Aus der Prämisse $\alpha \overset{L \cup O}{=} \beta$ folgt, dass $\beta \overset{L_1 \cup O_1}{=} \alpha \overset{L_1 \cup O_1}{=} \alpha_1$ und $\beta \overset{L_2 \cup O_2}{=} \alpha \overset{L_2 \cup O_2}{=} \alpha_2$. Daraus wiederum folgt $\big(\alpha \vdash \mathcal{I}_1 \wedge \mathcal{I}_2 \Leftrightarrow \beta \vdash \mathcal{I}_1 \wedge \mathcal{I}_2\big)$.
- Eigenschaft (4.3) besagt, dass keine Transition aus T ihre Eingaben beeinflussen und die bisherigen Ausgaben einschränken kann: $\forall \alpha, \beta \in \Lambda(V \cup V') : \big(\alpha \overset{I \cup L}{=} \beta \wedge \alpha \overset{L' \cup O'}{=} \beta\big) \Rightarrow \big((\alpha \vdash t) \Leftrightarrow (\beta \vdash t)\big)$. Angenommen, die Aussage $\big(\alpha \overset{I \cup L}{=} \beta \wedge \alpha \overset{L' \cup O'}{=} \beta\big)$ gilt für $\alpha, \beta \in \Lambda(V \cup V')$. Dann müssen zwei symmetrische Fälle betrachtet werden. Es wird gezeigt, dass wenn es eine Transition $t \in \mathcal{T}$ gibt, so dass $\alpha \vdash t$, dann gilt auch $\beta \vdash t$. Die andere Richtung ist analog zu beweisen.

  Gemäß der Definition der Kombination aus Gleichung (4.10) auf Seite 67 folgt aus $\alpha \vdash t$ die Aussage $\alpha \vdash t_1 \vee \alpha \vdash t_2$. Wegen der Symmetrie der beiden Fälle wird nur der erste Fall gezeigt, d.h. wenn $\alpha \vdash t_1$ gilt, dann gilt auch $\beta \vdash t_1$.

  Seien $\alpha_1 \overset{V_1 \cup V_1'}{=} \alpha$ und $\beta_1 \overset{V_1 \cup V_1'}{=} \beta$ gegeben. Dann gelten folgende Aussagen: $\beta_1 \overset{I_1 \cup L_1}{=} \alpha_1$, $\beta_1 \overset{L_1' \cup O_1'}{=} \alpha_1$ und $\alpha_1 \vdash t_1 \Leftrightarrow \beta_1 \vdash t_1$. Die linke Seite der letzten Aussage gilt wegen $\alpha \vdash t_1$ und $\alpha_1 \overset{V_1 \cup V_1'}{=} \alpha$. Dann gilt auch $\beta_1 \vdash t_1$. Wegen $\beta_1 \overset{V_1 \cup V_1'}{=} \beta$ gilt auch die Aussage $\beta \vdash t_1$.

□

Nun folgen die Eigenschaften, die sicherstellen, dass die Reihenfolge, in der Dienste

kombiniert werden, keine Auswirkung auf das resultierende Verhalten des kombinierten Dienstes haben.

Satz 4.3 (Kommutativität, Assoziativität, Idempotenz)**.** *Der Kombinationsoperator ist kommutativ, assoziativ und idempotent*[2]*. Das heißt, für paarweise kombinierbare Dienste S_1, S_2 und S_3 gelten folgende Gleichungen:*

$$S_1 \parallel S_2 = S_2 \parallel S_1,\ (S_1 \parallel S_2) \parallel S_3 = S_1 \parallel (S_2 \parallel S_3) \text{ und } S_1 \parallel S_1 = S_1.$$

Beweis. Die Kommutativität des Operators ist offensichtlich, da die Definition der Komposition symmetrisch für beide Argumente ist.

Für den Beweis der Assoziativität müssen folgende Relationen bewiesen werden: $\langle\!\langle (S_1 \parallel S_2) \parallel S_3 \rangle\!\rangle \subseteq \langle\!\langle S_1 \parallel (S_2 \parallel S_3) \rangle\!\rangle$ und $\langle\!\langle (S_1 \parallel S_2) \parallel S_3 \rangle\!\rangle \supseteq \langle\!\langle S_1 \parallel (S_2 \parallel S_3) \rangle\!\rangle$, woraus dann die Gleichung $\langle\!\langle (S_1 \parallel S_2) \parallel S_3 \rangle\!\rangle = \langle\!\langle S_1 \parallel (S_2 \parallel S_3) \rangle\!\rangle$ folgt.

$\subseteq$: In diesem Fall muss gezeigt werden, dass für jede Belegung $\alpha \in \Lambda(V_1 \cup V_2 \cup V_3)$ folgende Aussage gilt: $\mathrm{Succ}_{(S_1 \parallel S_2) \parallel S_3}(\alpha) \subseteq \mathrm{Succ}_{S_1 \parallel (S_2 \parallel S_3)}(\alpha)$. Angenommen, es gibt eine Belegung $\beta \in \Lambda(V_1 \cup V_2 \cup V_3)$ mit

$$\beta \in \mathrm{Succ}_{(S_1 \parallel S_2) \parallel S_3}(\alpha) \qquad \text{und} \qquad \beta \notin \mathrm{Succ}_{S_1 \parallel (S_2 \parallel S_3)}(\alpha).$$

Gemäß Definition (4.10) auf Seite 67 und wegen der Assoziativität des booleschen Operators $\wedge$ und des mengentheoretischen Operators $\cup$, bedeutet die Aussage $\beta \in \mathrm{Succ}_{(S_1 \parallel S_2) \parallel S_3}(\alpha)$, dass mindestens einer der drei Dienste ausführbar ist und folgende Aussage gilt: $\forall i \in \{1,2,3\} : \mathrm{En}_i(\alpha) \Rightarrow \exists t_i \in \mathcal{T}_i : \alpha, \beta' \vdash t_i$. Dies steht jedoch wegen der Assoziativität von $\wedge$ und $\cup$ im Widerspruch zur Aussage $\beta \notin \mathrm{Succ}_{S_1 \parallel (S_2 \parallel S_3)}(\alpha)$. Siehe Gleichung (4.5) auf Seite 58 für die Definition von En.

$\supseteq$: Analog zu „$\subseteq$“ führt der Widerspruch zwischen den Aussagen $\beta \in \mathrm{Succ}_{(S_1 \parallel S_2) \parallel S_3}(\alpha)$ und $\beta \notin \mathrm{Succ}_{(S_1 \parallel S_2) \parallel S_3}(\alpha)$ zu $\mathrm{Succ}_{(S_1 \parallel S_2) \parallel S_3}(\alpha) \supseteq \mathrm{Succ}_{S_1 \parallel (S_2 \parallel S_3)}(\alpha)$.

Wie bereits erklärt, gilt die Idempotenz nicht allgemein. Wegen der Kombinierbarkeit der Dienste, kann nur folgende Gleichung bewiesen werden: $S_a \parallel S_b = S_a = S_b$, wobei S_a und S_b bis auf die Namen der lokalen Variablen mit dem Dienst S identisch sind. Schließlich müssen folgende Relationen für den Beweis der Idempotenz bewiesen werden: $\langle\!\langle (S \parallel S) \rangle\!\rangle \subseteq \langle\!\langle S \rangle\!\rangle$ und $\langle\!\langle (S \parallel S) \rangle\!\rangle \supseteq \langle\!\langle S \rangle\!\rangle$, woraus die Gleichung $\langle\!\langle (S \parallel S) \rangle\!\rangle = \langle\!\langle S \rangle\!\rangle$ folgt. Der Beweis geht nach demselben Schema wie in den vorangehenden Teilen. Wegen der Idempotenz des Operators $\wedge$ gilt $\beta \in \mathrm{Succ}_{(S \parallel S)}(\alpha) \Leftrightarrow \beta \in \mathrm{Succ}_S(\alpha)$. □

Wie bereits in Abschnitt 3.3.2 auf Seite 32 erläutert, fügt ein Dienst ein Stück Verhalten der Gesamtfunktionalität hinzu – der Definitionsbereich des Systems wird erweitert und das Verhalten des Systems wird deterministischer. Nun wird gezeigt, dass die Dienstkombination ihre Teildienste verfeinert.

[2] Im Allgemeinen ist die Kombination nicht idempotent, da zwei identische Dienste nicht kombinierbar sind – die Mengen ihrer lokalen Variablen sind naturgemäß nicht disjunkt. Um die Kombinierbarkeit von zwei identischen Diensten herzustellen, müssen ihre lokalen Variablen umbenannt werden.

Satz 4.4 (Kombination verfeinert ihre Teildienste)**.** *Die Kombination von zwei Diensten $S = S_1 \parallel S_2$ ist eine Verfeinerung der beiden, d.h. es gilt $S_1 \succeq S$ und $S_2 \succeq S$.*

Beweis. Wegen der Symmetrie der beiden Fälle wird nur die Relation $S_1 \succeq S$ gezeigt. Gemäß Definition 4.14 auf Seite 65 müssen folgende Relationen nachgewiesen werden: I_1 `subtype` I und O_1 `subtype` O. Dies folgt trivialerweise aus Gleichungen $I = I_1 \cup I_2$, $L = L_1 \cup L_2$ und $O = O_1 \cup O_2$.

Außerdem muss die Gültigkeit folgender Relation gezeigt werden:

$$\{\rho \in \langle\!\langle S \rangle\!\rangle \mid \exists \rho_1 \in \langle\!\langle S_1 \rangle\!\rangle, \rho_1 \overset{I_1}{=} \rho\} \overset{I_1 \cup O_1}{\subseteq} \langle\!\langle S_1 \rangle\!\rangle.$$

Dafür werden gemäß Definition 4.6 auf Seite 58 folgende zwei Eigenschaften bewiesen:

$$\forall \rho \in \langle\!\langle S \rangle\!\rangle : \rho.0 \vdash \mathcal{I} \Rightarrow \rho.0 \vdash \mathcal{I}_1 \text{ und} \tag{4.13a}$$

$$\begin{aligned} &\forall \rho \in \langle\!\langle S \rangle\!\rangle, i \in \mathbb{N} : \\ &\quad \exists \rho_1 \in \langle\!\langle S_1 \rangle\!\rangle : \rho.i \overset{I_1}{=} \rho_1.i \Rightarrow \exists \beta_1 \in \mathrm{Succ}_1(\rho_1.i) : \rho.(i+1) \overset{I_1 \cup O_1}{=} \beta_1 \vee \\ &\quad \exists \rho_1 \in \langle\!\langle S_1 \rangle\!\rangle : \rho.i \overset{I_1}{=} \rho_1.i \wedge \neg \mathrm{En}(\rho.i) \Rightarrow \neg \mathrm{En}_1(\rho_1.i). \end{aligned} \tag{4.13b}$$

- Eigenschaft (4.13a) folgt trivialerweise aus der Definition der Kombination initialer Belegungen, nämlich aus $\mathcal{I} \overset{\text{def}}{=} \mathcal{I}_1 \wedge \mathcal{I}_2$ folgt $\mathcal{I} \Rightarrow \mathcal{I}_1$.
- Eigenschaft (4.13b) für unendliche Abläufe (d.h. 2. Zeile) ist induktiv über $\mathbb{N}$ zu beweisen. Zuerst wird die Belegung $\rho.0$ betrachtet. Aus $\rho \in \langle\!\langle S \rangle\!\rangle$ folgt $\rho.1 \in \mathrm{Succ}(\rho.0)$. Aus Eigenschaft (4.13a) folgt, dass $\forall \rho \in \langle\!\langle S \rangle\!\rangle : \exists \rho \in \langle\!\langle S \rangle\!\rangle, \rho_1 \in \langle\!\langle S_1 \rangle\!\rangle : \rho.0 \overset{V_1}{=} \rho_1.0$. Daraus folgt, dass gemäß Definition (4.10) auf Seite 67 die Belegung $\rho.1$ aus einer der folgenden drei Mengen sein muss.
  1. $\{\beta \mid \exists t_1 \in \mathcal{T}_1, t_2 \in \mathcal{T}_2 : \alpha, \beta' \vdash t_1 \wedge t_2\}$. Daraus folgt, dass $\exists t \in \mathcal{T} : \rho.0, \rho.1' \vdash t$, wobei α und β durch $\rho.0$ und $\rho.1$ entsprechend ersetzt werden. Aus $\rho.0 \overset{V_1}{=} \rho_1.0$ und $t = t_1 \wedge t_2$ folgt, dass $\exists t_1 \in \mathcal{T}_1 : \rho_1.0, \rho_1.1' \vdash t_1 \wedge \rho.1 \overset{V_1}{=} \rho_1.1$.
  2. $\{\beta \mid \exists t_1 \in \mathcal{T}_1 : \alpha, \beta' \vdash t_1 \wedge \neg \mathrm{En}_2(\alpha) \wedge \alpha \overset{L_2}{=} \beta\}$. Der Beweis ist analog zum vorherigen Fall.
  3. $\{\beta \mid \exists t_2 \in \mathcal{T}_2 : \alpha, \beta' \vdash t_2 \wedge \neg \mathrm{En}_1(\alpha) \wedge \alpha \overset{L_1}{=} \beta\}$. Dieser Fall kann nicht auftreten, wenn die Prämisse der Folgerung aus Eigenschaft (4.13b) erfüllt ist. Wenn es nämlich einen Systemablauf $\rho \in \langle\!\langle S \rangle\!\rangle$ und einen Systemablauf $\rho_1 \in \langle\!\langle S_1 \rangle\!\rangle$ mit $\rho \overset{I_1}{=} \rho_1$ gibt, kann das Prädikat $\neg \mathrm{En}_1(\rho_1.0)$ nicht erfüllt sein.

  Induktionschritt: Aus der Annahme, dass Eigenschaft (4.13b) für i gilt, folgt die Aussage $\forall \rho \in \langle\!\langle S \rangle\!\rangle : \exists \rho_1 \in \langle\!\langle S_1 \rangle\!\rangle : \rho.i \overset{V_1}{=} \rho_1.i$. Unter dieser Voraussetzung ist der Beweis für $(i+1)$ genau nach demselben Schema wie der für $i = 0$ zu führen.
- Eigenschaft (4.13b) für endliche Abläufe (d.h. 3. Zeile): aus $\neg \mathrm{En}(\rho.i)$ folgt gemäß Definition (4.10) einer der folgenden Fälle:

1. $\neg \mathrm{En}_1(\rho_1.i) \wedge \neg \mathrm{En}_2(\rho_2.i)$ mit $\rho.i \overset{V_1}{=} \rho_1.i$. Somit ist die Eigenschaft erfüllt.
2. $\mathrm{En}_1(\rho_1.i) \wedge \mathrm{En}_2(\rho_2.i) \wedge \forall t_1 \in \mathcal{T}_1, t_2 \in \mathcal{T}_2 : \rho.i, \rho.(i+1)' \vdash \neg(t_1 \wedge t_2)$. In diesem Fall blockieren sich die beiden Dienste, so dass die Kombination von zwei unendlichen Abläufen in $\langle\langle S_1 \rangle\rangle$ und $\langle\langle S_2 \rangle\rangle$ einen endlichen Ablauf in $\langle\langle S \rangle\rangle$ ergibt. Für diesen Fall gilt die Verfeinerung nicht und die Kombination wird als inkonsistent bezeichnet. Diese Sonderfälle werden in Abschnitt 5.2 behandelt.

Daraus folgt $S_1 \succeq S$, falls die Dienstkombination S konsistent ist (siehe Abschnitt 5.2). □

4.4.2. Priorisierte Kombination

Nun wird auf die priorisierte Kombination aus Abschnitt 4.3.3 eingegangen.

Die Kombination mehrerer Dienste unter Berücksichtigung von Priorisierungen zwischen diesen ergibt wieder einen Dienst.

Satz 4.5 (Wohldefinierte priorisierte Kombination)**.** *Seien zwei Dienste S_1 mit der Schnittstelle $(I_1 \blacktriangleright O_1)$ und S_2 mit der Schnittstelle $(I_2 \blacktriangleright O_2)$ jeweils durch ein Transitionssystem $\mathcal{S}_1 = (V_1, \mathcal{I}_1, \mathcal{T}_1)$ und $\mathcal{S}_2 = (V_2, \mathcal{I}_2, \mathcal{T}_2)$ gegeben. Außerdem sei ein Priorisierungsdienst S_P durch das Transitionssystem $S_P \overset{\text{def}}{=} (I_1 \uplus I_2 \uplus L_1 \uplus L_2, \mathcal{I}_P, \mathcal{T}_P, p)$ gegeben. Dann ist die priorisierte Kombination $S = S_1 \parallel^{S_P} S_2$ wiederum ein Dienst.*

Beweis. Der Beweis geht analog zu dem aus Satz 4.2. □

Die priorisierte Kombination ist nicht assoziativ, jedoch kommutativ und distributiv.

Satz 4.6 (Kommutativität, Distributivität, Idempotenz)**.** *Der priorisierte Kombinationsoperator ist kommutativ, distributiv und idempotent*[3]*. Das heißt, für paarweise kombinierbare Dienste S_1, S_2 und S_3 sowie zwei Priorisierungsdienste S_P und S_Q gelten folgende Gleichungen:*

$$S_1 \parallel^{S_P} S_2 = S_2 \parallel^{S_P} S_1,\ S_1 \parallel^{S_P} (S_2 \parallel^{S_Q} S_3) = (S_1 \parallel^{S_P} S_2) \parallel^{S_Q} (S_1 \parallel^{S_P} S_3) \text{ und } S \parallel^{S_P} S = S.$$

Beweis. Die Kommutativität des Operators ist offensichtlich. Die Definition der Succ-Funktion der priorisierten Kombination basiert auf drei Teilmengen (vgl. Gleichung (4.11) auf Seite 69). Bei den ersten zwei handelt es sich um den Fall, wenn keiner der beiden Dienste priorisiert ist, d.h. hier gelten die Regeln der kommutativen unpriorisierten Kombination. In der dritten Teilmenge bestimmt die Funktion p eindeutig, welcher der beiden Dienste priorisiert ist. Da sie symmetrisch für beide Argumente ist, ist auch die priorisierte Kombination kommutativ.

Wie bereits erwähnt, unterteilt ein Priorisierungsdienst die Menge von Belegungen in jedem Schritt in drei Teilmengen: einer oder keiner der beiden Dienste ist priorisiert.

[3] Genau wie in Satz 4.3 gilt die Idempotenz der priorisierten Kombination nicht allgemein. Wegen der Kombinierbarkeit, müssen alle lokalen Variablen von S entsprechend umbenannt werden.

Folglich kann eine priorisierte Kombination $S_1 \parallel^{S_P} S_2$ für einen Schritt mit einer der drei folgenden Formeln beschrieben werden: (1) $S_1 \parallel^{S_P} S_2 = S_1$ falls $\exists t_P \in \mathcal{T}_P : (\alpha, \beta' \vdash t_P) \wedge p(t_P) = 1$, (2) $S_1 \parallel^{S_P} S_2 = S_2$ falls $\exists t_P \in \mathcal{T}_P : (\alpha, \beta' \vdash t_P) \wedge p(t_P) = 2$ und (3) $S_1 \parallel^{S_P} S_2 = S_1 \parallel S_2$ falls $\exists t_P \in \mathcal{T}_P : (\alpha, \beta' \vdash t_P) \wedge p(t_P) = 0$ oder $\forall t_P \in \mathcal{T}_P : (\alpha, \beta' \nvdash t_P)$. Für den Beweis der Gleichung $S_1 \parallel^{S_P} (S_2 \parallel^{S_Q} S_3) = (S_1 \parallel^{S_P} S_2) \parallel^{S_Q} (S_1 \parallel^{S_P} S_3)$ mit zwei priorisierten Kombinationen müssen 3^2 mögliche Kombinationen betrachtet werden. Da sie alle jedoch symmetrisch sind, wird nur die Kombination betrachtet, in der S_P und S_Q keinen Dienst priorisieren. In diesem Fall stimmt das Verhalten der priorisierten Kombination mit der unpriorisierten überein und es muss gezeigt werden, dass die Gleichung $S_1 \parallel (S_2 \parallel S_3) = (S_1 \parallel S_2) \parallel (S_1 \parallel S_3)$ gilt. Da die unpriorisierte Kombination kommutativ, assoziativ und idempotent ist, erfolgt der Beweis durch eine einfachere Umformung. Die restlichen acht Fälle gehen analog.

Für den Beweis der Idempotenz werden zwei mögliche Situationen betrachtet. (1) S_P priorisiert keinen Dienst, d.h. $S \parallel^{S_P} S = S \parallel S$. Wegen der Idempotenz der unpriorisierten Kombination ist auch die priorisierte idempotent. (2) S_P priorisiert den Dienst S, d.h. das Verhalten von $S \parallel^{S_P} S$ stimmt mit dem Verhalten von S überein. □

Analog zur einfachen verfeinert die priorisierte Kombination ihre Teildienste. Allerdings gilt dies nicht für alle Eingaben. Es ist offensichtlich, dass wenn in der Kombination $S = S_1 \parallel^{S_P} S_2$ der Dienst S_1 für eine Belegung priorisiert ist (d.h. das Verhalten von S muss nur mit dem Verhalten von S_1 übereinstimmen), ist die Kombination S keine Verfeinerung von S_2. Das heißt, es handelt sich um eine eingeschränkte Verfeinerung.

Satz 4.7 (Priorisierte Kombination verfeinert ihre Teildienste eingeschränkt)**.** *Die priorisierte Kombination von zwei Diensten $S = S_1 \parallel^{S_P} S_2$ ist eine eingeschränkte Verfeinerung der beiden, d.h. es gibt zwei Mengen R_1 und R_2 von Systemabläufen, so dass die Relationen $S_1 \succeq_{R_1} S$ und $S_2 \succeq_{R_2} S$ gelten. Die Mengen R_1 und R_2 sind wie folgt definiert:*

$$R_1 \stackrel{\text{def}}{=} \{\rho \in \langle\!\langle S \rangle\!\rangle \mid \exists t \in \mathcal{T}_P, i \in \mathbb{N} : \rho.i, \rho.(i+1)' \vdash t \wedge p(t) = 2\}$$
$$R_2 \stackrel{\text{def}}{=} \{\rho \in \langle\!\langle S \rangle\!\rangle \mid \exists t \in \mathcal{T}_P, i \in \mathbb{N} : \rho.i, \rho.(i+1)' \vdash t \wedge p(t) = 1\}.$$

R_1 (bzw. R_2) umfasst alle Systemabläufe, in denen der Dienst S_2 (bzw. S_1) mindestens einmal priorisiert war.

Beweis. Wegen der Symmetrie der beiden Fälle wird nur die Relation $S_1 \succeq_{R_1} S$ gezeigt. Gemäß Definition 4.15 auf Seite 65 muss nachgewiesen werden, dass I_1 `subtype` I und O_1 `subtype` O gelten. Dies folgt aus Gleichungen $I = I_1 \cup I_2$ und $O = O_1 \cup O_2$.

Außerdem muss folgende Relation für die Menge R_1 gezeigt werden:

$$\{\rho \in \langle\!\langle S \rangle\!\rangle \setminus R_1 \mid \exists \rho_1 \in \langle\!\langle S_1 \rangle\!\rangle : \rho_1 \stackrel{I_1}{=} \rho\} \stackrel{I_1 \cup O_1}{\subseteq} \langle\!\langle S_1 \rangle\!\rangle.$$

Dafür sind gemäß Definition 4.6 auf Seite 58 folgende zwei Eigenschaften zu beweisen:

$$\forall \rho \in \langle\langle S \rangle\rangle \setminus R : \rho.0 \vdash \mathcal{I} \Rightarrow \rho.0 \vdash \mathcal{I}_1 \text{ und} \tag{4.14a}$$

$$\begin{aligned} &\forall \rho \in \langle\langle S \rangle\rangle \setminus R, i \in \mathbb{N}, \exists \rho_1 \in \langle\langle S_1 \rangle\rangle : \\ &\quad \rho.i \overset{I_1}{=} \rho_1.i \Rightarrow \exists \beta_1 \in \mathrm{Succ}_1(\rho_1.i) : \rho.(i+1) \overset{I_1 \cup O_1}{=} \beta_1 \vee \\ &\quad \rho.i \overset{I_1}{=} \rho_1.i \wedge \neg \mathrm{En}(\rho.i) \Rightarrow \neg \mathrm{En}_1(\rho_1.i). \end{aligned} \tag{4.14b}$$

- Eigenschaft (4.14a) folgt trivialerweise aus der Definition der Kombination initialer Belegungen, nämlich aus $\mathcal{I} \overset{\text{def}}{=} \mathcal{I}_P \wedge \mathcal{I}_1 \wedge \mathcal{I}_2$ folgt $\mathcal{I} \Rightarrow \mathcal{I}_1$.
- Eigenschaft (4.14b) für unendliche Abläufe ist induktiv über $\mathbb{N}$ zu beweisen. Zuerst wird die Belegung $\rho.0$ betrachtet. Aus $\rho \in \langle\langle S \rangle\rangle \setminus R$ folgt $\rho.1 \in \mathrm{Succ}(\rho.0) \wedge \forall t_p \in \mathcal{T}_P : \rho.0, \rho.1' \vdash t_p \Rightarrow p(t_p) \neq 2$. Aus Eigenschaft (4.14a) folgt, dass $\forall \rho \in \langle\langle S \rangle\rangle \setminus R : \exists \rho_1 \in \langle\langle S_1 \rangle\rangle : \rho.0 \overset{V_1}{=} \rho_1.0$. Daraus wiederum folgt, dass gemäß Gleichung (4.11) auf Seite 69 die Belegung $\rho.1$ aus einer der folgenden drei Mengen sein muss.
  1. $\{\beta \mid \exists t \in \mathcal{T} : \forall t_p \in \mathcal{T}_P : (\alpha, \beta' \vdash t \wedge \neg t_p)\}$. Daraus folgt, dass $\exists t \in \mathcal{T} : \rho.0, \rho.1' \vdash t$, wobei α und β durch $\rho.0$ und $\rho.1$ entsprechend ersetzt werden. Aus $\rho.0 \overset{I_1}{=} \rho_1.0$ und $t = t_1 \wedge t_2$ folgt, dass $\exists t_1 \in \mathcal{T}_1 : \rho_1.0, \rho_1.1' \vdash t_1 \wedge \rho.1 \overset{V_1}{=} \rho_1.1$. D.h. $\rho_1.1 \in \mathrm{Succ}(\rho_1.0)$.
  2. $\{\beta \mid \exists t \in \mathcal{T}, t_p \in \mathcal{T}_P : (\alpha, \beta' \vdash t \wedge t_p) \wedge p(t_p) = 0\}$. Der Beweis ist analog zum vorherigen Fall.
  3. $\{\beta \mid \exists t_1 \in \mathcal{T}_1, t_p \in \mathcal{T}_P : (\alpha, \beta' \vdash t_1 \wedge t_p) \wedge p(t_p) = 1\}$. Der Beweis ist analog zum vorherigen Fall.

  Induktionschritt: Aus der Annahme, dass Eigenschaft (4.14b) für i gilt, folgt die Aussage $\forall \rho \in \langle\langle S \rangle\rangle \setminus R : \exists \rho_1 \in \langle\langle S_1 \rangle\rangle : \rho.i \overset{V_1}{=} \rho_1.i$. Unter dieser Voraussetzung ist der Beweis für $(i+1)$ genau nach demselben Schema wie der für $i = 0$ zu führen.
- Eigenschaft (4.14b) für endliche Abläufe. Der Beweis geht analog zu dem letzten Fall aus dem Beweis von Satz 4.4.

Daraus folgt $S_1 \succeq_{R_1} S$. □

4.5. Systemmodi

In Abschnitt 3.4.3 auf Seite 47 wurde bereits erläutert, wie das Systemverhalten durch die priorisierte Dienstkombination in Systemmodi unterteilt werden kann. Im Folgenden wird der Begriff „Systemmodus“ formal definiert.

Ein Systemmodus ist durch ein Prädikat gegeben, das eine Teilmenge von Diensten beschreibt, welche für eine gegebene Belegung α *aktiviert* sind. Ein Dienst ist aktiviert, wenn eine Eingabe anliegt, für die die mit diesem Dienst verbundenen Priorisierungsdienste die Ausführung dieses Dienstes ermöglichen. Ein Priorisierungsdienst S_P in der

Kombination $PC = S_1 \parallel^{S_P} S_2$ unterteilt das Systemverhalten von PC in drei *Priorisierungsmodi*, in denen entweder S_1, S_2 oder beide aktiviert sind. Im Folgenden werden drei logische Prädikate definiert, die für eine priorisierte Dienstkombination und eine Belegung α die Priorisierungsmodi beschreiben. Das Prädikat $isAct_i$ (mit $i \in \{1,2\}$) legt fest, ob der Dienst S_i momentan aktiviert ist. Das Prädikat $isAct_0$ ist wahr genau dann, wenn die beiden Dienste gleichzeitig aktiviert sind:

$$isAct_0(S_P, \alpha) \overset{\text{def}}{\Leftrightarrow} \text{Succ}_P(\alpha) = \emptyset \vee \exists t \in \mathcal{T}_P, \beta \in \text{Succ}_P(\alpha) : \alpha, \beta' \vdash t \wedge p(t) = 0,$$
$$isAct_i(S_P, \alpha) \overset{\text{def}}{\Leftrightarrow} \exists t \in \mathcal{T}_P, \beta \in \text{Succ}_P(\alpha) : \alpha, \beta' \vdash t \wedge p(t) = i.$$

Die Priorisierungsmodi legen lediglich fest, ob ein Dienst in einer Kombination „lokal“ aktiviert ist. Aus Sicht der gesamten Diensthierarchie kann die Aktivierung eines Dienstes jedoch von mehreren Priorisierungsdiensten abhängen. Außerdem kann es vorkommen, dass ein aktivierter Dienst keine Transition ausführen kann (d.h. $\text{Succ}_i(\alpha) = \emptyset$).

Nun wird für das Gesamtverhalten *Sys* und die Belegung α die Menge derjenigen Dienste definiert, die im nächsten Schritt eine Transition ausführen können. Diese Menge ist durch $Act(Sys, \alpha)$ bezeichnet und ist induktiv über atomare und (priorisiert) kombinierte Teildienste von *Sys* definiert:

$$S \in Act(S, \alpha) \Leftrightarrow \text{Succ}_S(\alpha) \neq \emptyset,$$
$$S \in Act(S_1 \parallel S_2, \alpha) \Leftrightarrow S \in Act(S_1, \alpha) \vee S \in Act(S_2, \alpha),$$
$$S \in Act(S_1 \parallel^{S_P} S_2, \alpha) \Leftrightarrow \begin{cases} S \in Act(S_i, \alpha) & \text{falls } i \in \{1,2\} \wedge isAct_i(S_P, \alpha), \\ S \in Act(S_1 \parallel S_2, \alpha) & \text{falls } isAct_0(S_P, \alpha). \end{cases}$$

Dies bedeutet, ein Dienst ist genau dann ausführbar, wenn

- der Dienst eine ausführbare Transition hat und
- falls der Dienst ein Teildienst einer priorisierten Kombination ist, der Dienst in der Kombination aktiviert ist und
- alle Vorfahren des Dienstes in der Diensthierarchie in den entsprechenden Kombinationen aktiviert sind.

Nun wird die formale Definition eines Systemmodus eingeführt. Analog zu den traditionellen Systemmodi (wie z.B. denen von Leveson [Lev95, S. 370]), die Äquivalenzklassen von Systemzuständen definieren, beschreiben unsere Systemmodi Teilmengen von ausführbaren Diensten.

Definition 4.17 (Systemmodus)**.** Ein Modus des Systems *Sys* für die Belegung α definiert eine Teilmenge der für die Belegung α ausführbaren Teildienste von *Sys*. Ein Modus ist durch ein Prädikat $mode(\alpha)$ gegeben, das eine Teilmenge von $Act(Sys, \alpha)$ definiert. $Act(Sys, \alpha)$ ist die Supermenge der durch Priorisierungsdienste aktivierten Dienste. □

Beispiel 4.4 (Systemmodi)**.** Der Systemmodus „Aktiviert“ aus Beispiel 3.12 auf Seite 47 ist durch das Prädikat $mode_a(\alpha) \overset{\text{def}}{=} (Aktiviert \in Act(ACC, \alpha))$ gegeben. Dies

bedeutet, dass im Modus „Aktiviert“ und für die Belegung α Dienste `Eingeschaltet` und `Aktiviert` durch die entsprechenden Priorisierungsdienste aktiviert und der Dienst `Aktiviert` ausführbar sein müssen (vgl. Abbildung 3.18). □

In der vorliegenden Arbeit werden Systemmodi vor allem in der Korrektheitsprüfung verwendet, bei der die Erfüllung von Systemeigenschaften überprüft wird. Es gibt Eigenschaften, die nicht immer, sondern in definierten Systemmodi gelten müssen, d.h. sie sind keine Systeminvarianten. Dies bedeutet, statt einer logischen Formel φ muss die Formel $mode \Rightarrow \varphi$ erfüllt sein. Mehr Details dazu folgen in Abschnitt 5.3.

4.6. Zusammenfassung

Während die denotationale, FOCUS-basierte Semantik nur kurz eingeführt wird, liegt der Schwerpunkt dieses Kapitels auf der operationellen Semantik der dienstbasierten Spezifikationstechnik. Der wesentliche Vorteil einer zustandsbasierten Spezifikation ist die Möglichkeit, sie automatisch zu analysieren. Ein weiterer Vorteil der eingeführten Technik ist, dass es sich um einen konstruktiven Ansatz handelt. Durch die beiden formal definierten Operatoren kann ein kombinierter Dienst aus zwei gegebenen Diensten (und evtl. einer Priorisierung zwischen diesen) konstruiert werden.

Analog zu den traditionellen Systemmodi, wurden in diesem Kapitel Modi als eine Teilmenge von den in einem gegebenen Zustand ausführbaren Diensten definiert. Diese Modi werden im nächsten Kapitel bei der Korrektheitsprüfung eines Systems verwendet.

Weiterhin wurden die wichtigsten algebraischen Eigenschaften der Kombinationsoperatoren nachgewiesen. Die einfache Kombination ist kommutativ, assoziativ und idempotent. Unter anderem bedeutet das, dass die Reihenfolge, in der Dienste kombiniert werden, sich nicht auf das resultierende Gesamtergebnis auswirkt. Die priorisierte Kombination ist nicht assoziativ, jedoch kommutativ, distributiv und idempotent. Dies hat zur Folge, dass die Struktur der Diensthierarchie Auswirkung auf das Gesamtverhalten hat, falls die Spezifikation Priorisierungen enthält.

Außerdem wurde gezeigt, dass der kombinierte Dienst alle seine Teildienste (eingeschränkt) verfeinert. Dies bedeutet, dass das Hinzufügen eines Dienstes zu einer vorhandenen Spezifikation die Gesamtfunktionalität um weiteres Verhalten erweitert, d.h. die gesamte dienstbasierte Spezifikation einzelne Teilspezifikationen verfeinert.

Im folgenden Kapitel steht die Konsistenz der dienstbasierten Spezifikation im Mittelpunkt. Unter anderem wird nachgewiesen, dass das mit der Arbeit vorgestellte Engineeringmodell eine Instanz der Broyschen Systemtheorie ist, d.h. die eingeführte konsistente Diensthierarchie eine annotierte Diensthierarchie nach Broy ist.

KAPITEL 5

Konsistenz und Korrektheit

Der im vorgestellten Ansatz verwendete Kombinationsoperator lässt die Kombination von Diensten zu, die an denselben Ausgabeports im selben Zeitintervall unterschiedliche Werte erzeugen. Dadurch kann die Widerspruchsfreiheit (Konsistenz) der resultierenden Dienstkombination durch Konstruktion nicht garantiert werden. Um die Eindeutigkeit einer Spezifikation sicherzustellen, werden in diesem Kapitel mehrere automatische Analyseverfahren erarbeitet. Zusätzlich zur Konsistenz- und Korrektheitsprüfung von Anforderungen werden außerdem verschiedene Algorithmen zur Herstellung der Konsistenz und der Korrektheit einer Spezifikation vorgestellt.

Außerdem wird in diesem Kapitel die formale Integration des vorgestellten Ansatzes in die Broysche Theorie abgeschlossen. Es wird gezeigt, dass die eingeführte operationelle Diensthierarchie eine Instanz der deskriptiven Diensthierarchie von Broy ist.

Inhalt

Der dienstbasierte Ansatz ermöglicht eine Zergliederung der Gesamtfunktionalität eines Systems in einzelne Nutzerfunktionen. Der dabei verwendete Kombinationsoperator lässt die Kombination widersprüchlicher Dienste zu, um die Modellierung von potentiell widersprüchlichen Anforderung unterschiedlicher Stakeholder zu unterstützen. Dies hat zur Folge, dass die Widerspruchsfreiheit der resultierenden Dienstkombination durch Konstruktion nicht sichergestellt werden kann (engl. correct-by-construction). Eine inkonsistente Spezifikation birgt die Gefahr, dass sie mehrdeutig und formal nicht realisierbar ist. Daher sind Analyseverfahren zur Identifizierung von *Konflikten* zwischen Diensten (auch als unerwünschte „feature interactions“ bekannt) notwendig. Dadurch wird sichergestellt, dass das kombinierte Systemverhalten die Anforderungen einzelner Stakeholder erfüllt.

Laut Leveson [Lev95, S. 362] besteht eine Anforderungsspezifikation aus drei Komponenten: der Basisfunktionalität, Randbedingungen (engl. constraints) und Qualitätszielen. Die Basisfunktionalität, d.h. eine Sammlung geforderter Interaktionsmuster, ist im vorgestellten Ansatz durch eine Diensthierarchie gegeben. Die Randbedingungen schränken das Systemverhalten weiter ein und stammen typischerweise von Sicherheits- und Robustheitsanforderungen, physikalischen Begrenzungen der Umgebung oder Gesetzgebungen. Die Randbedingungen, oft als LTL-Formeln ausgedrückt, fügen der Spezifikation keine neue Funktionalität hinzu, sondern reduzieren die Anzahl der gültigen Systemabläufe. Qualitätsziele definieren nichtfunktionale Anforderungen an das System und müssen durch den Entwickler bei der Gestaltung des Systems berücksichtigt werden. Sie werden in der vorliegenden Arbeit nicht behandelt.

Angelehnt an die Definition von Leveson wird bei der eingeführten Anforderungsanalyse zwischen der Konsistenz- und der Korrektheitsprüfung unterschieden. In Abschnitt 5.2 wird auf die Konsistenz der Basisfunktionalität eingegangen. Angelehnt an Nuseibeh et al. [NER00] ist eine Spezifikation konsistent, wenn deren Dienste keine vordefinierte Konsistenzbedingung verletzen. Ein Beispiel einer *Inkonsistenz* ist ein Widerspruch zwischen Diensten, d.h. wenn zwei Dienste im selben Zeitintervall an einem gemeinsamen Ausgabeport unterschiedliche Werte erzeugen. In Abschnitt 5.3 steht die Korrektheit einer Spezifikation im Mittelpunkt. Eine Spezifikation wird als korrekt bezeichnet, wenn die Diensthierarchie alle Randbedingungen erfüllt. Nach Heitmeyer et al. [HJL96] heißen die Konsistenzbedingungen anwendungsunabhängige und die Randbedingungen anwendungsspezifische Eigenschaften.

Ein wesentlicher Vorteil eines operationellen gegenüber einem denotationalen Ansatz wie z.B. dem von Broy [Bro07a] ist die Möglichkeit, die Anforderungsanalyse automatisch durchführen zu können. Zu diesem Zweck werden sowohl der Nachweis der Konsistenz als auch der der Korrektheit einer Spezifikation auf das Erreichbarkeitsproblem in Transitionssystemen reduziert. Damit können Konsistenz- und Randbedingungen durch Model Checking [CGP99] automatisch überprüft werden.

Die Identifikation von Inkonsistenzen und Verletzungen von Randbedingungen ist nur der erste Schritt in Richtung einer konsistenten und korrekten Spezifikation. Im dar-

auffolgenden Schritt müssen identifizierte Konflikte (am besten automatisch) eliminiert werden. Zu diesem Zweck werden zunächst die Ursachen von Konflikten untersucht, um festzustellen, welche Arten von ihnen sich automatisch lösen lassen. In komplexen multifunktionalen Software-Systemen werden die meisten Konflikte durch unberücksichtigte Wechselwirkungen zwischen Nutzerfunktionen verursacht. Die Eliminierung dieser Art von Konflikten durch die automatische Generierung von Priorisierungsdiensten, welche die unberücksichtigten Wechselwirkungen darstellen, bildet den Schwerpunkt von Abschnitt 5.4. Dieses Verfahren wird sowohl zur Herstellung der Konsistenz als auch der Korrektheit einer Spezifikation eingesetzt. Der automatisch erzeugte Priorisierungsdienst stellt sicher, dass alle Randbedingungen erfüllt sind und keine Inkonsistenzen innerhalb einer Diensthierarchie auftreten.

In Abschnitt 5.5 wird die formale Integration des vorgestellten Ansatzes in die Broysche Theorie abgeschlossen, indem gezeigt wird, dass eine konsistente operationelle Diensthierarchie eine Instanz der deskriptiven Diensthierarchie von Broy ist.

5.1. Grundlagen

Dieser Abschnitt definiert verschiedene grundlegende Prädikate und Operationen auf Transitionssystemen, die im Rest des Kapitels verwendet werden. Zunächst werden lokale und erreichbare Zustände eines Systems definiert. Anschließend wird gezeigt, wie ein partielles System durch einen Fehlerzustand vervollständigt werden kann. Dies ist notwendig, um die Überprüfung einer Eigenschaft auf das Erreichbarkeitsproblem reduzieren zu können – die Eigenschaft ist verletzt, wenn ein Fehlerzustand erreicht ist.

5.1.1. Lokale und erreichbare Zustände

Ein *lokaler Zustand* eines Dienstes ist durch das Prädikat $loc(\alpha)$ mit $\alpha \in \Lambda(V)$ gegeben. Für eine gegebene Belegung α definiert das Prädikat die Menge aller Belegungen, welche die lokalen Variablen aus der Menge L auf dieselben Werte abbilden wie α:

$$loc(\alpha) \stackrel{\text{def}}{=} \{\beta \in \Lambda(V) \mid \beta \stackrel{L}{=} \alpha\}.$$

Beispielsweise gibt es im Zustandsdiagramm aus Abbildung 5.1 zwei lokale Zustände $z = 1$ und $z = 2$.

Die Menge aller *erreichbaren Zustände* des Dienstes S ist durch folgendes Prädikat gegeben:

$$\mathrm{Attr}(S) \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}, \alpha \vdash \mathcal{I}} \mathrm{Succ}^n(\alpha),$$

wobei die Menge $\mathrm{Succ}^n(\alpha)$ induktiv wie folgt definiert ist:

$$\mathrm{Succ}^0(\alpha) \stackrel{\text{def}}{=} \{\alpha\},$$
$$\mathrm{Succ}^{n+1}(\alpha) \stackrel{\text{def}}{=} \mathrm{Succ}(\mathrm{Succ}^n(\alpha)), \text{ wobei } \mathrm{Succ}(A) \stackrel{\text{def}}{=} \bigcup_{\gamma \in A} \mathrm{Succ}(\gamma).$$

5.1.2. Vervollständigung durch einen Fehlerzustand

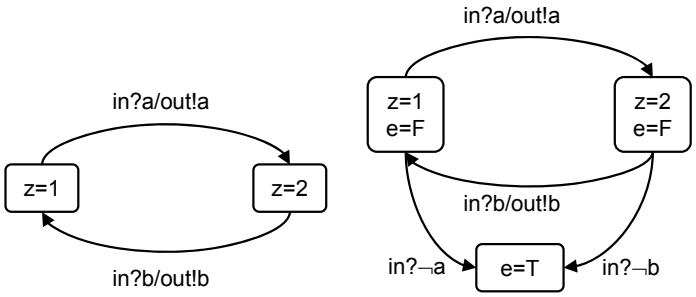
Ein partielles Transitionssystem kann durch einen *Fehlerzustand* vervollständigt werden. Fehlerzustände sind alle Belegungen α, die eine spezielle Variable e ($\mathit{type}(e) = \mathbb{B}\mathit{ool}$, $e \notin V$) auf den Wert *True* abbilden. Für eine Belegung geht das erweiterte Transitionssystem in den Fehlerzustand über, falls es im ursprünglichen System für diese Belegung keine ausführbare Transition gibt. Für ein System $S = (V, \mathcal{I}, \mathcal{T})$ ist seine Erweiterung um einen Fehlerzustand wie folgt definiert:

$$\hat{S} \stackrel{\text{def}}{=} (V \cup \{e\}, \hat{\mathcal{I}}, \hat{\mathcal{T}}),$$

wobei für alle $\alpha, \beta \in \Lambda(V \cup \{e\})$ folgende Aussagen gelten:

$$\alpha \vdash \hat{\mathcal{I}} \stackrel{\text{def}}{\Leftrightarrow} \big(\alpha \vdash \mathcal{I} \Leftrightarrow \neg\alpha(e)\big),$$
$$\alpha, \beta' \vdash \hat{\mathcal{T}} \stackrel{\text{def}}{\Leftrightarrow} \big(\alpha, \beta' \vdash \mathcal{T} \wedge \neg\alpha(e) \wedge \neg\beta(e)\big) \vee \big(\neg(\alpha, \beta' \vdash \mathcal{T}) \wedge \neg\alpha(e) \wedge \beta(e)\big).$$

Es ist offensichtlich, dass jeder Ablauf von S auch ein Ablauf von $\hat{S}$ ist und ein Ablauf aus $\langle\langle \hat{S} \rangle\rangle$ aber nicht aus $\langle\langle S \rangle\rangle$ eine Belegung α erreicht, für die $\alpha(e)$ zu wahr evaluiert. Die Letzteren werden *Fehlerabläufe* genannt. Es gibt keine Transition, die aus einem Fehlerzustand in einen anderen Zustand führt.



(a) Partielles System S (b) Erweitertes System $\hat{S}$

Abbildung 5.1.: Vervollständigung eines Transitionssystems

Beispiel 5.1 (Vervollständigung durch einen Fehlerzustand)**.** Sei ein partielles System S durch das Zustandsdiagramm aus Abbildung 5.1(a) gegeben. Das System hat keine ausführenden Transitionen für Belegungen α_1 mit $\alpha_1(z) = 1 \wedge \alpha_1(in) \neq a$ und α_2 mit

$\alpha_2(z) = 2 \wedge \alpha_2(in) \neq b$. Die Vervollständigung des Systems durch einen Fehlerzustand ist in Abbildung 5.1(b) dargestellt. Für die beiden Belegungen α_1 und α_2 hat das System $\hat{S}$ jeweils eine ausführbare Transition, die in den Fehlerzustand führt. □

5.2. Konsistenz

„Ein Sack voller guter Anforderungen macht noch lange keine gute Anforderungsspezifikation“ [Rup07, S. 31]. Einzelne Anforderungen dürfen bestimmte Konsistenzbedingungen nicht verletzen, welche die Eindeutigkeit der Spezifikation sicherstellen. Während Abschnitt 5.2.1 *Inkonsistenzen* zwischen Anforderungen informell beschreibt, werden sie in Abschnitt 5.2.2 formal definiert. Anschließend wird in Abschnitt 5.2.3 gezeigt, wie die Konsistenzprüfung anhand der Simulation automatisch durchgeführt werden kann.

5.2.1. Inkonsistenzen zwischen Anforderungen

Eines der wesentlichen Merkmale des vorgestellten Ansatzes ist die Möglichkeit, ein System aus unterschiedlichen Benutzerperspektiven zu spezifizieren. Anforderungen, die von unterschiedlichen Stakeholdern definiert sind, können sich widersprechen, d.h. sie können potenziell inkonsistent sein. Easterbrook et al. definieren eine Inkonsistenz als „the interference in the goals of one party caused by the actions of another“ [EFKN94]. Enthält eine Spezifikation mindestens eine Inkonsistenz, ist sie inkonsistent.

Nun stellt sich die Frage, wann zwei Anforderungen inkonsistent sind. Sind Anforderungen mit aussagenlogischen Ausdrücken beschrieben, ist die Definition einer Inkonsistenz relativ einfach. Zwei Anforderungen sind inkonsistent, falls aus den logischen Formeln ein Widerspruch abgeleitet werden kann, z.B. : $X \wedge \neg X$. Allerdings handelt es sich bei Anforderungen selten um Invarianten, die immer gelten müssen. Vielmehr wird durch ein Interaktionsmuster das Verhalten nur für eine Teilmenge aller Eingaben definiert. Für die Eingaben außerhalb seines Definitionsbereichs stellt ein Muster keine Anforderungen an das Verhalten und kann somit in keinem Widerspruch zu anderen Mustern stehen (vgl. Bemerkung 3.4 auf Seite 32). Um diese Tatsache zu berücksichtigen, wird die Definition einer Inkonsistenz verallgemeinert: Zwei Teile einer Spezifikation sind inkonsistent, falls sie eine gegebene Konsistenzbedingung verletzen.

Konsistenzbedingungen können sowohl syntaktische als auch semantische Eigenschaften der Spezifikation betreffen und sind meistens domänenspezifisch. Eine mögliche Klassifizierung von Inkonsistenzen zwischen Anforderungen wurde in [vLLD98] von van Lamsweerde et al. eingeführt. In der vorliegenden Arbeit werden ausschließlich funktionale Inkonsistenzen in den Spezifikationen reaktiver Systeme betrachtet.

5.2.2. Konsistente Spezifikation

Eine dienstbasierte Spezifikation ist eine Kombination von eigenständig definierten Teildiensten. Es kann dabei vorkommen, dass überlappende Dienste an ihren gemeinsamen Ausgabeports zum selben Zeitpunkt unterschiedliche Ausgabewerte erzeugen. Dies hat zur Folge, dass die Integration von Interaktionsmustern Widersprüche verursachen und infolgedessen zu einer inkonsistenten Gesamtspezifikation führen kann. Im Folgenden wird die Inkonsistenz zwischen zwei Diensten definiert.

***Bemerkung* 5.1** (Inkonsistenzen zwischen Systemen)**.** In der vorliegenden Arbeit werden ausschließlich Inkonsistenzen zwischen Interaktionsmustern ein und desselben Systems betrachtet, die bei der *Dienstkombination* entstehen. Inkonsistenzen zwischen Systemen in einer *Komposition* wurden bereits ausgiebig erforscht, z.B. in [dAH01, CFN03, GS05, GGMC+07], und liegen deswegen nicht im Fokus der Betrachtung. □

Gemäß der Definition aus Abschnitt 4.3.2 auf Seite 66 führen in einer Dienstkombination all diejenigen Dienste je eine Transition aus, welche für die aktuelle Eingabe ausführbar sind und deren auszuführende Transitionen untereinander nicht widersprüchlich sind. Daraus folgt, dass das kombinierte Transitionssystem keine Transitionen enthält, die aus den untereinander widersprüchlichen Transitionen der Teildienste kombiniert sind. Transitionen sind widersprüchlich, falls sie zum selben Zeitpunkt für dieselben Eingaben unterschiedliche Ausgaben fordern. Es kann dabei Eingaben geben, für die alle Transitionen der beiden Dienste paarweise widersprüchlich sind. Dies hat zur Folge, dass die Dienstkombination für diese Eingabe keine ausführbare Transition hat, obwohl die Teildienste diese Eingabe verarbeiten können – der Definitionsbereich der Kombination ist somit kleiner als die Vereinigung der Definitionsbereiche der beiden Teildienste. Dies widerspricht jedoch der Kernidee des dienstbasierten Ansatzes, die besagt, dass mit jedem weiteren Dienst das definierte Verhalten erweitert wird. Diese Beobachtung führt uns zur Definition einer Inkonsistenz zwischen zwei Diensten.

Definition 5.1 (Inkonsistenz in einfacher Kombination)**.** In einer Kombination $S = S_1 \| S_2$ sind die beiden Dienste in einem erreichbaren lokalen Zustand $loc(\alpha)$ inkonsistent, falls die Kombination S in diesem Zustand nicht alle Eingaben aus dem Definitionsbereich des Dienstes S_i mit $i \in \{1, 2\}$ verarbeiten kann:

$$loc(\alpha) \in Incons(S_i, S) \overset{\text{def}}{\Leftrightarrow} \{\beta \mid \beta \in loc(\alpha) \wedge \mathrm{En}_S(\alpha)\} \overset{I_i}{\subset} \{\beta \mid \beta \in loc(\alpha) \wedge \mathrm{En}_{S_i}(\alpha)\}.$$

Die Menge $Incons(S_i, S)$ umfasst alle lokalen Zustände, in denen zwischen dem Dienst S_i und den restlichen Diensten der Kombination eine Inkonsistenz vorliegt. □

***Bemerkung* 5.2** (Begriff „Inkonsistenz“)**.** Der Begriff „Inkonsistenz zwischen Anforderungen“ stammt aus den Arbeiten von Easterbrook et al. [EFKN94].

Darüber hinaus steht Definition 5.1 nicht im Widerspruch zur klassischen Verwendung des Begriffs in der Logik. Im Sinne der Theorie der Logik gesprochen, ist eine Menge logischer Formeln inkonsistent, falls es dafür kein Modell gibt. Gemäß der eingeführten

Definition sind zwei Dienste inkonsistent, falls es kein Modell gibt, das alle Eingaben aus den Definitionsbereichen der beiden Dienste gleichzeitig verarbeiten kann. □

In der priorisierten Kombination muss das Verhalten des kombinierten Dienstes nicht für alle Eingaben mit dem Verhalten der beiden Teildienste übereinstimmen. Für die restlichen Eingaben stellt der unpriorisierte Teildienst keine Anforderungen an die Ausgaben. Es ist dabei offensichtlich, dass nur im ersten Fall eine Inkonsistenz zwischen den Teildiensten auftreten kann. Demzufolge kann in einer priorisierten Kombination $S = S_1 \parallel^{S_P} S_2$ eine Inkonsistenz nur dann auftreten, wenn die beiden Teildienste durch den Priorisierungsdienst S_P gleichzeitig aktiviert werden, d.h. falls das Prädikat $isAtc_0(S_P, \alpha)$ aus Abschnitt 4.5 auf Seite 78 wahr ist. Entsprechend wird die Definition einer Inkonsistenz erweitert:

Definition 5.2 (Inkonsistenz in priorisierter Kombination)**.** In einer Kombination $S = S_1 \parallel^{S_P} S_2$ sind die beiden Dienste im erreichbaren lokalen Zustand $loc(\alpha)$ inkonsistent, falls die beiden gleichzeitig aktiviert und nach Definition 5.1 inkonsistent sind:

$$loc(\alpha) \in Incons(S_i, S) \overset{\text{def}}{\Leftrightarrow} isAct_0(S_P, \alpha) \\ \wedge \{\beta \mid \beta \in loc(\alpha) \wedge \mathrm{En}_S(\alpha)\} \overset{I_i}{\subset} \{\beta \mid \beta \in loc(\alpha) \wedge \mathrm{En}_{S_i}(\alpha)\}.$$

□

Gemäß Definitionen 5.1 und 5.2 ist die Menge aller Inkonsistenzen zwischen zwei Diensten in einer Kombination $S_1 \parallel S_2$ oder $S_1 \parallel^{S_P} S_2$ wie folgt definiert:

$$\mathcal{INC}(S_1, S_2) \overset{\text{def}}{=} Incons(S_1, S_1 \parallel S_2) \cup Incons(S_2, S_1 \parallel S_2) \text{ und} \tag{5.1a}$$

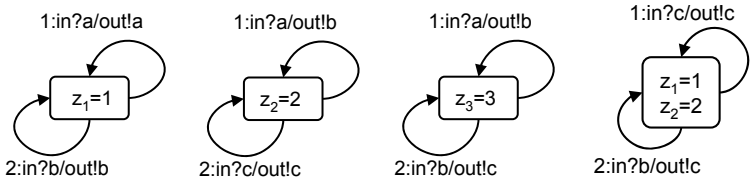
$$\mathcal{INC}(S_1, S_2, S_P) \overset{\text{def}}{=} Incons(S_1, S_1 \parallel^{S_P} S_2) \cup Incons(S_2, S_1 \parallel^{S_P} S_2). \tag{5.1b}$$

Eine Unterklasse der oben eingeführten Inkonsistenzen ist die Menge von Deadlocks (oder Verklemmungen). Obwohl die eingeführte Semantik endliche Abläufe zulässt, sollen die meisten reaktiven Systeme unendlich lang laufen. Folglich sind endliche Abläufe unerwünscht und weisen auf Inkonsistenzen in der Spezifikation hin. Ein Zustand, für den in der Kombination von zwei Diensten kein Nachfolgezustand existiert, wird als Deadlock bezeichnet.

Definition 5.3 (Deadlock)**.** Ein lokaler Zustand $loc(\alpha)$ ist ein Deadlock zwischen zwei Diensten, falls es in diesem Zustand keine Eingabe gibt, für welche ihre Kombination eine Transition ausführen kann. Mit anderen Worten ist die Kombination S für keine Belegung aus $loc(\alpha)$ ausführbar: $\bigwedge_{\beta \in loc(\alpha)} \neg \mathrm{En}_S(\beta)$. □

Die Inkonsistenzen werden zunächst anhand eines einfachen Beispiels erläutert. Anschließend wird ein Ausschnitt aus der Spezifikation der laufenden Fallstudie gezeigt.

Beispiel 5.2 (Inkonsistenzen)**.** Seien Dienste `D1`, `D2` und `D3` durch die entsprechenden Zustandsdiagramme aus Abbildungen 5.2(a), 5.2(b) und 5.2(c) gegeben. Die Kombination der Dienste `D1` und `D2` ergibt das Zustandsdiagramm aus Abbildung 5.2(d). Die

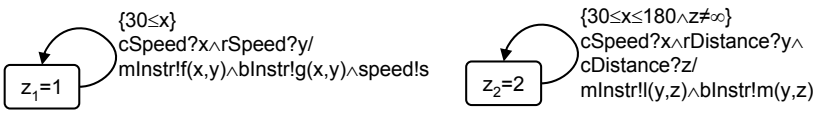


(a) `D1` (b) `D2` (c) `D3` (d) `D1‖D2`

Abbildung 5.2.: Dienste aus Beispiel 5.2

Transitionen $in?a/out!a$ des Dienstes `D1` und $in?a/out!b$ des Dienstes `D2` widersprechen sich, da sie auf dieselbe Eingabe unterschiedliche Ausgaben fordern. Gemäß der Definition der Succ-Funktion aus Gleichung (4.10) auf Seite 67 gehört die Kombination der beiden nicht zur Transitionsmenge des kombinierten Transitionssystems. Damit gibt es in der Dienstkombination keine ausführbare Transition für die Eingabe a, obwohl diese Eingabe im Definitionsbereich beider Teildienste liegt. Da die Belegung $\alpha(in) = a$ erreichbar ist und Folgendes gilt: $\mathrm{En}_{D1}(\alpha) \neq \emptyset \wedge \mathrm{En}_{D2}(\alpha) \neq \emptyset \wedge \mathrm{En}_{D1\|D2}(\alpha) = \emptyset$, liegt im Zustand $loc(\alpha)$ mit $\alpha(z_1) = 1 \wedge \alpha(z_2) = 2$ eine Inkonsistenz vor.

Die Kombination der Dienste `D1` und `D3` ergibt ein Transitionssystem, dessen Transitionsmenge leer ist. Da die beiden Dienste auf jede Eingabe unterschiedliche Ausgaben fordern, sind keine Transitionen kombinierbar. Demzufolge gibt es in der Kombination für keine Eingabe eine ausführbare Transition, d.h. die Dienste sind in einem Deadlock. □



(a) `Tempomat/G` (b) `Abstandsregelung/G`

Abbildung 5.3.: Dienste aus Beispiel 5.3

Beispiel 5.3 (Inkonsistenz zwischen `Tempomat` und `Abstandsregelung`)**.** Als Beispiel dienen wieder die beiden Dienste `Tempomat/G` und `Abstandsregelung/G` aus Beispielen 3.7 auf Seite 32 und 3.9 auf Seite 36. Zur Erinnerung sind ihre Verhaltensspezifikationen in Abbildung 5.3 erneut skizziert. Die beiden Dienste haben drei gemeinsame Ports `cSpeed`, `mInstr` und `bInstr`. Liegt der Wert am Port `cSpeed` im Intervall zwischen 30 und 180 und ist der Wert am Port `cDistance` ungleich ∞, haben die beiden Dienste jeweils eine ausführbare Transition. Es gibt jedoch Eingaben, für welche sie gleichzeitig unterschiedliche Werte an den Ausgabeports `mInstr` und `bInstr` fordern. Daraus folgt, dass für diese Eingaben das kombinierte Transitionssystem keine ausführbare Transition hat. Das wiederum bedeutet, dass es Eingaben gibt, für welche die Teildienste eine

Transition ausführen können, deren Kombination jedoch nicht. Dies hat zur Folge, dass die Dienste `Tempomat/G` und `Abstandsregelung/G` inkonsistent sind. □

Aufbauend auf den Definitionen von Inkonsistenzen wird im Folgenden die konsistente dienstbasierte Spezifikation definiert.

Definition 5.4 (Konsistente dienstbasierte Spezifikation)**.** Eine Diensthierarchie $(D \cup P, w, V)$ ist konsistent, falls folgende Eigenschaften gelten:

- Es gibt eine Belegung, welche das Initialprädikat des Wurzeldienstes w erfüllt: $\exists \alpha \in \Lambda(V_w) : \alpha \vdash \mathcal{I}_w$.
- Für jeden kombinierten Dienst S aus D dürfen zwischen seinen Teildiensten keine Inkonsistenzen vorliegen. Abhängig von der Art der Kombination $S = S_1 \parallel S_2$ oder $S = S_1 \parallel^{S_P} S_2$ muss eine der folgenden Aussagen gelten.

$$\mathcal{INC}(S_1, S_2) = \emptyset \qquad \text{oder} \qquad \mathcal{INC}(S_1, S_2, S_P) = \emptyset.$$

□

Die zweite Eigenschaft der obigen Definition lässt sich auf n Teildienste verallgemeinern. Dafür müssen die beiden Definitionen aus Gleichungen (5.1a) und (5.1b) um weitere Mengen von Inkonsistenzen erweitert werden.

5.2.3. Konsistenzprüfung durch Simulation

Die Konsistenzbedingungen aus Definitionen 5.1 und 5.2 lassen sich durch den Nachweis der Simulation zwischen Transitionssystemen automatisch nachweisen. Eine Simulation ist eine Relation, die Zustände von zwei Transitionssystemen miteinander in Beziehung setzt. Bei der Konsistenzprüfung wird überprüft, ob ein Fehlerzustand in der Kombination von zwei Diensten erreicht werden kann, während er in einem einzelnen Dienst nicht erreichbar ist. Die Konsistenzbedingungen aus Definition 5.4 werden durch das Erreichen eines Fehlerzustands in der Dienstkombination überprüft.

Einfache Kombination Sei eine Dienstkombination $S = S_1 \parallel S_2$ gegeben. Um die Konsistenz zwischen den beiden Teildiensten zu überprüfen, werden die Dienste S, S_1 und S_2 jeweils um einen Fehlerzustand erweitert. Gemäß Definition 5.1 sind S_1 und S_2 in einem erreichbaren lokalen Zustand $loc(\alpha)$ inkonsistent, falls die Kombination S in diesem Zustand nicht alle Eingaben verarbeiten kann, die der Dienst S_1 ohne S_2 (bzw. S_2 ohne S_1) verarbeiten kann. Gemäß der Definition aus Abschnitt 5.1.2 erreicht der vervollständigte Dienst $\hat{S}$ für eine Belegung α einen Fehlerzustand, wenn der ursprüngliche Dienst S für diese Belegung keine ausführbare Transition hat. Daraus folgt, dass der Dienst S_1 mit dem Dienst S_2 in $loc(\alpha)$ inkonsistent ist, falls die vervollständigte Kombination $\hat{S}$ für eine der Belegungen aus $loc(\alpha)$ den Fehlerzustand erreicht, während der vervollständigte Teildienst $\hat{S}_1$ für dieselbe Belegung keinen Fehlerzustand erreicht.

Priorisierte Kombination Sei eine priorisierte Dienstkombination $S = S_1 \|^{S_P} S_2$ gegeben. Im Falle der priorisierten Kombination müssen gemäß Definition 5.2 zusätzlich zur Bedingung aus dem letzten Paragraphen die beiden Teildienste gleichzeitig aktiviert sein. Für die automatische Konsistenzprüfung wird der Priorisierungsdienst S_P um eine lokale Variable p mit $type(p) = \{0, 1, 2\}$ erweitert, die speichert, ob S_1, S_2 oder keiner der beiden priorisiert wird (vgl. Definition der Funktion p in Abschnitt 4.3.3 auf Seite 68). Demzufolge entspricht der Ausdruck $\alpha(p) = 0$ der Evaluation von $isAct_0(S_P, \alpha)$ zu wahr. Daraus folgt, dass in $loc(\alpha)$ eine Inkonsistenz vorliegt, falls die Bedingung aus dem letzten Paragraphen erfüllt und $\alpha(p) = 0$ sind.

Deadlocks Die Kombination $S = S_1 \| S_2$ bzw. $S = S_1 \|^{S_P} S_2$ befindet sich im Zustand $loc(\alpha)$ in einem Deadlock, falls in der vervollständigen Kombination $\hat{S}$ für jede Belegung aus $loc(\alpha)$ der Fehlerzustand der einzige Nachfolgezustand ist.

5.3. Korrektheit

Während das Hauptaugenmerk des vorangehenden Abschnitts auf der Konsistenz einer Diensthierarchie lag, steht nun die Korrektheit der Spezifikation im Mittelpunkt. Wie bereits in der Einführung dieses Kapitels erwähnt, besteht eine funktionale Spezifikation aus einer Diensthierarchie und einer Menge von Randbedingungen, die das Verhalten des Systems zusätzlich einschränken. Eine Spezifikation ist korrekt, wenn die Diensthierarchie alle Randbedingungen erfüllt.

Im folgenden Abschnitt werden verschiedene Klassen von Randbedingungen vorgestellt. In Abschnitt 5.3.2 stehen die formale Definition einer korrekten dienstbasierten Spezifikation sowie die Erfüllung einer Randbedingung durch eine Diensthierarchie im Mittelpunkt. In Abschnitt 5.3.3 wird der Nachweis der Korrektheit einer Spezifikation auf das Erreichbarkeitsproblem in Transitionssystemen reduziert.

5.3.1. Randbedingungen

Eine konsistente Spezifikation formalisiert eine Menge von Interaktionsmustern zwischen dem System und seiner Umgebung und garantiert, dass diese Muster sich nicht widersprechen. Häufig liegt jedoch der Grund des Fehlverhaltens eines Systems nicht in einem Widerspruch zwischen Interaktionsmustern, sondern in der Tatsache, dass das Systemverhalten *funktionsübergreifende* Randbedingungen verletzt. Diese Randbedingungen betreffen vielmehr das Gesamtverhalten des Systems als ein einzelnes Interaktionsmuster und stehen orthogonal zur Diensthierarchie. An dieser Stelle ist es wichtig, den Unterschied zwischen einem Dienst und einer Randbedingung hervorzuheben. Ein Dienst definiert ein Beispiel des Systemverhaltens, d.h. ein mögliches Interaktionsmuster, das das System aufweisen *kann*. Mit jedem weiteren Dienst wird der Gesamtspezifikation Verhalten hinzugefügt, d.h. die Anzahl von Abläufen wird erhöht (vgl. Abschnitt 4.3.2).

Eine Randbedingung schränkt das Gesamtverhalten des Systems ein, indem sie von jedem Interaktionsmuster erfüllt werden *muss* – ein Ablauf, der die Randbedingung verletzt, gehört nicht zum Systemverhalten. Somit wird die Anzahl von Systemabläufen reduziert. Bei den meisten Randbedingungen handelt es sich um die so genannten *safety*-Eigenschaften [Lev95, Kapitel 15].

Beispiel 5.4 (*Safety*-Eigenschaft)**.** Eine mögliche *safety*-Eigenschaft fordert, dass die vom ACC-System gesteuerte Geschwindigkeit nie über 250 km/h liegt. Dies gilt für alle Nutzerfunktionen (z.B. *Tempomat* oder *Abstandsregelung*) und alle Systemmodi (z.B. *Ausgeschaltet* oder *Aktiviert*). □

Nun stellt sich die Frage, wie eine Randbedingung in einer dienstbasierten Spezifikation konkret aussehen soll. Dabei muss berücksichtigt werden, dass bestimmte Eigenschaften immer und andere nur in bestimmten Systemmodi gelten müssen. Um dieser Tatsache Rechnung zu tragen, wird bei der Beschreibung von Randbedingungen zwischen Modus- und Systeminvarianten unterschieden.

Eine *Systeminvariante* muss immer erfüllt sein, d.h. unabhängig davon, welche Dienste gerade durch Priorisierungsdienste aktiviert sind. Für eine Diensthierarchie $(D \cup P, w, V)$ und eine Randbedingung φ muss folgende Aussage gelten: $\forall \rho \in \langle\langle w \rangle\rangle, i \in \mathbb{N} : \rho_i \vdash \varphi$. Das heißt, jede erreichbare Belegung im Wurzeldienst w muss die Randbedingung φ erfüllen.

Eine *Modusinvariante* muss nur in einem bestimmten Systemmodus gelten (vgl. Abschnitt 4.5 auf Seite 78). Für eine Diensthierarchie $(D \cup P, w, V)$, einen Systemmodus $mode(\alpha)$ und eine Randbedingung φ muss folgende Aussage gelten: $\forall \rho \in \langle\langle w \rangle\rangle, i \in \mathbb{N} : mode(\rho_i) \Rightarrow \rho_i \vdash \varphi$. Das heißt, jede erreichbare Belegung im Wurzeldienst w muss die Randbedingung φ erfüllen, falls die Diensthierarchie sich für diese Belegung im Modus $mode(\alpha)$ befindet.

Beispiel 5.5 (Randbedingungen)**.** Als Beispiel dient wieder die Diensthierarchie aus Abbildung 3.18 auf Seite 47.

Für die ACC-Steuerung wird folgende Systeminvariante definiert. Unabhängig von dem aktuellen Systemmodus, z.B. ob der Tempomat oder die Abstandsregelung aktiviert ist, darf die ACC-Steuerung das Fahrzeug nicht weiter beschleunigen, falls die vom Fahrer eingegebene Wunschgeschwindigkeit erreicht oder überschritten wurde: $\varphi \stackrel{\text{def}}{=} cSpeed \geq rSpeed \Rightarrow mInstr' \leq 0 \vee mInstr' = \varepsilon$. Eine negative Zahl am Port `mInstr` bedeutet, dass die Motorleistung für den Bremsvorgang verwendet wird. Die Systeminvariante ist dann wie folgt definiert: $\forall \rho \in \langle\langle w \rangle\rangle, i \in \mathbb{N} : \rho_i(cSpeed) \geq \rho_i(rSpeed) \Rightarrow \rho_{i+1}(mInstr) \leq 0 \vee \rho_{i+1}(mInstr) = \varepsilon$.

Im Gegensatz zur Systeminvariante muss folgende Eigenschaft nur im Modus „Deaktiviert“ gelten. Die Modusinvariante fordert, dass die ACC-Steuerung keine Steuerbefehle an die Motor- und Bremssteuerung verschickt, falls das ACC deaktiviert ist. Gemäß Definition 4.17 auf Seite 79 wird der Modus „Deaktiviert“ wie folgt definiert: $mode_d(\alpha) \stackrel{\text{def}}{=} (Deaktiviert \in Act(ACC, \alpha))$. Die Eigenschaft an sich ist wie folgt definiert: $\varphi_d \stackrel{\text{def}}{=} mInstr = \varepsilon \wedge bInstr = \varepsilon$. Daraus folgt die Modusinvariante $\forall \rho \in \langle\langle w \rangle\rangle, i \in \mathbb{N} : mode_d(\rho_i) \Rightarrow \rho_i(mInstr) = \varepsilon \wedge \rho_i(bInstr) = \varepsilon$. □

Eine weitere Klasse von Randbedingungen befasst sich mit dem korrekten Wechsel zwischen Systemmodi. Das Systemverhalten wird zu jedem Zeitpunkt durch das mit dem aktiven Systemmodus assoziierten Verhalten definiert. Abhängig von den aktuellen Eingaben wird zwischen alternativen Modi zur Laufzeit hin- und hergeschaltet. Ein korrekter Moduswechsel spielt eine wesentliche Rolle für die Korrektheit einer dienstbasierten Spezifikation. Demzufolge muss sichergestellt werden, dass das System in einen bestimmten Modus wechselt, wenn bestimmte Eingaben eintreffen. Dies wird in einer *Moduswechsel-Bedingung* festgelegt, die den Modus definiert, in dem sich das System befinden muss, wenn die durch die logische Formel φ definierten Eingaben eintreffen: $\forall \rho \in \langle\langle w \rangle\rangle, i \in \mathbb{N} : \varphi \Rightarrow mode(\rho_i)$.

Beispiel 5.6 (Moduswechsel)**.** Die Spezifikation der ACC-Steuerung (ohne die *Stop&Go*-Funktion) fordert, dass das ACC deaktiviert bleibt, solange die momentane Geschwindigkeit kleiner 30 km/h ist. Die entsprechende Moduswechsel-Bedingung lautet, dass das System sich im Modus „Deaktiviert“ befinden muss, wenn am Eingabeport `cSpeed` ein Wert kleiner 30 anliegt: $\forall \rho \in \langle\langle w \rangle\rangle, i \in \mathbb{N} : \rho_i(cSpeed) < 30 \Rightarrow mode_d(\rho_i)$, wobei $mode_d$ wie in Beispiel 5.5 definiert ist. □

5.3.2. Korrektheitsprüfung

Nach der Vorstellung verschiedener Klassen von Randbedingungen werden nun die Definition der Erfüllung einer Randbedingung von einem Dienst und die einer korrekten Spezifikation eingeführt.

Es wird angenommen, dass jede *safety*-Eigenschaft φ über $V_\varphi = I \uplus L \uplus O$ durch einen so genannten *Eigenschaftsdienst* $S_\varphi = (V_\varphi, \mathcal{I}, \mathcal{T})$ gegeben ist. Diese Annahme basiert auf der Feststellung von Vardi und Wolper, dass für jede temporale Formel ein Automat konstruiert werden kann, der genau diejenigen Berechnungssequenzen erzeugt, die diese Formel erfüllen [VW86]. Da in diesem Abschnitt ausschließlich *safety*-Eigenschaften betrachtet werden, ist das konstruierte Transitionssystem endlich. Außerdem wird angenommen, dass der Eigenschaftsdienst S_φ deterministisch und eingabevollständig (engl. input complete) ist. Das heißt, für alle erreichbaren Belegungen $\alpha, \beta, \gamma \in \mathrm{Attr}(S_\varphi)$ gelten folgende Relationen:

$$\big(\beta \vdash \mathcal{I} \wedge \gamma \vdash \mathcal{I} \Rightarrow \beta \overset{L}{=} \gamma\big) \wedge \big(\beta, \gamma \in \mathrm{Succ}_{S_\varphi}(\alpha) \Rightarrow \beta \overset{L}{=} \gamma\big), \tag{5.2a}$$

$$\mathrm{Succ}_{S_\varphi}(\alpha) \neq \emptyset. \tag{5.2b}$$

Der Determinismus des Eigenschaftsdienstes erleichtert den Nachweis der Erfüllung einer Eigenschaft, indem die Entscheidung, ob die Eigenschaft verletzt ist, lokal getroffen wird. Das heißt, wenn der Fehlerzustand von der Belegung α aus nicht erreichbar ist, dann verletzt das System die Eigenschaft nach dem Erreichen der Belegung α nie. Das ist nicht immer der Fall, wenn der Eigenschaftsdienst nichtdeterministisch ist. Die Determinisierung von Diensten erfolgt durch die Potenzmengenkonstruktion [HU79]. Dieser Vorgang liegt jedoch nicht im Fokus der vorliegenden Arbeit und wird in [BH09c] behandelt. Ein eingabevollständiger Dienst definiert explizit eine beliebige Reaktion auf

diejenigen Eingaben, welche von der Eigenschaft nicht betroffen sind (wir sprechen von der Chaos-Vervollständigung).

Im Folgenden wird die Bedingung für die Erfüllung einer Eigenschaft von einem Dienst definiert. Dabei werden ausschließlich Eigenschaften betrachtet, welche die Relationen zwischen Ein- und Ausgabeports definieren. Lokale Variablen von Diensten werden von Eigenschaften nicht eingeschränkt (vgl. Bemerkung 4.2 auf Seite 65). Die Definition bezieht sich auf die Abläufe aus Definition 4.6 auf Seite 58.

Definition 5.5 (Erfüllung einer Eigenschaft)**.** Ein Dienst S über V erfüllt die Eigenschaft φ über V_φ (bezeichnet durch $S \models \varphi$), falls alle Abläufe von S aus der Menge der Abläufe von S_φ sind:

$$\langle\!\langle S \rangle\!\rangle \overset{V \cap V_\varphi}{\subseteq} \langle\!\langle S_\varphi \rangle\!\rangle. \tag{5.3}$$

□

In der Definition wird angenommen, dass der Dienst S und der Eigenschaftsdienst S_φ gemäß Gleichung 4.9 auf Seite 67 kombinierbar sind.

Aus der Definition der Erfüllung einer Eigenschaft lässt sich die Definition einer korrekten dienstbasierten Spezifikation herleiten.

Definition 5.6 (Korrekte Spezifikation)**.** Seien eine Diensthierarchie durch den Graphen $(D \cup P, w, V)$ und eine Randbedingung durch die logische Formel φ gegeben. Die Diensthierarchie ist bezüglich der Randbedingung φ korrekt, falls der Wurzeldienst w ein Modell von φ ist: $w \models \varphi$. □

5.3.3. Reduktion auf das Erreichbarkeitsproblem

Der Nachweis der Erfüllung einer Eigenschaft wird auf das Erreichbarkeitsproblem in Transitionssystemen reduziert.

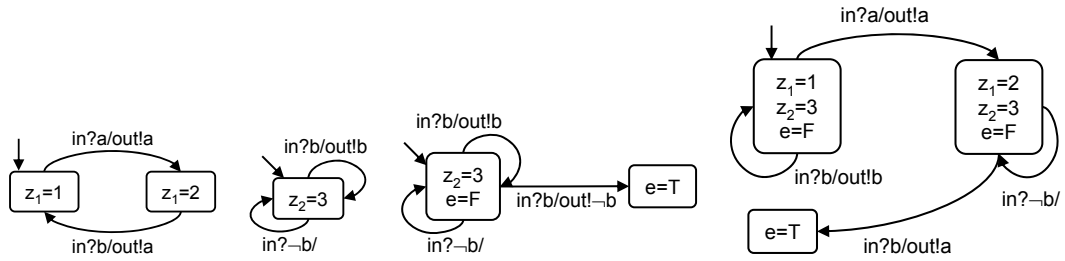
Der Dienst $S = (V, \mathcal{I}, \mathcal{T})$ erfüllt die Eigenschaft φ genau dann, wenn die Dienstkombination $C \overset{\text{def}}{=} \hat{S}_\varphi \parallel S$ von dem Dienst S und dem vervollständigten Eigenschaftsdienst $\hat{S}_\varphi$ keinen Fehlerzustand erreicht:

$$\begin{aligned} S \models \varphi \overset{\text{def}}{\Leftrightarrow} \forall \alpha, \beta \in \mathrm{Attr}(C) : \alpha \vdash \mathcal{I} \Rightarrow \neg\alpha(e) \\ \wedge \big(\mathrm{En}_S(\alpha) \wedge \beta \in \mathrm{Succ}_C(\alpha)\big) \Rightarrow \big(\alpha(e) \Leftrightarrow \beta(e)\big). \end{aligned} \tag{5.4}$$

Aus Satz A.1 auf Seite 149 folgt, dass die Eigenschaft aus Gleichung (5.3) genau dann gilt, wenn die Eigenschaft aus Gleichung (5.4) gilt.

Im Folgenden wird der Erfüllungsnachweis einer Eigenschaft anhand eines sehr einfacheren Beispiels erläutert. Ein Beispiel aus der Fallstudie folgt in Abschnitt 5.4.

Beispiel 5.7 (Erfüllung einer Eigenschaft)**.** Seien der Dienst S durch das Zustandsdiagramm aus Abbildung 5.4(a) und die Eigenschaft φ durch die logische Formel $in =$



(a) S (b) S_φ (c) $\hat{S}_\varphi$ (d) $S \parallel \hat{S}_\varphi$

Abbildung 5.4.: Zustandsdiagramme aus Beispiel 5.7

$b \Rightarrow out' = b$ gegeben, wobei $type(in) = type(out) = \{a, b\}$. Aus der Formel wird der Eigenschaftsdienst S_φ aus Abbildung 5.4(b) generiert. Der Dienst ist eingabevollständig: für die Eingabe `b` wird die Ausgabe `b` gefordert, für alle anderen Eingaben kann am Port `out` ein beliebiger Wert anliegen. Entsprechend der Definition aus Abschnitt 5.1.2 wird der Dienst um einen Fehlerzustand erweitert (vgl. Abbildung 5.4(c)). Der Dienst geht in den Fehlerzustand ($e = T$) über, wenn als Reaktion auf die Eingabe `b` eine andere Ausgabe als `b` erzeugt wird. Die Kombination des Dienstes S mit dem um den Fehlerzustand erweiterten Eigenschaftsdienst ist in Abbildung 5.4(d) dargestellt.

Immer wenn die Kombination $S \parallel \hat{S}_\varphi$ für die Belegung α mit $\alpha(z_1) = 2 \wedge \alpha(in) = b$ ausführbar ist, erreicht sie einen Fehlerzustand β mit $\beta(e) = T \wedge \beta(out) = a$. Genau diese Abfolge von Belegungen verletzt die Eigenschaft φ. □

***Bemerkung* 5.3** (Schnittstellenprojektion)**.** Um das Verständnis zu erleichtern, wird in Gleichung (5.4) angenommen, dass der Dienst und die Eigenschaft auf derselben Menge von Variablen definiert sind. Allerdings betreffen sowohl Randbedingungen als auch (Teil-)Dienste typischerweise nur einen Aspekt des Systemverhaltens und können demzufolge auf unterschiedlichen Teilschnittstellen des Systems definiert sein. Gleichung (5.3) berücksichtigt diesen Umstand, indem die Abläufe des Dienstes und die des Eigenschaftsdienstes auf die Schnittmenge der Variablen projiziert werden. Um dies auch in der algorithmischen Analyse (d.h. in Gleichung (5.4)) zu berücksichtigen, wird die *Schnittstellenprojektion* eines Transitionssystems auf eine Variablemenge eingeführt.

Die Schnittstellenprojektion eines Systems $S = (V, \mathcal{I}, \mathcal{T})$ auf die Variablemenge $V \setminus W$ ist wie folgt definiert:

$$S|_W \stackrel{\text{def}}{=} (V \setminus W, \mathcal{I}|_W, \mathcal{T}|_W),$$

wobei für alle $\alpha, \beta \in \Lambda(V \setminus W)$ folgende Relationen gelten:

$$\alpha \vdash \mathcal{I}|_W \stackrel{\text{def}}{\Leftrightarrow} \exists \gamma \in \Lambda(V) : \gamma \vdash \mathcal{I} \wedge \alpha \stackrel{V \setminus W}{=} \gamma \text{ und}$$
$$\alpha, \beta' \vdash \mathcal{T}|_W \stackrel{\text{def}}{\Leftrightarrow} \exists \gamma, \delta \in \Lambda(V) : \gamma, \delta' \vdash \mathcal{T} \wedge \alpha \stackrel{V \setminus W}{=} \gamma \wedge \beta \stackrel{V \setminus W}{=} \delta.$$

Die Relation $\langle\!\langle S \rangle\!\rangle \stackrel{V \setminus W}{=} \langle\!\langle S|_W \rangle\!\rangle$ zwischen einem System und seiner Projektion ist offensichtlich. Die Schnittstellenprojektion erfolgt automatisch, indem alle logischen Aus-

drücke, welche die Variablen aus W einschränken, durch den Ausdruck *True* ersetzt werden.

Sind der Dienst und die Eigenschaft auf zwei unterschiedlichen Variablenmengen definiert, müssen die beiden Transitionssysteme in Gleichung (5.4) auf die Schnittmenge ihrer Variablen reduziert werden. □

5.4. Konfliktlösung

In den vorangehenden Abschnitten wurde gezeigt, wie *Konflikte* – Inkonsistenzen innerhalb einer Diensthierarchie und Verletzungen von Randbedingungen – automatisch entdeckt werden können, um die Konsistenz und die Korrektheit einer Spezifikation zu überprüfen. Nun gehen wir einen Schritt weiter und stellen mehrere Verfahren zur algorithmischen Konfliktlösung bereit. Zu diesem Zweck werden zunächst die Ursachen von Konflikten untersucht, um festzustellen, welche Arten von Konflikten sich automatisch lösen lassen. Anschließend werden die automatischen Konfliktlösungen vorgestellt.

Im Rahmen der beiden Fallstudien aus Anhang C wurden folgende Klassen von Konflikten (oder besser gesagt Konfliktursachen) identifiziert.

- *Missverstandene Anforderungen.* Anforderungen eines Stakeholders wurden falsch interpretiert oder dargestellt. Dadurch widersprechen sie den Anforderungen eines anderen Stakeholders oder verletzten eine Randbedingung.
- *Widersprüchliche Anforderungen.* Viele Inkonsistenzen haben ihren Ursprung in widersprüchlichen Vorstellungen einzelner Stakeholder von dem System.
- *Überspezifikation.* Die Lösungskompetenz des Entwicklers wird durch zu restriktive Anforderungen eingeschränkt. In der Domäne reaktiver Software-Systeme handelt es sich dabei meistens um zu restriktive Anforderungen an die Reaktionszeit des Systems. Statt zu fordern, dass die Reaktion auf eine bestimmte Eingabe *irgendwann* erfolgt, wird die geforderte Ausgabe gleich im nächsten Zeitintervall erzwungen. In LTL-Formeln ausgedrückt, statt der Eigenschaft $\Box(p \rightarrow \Diamond\, q)$ wird die Eigenschaft $\Box(p \rightarrow \circ\, q)$ gefordert[1].
- *Unberücksichtigte Wechselwirkungen.* Die meisten Konflikte in dienstbasierten Spezifikationen werden durch unberücksichtigte Wechselwirkungen zwischen Nutzerfunktionen verursacht. Es handelt sich um den klassischen Fall des *feature-interaction*-Problems [Zav03]. In der Nomenklatur dieser Arbeit bedeutet das, dass in der Spezifikation ein Priorisierungsdienst fehlt, welcher eine Wechselwirkung zwischen zwei Diensten definieren soll.

Konflikte, die durch falsche oder widersprüchliche Anforderungen verursacht werden, können weder automatisch noch schematisch gelöst werden. Zur Lösung eines solchen Konflikts müssen die betroffenen Dienste von Stakeholdern per Hand modifiziert werden.

[1] In den Formeln wird die klassische LTL-Syntax benutzt: Der Operator □ steht für „Global“, ∘ für „Nachfolger“ und ◊ für „Irgendwann“.

Demzufolge wird diese Art von Konflikten in der vorliegenden Arbeit nicht weiter behandelt. Die durch Überspezifikationen verursachten Konflikte können manuell jedoch schematisch durch die so genannte Dienstabstraktion gelöst werden (vgl. Abschnitt 5.4.1). Das Problem der unberücksichtigten Wechselwirkungen kann in vielen Fällen voll automatisch gelöst werden. Die automatische Generierung fehlender Priorisierungen bildet den Schwerpunkt dieses Abschnittes. Dabei unterscheiden wird zwischen der Generierung einer dynamischen und der einer statischen Priorisierung. Im ersten Fall wird die Erfüllung von Randbedingungen sichergestellt. Dafür wird eine Priorisierung erzeugt, die unter Berücksichtigung der Randbedingungen zwischen Diensten zur Laufzeit hin- und herschaltet (vgl. Abschnitt 5.4.2). Im Falle einer statischen Priorisierung werden Inkonsistenzen innerhalb einer Diensthierarchie beseitigt. Immer wenn zwei Dienste widersprüchliche Ausgaben erzeugen, gibt die statische Priorisierung immer demselben Dienst Vorrang (vgl. Abschnitt 5.4.3).

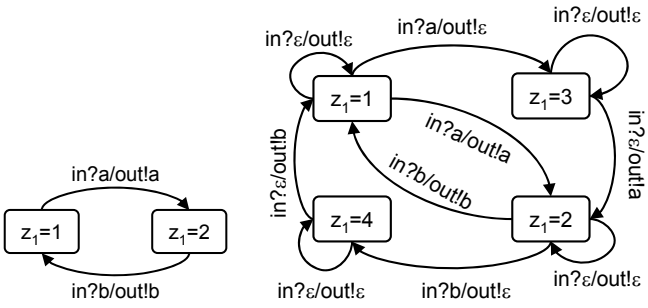
5.4.1. Dienstabstraktion

In vielen Fällen können Inkonsistenzen innerhalb einer Diensthierarchie durch die Abstraktion (oder Abschwächung) eigenständiger Dienste beseitigt werden, ohne dabei den Sinngehalt der ursprünglichen Anforderungen zu verzerren.

Es wird angenommen, dass ein informeller Anwendungsfall die abstrakteste Form einer funktionalen Anforderung ist während der entsprechende Dienst *eine* seiner Verfeinerungen darstellt. Häufig kommt es vor, dass die gewählte Verfeinerung das Systemverhalten stärker einschränkt als es in der ursprünglichen Anforderung gemeint war. Beispielsweise wird der Anwendungsfall „das System muss auf die Eingabe `a` mit der Ausgabe `b` reagieren“ oft als „das System muss auf die Eingabe `a` mit einer Ausgabe `b` im nächsten Zeitintervall reagieren“ interpretiert. Das liegt in erster Linie daran, dass sich eine Eigenschaft der Form $(p \rightarrow \circ q)$ viel einfacher durch einen Automaten modellieren lässt als eine der Form $(p \rightarrow \Diamond q)$ (vgl. Abbildung 5.5). Während die erste durch einen deterministischen Automaten modelliert wird, benötigt man für die zweite einen nichtdeterministischen Automaten mit deutlich größeren Zustands- und Transitionsmengen.

Durch manuelle Reviews von Inkonsistenzen können solche Unstimmigkeiten zwischen informellen Anforderungen und Diensten entdeckt werden. Um eine Inkonsistenz zwischen zwei Diensten zu beseitigen, werden abstraktere Verfeinerungen der ursprünglichen Anforderungen modelliert, d.h. Dienste, die mehr Verhalten zulassen. Aus der Semantik des Kombinationsoperators folgt, dass je abstrakter (nichtdeterministischer) Dienste sind, desto geringer ist die Anzahl von Inkonsistenzen zwischen diesen.

Beispiel 5.8 (Dienstabstraktion)**.** Sei der Anwendungsfall „Auf die Eingabe `a` reagiert das System mit der Ausgabe `a`, danach wartet das System auf die Eingabe `b`, auf die es mit der Ausgabe `b` reagiert“ durch das Zustandsdiagramm aus Abbildung 5.5(a) formalisiert. Der Anwendungsfall fordert jedoch nicht, dass die Reaktion des Systems immer im nächsten Zeitintervall erfolgen muss – das Systemverhalten wird durch ein zu restriktives Zustandsdiagramm unnötig stark eingeschränkt. Das nichtdeterministische Zustandsdia-



(a) $(p \rightarrow \circ q)$ (b) $(p \rightarrow \Diamond q)$

Abbildung 5.5.: Dienstabstraktion

gramm aus Abbildung 5.5(b) ist eine andere Formalisierung desselben Anwendungsfalls, die jedoch abstrakter als die deterministische ist. Das nichtdeterministische Zustandsdiagramm erlaubt die Systemreaktion in einem beliebigen nachfolgenden Zeitintervall. □

5.4.2. Dynamische Priorisierung

Im Folgenden wird ein Algorithmus zur automatischen Erzeugung einer Priorisierung vorgestellt, welche die Erfüllung einer Randbedingung durch die Kombination von zwei Diensten sicherstellt. Die somit erzeugte Priorisierung deaktiviert einen der beiden Dienste, um die Verletzung der Randbedingung zu verhindern. Der Algorithmus erzeugt die schwächste Priorisierung, d.h. diejenige, die ausschließlich im Falle einer möglichen Verletzung interveniert. Wird die Randbedingung im Falle einer unpriorisierten Kombination verletzt, gibt die erzeugte Priorisierung einem der beiden Dienste nichtdeterministisch Vorrang, so dass die Verletzung verhindert wird. Wird die Randbedingung von der unpriorisierten Kombination erfüllt, wird kein Dienst priorisiert.

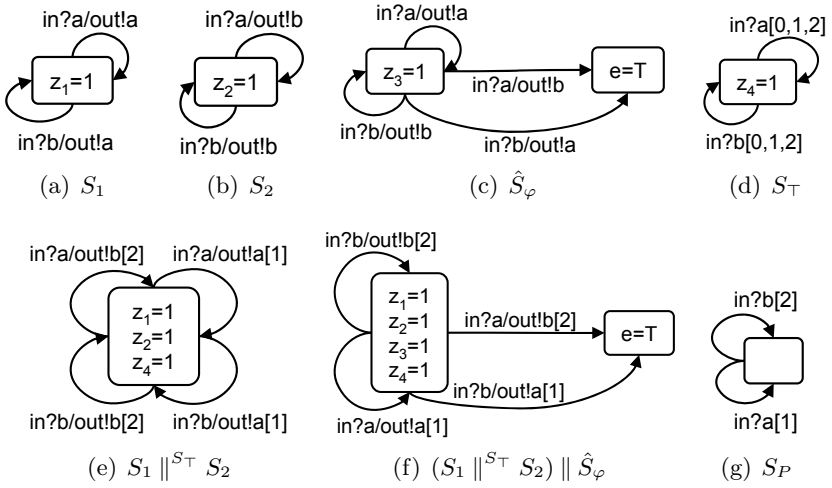
Im Folgenden wird ein kurzer Überblick über die Generierung eines Priorisierungsdienstes zwischen zwei Diensten gemäß einer Randbedingung gegeben. Es wird angenommen, dass die entsprechenden Transitionssysteme S_1, S_2 und S_φ deterministisch sind (siehe [BH09c] für mehr Details zur Determinisierung von Diensten). Im ersten Schritt wird der Suchraum für den Algorithmus berechnet, d.h. die Menge aller möglichen Priorisierungskombinationen: für jede Belegung kann S_1, S_2 oder keiner der beiden priorisiert werden. Anschließend erzeugt der Algorithmus aus diesen Kombinationen denjenigen Priorisierungsdienst, der nur wenn nötig einem Dienst Vorrang gibt. Der Algorithmus gibt eines der folgenden drei Ergebnisse zurück: (1) $S_1 \parallel S_2 \models \varphi$, d.h. kein Priorisierungsdienst ist nötig, (2) den Priorisierungsdienst P, so dass $S_1 \parallel^P S_2 \models \varphi$, (3) S_1 und S_2 verletzten die Eigenschaft φ bei jeder Priorisierung. Die Berechnung des Suchraumes erfolgt iterativ, d.h. nach jeder erzeugten Transition des Priorisierungsdienstes werden die nächsten möglichen Kombinationen berechnet, so dass der Algorithmus eine Laufzeit von $\mathcal{O}(n)$ hat, wobei n das Produkt der Anzahl der Transitionen in S_1, S_2 und S_φ ist.

***Bemerkung* 5.4** (Nichtdeterministische Priorisierung)**.** Um das Verständnis des Algorithmus zu verbessern, wird die Funktion p des Priorisierungsdienstes zunächst ein wenig umformuliert. Die Funktion $p\colon \mathcal{T}_P \to \{0,1,2\}$ aus Abschnitt 4.3.3 auf Seite 68 legt fest, ob eine Transition aus $\mathcal{T}_P$ den Dienst S_1, S_2 oder keinen der beiden priorisiert. Stattdessen wird im Folgenden $p\colon \Lambda(V_{PC}) \times \Lambda(V_{PC}) \to \mathcal{P}(\{0,1,2\}) \setminus \emptyset$ geschrieben, um die nichtdeterministische Priorisierung zwischen Diensten einfacher ausdrücken zu können. Beispielsweise bedeutet der Ausdruck $p(\alpha,\beta) = \{0,2\}$ mit $\alpha,\beta \in \Lambda(V_{PC})$ und $\alpha,\beta' \vdash \mathcal{T}_P$, dass beim Übergang (α,β) entweder kein Dienst oder S_2 Vorrang haben kann.

Entsprechend bezeichnet der Ausdruck $\mathrm{Succ}_i(\alpha)$ mit $\mathrm{Succ}_i(\alpha) \subseteq \mathrm{Succ}_C(\alpha)$ und $i \in \{0,1,2\}$ die Menge aller Nachfolger β für die Belegung α in der Dienstkombination C, für welche die Formel $i \in p(\alpha,\beta)$ gilt. □

Suchraum

Es wird angenommen, dass für die Dienste S_1, S_2 und die Eigenschaft φ folgende Gleichung gilt: $V_\varphi = V_1 \cup V_2$ (vgl. Bemerkung 5.3 auf Seite 94). Der Synthesealgorithmus führt die Suche auf der Dienstkombination $C = (S_1 \|^{S_\top} S_2) \| \hat{S}_\varphi$ durch. Für jede Belegung priorisiert der Dienst $S_\top$ nichtdeterministisch S_1, S_2 oder keinen der beiden und ist wie folgt definiert $S_\top \stackrel{\text{def}}{=} (I_1 \cup I_2 \cup I_\varphi, \mathcal{I}_1 \wedge \mathcal{I}_2 \wedge \mathcal{I}_\varphi, \mathit{True}, p)$ und $\forall \alpha,\beta : p(\alpha,\beta) = \{0,1,2\}$. Der Dienst $\hat{S}_\varphi$ stellt sicher, dass der Fehlerzustand erreicht wird, wenn immer die Eigenschaft φ verletzt wird und ist wie in Abschnitt 5.1.2 definiert. Die Kombination C umfasst alle sowohl priorisierten als auch unpriorisierten Verhalten und ist somit der Suchraum für den Synthesealgorithmus.



(a) S_1 (b) S_2 (c) $\hat{S}_\varphi$ (d) $S_\top$

(e) $S_1 \|^{S_\top} S_2$ (f) $(S_1 \|^{S_\top} S_2) \| \hat{S}_\varphi$ (g) S_P

Abbildung 5.6.: Synthese des Priorisierungsdienstes

Beispiel 5.9 (Suchraum)**.** Seien zwei Dienste S_1, S_2 und die Eigenschaft φ durch die entsprechenden Zustandsdiagramme aus Abbildungen 5.6(a), 5.6(b) und 5.6(c) gegeben,

wobei $type(in) = type(out) = \{a, b\}$. Der Eigenschaftsdienst aus Abbildung 5.6(c) ist gemäß der Definition aus Abschnitt 5.1.2 durch einen Fehlerzustand vervollständigt. Der initiale Priorisierungsdienst $S_\top$ ist durch das Zustandsdiagramm aus Abbildung 5.6(d) gegeben. Für jede Eingabe (`a` oder `b`) priorisiert der Dienst nichtdeterministisch S_1, S_2 oder keinen der beiden (`[0,1,2]` im Diagramm). Die Dienstkombination $S_1 \parallel^{S_\top} S_2$ ist in Abbildung 5.6(e) dargestellt. Da alle Transitionen der beiden Dienste paarweise widersprüchlich sind, gehört das unpriorisierte Verhalten (d.h. wenn $p(t) = 0$) nicht zum kombinierten Verhalten. Demzufolge umfasst die Transitionsmenge des kombinierten Dienstes nur diejenigen Transitionen, für die entweder $p(t) = 1$ oder $p(t) = 2$ gilt (`[1]` oder `[2]` im Diagramm). Die Kombination $(S_1 \parallel^{S_\top} S_2) \parallel \hat{S}_\varphi$ aus Abbildung 5.6(f) bildet dann den Suchraum für den Synthesealgorithmus. Alle Transitionen, welche die Eigenschaft φ verletzen, führen in dieser Kombination zu einem Fehlerzustand. □

Synthese des Priorisierungsdienstes

Im Folgenden wird die Grundidee des Algorithmus zur automatischen Synthese eines Priorisierungsdienstes erläutert. Der Algorithmus versucht durch die geeignete Festlegung der Priorisierung das Erreichen eines Fehlerzustands zu verhindern. Wenn der Algorithmus die Wahl zwischen dem unpriorisierten und dem priorisierten Verhalten hat, so wird Ersteres bevorzugt, d.h. der resultierende Priorisierungsdienst gibt in diesem Fall keinem der beiden Dienste Vorrang. Wenn für einen Zustand α alle möglichen Priorisierungsvarianten zu einem Fehlerzustand führen, wird der Fehler zu α propagiert. Wenn der Fehler zum Initialzustand propagiert ist, gibt es keinen Priorisierungsdienst, der die Erfüllung der Eigenschaft sicherstellen kann. Wenn der resultierende Priorisierungsdienst nie einem der Dienste Vorrang gibt, kann er weggelassen werden – die unpriorisierte Kombination erfüllt die Eigenschaft.

Die oben erläuterte Idee ist durch Algorithmus 1 realisiert, der für eine Menge von Initialzuständen A in der Dienstkombination C jeweils eine Menge von *Prioritäten* *Pri* und Fehlerzuständen *Err* berechnet. In diesem Kontext ist eine Priorität ein 3-Tupel (α, β, i), das bezeichnet, dass beim Schritt von α nach β der Dienst S_i mit $i \in \{1, 2\}$ oder keiner der beiden ($i = 0$) Vorrang hat. *Vis* bezeichnet die Menge der vom Algorithmus bereits analysierten Zustände.

Der Algorithmus führt auf einer gegebenen Zustandsmenge eine *Tiefensuche* durch. Er bekommt als Eingabe jeweils eine Menge von Initial- und Fehlerzuständen (vgl. Abbildung 5.7(a)). Für jeden Zustand $\alpha \in A$ werden zunächst die Prioritäten für seine unpriorisierten Nachfolgezustände aus $\mathrm{Succ}_0(\alpha)$ rekursiv berechnet (vgl. Zeile 4 im Algorithmus und Abbildung 5.7(b)). Wenn alle diese Nachfolger zu keinem Fehlerzustand führen, ist keine Priorisierung erforderlich – alle Transitionen von α nach $\beta \in \mathrm{Succ}_0(\alpha)$ werden mit $p(t) = 0$ versehen (vgl. Zeile 6). Anderenfalls werden die Prioritäten für die Nachfolger aus $\mathrm{Succ}_1(\alpha)$ und $\mathrm{Succ}_2(\alpha)$ rekursiv berechnet (vgl. Zeilen 8 und 9 und Abbildung 5.7(c)). Wenn auch von diesen priorisierten Nachfolgern aus ein Fehlerzustand erreichbar ist (vgl. Zeile 11), kann keine Priorisierung die Dienstkombination vom Errei-

Algorithm 1 Synthesealgorithmus

Ensure: $prio(A, Vis, Err) = (Pri, Vis^n, Err^n)$

1: $Pri \leftarrow \emptyset,\ Err^n \leftarrow Err,\ Vis^n \leftarrow Vis$
2: **for all** $\alpha \in A \setminus (Vis^n \cup Err^n)$ **do**
3: $\quad Vis^n \leftarrow Vis^n \cup \{\alpha\}$
4: $\quad (P_0, Vis^n, Err^n) \leftarrow prio(\mathrm{Succ}_0(\alpha), Vis^n, Err^n)$
5: $\quad$ **if** $\mathrm{Succ}_0(\alpha) \cap Err^n = \emptyset$ **then**
6: $\quad\quad Pri \leftarrow Pri \cup \{(\alpha, \beta, 0) \mid \beta \in \mathrm{Succ}_0(\alpha)\} \cup P_0$
7: $\quad$ **else** {Priorisierung ist erforderlich}
8: $\quad\quad (P_1, V_1, E_1) \leftarrow prio(\mathrm{Succ}_1(\alpha), Vis^n, Err^n)$
9: $\quad\quad (P_2, V_2, E_2) \leftarrow prio(\mathrm{Succ}_2(\alpha), Vis^n, Err^n)$
10: $\quad\quad Err^n \leftarrow E_1 \cup E_2,\ Vis^n \leftarrow V_1 \cup V_2$
11: $\quad\quad$ **if** $\forall i \in \{1,2\} : \mathrm{Succ}_i(\alpha) \cap Err^n \neq \emptyset$ **then** {Ein Fehler ist unvermeidbar}
12: $\quad\quad\quad$ **return** $(Pri, Vis^n, Err^n \cup A)$
13: $\quad\quad$ **end if**
14: $\quad\quad$ **for all** $i \in \{1,2\}$ **do**
15: $\quad\quad\quad$ **if** $\mathrm{Succ}_i(\alpha) \cap Err^n = \emptyset$ **then**
16: $\quad\quad\quad\quad Pri \leftarrow Pri \cup \{(\alpha, \beta, i) \mid \beta \in \mathrm{Succ}_i(\alpha)\} \cup P_i$
17: $\quad\quad\quad$ **end if**
18: $\quad\quad$ **end for**
19: $\quad$ **end if**
20: **end for**
21: **return** (Pri, Vis^n, Err^n)

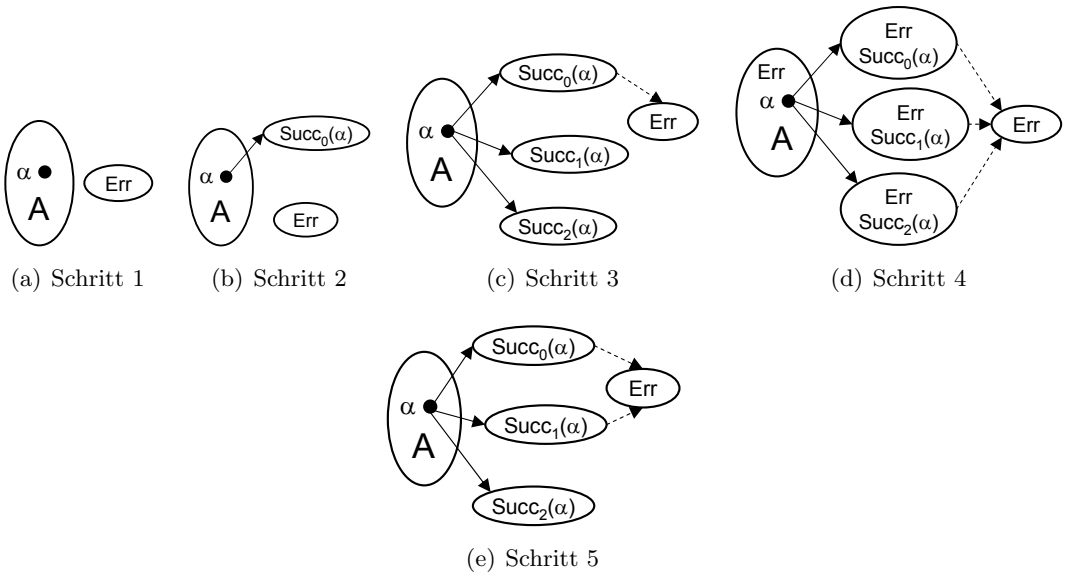
chen eines Fehlerzustands abhalten. In diesem Fall wird der Fehler zu allen Zuständen aus A propagiert (vgl. Zeile 12 und Abbildung 5.7(d)). Anderenfalls wird diejenige Priorität gespeichert, die zu keinem Fehlerzustand führt (vgl. Zeile 16 und Abbildung 5.7(e)).

Für eine gegebene Dienstkombination $C = (S_1 \|^{S_\top} S_2) \| \hat{S_\varphi}$ wird die Menge von Prioritäten (α, β, i) mit folgendem Funktionsaufruf berechnet:

$$(Pri, Vis, Err) = prio(\{\alpha \vdash \mathcal{I}_C\}, \emptyset, \{\alpha \in \Lambda(V_C) \mid \alpha(e)\}),$$

wobei *prio* durch Algorithmus 1 definiert, $\{\alpha \vdash \mathcal{I}_C\}$ die Menge der Initialzustände und $\{\alpha \in \Lambda(V_C) \mid \alpha(e)\}$ die Menge der (initialen) Fehlerzustände in C ist. Die resultierende Menge Pri wird anschließend in einen Priorisierungsdienst $P \stackrel{\text{def}}{=} (I_1 \cup I_2 \cup I_\varphi \cup L_1 \cup L_2 \cup L_\varphi, \mathcal{I}_C, \mathcal{T}_P, p)$ aus Abschnitt 4.3.3 überführt. Seine Transitionsmenge $\mathcal{T}_P$ ist durch folgende Succ-Funktion definiert: $\mathrm{Succ}(\alpha) \stackrel{\text{def}}{=} \{\beta \mid \exists(\alpha, \beta, i) \in Pri\}$. Die Funktion p ist wie folgt definiert: $p(\alpha, \beta) \stackrel{\text{def}}{=} \{i \mid (\alpha, \beta, i) \in Pri\}$.

Der Algorithmus terminiert für Variablen mit endlichen Definitionsbereichen, da Fehlerzustände und bereits analysierte Zustände nicht wiederholt behandelt werden. Satz A.2 auf Seite 150 zeigt, dass der Algorithmus korrekt ist, d.h. für S_1, S_2, φ und den resultierenden Priorisierungsdienst P folgende Aussagen gelten. (a) $S_1 \|^P S_2 \models \varphi$ und (b) P ist der schwächste Priorisierungsdienst, der die Erfüllung der Eigenschaft sicherstellt. Das



(a) Schritt 1 (b) Schritt 2 (c) Schritt 3 (d) Schritt 4

(e) Schritt 5

Abbildung 5.7.: Ein Zyklus von Algorithmus 1

bedeutet, dass für jeden anderen Priorisierungsdienst R, der der Priorisierungsordnung „0 vor $1, 2$“ folgt, folgende Aussage gilt: wenn $S_1 \|^R S_2 \models \varphi$ dann $\langle\langle S_1 \|^R S_2 \rangle\rangle \subseteq \langle\langle S_1 \|^P S_2 \rangle\rangle$.

Beispiel 5.10 (Synthesealgorithmus)**.** Der Algorithmus analysiert die Dienstkombination aus Abbildung 5.6(f) und erzeugt den Priorisierungsdienst aus Abbildung 5.6(g).

Für jeden Initialzustand α werden zunächst die Prioritäten für ihre unpriorisierten Nachfolgezustände aus $\mathrm{Succ}_0(\alpha)$ berechnet, d.h. die Tupel der Form $(\alpha, \beta, 0)$. Da im ganzen Suchraum keine Transitionen mit $p(t) = 0$ existiert, ist diese Menge leer. Im zweiten Schritt werden die Prioritäten für die Nachfolger aus $\mathrm{Succ}_1(\alpha)$ und $\mathrm{Succ}_2(\alpha)$ berechnet, d.h. die Tupel der Form $(\alpha, \beta, 1)$ und $(\alpha, \beta, 2)$. Alle Tupel, die zu einem Fehlerzustand führen (d.h. mit $\beta(e) = \mathit{True}$) werden eliminiert. Somit ergibt die Synthese zwei Tupel $(\alpha_1, \beta_1, 1)$ mit $\alpha_1(z_j) = 1 \wedge \alpha_1(in) = a$, $\beta_1(z_j) = 1$ und $(\alpha_2, \beta_2, 2)$ mit $\alpha_2(z_j) = 1 \wedge \alpha_1(in) = b$, $\beta_1(z_j) = 1$, wobei $j \in \{1, 2, 3, 4\}$. Die Kombination der beiden Tupel ergibt den Priorisierungsdienst aus Abbildung 5.6(g). □

Deadlockfreie Kombination Der resultierende Priorisierungsdienst P stellt sicher, dass die Dienstkombination $S_1 \|^P S_2$ die Eigenschaft φ nicht verletzt. Allerdings kann es durchaus dazu kommen, dass der Priorisierungsdienst die Kombination in einen Deadlock führt, um sie vom Erreichen eines Fehlerzustands abzuhalten. Da Deadlocks in der Domäne reaktiver Systeme unerwünscht sind, wird die Initialmenge von Fehlerzuständen um die Deadlocks aus Definition 5.3 auf Seite 87 erweitert: $\mathit{Err} \stackrel{\text{def}}{=} \{\alpha \mid \alpha(e)\} \cup \{\alpha \mid \forall \beta \in \mathit{loc}(\alpha) : \mathrm{Succ}(\beta) = \emptyset\}$.

Man beachte, dass die Menge von Deadlocks nicht im Voraus, sondern einher mit der

Synthese des Priorisierungsdienstes berechnet wird, da die Entscheidung, ob ein Zustand ein Deadlock ist, lokal getroffen werden kann.

Modulare Priorisierung

Bisher wurde bei der Synthese nur eine einzige Randbedingung berücksichtigt. Es ist jedoch offensichtlich, dass es in einer Spezifikation mehrere Randbedingungen geben kann, die alle erfüllt werden müssen. Im Folgenden wird die Situation betrachtet, in der zwei Dienste S_1 und S_2 die Eigenschaften φ, ψ, ... erfüllen müssen. In diesem Fall wird für jede einzelne Eigenschaft ein separater Priorisierungsdienst erzeugt, die anschließend zu einem Dienst kombiniert werden. Der kombinierte Priorisierungsdienst stellt sicher, dass die Konjunktion $\varphi \wedge \psi \wedge \ldots$ erfüllt wird.

Die Kombination der Priorisierungsdienste $P_1 = (V_1, \mathcal{I}_1, \mathcal{T}_1, p_1)$ und $P_2 = (V_2, \mathcal{I}_2, \mathcal{T}_2, p_2)$ ergibt den Dienst $P = P_1 \parallel P_2 \stackrel{\text{def}}{=} (V, \mathcal{I}, \mathcal{T}, p)$. Die Mengen V, $\mathcal{I}$ und $\mathcal{T}$ sind genau wie in der Dienstkombination aus Abschnitt 4.3.2 auf Seite 66 definiert. Die Funktion p ist für alle $\alpha, \beta \in \Lambda(V)$ definiert, so dass $p(\alpha, \beta) \stackrel{\text{def}}{\Leftrightarrow} p_1(\alpha, \beta) \cap p_2(\alpha, \beta)$ gilt. Das Verhalten der Kombination $S_1 \parallel^P S_2$ ist undefiniert genau dann, wenn p_1 und p_2 für ein und dieselbe Transition unterschiedlichen Diensten Vorrang geben (d.h. $p_1(\alpha, \beta) \cap p_2(\alpha, \beta) = \emptyset$).

Satz A.3 auf Seite 151 zeigt, dass die Kombination mehrerer Priorisierungsdienste die Erfüllung der entsprechenden Eigenschaften sicherstellt. Konkret heißt es, dass für zwei Dienste S_1, S_2 und zwei Eigenschaften φ, ψ mit $S_1 \parallel^{P_\varphi} S_2 \models \varphi$ und $S_1 \parallel^{P_\psi} S_2 \models \psi$ folgende Aussage gilt: $S_1 \parallel^{P_\varphi \| P_\psi} S_2 \models \varphi \wedge \psi$.

Beispiel 5.11 (Zusammenspiel von Tempomat und Abstandsregelung)**.** Auf die Nutzerfunktionen *Tempomat* und *Abstandsregelung* wurde bereits in Beispielen 3.7 auf Seite 32 und 3.9 auf Seite 36 eingegangen. Eine Randbedingung, welche die Beachtung der Wunschgeschwindigkeit fordert, wurde in Beispiel 5.5 auf Seite 91 definiert. Eine weitere Randbedingung fordert die Beachtung des vom Fahrer gewählten Abstands zum vorausfahrenden Objekt. Für die beiden Eigenschaften erzeugt der Synthesealgorithmus zwei separate Priorisierungsdienste, deren Kombination den Priorisierungsdienst aus Beispiel 3.10 auf Seite 39 ergibt. □

5.4.3. Statische Priorisierung

Im Gegensatz zum Verfahren aus dem vorangehenden Abschnitt, das die Erfüllung von Randbedingungen durch eine Diensthierarchie sicherstellt, werden an dieser Stelle wieder die Inkonsistenzen innerhalb einer Diensthierarchie behandelt. Das Ziel ist die Erzeugung eines Priorisierungsdienstes, der in einer Dienstkombination das Erreichen von Inkonsistenzen verhindert. Bei diesem Verfahren wird angenommen, dass die beteiligten Stakeholder bereits im Voraus entscheiden, welche Interaktionsmuster den Vorrang vor den anderen haben. Es ist z.B. offensichtlich, dass das Systemverhalten im Notfall Vorrang vor dem regulären Verhalten haben muss. In diesem Fall soll der erzeugte Priorisierungsdienst dem Dienst `Notfall` immer dann Vorrang geben, wenn zwischen den

beiden Diensten eine Inkonsistenz auftritt. In allen anderen Fällen gelten die Regeln der unpriorisierten Kombination. Im Folgenden wird gezeigt, wie Inkonsistenzen in einer Kombination von zwei Diensten durch eine statische Priorisierung beseitigt werden können.

Es ist offensichtlich, dass diese Art der Konfliktlösung einen Sonderfall des Algorithmus aus dem vorangehenden Abschnitt darstellt. Dafür muss Algorithmus 1 an einer einzigen Stelle modifiziert werden, in dem die Funktion p des Priorisierungsdienstes $S_\top$ von $\forall \alpha, \beta : p(\alpha, \beta) = \{0, 1, 2\}$ auf $\forall \alpha, \beta : p(\alpha, \beta) = \{0, i\}$ eingeschränkt wird, wobei S_i der statisch vorrangige Dienst ist. Die Zustände aus Gleichungen (5.1a) und (5.1b) auf Seite 87 bilden die initiale Menge *Err* für den Algorithmus. Dadurch lassen sich die Inkonsistenzen innerhalb einer Diensthierarchie durch dasselbe Verfahren wie die Verletzungen einer Randbedingung beseitigen, allerdings mit einem anderen Suchraum.

5.5. Integration in die Broysche Theorie

In diesem Abschnitt steht die formale Integration des vorgestellten Ansatzes in die Broysche Theorie im Mittelpunkt. Im Folgenden wird gezeigt, dass die *konsistente* operationelle Diensthierarchie aus Definition 5.4 auf Seite 89 eine Instanz der deskriptiven Diensthierarchie von Broy aus Definition 4.13 auf Seite 63 ist.

Zu diesem Zweck wird zunächst die Teildienst-Relation zwischen zwei konsistenten Diensten und deren unpriorisierten Kombination nachgewiesen.

Satz 5.1 (Kombination und Teildienst-Relation)**.** *Seien S eine konsistente Kombination von zwei Diensten $S = S_1 \parallel S_2$ und F, F_1 und F_2 ihre entsprechenden Schnittstellenabstraktionen (vgl. Gleichung (4.8) auf Seite 66). Dann sind F_1 und F_2 zwei Teildienste von F gemäß Definition 4.12 auf Seite 63:*

$$S = S_1 \parallel S_2 \Rightarrow F_1 \sqsubseteq_{sub} F \wedge F_2 \sqsubseteq_{sub} F.$$

Beweis. Wegen der Symmetrie der beiden Fälle wird nur $F_1 \sqsubseteq_{sub} F$ bewiesen. Aus Satz 4.4 auf Seite 75 und Korollar 4.1 auf Seite 66 folgt, dass die Kombination der beiden Dienste ihre Verfeinerung ist: $S = S_1 \parallel S_2 \Rightarrow F_1 \succeq F$. Es muss noch folgende Relation gezeigt werden: $dom(F_1) \subseteq dom(F\dagger(I_1 \blacktriangleright O_1))$. Gemäß Definition 4.10 auf Seite 62 und Gleichung (4.8) auf Seite 66 folgt dies aus der Relation:

$$\langle\!\langle S_1 \rangle\!\rangle \overset{I_1}{\subseteq} \langle\!\langle S \rangle\!\rangle.$$

Für den Beweis dieser Relation sind gemäß Definition 4.6 auf Seite 58 folgende zwei

Eigenschaften zu zeigen:

$$\forall \rho_1 \in \langle\langle S_1 \rangle\rangle, \exists \alpha \in \Lambda(V) : \rho_1.0 \overset{I_1}{=} \alpha \wedge (\rho_1.0 \vdash \mathcal{I}_1 \Rightarrow \alpha \vdash \mathcal{I}) \text{ und} \tag{5.5a}$$

$$\begin{aligned} &\forall \rho_1 \in \langle\langle S_1 \rangle\rangle, \exists \rho \in \langle\langle S \rangle\rangle : \forall i \in \mathbb{N} : \\ &\quad \rho.i \overset{I_1}{=} \rho_1.i \Rightarrow \rho_1.(i+1) \overset{I_1}{=} \rho.(i+1) \vee \\ &\quad \rho.i \overset{I_1}{=} \rho_1.i \wedge (\neg \mathrm{En}(\rho_1.i) \Rightarrow \neg \mathrm{En}_1(\rho.i)). \end{aligned} \tag{5.5b}$$

- Eigenschaft (5.5a) besagt, dass es für jeden Systemablauf aus S_1 eine Belegung α über Variablen von S gibt, die gleiche Eingabewerte wie $\rho_1.0$ hat und $\mathcal{I}$ erfüllt. Die Prädikate $\mathcal{I}$ und $\mathcal{I}_1$ schränken die Variablen aus I beziehungsweise I_1 nicht ein. Demzufolge muss lediglich gezeigt werden, dass es für jeden Ablauf $\rho_1 \in \langle\langle S_1 \rangle\rangle$ mit $\rho_1.0 \vdash \mathcal{I}$ eine Belegung $\alpha \in \Lambda(V)$ mit $\alpha \vdash \mathcal{I}$ gibt. In diesem Fall folgt aus Gleichung (4.2) auf Seite 57, dass es auch eine Belegung $\alpha \in \Lambda(V)$ gibt, die sowohl $\rho_1.0 \vdash \mathcal{I}_1 \Rightarrow \alpha \vdash \mathcal{I}$ als auch $\rho_1.0 \overset{I_1}{=} \alpha$ erfüllt. Da $\rho_1.0$ und α unabhängig voneinander sind, bleibt zu zeigen, dass es eine $\mathcal{I}$ erfüllende Belegung $\alpha \in \Lambda(V)$ gibt. Dies folgt direkt aus der Annahme einer konsistenten Kombination und der Definition der Inkonsistenz (vgl. Definition 5.1 auf Seite 86).
- Eigenschaft (5.5b) folgt direkt aus der Annahme einer konsistenten Kombination und der Definition der Inkonsistenz. Denn aus Definition 5.1 folgt, dass $\forall \rho \in \langle\langle S \rangle\rangle, \rho_1 \in \langle\langle S_1 \rangle\rangle, i \in \mathbb{N} : \rho.i \overset{I_1}{=} \rho_1.i \Rightarrow \mathrm{Succ}_1(\rho_1.i) \overset{I_1}{\subseteq} \mathrm{Succ}(\rho.i)$.

□

Nun wird die eingeschränkte Teildienst-Relation zwischen zwei konsistenten Diensten und deren priorisierten Kombination nachgewiesen.

Satz 5.2 (Priorisierte Kombination und eingeschränkte Teildienst-Relation)**.** *Seien S eine konsistente priorisierte Kombination von zwei Diensten $S = S_1 \parallel^{S_P} S_2$ und F, F_1 und F_2 ihre entsprechenden Schnittstellenabstraktionen. Dann sind F_1 und F_2 zwei eingeschränkte Teildienste von F gemäß Definition 4.12 auf Seite 63. Das heißt, es gibt zwei Mengen $H_1, H_2 \subseteq dom(F)$, so dass folgende Aussage gilt:*

$$S = S_1 \parallel^{S_P} S_2 \Rightarrow F_1 \sqsubseteq_{sub} F|(dom(F) \setminus H_1) \wedge F_2 \sqsubseteq_{sub} F|(dom(F) \setminus H_2).$$

Die Historienmengen H_i mit $i \in \{1, 2\}$ sind wie folgt definiert:

$$H_i \overset{\text{def}}{=} \{h \in dom(F) \mid \exists \rho \in R_i : h \overset{I_i}{=} \rho\},$$

wobei die Mengen von Systemabläufen R_i wie in Satz 4.7 auf Seite 77 definiert sind:

$$\begin{aligned} R_1 &\overset{\text{def}}{=} \{\rho \in \langle\langle S \rangle\rangle \mid \exists t \in T_P, i \in \mathbb{N} : \rho.i, \rho.(i+1)' \vdash t \wedge p(t) = 2\} \text{ und} \\ R_2 &\overset{\text{def}}{=} \{\rho \in \langle\langle S \rangle\rangle \mid \exists t \in T_P, i \in \mathbb{N} : \rho.i, \rho.(i+1)' \vdash t \wedge p(t) = 1\}. \end{aligned}$$

Beweis. Wegen der Symmetrie der beiden Fälle wird nur $F_1 \sqsubseteq_{sub} F|H_1$ bewiesen. Aus Satz 4.7 auf Seite 77 und Korollar 4.1 auf Seite 66 folgt, dass die priorisierte Kombination der beiden Dienste ihre eingeschränkte Verfeinerung ist: $S = S_1 \parallel^{S_P} S_2 \Rightarrow F_1 \succeq_{R_1} F$. Es muss noch gezeigt werden, dass die Relation $dom(F_1) \subseteq dom(F\dagger(I_1 \blacktriangleright O_1))$ unter Ausschluss der Historien aus H_1 gilt. Gemäß Definition 4.10 auf Seite 62 und Gleichung (4.8) auf Seite 66 folgt dies aus der Relation:

$$\langle\!\langle S_1 \rangle\!\rangle \overset{I_1}{\subseteq} \langle\!\langle S \rangle\!\rangle \setminus R_1.$$

In der Menge R_1 sind alle Systemabläufe zusammen gefasst, in denen S_2 mindestens zu einem Zeitintervall $t \in \mathbb{N}$ priorisiert ist. Folglich umfasst die Menge $\langle\!\langle S \rangle\!\rangle \setminus R_1$ alle Abläufe, in denen zu jedem Zeitintervall entweder kein Dienst oder S_1 priorisiert ist, d.h. $\forall i \in \mathbb{N} : \neg isAct_2(S_P, \rho.i)$. Unter der Annahme einer konsistenten priorisierten Kombination folgt aus Definition 5.2 auf Seite 87, dass der Beweis der oberen Relation jenem aus Satz 5.1 identisch ist. □

Man beachte, dass eine der beiden Mengen R_1 oder R_2 leer sein kann, wenn der Priorisierungsdienst S_P einem der beiden Dienste nie Vorrang gibt. Dann steht dieser Dienst in einer uneingeschränkten Teildienst-Relation mit dem kombinierten Dienst.

Anschließend wird die Relation zwischen der operationellen Diensthierarchie aus Definition 5.4 und der denotationalen Diensthierarchie aus Definition 4.13 definiert.

Satz 5.3 (Operationelle und denotationale Diensthierarchie)**.** *Sei die operationelle Diensthierarchie $\mathcal{DH}_o = (D_o \cup P_o, w_o, V_o)$ nach Definition 5.4 konsistent. Dann ist die Schnittstellenabstraktion dieser Hierarchie $\mathcal{DH}_d = (D_d, w_d, V_d, H_d)$ eine denotationale Diensthierarchie nach Definition 4.13, wobei die Schnittstellenabstraktion einer operationellen Diensthierarchie wie folgt definiert ist:*

- *D_d ist die Menge der Schnittstellenabstraktionen der Dienste aus D_o, wobei für jeden Dienst $S_i \in D_o$ genau eine Schnittstellenabstraktion $F_i \in D_d$ existiert;*
- *w_d ist die Schnittstellenabstraktion von w_o;*
- *Es gibt eine vertikale Kante in $\mathcal{DH}_d$ genau dann, wenn es eine Kante zwischen den entsprechenden Diensten in $\mathcal{DH}_o$ gibt: $(F_1, F_2) \in V_d \overset{\text{def}}{\Leftrightarrow} (S_1, S_2) \in V_o$;*
- *Es gibt eine horizontale Kante in $\mathcal{DH}_d$ genau dann, wenn es einen Priorisierungsdienst zwischen den entsprechenden Diensten in $\mathcal{DH}_o$ gibt: $(F_1, F_2) \in H_d \overset{\text{def}}{\Leftrightarrow} \exists (S_3, S_1), (S_3, S_2), (S_3, S_p) \in V_o : S_3 = (S_1 \parallel^{S_p} S_2 \parallel \dots)$.*

Beweis. Der Teilgraph (D_d, w_d, V_d) ist ein Baum, denn gemäß Definition 4.16 die Hierarchie $\mathcal{DH}_o$ ein Baum ist.

Die Knoten aus einer vertikalen Kante $(F_1, F_2) \in V_d$ stehen in einer (eingeschränkten) Teildienst-Relation zueinander, da die Dienste aus der entsprechenden Kante (S_1, S_2) aus V_o in einer Kombination-Relation und ihre Schnittstellenabstraktionen gemäß Sätzen 5.1 und 5.2 in einer (eingeschränkten) Teildienst-Relation zueinander stehen.

Mindestens einer der beiden Knoten aus einer horizontalen Kante $(F_1, F_2) \in H_d$ steht in einer eingeschränkten Teildienst-Relation zu ihrem unmittelbaren gemeinsamen Vater F, da es in $\mathcal{DH}_o$ einen Priorisierungsdienst S_p gibt, so dass $S = (S_1 \parallel^{S_p} S_2 \parallel \dots)$ gilt. Gemäß Satz 5.2 stehen die Schnittstellenabstraktionen des kombinierten Dienstes und mindestens eines ihrer Teildienste in einer eingeschränkten Teildienst-Relation zueinander. □

5.6. Zusammenfassung

In diesem Kapitel wurden zwei Arten von Konflikten in einer funktionalen Spezifikation eingeführt, die Inkonsistenzen zwischen Diensten und die Verletzungen von Randbedingungen. Die Ersteren sind anwendungsunabhängig und führen zu einer inkonsistenten Spezifikation, die mehrdeutig und nicht implementierbar ist. Die Letzteren (anwendungsspezifischen) führen zu einer inkorrekten Spezifikation, die den Anforderungen von Stakeholdern nicht entspricht.

Für die beiden Arten von Konflikten wurden Suchalgorithmen bereitgestellt. Die Nachweise der Konsistenz und der Korrektheit wurden auf das Erreichbarkeitsproblem in Automaten reduziert, so dass die Analyse automatisch durchgeführt werden kann.

Außerdem wurden die Ursachen von Konflikten untersucht, um festzustellen, welche Arten von ihnen sich automatisch lösen lassen. In komplexen multifunktionalen Software-Systemen ist die meist verbreitete Konfliktursache eine unberücksichtigte Wechselwirkung zwischen Nutzerfunktionen – das sogenannte *feature-interaction*-Problem. Die Beseitigung solcher Konflikte durch die automatische Generierung von Priorisierungsdiensten zur Darstellung der unberücksichtigten Wechselwirkungen ist ein wesentlicher Beitrag dieses Kapitels. Die resultierenden Priorisierungsdienste stellen sicher, dass alle Randbedingungen erfüllt sind und keine Inkonsistenzen innerhalb einer Diensthierarchie auftreten.

In diesem Kapitel wurde die formale Integration des vorgestellten Ansatzes in die Broysche Theorie abgeschlossen, indem gezeigt wurde, dass zwei Dienste und ihre Kombination jeweils in einer Teildienst-Relation zueinander stehen.

KAPITEL 6

Übergang zur logischen Architektur

Außer funktionalen Anforderungen sind Qualitätsmerkmale [BCK98, Kapitel 4] wie z.B. Performance, Wiederverwendbarkeit oder Wartbarkeit weitere wichtige Kriterien für die Akzeptanz eines Systems. Diese Merkmale haben keinen Einfluss auf die Funktionalität, bestimmen jedoch die Struktur des Systems und werden im vorgestellten modellbasierten Ansatz in der logischen Architektur betrachtet. Nachdem auf die Gestalt der logischen Architektur ganz andere Anforderungen (Architekturtreiber genannt) Auswirkungen haben als auf die funktionale Spezifikation, ist ihre Struktur prinzipiell unabhängig von der Struktur der Diensthierarchie.

Im vorliegenden Kapitel wird die automatische Generierung einer logischen Architektur aus einer funktionalen Spezifikation heraus erläutert. Die erarbeitete Transformation ist eigenschaftserhaltend (engl. *property preserving*), d.h. sie garantiert die Erfüllung aller in der Diensthierarchie spezifizierten Eigenschaften im resultierenden Netzwerk logischer Komponenten. Dieses Netzwerk dient als Basis für die Restrukturierung der Architektur gemäß weiteren Architekturtreibern.

Inhalt

Im modellbasierten Ansatz aus [BFG+08] werden Qualitätsmerkmale in der logischen Architektur betrachtet. Die logische Architektur ist ein Netzwerk kommunizierender logischer Komponenten, die durch gerichtete, getypte Kanäle miteinander verbunden sind. Die logische Kommunikation zwischen Komponenten erfolgt ausschließlich über diese Kanäle. Im Vergleich zur Spezifikation steht in der Architektur nicht mehr die Formalisierung der an der Systemgrenze beobachtbaren Funktionalität im Vordergrund, sondern vielmehr die Strukturierung des Systems in logische Einheiten, deren Gesamtverhalten das in der Spezifikation festgelegte Verhalten realisiert.

Bass et al. [BCK98, S. 21] definieren eine Architektur[1] wie folgt:

> „The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them."

Aus der Definition ist ersichtlich, dass ein System mehrere Strukturen haben kann und keine von ihnen *die* Architektur genannt werden kann. Außerdem definieren Bass et al. keine universalen Kriterien für eine gute Architektur. Stattdessen erfolgt die Strukturierung anhand unterschiedlicher Kriterien wie z.B. der geometrischen Struktur des Systems, der organisatorischen Struktur im Unternehmen oder eines der anwendungsspezifischen Qualitätsmerkmale.

Nachdem die logische Architektur prinzipiell nach ganz anderen Kriterien strukturiert wird als die funktionale Spezifikation, stellt sich die Frage, wie man von einer Hierarchie von Interaktionsmustern zu einem Netzwerk logischer Komponenten gelangt. In dem vorliegenden Kapitel wird ein formaler Übergang von einer Diensthierarchie zu einer logischen Architektur vorgestellt. Man beachte, dass die Erstellung einer guten (d.h. bestimmte Qualitätsmerkmale erfüllenden) Architektur nicht im Fokus der vorliegenden Arbeit liegt. Das angestrebte Ziel ist die Bereitstellung einer so genannten initialen Architektur, die sich in weiteren Schritten gemäß einer Reihe von *Architekturtreibern* leicht umstrukturieren lässt. Nach Bass et al. [BCK98, S. 154] bezeichnen Architekturtreiber Anforderungen, die Auswirkungen auf die Gestalt der Architektur haben.

Es existieren zwei prinzipiell unterschiedliche Vorgehensweisen, von einer funktionalen Spezifikation zu einer logischen Architektur zu gelangen. Beim manuellen Übergang wird die Architektur von Hand erstellt und anschließend überprüft, ob sie die gegebene Spezifikation verfeinert. Beim automatischen Übergang wird ein Netzwerk von Komponenten aus einer gegebenen Diensthierarchie generiert, so dass alle spezifizierten Eigenschaften per Konstruktion erhalten bleiben (engl. *correct-by-construction*). In einem automatisch generierten Netzwerk werden nur funktionale Anforderungen berücksichtigt, so dass wei-

[1] In der Definition von Bass et al. wird der Begriff „Software-Architektur" verwendet. Allerdings meinen die Autoren damit nicht eine konkrete Form der Architektur, sondern allgemein eine Struktur des Systems. Deswegen benutzen sie für die Bezeichnung eines Bausteins der Architektur den allgemeinen Begriff „Element" statt z.B. einer Software-Komponente oder einer Klasse. Der in der vorliegenden Arbeit verwendete Begriff „logische Architektur" bezeichnet eine Unterklasse der Architektur aus dieser Definition.

tere Umstrukturierungsschritte notwendig sind, um geforderte Qualitätsmerkmale zu erfüllen.

In dieser Arbeit wird nur der automatische Übergang betrachtet. Das generierte Netzwerk logischer Komponenten soll dabei folgende zwei Eigenschaften aufweisen:

- *Totales Verhalten:* Das Verhalten des Netzwerks soll im Vergleich zum Verhalten der Diensthierarchie total sein[2]. Durch die totale Definition des Verhaltens stellt die logische Architektur ein vollständiges und implementierungsunabhängiges Modell der Systemfunktionalität dar.
- *Assoziativität:* Im Gegensatz zur priorisierten Dienstkombination soll die Komposition von Komponenten assoziativ sein. Diese in der Diensthierarchie fehlende Eigenschaft ist für eine Restrukturierung der Architektur gemäß Architekturtreibern unerlässlich.

Dank der Assoziativität der Komposition dient dieses Netzwerk als Basis für die Strukturierung der Architektur gemäß weiterer Architekturtreiber. Demzufolge ist die in diesem Kapitel vorgestellte Transformation nur ein erster Schritt in Richtung einer guten logischen Architektur.

Im folgenden Abschnitt werden Unterschiede zwischen der Spezifikation und der logischen Architektur untersucht. In Abschnitt 6.2 wird ein Algorithmus zur eigenschaftserhaltenden Transformation einer Diensthierarchie in ein Netzwerk von Komponenten eingeführt.

6.1. Abgrenzung zur funktionalen Spezifikation

In diesem Abschnitt werden die funktionale Spezifikation und die logische Architektur gegenübergestellt. Außerdem wird die Rolle der logischen Architektur im modellbasierten Entwicklungsprozess erläutert.

6.1.1. Gegenüberstellung

Die Diensthierarchie und das Netzwerk von Komponenten sind zwei orthogonale Strukturen der Funktionalität. Die beiden Modelle sind in Tabelle 6.1 gegenübergestellt.

- *Black- vs. White-Box-Sicht:* Die funktionale Spezifikation definiert eine Struktur des Systemverhaltens, wie es an der Systemgrenze durch Benutzer wahrgenommen wird. Die Architektur definiert eine innere Struktur des Systems.
- *Interaktionsmuster vs. Komponenten:* Interaktionsmuster sind partielle Aspekte des Systemverhaltens. Sie können überlappen. Komponenten beschreiben ein totales Verhalten und haben keine gemeinsamen Ausgabeports.

[2] Während in der Diensthierarchie Eingaben existieren, für die es kein definiertes Verhalten gibt, sollen diese Eingaben im Netzwerk auf den vordefinierten $\perp$-Wert abgebildet werden.

Funktionale Spezifikation	Logische Architektur
Beschreibt das System aus der Black-Box-Sicht.	Beschreibt das System aus der White-Box-Sicht.
Das Gesamtverhalten wird in partielle Interaktionsmuster unterteilt.	Das Gesamtverhalten wird in totale Komponenten unterteilt.
Wird als ein Kontrakt zwischen dem Benutzer und den Entwicklern verwendet. Definiert die Problemdomäne.	Wird von den Entwicklern erstellt, um die innere Struktur des Systems (die Lösungsdomäne) besser zu verstehen.
Kann redundante Interaktionsmuster enthalten.	Funktionale Redundanz soll vermieden werden.
Modelliert ausschließlich die Systemfunktionalität.	Berücksichtigt Qualitätsmerkmale des Systems.

Tabelle 6.1.: Abgrenzung der logischen Architektur von der Spezifikation

- *Problem- vs. Lösungsdomäne:* Die Spezifikation definiert ein Modell der Problemdomäne. Die logische Architektur ist das erste Modell der Lösungsdomäne. Ihre Gestaltung hat Auswirkung auf die Implementierung des Systems.
- *Redundanz:* Interaktionsmuster sollten nicht in sehr kleine und abstrakte Muster zerlegt werden – ein Muster soll eine für den Benutzer intuitive Funktionalität darstellen. Demzufolge kann eine Spezifikation mehrere identische oder ähnliche Muster enthalten. In der logischen Architektur soll die funktionale Redundanz (falls durch Qualitätsmerkmale nicht explizit gefordert) vermieden werden.
- *Qualitätsmerkmale:* Im Gegensatz zur funktionalen Spezifikation, in der ausschließlich Nutzerfunktionen modelliert werden, werden in der logischen Architektur weitere Architekturtreiber wie z.B. Qualitätsmerkmale berücksichtigt.

6.1.2. Die Rolle der logischen Architektur

Im Folgenden wird die Rolle der Architektur in der modellbasierten Entwicklung erläutert.

- *Strukturierung der Spezifikation:* Die Annahme, dass eine Spezifikation keinerlei Festlegungen in Bezug auf die Architektur enthält, ist in der Praxis außer bei sehr kleinen Systemen unrealistisch. Wie bereits in Abschnitt 1.3.2 erläutert, ist oft eine Dekomposition des Systems notwendig, um die Spezifikation zu strukturieren. Die logische Architektur ist häufig der Ausgangspunkt für die Spezifikation verschiedener Teile des Systems (vgl. die Matrix aus Abbildung 3.2 auf Seite 22).
- *Organisation der Entwicklung:* Ein System wird selten von einem einzigen Team entwickelt, vielmehr ist die Entwicklung komplexer Systeme durch den klassischen Zulieferprozess geprägt. Die Gliederung des Systems in handhabbare Teilsysteme mit fest definierten Schnittstellen reduziert den Aufwand für die Kommunikation

zwischen den Entwicklern. Diese Zerlegung ist nicht alleine aus der funktionalen Spezifikation abzuleiten und erfordert weitere Informationen, die in die logische Architektur einfließen.

- *Traceability:* Bei der Entwicklung komplexer Systeme ist es üblich, dass die Anforderungen sowie die technische Infrastruktur der zu entwickelnden Systeme sich kontinuierlich ändern. Für die Wartung solcher Systeme ist es wichtig, die Auswirkungen dieser Änderungen rückverfolgen zu können. Komponenten der logischen Architektur verbinden Anforderungen mit Elementen der Implementierung.

6.2. Generierung der logischen Architektur

Die automatische Generierung einer logischen Architektur aus einer funktionalen Spezifikation heraus steht im Mittelpunkt dieses Abschnitts. Die erarbeitende Transformation ist eigenschaftserhaltend (engl. *property preserving*), d.h. sie gewährleistet die Erhaltung aller in der Diensthierarchie spezifizierten Eigenschaften im resultierenden Netzwerk per Konstruktion.

Im folgenden Abschnitt wird die logische Architektur formal definiert, während in Abschnitt 6.2.2 der Transformationsalgorithmus eingeführt und erläutert wird.

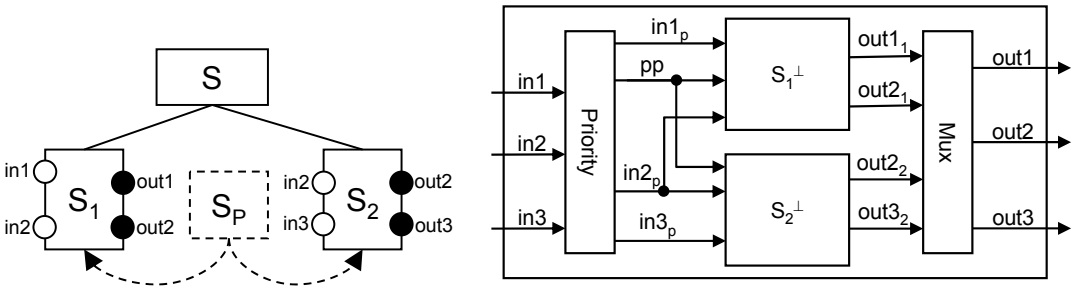
6.2.1. Logische Architektur

Die Definition der logischen Architektur ist an die von *System Structure Diagrams* von AutoFocus [HSE97] angelehnt. Sie besteht aus einem Netzwerk logischer Komponenten, die durch gerichtete getypte Kanäle miteinander verbunden sind. Logische Komponenten wurden bereits in Abschnitt 3.2.1 auf Seite 23 eingeführt. Zur Erinnerung: Eine Komponente hat eine syntaktische Schnittstelle und eine Verhaltensspezifikation. Die syntaktische Schnittstelle besteht jeweils aus einer Menge von Ein- und Ausgabeports. Die Verhaltensspezifikation einer Komponente ist durch ein Transitionssystem gegeben, das Eingabewerte auf Ausgabewerte abbildet. Ein Netzwerk von Komponenten hat interne sowie externe, an der Netzwerkgrenze sichtbare Kanäle. Ein Kanal stellt die Kommunikation zwischen zwei Komponenten dar, indem er zwei Kommunikationspartner (Ports von Komponenten) miteinander verbindet. Komponenten können mittels des Kompositionsoperators aus Abschnitt 4.3.5 auf Seite 72 hierarchisch strukturiert werden.

***Bemerkung* 6.1** (Keine explizite Angabe von Kanälen)**.** Obwohl in folgenden Abbildungen dargestellt, werden Kanäle im vorgestellten Formalismus nicht explizit verwendet – gleichnamige Ports sind implizit durch einen Kanal verbunden. Ist ein Port mit keinem anderen verbunden, ist er implizit mit der Systemumgebung verbunden. □

6.2.2. Transformation

Im Wesentlichen besteht die Transformation einer Diensthierarchie in ein Netzwerk logischer Komponenten aus zwei Schritten, der Totalisierung partieller Dienste und der Erzeugung mehrerer synchronisierender Komponenten, welche die Simulation des Verhaltens der Dienstkombination sicherstellen. Im Folgenden wird der Transformationsalgorithmus für eine priorisierte Kombination von zwei Diensten erläutert. Auf einen kleinen Unterschied zu einer unpriorisierten Kombination wird an der entsprechenden Stelle hingewiesen.



(a) Dienstkombination

(b) Netzwerk von Komponenten

Abbildung 6.1.: Transformation

Für eine gegebene Diensthierarchie $S = S_1 \parallel^{S_P} S_2$ aus Abbildung 6.1(a) erzeugt der Algorithmus das Netzwerk von Komponenten aus Abbildung 6.1(b), das der Diensthierarchie verhaltensäquivalent ist. Die Komponente `Priority` entscheidet anhand von Eingabewerten und gemäß der Semantik der priorisierten Dienstkombination, ob eine der Komponenten $S_1^\perp$ und $S_2^\perp$ zeitlich deaktiviert wird. Diese Priorisierungsnachricht wird am Port `pp` ausgegeben. Eine deaktivierte Komponente erzeugt an ihren Ausgabeports den Wert $\perp$. Die Komponente $S_i^\perp$ ist die eigenschaftserhaltende Totalisierung des Dienstes S_i. Die Ausgaben von $S_1^\perp$ und $S_2^\perp$ werden in der Komponente `Mux` (für Multiplexer) fusioniert, da zwei logische Komponenten keine gemeinsamen Ausgabeports haben dürfen. Der Multiplexer stellt außerdem sicher, dass $S_1^\perp$ und $S_2^\perp$ keine widersprüchlichen Werte erzeugen. Wegen der starken Kausalität verursacht die Komponente `Mux` bei der Verarbeitung von Daten eine Verzögerung von einem Zeitintervall. Um die Synchronisation zwischen Ausgabewerten nicht zu verlieren, werden alle Nachrichten durch die Komponente `Mux` geleitet, unabhängig davon, ob sie durch die Komponente `Mux` zusammengeführt werden müssen oder nicht.

Man beachte, dass die Priorisierungsnachricht und die Eingaben, auf deren Basis sie erzeugt wurde, bei den Komponenten $S_1^\perp$ und $S_2^\perp$ gleichzeitig ankommen. Dies stellt sicher, dass die Deaktivierung einer Komponente gleichzeitig mit den Eingaben erfolgt. Gingen die Eingaben an den Ports `in1`, `in2` und `in3` direkt an die Komponenten (und nicht wie in Abbildung 6.1(b) durch `Priority`), wäre die Priorisierungsnachricht wegen der starken Kausalität immer um ein Zeitintervall verzögert.

Anhand dieses kleinen Beispiels wird im Folgenden der Transformationsalgorithmus eingeführt. Die Transformation besteht aus folgenden Schritten:

1. Partielle Dienste werden zu Komponenten totalisiert.
2. Für die Kompositionalität von Komponenten – sie dürfen keine gemeinsamen Ausgabeports haben – werden ihre Ausgabeports in schematischer Weise umbenannt.
3. Für die Steuerung der Priorisierung zwischen den Komponenten wird eine Koordinationskomponente erzeugt, welche anhand der aktuellen Eingaben und gemäß der Semantik der priorisierten Dienstkombination die Komponenten deaktiviert.
4. Um sich deaktivieren lassen zu können, bekommen die Komponenten $S_1^\perp$ und $S_2^\perp$ jeweils einen zusätzlichen Eingabeport, dessen Wert bestimmt, ob eine ursprüngliche Transition ausgeführt oder an allen Ausgabeports der Wert $\perp$ erzeugt wird.
5. Es wird eine neue Komponente erzeugt, welche die Ausgaben von $S_1^\perp$ und $S_2^\perp$ gemäß der Semantik der Dienstkombination fusioniert.
6. Alle modifizierten und neu erzeugten Komponenten werden komponiert.

Im Folgenden werden die einzelnen Schritte der Transformation im Detail erläutert.

Totalisierung Im ersten Schritt werden partielle Dienste totalisiert. Für einen gegebenen Dienst S wird seine Totalisierung $S^\perp$ wie folgt definiert. Die totale Komponente $S^\perp$ erzeugt an ihren Ausgabeports den Wert $\perp$ für alle Eingaben, die außerhalb des Definitionsbereichs von S liegen. Für die restlichen Eingaben verhalten sich $S^\perp$ und S identisch. Um einen partiellen Dienst $S = (V, \mathcal{I}, \mathcal{T})$ zu einer Komponente $S^\perp = (V, \mathcal{I}, \mathcal{T}^\perp)$ durch den Wert $\perp$ zu totalisieren, werden die Typen aller Variablen aus V um den Wert $\perp$ erweitert. Die Transitionsmenge $\mathcal{T}^\perp$ wird durch folgende Succ-Funktion definiert:

$$\mathrm{Succ}_{S^\perp}(\alpha) \stackrel{\text{def}}{=} \mathrm{Succ}_S(\alpha) \cup \{\beta \mid \neg \mathrm{En}_S(\alpha) \wedge \alpha \stackrel{L}{=} \beta \wedge \forall o \in O : \beta(o) = \perp\}.$$

Für die Eingaben außerhalb des Definitionsbereichs von S macht die Komponente $S^\perp$ einen Stotterschritt (engl. *stuttering step* [Lam83]): sie gibt den Wert $\perp$ aus und modifiziert ihre lokalen Variablen nicht.

Indexierung von Variablen Im Gegensatz zur Dienstkombination dürfen Komponenten in einer Komposition keine gemeinsamen Ausgabeports haben. Um die Kompositionalität der Komponenten sicherzustellen, werden alle Ausgabeports der totalen Komponenten aus dem letzten Paragraphen in einheitlicher Weise umbenannt. Sie werden durch den Indikator (im Beispiel durch den Index) des entsprechenden Dienstes indexiert. Beispielsweise wird der Port `out2` des Dienstes S_1 in $\texttt{out2}_1$ und der gleichnamige Port von S_2 in $\texttt{out2}_2$ umbenannt (vgl. Abbildung 6.1).

Zu diesem Zweck wird der Operator [./.] zur Umbenennung von Variablen in einer logischen Formel definiert. Der Ausdruck $\Phi[w/v]$ bezeichnet das Ersetzen aller Auftreten der Variable v durch die Variable w und v' durch w' in der Formel Φ.

Die Indexierung einer Variable v in einer Komponente S_i ist wie folgt definiert: $S_i[v_i/v] \stackrel{\text{def}}{=} (V_{v_i}, \mathcal{I}_{v_i}, \mathcal{T}_{v_i})$ mit $V_{v_i} \stackrel{\text{def}}{=} (V \setminus \{v\}) \cup \{v_i\}$, $\mathcal{I}_{v_i} \stackrel{\text{def}}{=} \mathcal{I}[v_i/v]$ und $\mathcal{T}_{v_i} \stackrel{\text{def}}{=} \{t[v_i/v] \mid t \in \mathcal{T}\}$, wobei $v \in V$ und $v_i \notin V$ ein Paar von Variablen gleichen Typs sind. Für eine Teilmenge von Variablen $U \subseteq V$ wird der Operator $S_i[v_i/v]_{v \in U}$ als eine sukzessive Umbenennung aller Variablen aus U definiert. Demzufolge bezeichnet der Ausdruck $S_i[o_i/o]_{o \in O}$ eine Komponente S_i, in der alle Ausgabeports durch i indexiert sind. S_i und $S_i[o_i/o]_{o \in O}$ sind verhaltensäquivalent.

Priorisierung Zur Erinnerung: Die Schnittstelle des Priorisierungsdienstes umfasst alle Eingabeports der beiden Teildienste und keine Ausgabeports. Die Funktion $p\colon \mathcal{T}_P \to \{0, 1, 2\}$ legt fest, welcher Teildienst für die aktuellen Eingaben priorisiert wird (vgl. Abschnitt 4.3.3 auf Seite 68).

Demzufolge erzeugt der Algorithmus für eine Dienstkombination $S = S_1 \parallel^{S_P} S_2$ mit dem Priorisierungsdienst $S_P = (I_P \uplus L_P, \mathcal{I}_P, T_P, p)$ eine Komponente $Prio \stackrel{\text{def}}{=} (I_P \uplus L_P \uplus O, \overline{\mathcal{I}}_P, \overline{\mathcal{T}}_P)$, wobei $O \stackrel{\text{def}}{=} I_1 \cup I_2 \cup \{pp\}$ mit $pp \notin (I_P \cup L_P)$ und $type(pp) = \{0, 1, 2, \bot\}$. Für die Initialbelegungen von *Prio* wird zusätzlich zum Prädikat $\mathcal{I}_P$ gefordert, dass der Ausgabeport `pp` mit dem Wert $\bot$ belegt wird: $\overline{\mathcal{I}}_P \stackrel{\text{def}}{=} \mathcal{I}_P \wedge pp = \bot$. An den restlichen Ausgabeports werden im Initialzustand beliebige Werte erzeugt. Die Komponente *Prio* verhält sich genauso wie der Dienst S_P und erzeugt außerdem die Priorisierungsnachricht am Port `pp` gemäß der Funktion $p(t)$. Außerdem werden alle Eingaben im darauf folgenden Zeitintervall an den entsprechenden Ausgabeports ausgegeben. Um zwischen den Ein- und Ausgabeports der Komponente *Prio* unterscheiden zu können, werden alle Ausgabeports dieser Komponente durch den Indikator p indexiert: $Prio[o_p/o]_{o \in O}$. Die Transitionsmenge $\overline{\mathcal{T}}_P$ von *Prio* ist durch folgende Succ-Funktion definiert:

$$
\begin{aligned}
&\mathrm{Succ}(\alpha) \stackrel{\text{def}}{=} \\
&\{\beta \mid \exists t \in \mathcal{T}_P : \alpha, \beta' \vdash t \wedge \beta(pp) = p(t) \wedge \forall in \in I : \exists in_p \in O : \alpha(in) = \beta(in_p)\} \cup \\
&\{\beta \mid \forall t \in \mathcal{T}_P : \alpha, \beta' \vdash \neg t_P \wedge \beta(pp) = 0 \wedge \forall in \in I : \exists in_p \in O : \alpha(in) = \beta(in_p) \wedge \alpha \stackrel{L}{=} \beta\}.
\end{aligned}
$$

Die erste Teilmenge beschreibt den Fall, in dem der ursprüngliche Priorisierungsdienst eine Transition ausführen kann. Die Komponente *Prio* verhält sich identisch und gibt am Port `pp` den Wert $p(t)$ aus. Im zweiten Fall hat der Priorisierungsdienst keine ausführbare Transition – die Komponente modifiziert ihre lokalen Variablen nicht und gibt am Port `pp` den Wert 0 aus.

Man beachte, dass im Falle einer unpriorisierten Dienstkombination $S = S_1 \parallel S_2$ die Komponente *Prio* für die Koordination zwischen $S_1^{\perp}$ und $S_2^{\perp}$ nicht notwendig ist. Allerdings, damit die Verarbeitung von Daten im resultierenden Netzwerk die gleiche Anzahl von Zeitintervallen dauert wie in einer Komposition mit Priorisierung, wird auf diese Komponente nicht verzichtet. Ihre Transitionsmenge ist wie folgend definiert:

$$\mathrm{Succ}(\alpha) \stackrel{\text{def}}{=} \{\beta \mid \beta(pp) = 0 \wedge \forall in \in I : \exists in_p \in O : \alpha(in) = \beta(in_p)\}.$$

In jedem Schritt erzeugt die Komponente am Port `pp` den Wert 0, priorisiert damit weder

$S_1^\perp$ noch $S_2^\perp$ und gibt die empfangenen Eingaben aus. Es ist offensichtlich, dass *Prio* in dieser Form die Daten nicht modifiziert, sondern lediglich um ein Zeitintervall verzögert.

Erweiterung von Komponenten Für den Empfang der Priorisierungsnachricht werden die Schnittstellen der Komponenten $S_1^\perp$ und $S_2^\perp$ jeweils um den Eingabeport `pp` erweitert. Außerdem werden alle Transitionen der Komponenten jeweils um eine Vorbedingung erweitert, die überprüft, ob die jeweilige Komponente aktuell aktiviert ist oder nicht. Eine Transition der Komponente $S_i^\perp$ wird nur dann ausgeführt, wenn der aktuelle Wert am Eingabeport `pp` 0 oder i ist. Anderenfalls wird eine neue Transition ausgeführt, die an allen Ausgabeports den Wert $\perp$ erzeugt und die lokalen Variablen nicht modifiziert – die Komponente macht einen Stotterschritt.

Demzufolge ist die Erweiterung der Komponente $S_i^\perp = (V, \mathcal{I}, \mathcal{T})$ um den Eingabeport `pp` durch folgende Komponente gegeben: $S_i^p \stackrel{\text{def}}{=} (\overline{V}, \overline{\mathcal{I}}, \overline{\mathcal{T}})$ wobei $\overline{V} \stackrel{\text{def}}{=} I \cup \{pp\} \cup L \cup O$ und $\overline{\mathcal{I}} \stackrel{\text{def}}{=} \mathcal{I} \wedge (\forall o \in O : o = \perp)$. Die Transitionsmenge $\overline{\mathcal{T}}$ ist wie folgt definiert:

$$\begin{aligned}\text{Succ}(\alpha) \stackrel{\text{def}}{=} &\{\beta \mid \exists t \in \mathcal{T} : \alpha, \beta' \vdash t \wedge \alpha(pp) \in \{0, i\}\} \\ &\cup \{\beta \mid \alpha(pp) \notin \{0, i\} \wedge \forall o \in O : \beta(o) = \perp \wedge \alpha \stackrel{L}{=} \beta\}.\end{aligned}$$

Da die Ausgabeports der Komponente *Prio* umbenannt wurden, müssen die Eingabeports der Komponente S_i^p durch denselben Indikator p indexiert werden, um die Kommunikation zwischen den Komponenten herzustellen (vgl. Abbildung 6.1(b)). Die Indexierung ergibt die Komponente $S_i^p[i_p/i]_{i \in I}$.

Multiplexer Für zwei Dienste S_1 und S_2 und jeden ihrer gemeinsamen Ausgabeports `o` erzeugt der Algorithmus jeweils einen Multiplexer, der die Ausgabewerte der beiden fusioniert. Der Übersichtlichkeit halber werden in Abbildung 6.1(b) die drei notwendigen Multiplexer (gemäß der Anzahl der Ausgabeports) durch die Komponente `Mux` dargestellt. Ein Multiplexer hat einen Ausgabeport `o` und zwei Eingabeports $\texttt{o}_1$ und $\texttt{o}_2$. Eingaben werden ohne Änderungen am Ausgabeport `o` ausgegeben, falls sie gleich sind. Empfängt die Komponente an einem der Eingabeports den Wert $\perp$, wird entsprechend die andere Eingabe ausgegeben. Für zwei unterschiedliche Eingaben (beide ungleich $\perp$) hat die Komponente keine ausführbare Transition.

Der Multiplexer für einen gemeinsamen Ausgabeport `o` von zwei Diensten S_1 und S_2 ist wie folgt definiert: $mux(S_1, S_2, o) \stackrel{\text{def}}{=} (V_m, \mathcal{I}_m, \mathcal{T}_m)$ mit $V_m \stackrel{\text{def}}{=} I_m \uplus O_m$, $I_m \stackrel{\text{def}}{=} \{o_1, o_2\}$, und $O_m \stackrel{\text{def}}{=} \{o\}$, wobei $o_1, o_2 \notin (V_1 \cup V_2)$, $type(o) = type(o_1) = type(o_2)$. Die Initialbelegung für den Port `o` im Multiplexer und die für den gleichnamigen Port in der Dienstkombination müssen gleich sein: $\alpha \vdash \mathcal{I}_m \stackrel{\text{def}}{\Leftrightarrow} \exists \varphi \vdash \mathcal{I}_1 \wedge \mathcal{I}_2 : \varphi(o) = \alpha(o)$. Die Transitionsmenge $\mathcal{T}_m$ ist durch folgende Succ-Funktion definiert:

$$\begin{aligned}\text{Succ}(\alpha) \stackrel{\text{def}}{=} &\{\beta \mid \alpha(o_1) = \alpha(o_2) = \beta(o)\} \\ &\cup \{\beta \mid \beta(o) = \alpha(o_1) \wedge \alpha(o_2) = \perp\} \cup \{\beta \mid \beta(o) = \alpha(o_2) \wedge \alpha(o_1) = \perp\}.\end{aligned}$$

Man beachte, dass das Verhalten des Multiplexers partiell ist – die Reaktion auf zwei ungleiche Eingaben ist nicht definiert. Dies steht im Widerspruch zur Annahme, dass logische Komponenten immer total sind. Allerdings handelt es sich dabei um ein Hilfskonstrukt – die Totalität der Komponenten *Prio*, $S_1^{\perp}$ und $S_2^{\perp}$ hat zur Folge, dass auch das gesamte Netzwerk total ist.

Die eingeführte Synthese eines Multiplexers lässt sich auf eine beliebige Anzahl von Diensten mit einem gemeinsamen Ausgabeport erweitern. Außerdem wird für einen Port, der nur zu einem Dienst gehört, ein Multiplexer nach demselben Schema erzeugt. In diesem Fall stellt der Multiplexer eine Identitätsfunktion mit der Verzögerung von einem Zeitintervall dar. Somit erzeugt der Transformationsalgorithmus jeweils einen Multiplexer für jeden einzelnen Ausgabeport aus der Schnittstelle der Dienstkombination $S = S_1 \parallel^{S_P} S_2$.

Komposition Im letzten Schritt werden die erzeugten Komponenten komponiert. Die Transformation der Dienstkombination $S = S_1 \parallel^{S_P} S_2$ ergibt folgende Komposition von Komponenten (der Kompositionsoperator $\oplus$ ist in Abschnitt 4.3.5 auf Seite 72 definiert):

$$C_S \stackrel{\text{def}}{=} \overline{Prio} \oplus \overline{S}_1^p \oplus \overline{S}_2^p \oplus MUX,$$

wobei $\overline{Prio} \stackrel{\text{def}}{=} Prio[o_p/o]_{o \in O}$ die Komponente *Prio* mit den indexierten Ausgabeports und $\overline{S}_i^p \stackrel{\text{def}}{=} S_i^p[o_i/o, in_p/in]_{o \in O_i, in \in I_i}$ die Totalisierung des Dienstes S_i mit den indexierten Ein- und Ausgabeports und dem neuen Eingabeport `pp` ist. *MUX* bezeichnet die Komposition aller Multiplexer:

$$MUX \stackrel{\text{def}}{=} \bigoplus_{o \in O_1 \cup O_2} mux(S_1, S_2, o).$$

Man beachte, dass dank der *input-disabled* Semantik der Multiplexer die Komponenten S_1^p und S_2^p daran gehindert werden, unterschiedliche Werte an gleichnamigen Ausgabeports zu erzeugen. Die Multiplexer haben keine ausführbaren Transitionen für unterschiedliche Werte und blockieren dadurch die Transitionen in S_1^p und S_2^p, die diese Werte verursachen. Demzufolge verhält sich die Komposition logischer Komponenten gemäß der Semantik der Dienstkombination.

Wie bereits erwähnt, liegt der einzige Unterschied zwischen der Transformation einer priorisierten und der einer unpriorisierten Dienstkombination in der Verhaltensspezifikation der Komponente *Prio*. In dem unpriorisierten Fall erzeugt *Prio* in jedem Zeitintervall den Wert 0 am Port `pp` und priorisiert dadurch keine der Komponenten.

Wegen der starken Kausalität braucht die resultierende Komposition für die Verarbeitung von Daten genau drei Zeitintervalle, während die Dienstkombination nur eins benötigt. Auf die Eingaben im Zeitintervall t erfolgt die Reaktion der Dienstkombination im Zeitintervall $t + 1$, während die Komposition erst im Zeitintervall $t + 3$ darauf reagiert. Im allerersten Zeitintervall $t = 0$ sind die Werte an den Ausgabeports der Dienstkombination und den entsprechenden Ports des Netzwerks identisch, da die Initialbelegungen in der Kombination und in den Multiplexern gleich sind. In den folgenden

zwei Zeitintervallen $t = 1$ und $t = 2$ macht das Netzwerk – dank den Initialbelegungen der Ausgabeports von *Prio*, $S_1^{\perp}$ und $S_2^{\perp}$ – zwei Stotterschritte. Ab dem Zeitintervall $t = 3$ erfolgen die Ausgaben des Netzwerks gemäß der Semantik der Dienstkombination, allerdings mit einer Verzögerung von zwei Zeitintervallen.

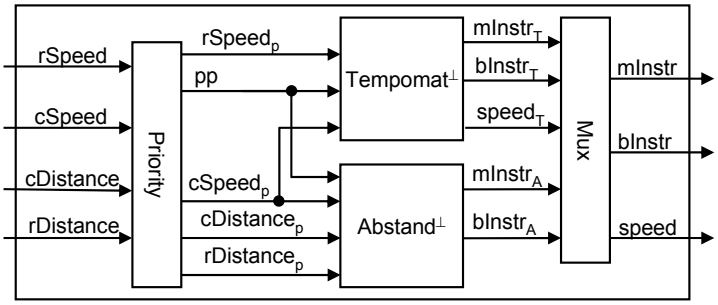


Abbildung 6.2.: Logische Architektur der ACC-Steuerung

Beispiel 6.1 (Transformation)**.** In diesem Beispiel wird die priorisierte Kombination der Dienste `Tempomat/G` aus Abbildung 3.8 auf Seite 33 und `Abstand/G` aus Abbildung 3.11 auf Seite 36 transformiert. Der Priorisierungsdienst ist durch das Zustandsdiagramm aus Abbildung 3.14(b) auf Seite 39 gegeben. Das Ergebnis der Transformation ist in Abbildung 6.2 illustriert.

Zuerst werden die beiden Dienste totalisiert. Die Totalisierung des Dienstes `Tempomat/G` ist durch das Zustandsdiagramm aus Abbildung 6.3(a) gegeben. Für alle Eingaben außerhalb des Definitionsbereichs des Dienstes macht die Komponente `Tempomat`$^{\perp}$ einen Stotterschritt. Die Totalisierung des Dienstes `Abstand/G` erfolgt analog.

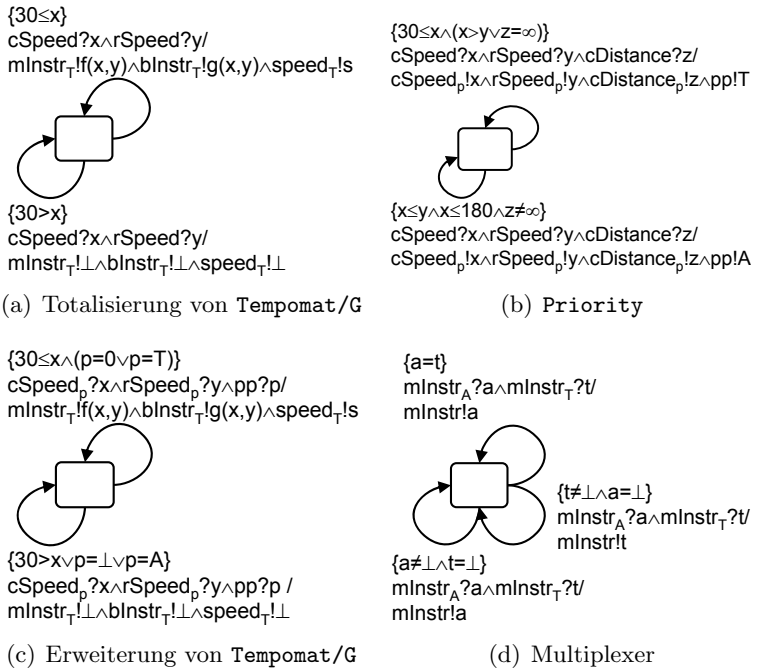
Im zweiten Schritt wird die Komponente `Priority` erzeugt (vgl. Abbildung 6.3(b)). Sie verhält sich genauso wie der Priorisierungsdienst und gibt außerdem das Ergebnis der Funktion $p(t)$ am Port `pp` (die Priorisierungsnachricht `A` oder `T`) und alle Eingaben an den entsprechenden Ausgabeports aus. Die logische Konjunktion der Vorbedingungen der beiden Transitionen aus Abbildung 6.3(b) ergibt eine Tautologie, d.h. die Komponente ist total.

Im dritten Schritt werden die beiden Komponenten `Tempomat`$^{\perp}$ und `Abstand`$^{\perp}$ um den Port `pp` erweitert. Die Erweiterung der Komponente `Tempomat`$^{\perp}$ ist durch das Zustandsdiagramm aus Abbildung 6.3(c) gegeben. Die Erweiterung verhält sich genauso wie die ursprüngliche Komponente bis auf den Fall, in dem am Port `pp` der Wert `A` ankommt (d.h. die Komponente `Abstand`$^{\perp}$ hat Vorrang). In diesem Fall macht `Tempomat`$^{\perp}$ einen Stotterschritt. Die Erweiterung der Komponente `Abstand`$^{\perp}$ erfolgt analog.

Im letzten Schritt wird für jeden Ausgabeport je ein Multiplexer erzeugt. In Abbildung 6.2 sind drei Multiplexer durch eine Komponente `Mux` dargestellt. Der Multiplexer für den Port `mInstr` ist durch das Zustandsdiagramm aus Abbildung 6.3(d) gegeben. Das Verhalten der Komponente ist partiell, da sie keine ausführbare Transition für die Eingaben $mInstr_A \neq \perp \wedge mInstr_T \neq \perp \wedge mInstr_A \neq mInstr_T$ hat. Dadurch blockiert

sie die beiden Komponenten $\texttt{Tempomat}^{\perp}$ und $\texttt{Abstand}^{\perp}$, wenn sie unterschiedliche Werte an den entsprechenden Ports $\texttt{mInstr}_{\texttt{A}}$ und $\texttt{mInstr}_{\texttt{T}}$ erzeugen wollen. Die Multiplexer für die zwei anderen Ausgabeports werden analog definiert.

Das resultierende Netzwerk aus Abbildung 6.2 verhält sich bis auf die Verzögerung von zwei Zeitintervallen genauso wie die priorisierte Kombination der beiden Dienste. □



(a) Totalisierung von `Tempomat/G` (b) `Priority`

(c) Erweiterung von `Tempomat/G` (d) Multiplexer

Abbildung 6.3.: Verhaltensspezifikationen der resultierenden Komponenten

Erhaltung von Eigenschaften

Die resultierende logische Architektur erfüllt alle in der Diensthierarchie spezifizierten Eigenschaften per Konstruktion (d.h. sie ist *correct-by-construction*). Dies wird in Satz A.4 auf Seite 151 durch den Nachweis der Simulationsrelation aus folgender Definition gezeigt. Diese Simulationsrelation setzt die Abläufe eines Dienstes mit den Abläufen einer Komponente in Beziehung.

Definition 6.1 (Simulation von Diensten durch Komponenten)**.** Seien eine Komponente $C = (V_C, \mathcal{I}_C, \mathcal{T}_C)$ und ein Dienst $S = (V_S, \mathcal{I}_S, \mathcal{T}_S)$ gegeben, so dass $V_C = I \uplus L_C \uplus O$ und $V_S = I \uplus L_S \uplus O$ und $L_S \subseteq L_C$. Ein Ablauf der Komponente $\rho_C = \beta_0\beta_1\ldots$ *simuliert* einen unendlichen Ablauf des Dienstes $\rho_S = \alpha_0\alpha_1\ldots$, bezeichnet durch $sim(\rho_C, \rho_S)$, falls Folgendes gilt: $\alpha_0 \overset{V_S}{=} \beta_0$, $\alpha_i \overset{I}{=} \beta_i$, $\alpha_{i+1} \overset{L_S}{=} \beta_{i+2}$ und $\alpha_{i+1} \overset{O}{=} \beta_{i+3}$ für alle $i \in \mathbb{N}$.

Das Verhalten der Komponente C *simuliert* das Verhalten des Dienstes S, wenn für jeden Ablauf $\rho_C = \beta_1\beta_2\ldots$ mit $\beta_i(v) \neq \perp$ für alle $i \in \mathbb{N}$ und $v \in V_S$ ein Ablauf ρ_S vom

Dienst S existiert, so dass $sim(\rho_C, \rho_S)$. Die Simulationsrelation wird durch den Ausdruck $sim(C, S)$ bezeichnet. □

Die Definition der Simulation $sim(C, S)$ basiert auf der *Reduktion von Stotterschritten* [Lam83], bei der Äquivalenzklassen von Abläufen gebildet werden. Laut Lamport umfasst eine Äquivalenzklasse dieselben Sequenzen von Transitionsschritten, die durch beliebig lange aber endliche Sequenzen von $\perp$-Transitionen unterbrochen werden. Der Repräsentant einer solchen Klasse ist der Ablauf ohne $\perp$-Transitionen.

Korollar 6.1 (Simulations- und Verfeinerungsrelation)**.** *Die Komponente C ohne Stotterschritte verfeinert den Dienst S im Sinne der Definition 4.14 auf Seite 65. Demzufolge ist die Schnittstellenabstraktion von C mit reduzierten Stotterschritten eine* FOCUS*-Verfeinerung der Schnittstellenabstraktion von S nach Definition 4.11 auf Seite 62.*

Beweis. C und S haben dieselbe syntaktische Schnittstelle. Aus Satz A.4 folgt, dass für jeden Ablauf $\rho_C = \beta_1\beta_2\ldots$ mit $\beta_i(v) \neq \perp$ für alle $i \in \mathbb{N}$ und $v \in V_S$ ein Ablauf ρ_S von S existiert, so dass $\alpha_0 \overset{V_S}{=} \beta_0$, $\alpha_i \overset{I}{=} \beta_i$, $\alpha_{i+1} \overset{L_S}{=} \beta_{i+2}$ und $\alpha_{i+1} \overset{O}{=} \beta_{i+3}$ für alle $i \in \mathbb{N}$. Durch die Reduktion der Stotterschritte in den Abläufen von C, erfüllen S und C die Bedingung aus Definition 4.14. Aus Korollar 4.1 auf Seite 66 folgt, dass die Schnittstellenabstraktion von C eine FOCUS-Verfeinerung der Schnittstellenabstraktion von S ist. □

6.3. Zusammenfassung

In diesem Kapitel wurden die Rolle der logischen Architektur – eines implementierungsneutralen Modells in der Lösungsdomäne – in der modellbasierten Entwicklung und ihre Abgrenzung zur Diensthierarchie erläutert. Es wurde erklärt, dass weitere Architekturtreiber außer funktionalen Anforderungen die Gestalt der logischen Architektur beeinflussen können. Zu diesen Architekturtreibern zählen z.B. Qualitätsmerkmale oder eine bestehende Hardware-Topologie. Daraus folgte, dass die Gestalt einer logischen Architektur prinzipiell unabhängig von der Struktur einer Diensthierarchie ist.

Das Hauptaugenmerk dieses Kapitels lag auf einer eigenschaftserhaltenden Transformation einer Diensthierarchie in ein Netzwerk logischer Komponenten. Die eingeführte Transformation gewährleistet die Erhaltung aller spezifizierten Eigenschaften in der resultierenden Architektur. Genauer gesagt, das Verhalten des generierten Netzwerks simuliert das Verhalten der Diensthierarchie. Dank der Assoziativität der Komposition ist ein Netzwerk logischer Komponenten einfacher zu restrukturieren als eine Diensthierarchie. Dieses Netzwerk dient als Basis für die Restrukturierung der Architektur gemäß weiteren Architekturtreibern. Demzufolge stellt die Transition nur den ersten Schritt in Richtung einer guten Architektur dar.

KAPITEL 7

CASE-Tool-Unterstützung

Im Mittelpunkt dieses Kapitels steht die Realisierung des vorgestellten Ansatzes in einem CASE-Tool. In Abschnitt 7.1 wird eine Erweiterung des Werkzeugs AutoFocus um die dienstbasierte Modellierungstechnik vorgestellt. In Abschnitt 7.2 wird erläutert, wie die Konsistenz dienstbasierter Modelle mittels eines Model Checkers überprüft werden kann.

Inhalt

7.1. Dienstbasierte Erweiterung von AutoFocus

Das am Lehrstuhl für Software & Systems Engineering der TU München entwickelte Werkzeug AutoFocus[1] wurde im Rahmen dieser Arbeit um die dienstbasierte Modellierungstechnik erweitert.

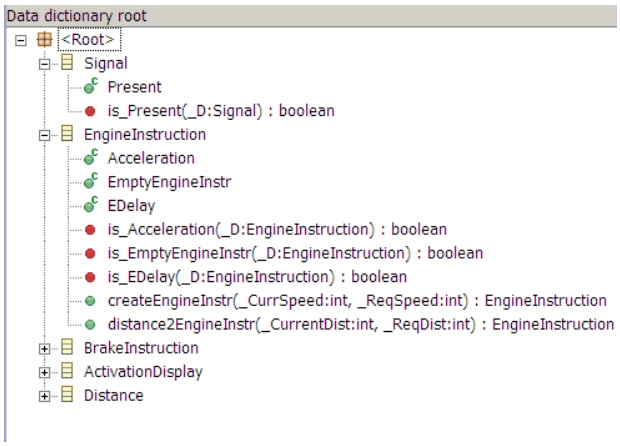


Abbildung 7.1.: Definition von Datentypen

AutoFocus ist ein Eclipse-basiertes CASE-Tool zur Modellierung und Analyse verteilter, reaktiver und digitaler Systeme. Das Tool wurde bis heute in erster Linie für die Modellierung der in Abschnitt 6.2.1 eingeführten komponentenbasierten Architektur entwickelt. In dem Tool können hierarchische Netzwerke logischer Komponenten modelliert werden. Eine Komponente hat eine syntaktische Schnittstelle und eine Verhaltensspezifikation. Die Verhaltensspezifikation einer atomaren Komponente wird durch einen Automaten (wie in Abschnitt 3.3.1 eingeführt) modelliert. Das Verhalten einer hierarchischen (zusammengesetzten) Komponente ergibt sich aus der in Abschnitt 4.3.5 definierten Komposition ihrer Teilkomponenten. Für die Analyse steht in AutoFocus eine Simulationsumgebung zur Verfügung, in der Modelle interaktiv ausgeführt werden können. Mehr dazu findet sich auf der Internetseite des Tools.

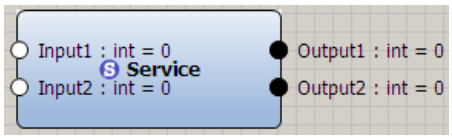


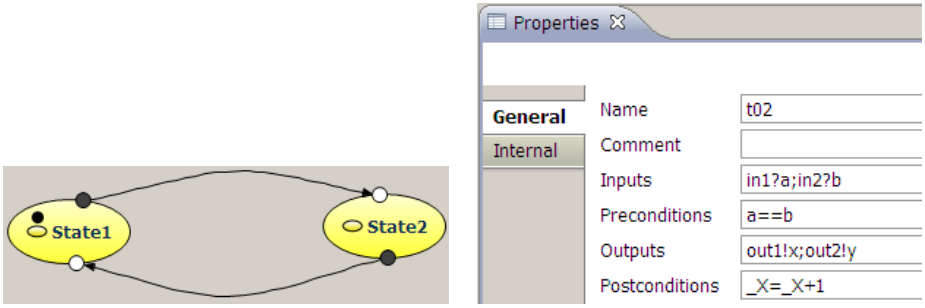
Abbildung 7.2.: Syntaktische Schnittstelle eines Dienstes

Im Folgenden wird die Erweiterung des Tools um die Modellierungstechnik zur Spezifikation von Nutzerfunktionen vorgestellt.

[1] http://af3.in.tum.de

Modellierung

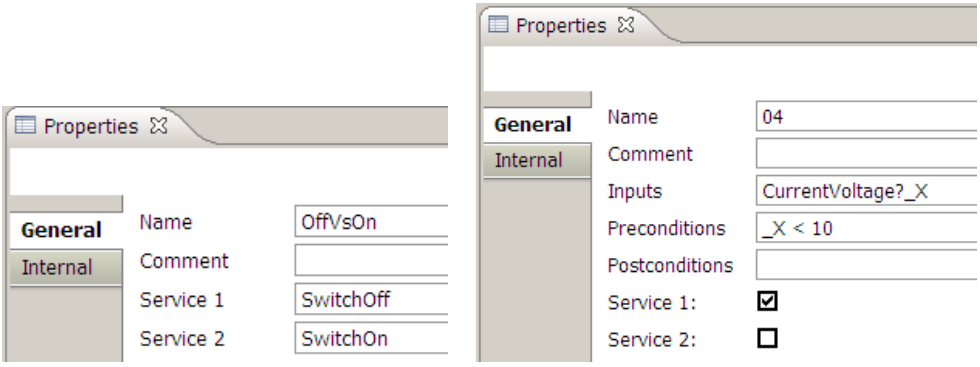
In AutoFocus können einfache und zusammengesetzte Datentypen sowie Familien darauf operierender Funktionen im *Data Dictionary* definiert werden (vgl. Abbildung 7.1). Während ein einfacher Datentyp eine Menge von Konstanten definiert, ist ein zusammengesetzter Datentyp auf anderen Datentypen aufgebaut.



(a) Automat (b) Transition

Abbildung 7.3.: Verhaltensspezifikation

Die syntaktische Schnittstelle eines Dienstes wird durch eine Menge von Ein- und Ausgabeports modelliert (vgl. Abbildung 7.2). Jeder Port hat einen Namen, einen Datentyp und einen Initialwert. Die Verhaltensspezifikation eines Dienstes wird durch einen Automaten modelliert (vgl. Abbildung 7.3). Der Automat besteht aus einer Menge von Kontrollzuständen und Transitionen zwischen diesen (vgl. Abbildung 7.3(a)). Transitionen wiederum bestehen aus Ein- und Ausgabemustern sowie Vor- und Nachbedingungen, die in einem separaten Editor definiert werden (vgl. Abbildung 7.3(b)). Die Syntax der Definition von Transitionen wurde bereits in Abschnitt 3.3.1 eingeführt.



(a) Zwei Dienste (b) Transition

Abbildung 7.4.: Priorisierungsdienst

Ein Priorisierungsdienst wird im Tool zwischen zwei Diensten definiert. In den Eigen-

schaften jedes Priorisierungsdienstes werden jeweils zwei Dienste in den obligatorischen Feldern `Service1` und `Service2` vermerkt (vgl. Abbildung 7.4(a)). Ein Priorisierungsdienst wird durch einen Automaten modelliert und hat keine eigene syntaktische Schnittstelle. Der Automat eines Priorisierungsdienstes ist dem Automaten eines Dienstes sehr ähnlich, seine Transitionen haben aber keine Ausgabemuster (vgl. Abbildung 7.4(b)). Die Eingabemuster seiner Transitionen sind auf den Eingabeports der in den Feldern `Service1` und `Service2` vermerkten Diensten definiert. In den Eigenschaften einer Transition wird vermerkt, ob sie den Dienst `Service1`, `Service2` oder keinen der beiden priorisiert.

Dienste können in AutoFocus hierarchisch strukturiert werden – die Kombination der Dienste ergibt die Verhaltensspezifikation des kombinierten Dienstes. Die Diensthierarchie wird im *Project Explorer* dargestellt (vgl. Abbildung 7.5).

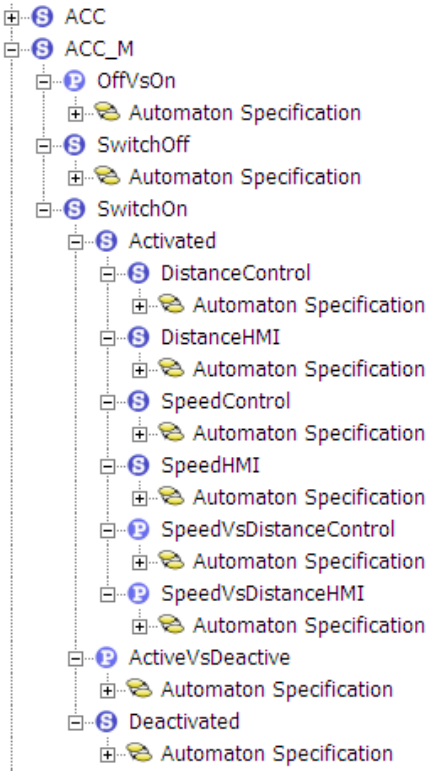


Abbildung 7.5.: Diensthierarchie

Die Modellierung der Dienstkombination in AutoFocus weicht von der Definition aus Abschnitt 4.3.1 auf Seite 64 unwesentlich ab. Nach dieser Definition ergibt sich die syntaktische Schnittstelle des kombinierten Dienstes aus der Vereinigung der Schnittstellen seiner Teildienste. In AutoFocus hat ein kombinierter Dienst seine eigene Schnittstelle – die Ports des kombinierten Dienstes werden durch logische Kanäle mit den entsprechenden Ports der Teildienste verbunden (vgl. Abbildung 7.6). Allerdings wird im Tool

sichergestellt, dass die Werte an den durch einen Kanal verbundenen Ports immer gleich sind, d.h. semantisch gesehen handelt es sich in diesem Fall um denselben Port. Im Gegensatz zu logischen Komponenten können die Ausgabeports mehrerer Teildienste mit demselben Ausgabeport des kombinierten Dienstes verbunden werden. Das entspricht genau der Definition der Dienstkombination, gemäß der Teildienste sowohl Ein- als auch Ausgabeports teilen können.

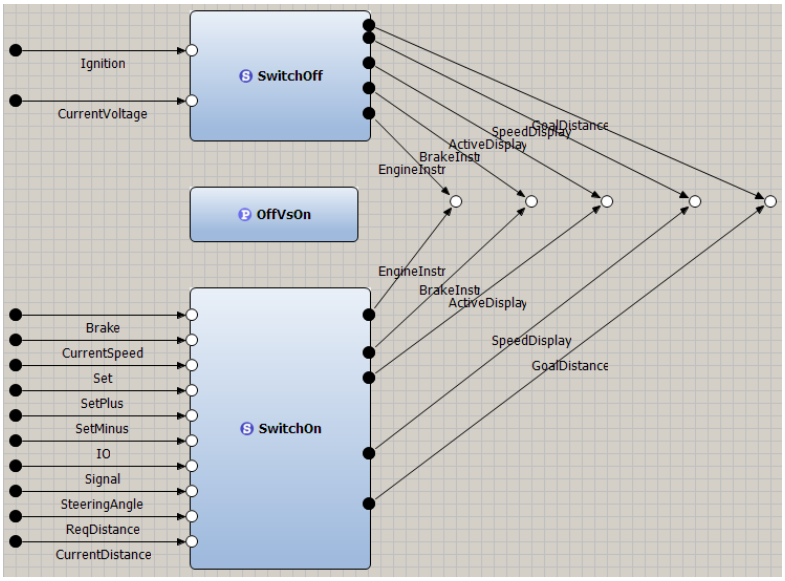


Abbildung 7.6.: Priorisierte Dienstkombination

Simulation

In der Simulationsumgebung von AutoFocus können dienstbasierte Modelle interaktiv ausgeführt werden (vgl. Abbildung 7.7). Sowohl ganze Diensthierarchien als auch Teilhierarchien können in die Simulationsumgebung geladen werden. In der *Running*-Sicht wird die Simulation gesteuert: sie kann gestartet, unterbrochen oder gestoppt werden. Außerdem kann die Simulation schrittweise vorwärts oder rückwärts ausgeführt werden.

In der *Input*-Sicht wird die Systemumgebung durch den Benutzer von AutoFocus simuliert – in jedem Simulationsschritt werden die aus der Umgebung zu erwartenden Daten per Hand eingegeben. Die Reaktion des Systems auf diese Eingaben (d.h. daraus resultierende Belegungen der Ausgabeports) wird in der *Output*-Sicht angezeigt.

In der *ControlState*-Sicht werden die Kontrollzustände angezeigt, in denen sich die laufenden Dienste aktuell befinden. In der *Graphical-Simulation-State*-Sicht wird der Automat eines ausgewählten Dienstes mit dem markierten aktuellen Kontrollzustand graphisch dargestellt. Die syntaktischen Schnittstellen der simulierten Dienste zusammen mit den aktuellen Portbelegungen werden in der *Graphical-Simulation-Structure*-Sicht

angezeigt.

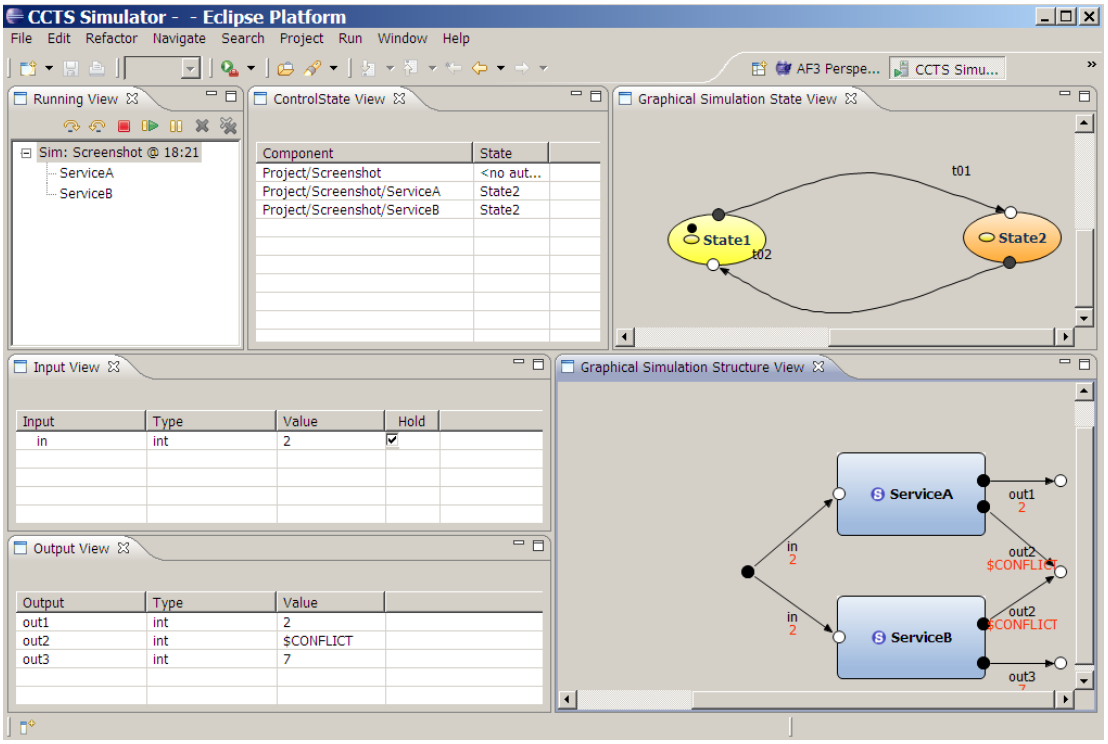
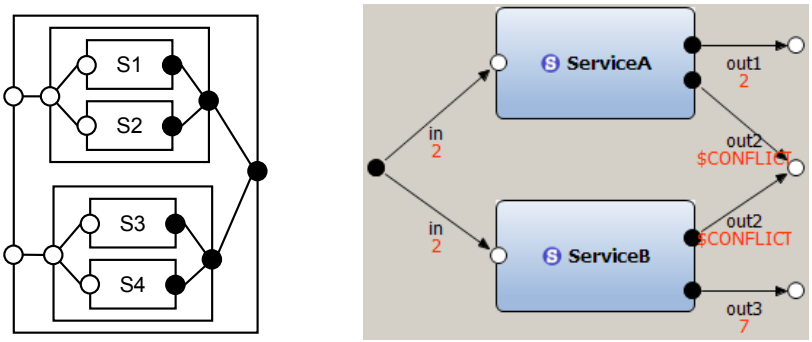


Abbildung 7.7.: Simulationsumgebung

Die Ausführung eines dienstbasierten Modells erfolgt gemäß den Definitionen aus Abschnitt 4.3 auf Seite 64. Unter anderem stellt die Simulationsumgebung sicher, dass Dienste an ihren gemeinsamen Ausgabeports immer dieselben Werte erzeugen. Wenn (nichtdeterministische) Dienste auf dieselben Eingaben an ihren gemeinsamen Ausgabeports mehrere Werte erzeugen können, wird die Schnittmenge dieser Ausgaben berechnet und ein Wert aus dieser Menge ausgegeben. Die Berechnung der Schnittmenge erfolgt induktiv über die Tiefe der Hierarchie. Beispielsweise können die Dienste `S1`, `S2`, `S3` und `S4` aus Abbildung 7.8(a) jeweils nur die Transitionen ausführen, die am gemeinsamen Ausgabeport denselben Wert erzeugen. Ist die Schnittmenge der möglichen Ausgaben leer, sind die Dienste gemäß den Definitionen aus Abschnitt 5.2.2 auf Seite 86 inkonsistent – am betroffenen Ausgabeport wird die Meldung `$CONFLICT` angezeigt (vgl. Abbildung 7.8(b)). Auf alle danach folgenden Eingaben reagiert das System mit dieser Meldung – die Eingabeströme mit dem eingegebenen Präfix liegen außerhalb des Definitionsbereichs des kombinierten Dienstes und die Reaktion auf sie ist undefiniert.



(a) Hierarchische Überlappung (b) Inkonsistenz zwischen Diensten

Abbildung 7.8.: Überlappende Dienste

7.2. Konsistenzprüfung in NuSMV

In Abschnitt 5.2.3 auf Seite 89 wurde bereits erläutert, wie die Konsistenz einer Diensthierarchie automatisch überprüft werden kann. Um die Umsetzbarkeit dieser Konzepte in einem Tool zu überprüfen, wurde im Rahmen der vorliegenden Arbeit eine Machbarkeitsstudie zur Analyse dienstbasierter Modelle in einem Model Checker erarbeitet. Die Wahl fiel auf NuSMV[2], da dieser symbolische Model Checker für die Analyse synchroner und asynchroner endlicher Transitionssysteme bestens geeignet ist. Ähnlich wie in AutoFocus haben Module (Transitionssysteme in NuSMV) Eingabe-, Ausgabe- und lokale Variablen. Eine Transition eines Moduls stellt einen Übergang von einer Variablenbelegung zur nächsten dar. Module können hierarchisch strukturiert werden, wobei ihre Synchronisation über gleichnamige Variablen erfolgt.

In Anhang B wird das Schema der Transformation dienstbasierter Modelle in die Eingabesprache von NuSVM anhand eines Beispiels im Detail erläutert. Die Transformation stellt sicher, dass das Verhalten kombinierter Module in NuSMV und das Verhalten der Dienstkombination in AutoFocus identisch sind. Außerdem werden kombinierte Module um einen Fehlerzustand erweitert, dessen Erreichbarkeit vom Model Checker überprüft wird.

Die Machbarkeitsstudie zeigt, dass die theoretischen Konzepte aus Abschnitt 5.2 in einem Model Checker mit wenig Aufwand realisiert werden können. Ein NuSMV-Back-End für AutoFocus wurde im Rahmen dieser Arbeit nicht erarbeitet.

[2] http://nusmv.irst.itc.it/

KAPITEL 8

Verwandte Arbeiten

Die vorliegende Arbeit hat viele Berührungspunkte zu anderen Arbeiten aus dem Bereich des Requirements Engineering und der modellbasierten Entwicklung. Diese Ansätze werden im Folgenden vorgestellt und von der vorliegenden Arbeit abgegrenzt. Die verwandten Arbeiten lassen sich in drei Klassen einteilen. In Abschnitt 8.1 werden Modellierungstechniken zur Spezifikation funktionaler Anforderungen dargestellt. Abschnitt 8.2 befasst sich mit den Ansätzen zur Konsistenzprüfung von Spezifikationen. In Abschnitt 8.3 werden verwandte Ansätze zur modellbasierten Entwicklung untersucht.

Inhalt

8.1. Anforderungsspezifikationen

In diesem Abschnitt werden die wichtigsten verwandten Ansätze zur Spezifikation funktionaler Anforderungen dargestellt. Eine umfangreiche Übersicht über aktuelle Arbeiten im Bereich des Requirements Engineering findet sich in der Arbeit von Cheng und Atlee [CA07].

Strukturierung von Anforderungen

Die Idee einer formalen Anforderungsspezifikation ist nicht neu und geht auf das 4-Variablen-Modell von Parnas und Madey [PM95] zurück. Das Modell beschreibt das Systemverhalten aus der Black-Box-Sicht durch eine Menge mathematischer Relationen, welche auf vier Mengen von Variablen definiert sind. Im Gegensatz zur vorliegenden Arbeit können Anforderungen im 4-Variablen-Modell nicht modular spezifiziert werden – es sieht keinen Operator zur Kombination von Anforderungen vor. Aus diesem Grund ist der Ansatz nicht skalierbar und kann zur Spezifikation heutiger Systeme nicht eingesetzt werden. Außerdem ist im 4-Variablen-Modell keine Verfeinerung zwischen Funktionen definiert, was die Integration der Anforderungsspezifikation in die modellbasierte Entwicklung erschwert.

Die Grundlage für die in dieser Arbeit vorgestellte dienstbasierte Modellierung ist die FOCUS-Theorie von Broy und Stølen [BS01], die ursprünglich zur Beschreibung logischer Architekturen entwickelt wurde. Die Weiterentwicklung der Theorie [Bro03] eignet sich jedoch nicht nur für die Spezifikation totaler Komponenten, sondern auch zur Beschreibung partieller Verhalten. Die hierarchische Strukturierung von Nutzerfunktionen – die Diensthierarchie – wurde in [Bro07a] formal definiert. Die Vorteile der in der vorliegenden Arbeit eingeführten operationellen Semantik gegenüber der denotationalen Semantik von Broy wurden in der Einführung zu Kapitel 4 erläutert.

Ein etablierter Ansatz zur Strukturierung von Anforderungen ist die zielorientierte Vorgehensweise von van Lamsweerde et al. [vL01], auch unter dem Namen KAOS bekannt. Ausgehend von Absichten von Stakeholdern werden in diesem Ansatz konkrete Anforderungen abgeleitet. So genannte „weiche“ Ziele werden schrittweise in „harte“ Ziele verfeinert, so dass die Spezifikation durch einen AND/OR-Graphen dargestellt wird. Die Knoten des Graphen sind Ziele (formalisiert in der Prädikatlogik bzw. LTL), zwischen denen zwei Arten von Verfeinerungsbeziehungen bestehen können: ein AND-Ziel ist erfüllt genau dann, wenn alle seine Teilziele erfüllt sind, ein OR-Ziel ist erfüllt genau dann, wenn eines seiner Teilziele erfüllt ist. Der KAOS-Ansatz ist für die Erhebung und Strukturierung von Anforderungen gut geeignet, bis auf die fehlende Möglichkeit, Wechselwirkungen zwischen Anforderungen zu modellieren. Unter anderem können aus explizit modellierten Widersprüchen zwischen Zielen neue Ziele bzw. Anforderungen abgeleitet werden. Für die automatische Konsistenzprüfung von Spezifikationen ist der Ansatz nicht geeignet. Im Gegensatz zur vorliegenden Arbeit können Widersprüche zwischen Zielen nur manuell entdeckt werden, was den KAOS-Ansatz praxisuntauglich macht.

Ein weiterer etablierter Ansatz im Requirements Engineering ist die *Feature Oriented Domain Analysis* (FODA) von Kang et al. [KCH+90] zur hierarchischen Strukturierung von Features einer Produktlinie. Nach Kang et al. sind Features „user-visible aspects or characteristics of the domain“. Die Features eines Systems werden grafisch und ohne eine Semantik mittels Kompositions- und Querbeziehungen strukturiert. Einige weiterführende Ansätze beschäftigen sich mit der Formalisierung von Feature-Bäumen mittels Grammatiken [BO92, CHE05] oder der Prädikatenlogik [Man02, SZW05]. In all diesen Ansätzen liegt das Hauptaugenmerk auf der Formalisierung von Beziehungen zwischen den Features. Beziehungen spielen auch im dienstbasierten Ansatz eine bedeutende Rolle, jedoch liegt der Schwerpunkt zunächst auf der formalen Spezifikation des Verhaltens der eigentlichen Features. Dies wird von keinem der existierenden Ansätze ausreichend behandelt. „As a consequence, these approaches focus on the analysis of dependencies, however abstracting away from the causes for these dependencies“ [Sch07].

In [Sch08] wird die Funktionalität eines reaktiven Systems in eine Hierarchie von Funktionen unterteilt. Genau wie in der vorliegenden Arbeit abstrahiert Schätz von einem komponentenbasierten Entwurf und befasst sich mit Nutzerfunktionen. Die Funktionen von Schätz haben eine Datenfluss-Schnittstelle (engl. data flow interface) zur Definition von Ein- und Ausgaben sowie eine Kontroll-Schnittstelle (engl. control interface) zur Aktivierung bzw. Deaktivierung von Funktionen. Das Verhalten einer Funktion wird mittels eines Transitionssystems definiert. Zur parallelen bzw. alternativen Kombination von Funktionen werden in [Sch08] zwei entsprechende Operatoren definiert. Im Gegensatz zu Diensten dürfen Funktionen direkt miteinander kommunizieren – durch ihre Kontroll-Schnittstelle kann eine Funktion eine andere (de-)aktivieren. In diesem Fall muss sich der Benutzer mit der internen Struktur des Systems (d.h. Schnittstellen zwischen Funktionen) befassen, was dem Prinzip der Black-Box-Spezifikation widerspricht (vgl. Abschnitt 3.3.2 auf Seite 32).

Interaktionsmuster

Die Interaktion eines Systems mit seiner Umgebung wird häufig durch UML-Sequenzdiagramme oder Message Sequence Charts [IT99] (MSCs) modelliert. Allerdings wird in der Praxis oft nur die graphische Notation dieser Modellierungstechnik verwendet, ohne eine fundamental ausgearbeitete, verstandene und allgemein anerkannte Semantik. Eine mögliche Semantik von MSCs wurde von Broy in [Bro05b] definiert. Allerdings wird in diesem Ansatz eine denotationale Semantik definiert, die für automatische Analysen von MSCs nicht ausreicht.

Im Ansatz von Sinnig et al. [SCK09] werden Transitionssysteme zur Definition der Semantik von Use Cases verwendet. Die Szenarien eines Use Cases werden zuerst durch separate Transitionssysteme formalisiert und anschließend zu einem Gesamtverhalten integriert. Allerdings sieht dieser Ansatz nur eine Interleaving-Semantik für die Kombination von Transitionssystemen vor. Das hat zur Folge, dass mehrere Szenarien, die das Verhalten an einem Ausgabeport gleichzeitig spezifizieren, nicht modelliert werden

können. Ähnliche Ansätze von Sinha et al. [SPW07] und Mizouni et al. [MSKD07] haben denselben Mangel.

In [Krü03, BKM07] stellen Krüger et al. „cross-cutting aspects of the corresponding architecture“ in den Mittelpunkt der Modellierung. Diese Aspekte (auch Dienste genannt) werden als Interaktionsmuster zwischen mehreren Komponenten definiert und mittels MSCs beschrieben. Genau wie in der vorliegenden Arbeit wird im Ansatz von Krüger et al. die Rolle überlappender und partieller Dienste hervorgehoben: „each individual service only represents a partial view on the collaborations within the system under consideration“. Die Gesamtspezifikation ergibt sich aus der Kombination einzelner Dienste. Außerdem wurde in [Krü03] der Begriff der Bevorrechtigung (engl. *preemption*) eingeführt, der in etwa dem Begriff der Priorisierung der vorliegenden Arbeit entspricht. Der Ansatz von Krüger setzt auf einer denotationalen Semantik auf, deren Nachteile bereits erläutert wurden.

Verschiedene Benutzerperspektiven

Die Idee, das Systemverhalten aus verschiedenen Benutzerperspektiven zu beschreiben, geht auf die Arbeiten von Finkelstein et al. [FGH+94] zurück. Eine Benutzerperspektive (engl. *viewpoint*) ist von Easterbrook und Nuseibeh in [EN95] wie folgt definiert. “ViewPoints are loosely coupled, locally managed, distributable objects which encapsulate partial knowledge about a system and its domain.“ Die Autoren definieren verschiedene Arten von Inkonsistenzen zwischen Benutzerperspektiven und stellen mehrere Methoden zur Verwaltung von Inkonsistenzen bereit (siehe auch [SNL+07, SFG10] für eine Übersicht). Im Gegensatz zur vorliegenden Arbeit sieht der Ansatz von Easterbrook und Nuseibeh eine automatische Konsistenzprüfung nicht vor. Außerdem werden Viewpoints in der Prädikatlogik ausgedrückt, was im Gegensatz zu Interaktionsmustern die direkte Verbindung zwischen Modellen und Benutzern des Systems erschwert.

Eine Voraussetzung für die Beschreibung eines Systems aus verschiedenen Perspektiven ist die Existenz eines Operators zur Kombination von (teils überlappenden) Modellen. Die Idee der Kombination partieller Verhalten ist nicht neu (siehe z.B. die Arbeiten von Hunter und Nuseibeh [HN98], Sabetzadeh und Easterbrook [SE06], Sabetzadeh et al. [SNEC07] oder Nejati und Chechik [NC08]). Die interessanteste Arbeit in diesem Bereich ist der Ansatz von Uchitel und Chechik [UC04], in dem die Kombination mehrerer Verhaltensmodelle derselben Komponente definiert wird. Ähnlich wie in der vorliegenden Arbeit ist diese Kombination als die minimale gemeinsame Verfeinerung der Teilmodelle definiert. Allerdings ist diese Kombination nur für konsistente Modelle definiert, d.h. für Modelle mit einer existierenden gemeinsamen Verfeinerung. Dies ist aber in der 3-wertigen Semantik von Uchitel und Chechik nicht immer garantiert. Der dienstbasierte Ansatz hat diese Einschränkung nicht. Außerdem sind in der dienstbasierten Spezifikation nichtdeterministische Interaktionsmuster zugelassen, was in [UC04] nicht der Fall ist. Darüber hinaus werden in der vorliegenden Arbeit Wechselwirkungen zwischen Modellen definiert, die in [UC04] nicht vorgesehen sind. Ein ähnlicher Ansatz ist in der Arbeit von

Fischbein und Uchitel [FU08] zu finden. Zusätzlich zu den oben genannten Mängeln ist der Kombinationsoperator in diesem Ansatz nur für Transitionssysteme über demselben Alphabet definiert.

Zustandsbasierte Ansätze

Die Automaten-Theorie [HU79], auf der die dienstbasierte Modellierung aufbaut, ist eine etablierte theoretische Fundierung vieler Ansätze im Software Engineering. Die meisten dieser Ansätze sind dem Entwurf und der Implementierung von Softwaresystemen zuzuordnen. Ihre Gemeinsamkeiten liegen in der Dekomposition eines Systems in Teilsysteme und der Kommunikation zwischen diesen. Diese Ansätze nehmen an, dass Automaten ein totales Modell des Systems darstellen, d.h. die Reaktion auf alle Eingaben aus dem Alphabet definieren [UC04]. Ansätze wie I/O-Automaten von Lynch und Tuttle [LT89] oder Interface-Automaten von de Alfaro und Henzinger [dAH01] sind für Beschreibung partieller Interaktionsmuster nicht geeignet. Die Eigenschaften wie *input-enabledness* von I/O-Automaten oder die Disjunktion von Ausgabeports von Interface-Automaten erschweren die Modellierung von Anforderungen aus verschiedenen Benutzerperspektiven, da das Verhalten an einem Ausgabeport durch unterschiedliche Automaten nicht definiert werden kann.

Aspektorientierte Entwicklung

Der Begriff des Dienstes aus der vorliegenden Arbeit lässt sich mit dem Begriff des Aspekts aus der aspektorientierten Entwicklung [AOS08] vergleichen. Aspektorientierte Entwicklung ist eine Methode der Systementwicklung, mit der verschiedene Aspekte eines Systems getrennt voneinander spezifiziert, entworfen und entwickelt werden. Die getrennt entwickelten Aspekte werden anschließend zum Gesamtsystem integriert. Dieses Paradigma hat seinen Ursprung in der aspektorientierten Programmierung (siehe z.B. die Arbeit von Aksit et al. [ARS09]) und dem aspektorientierten Design (siehe z.B. die Arbeit von Kienzle et al. [KAAK09]). In den letzten 10 Jahren wurde dem aspektorientierten Ansatz auch im Requirements Engineering viel Aufmerksamkeit gewidmet [EA009].

Der Begriff des aspektorientierten Requirements Engineering (AORE) geht auf die Arbeit von Rashid et al. [RSMA02] zurück, in der Querschnittsaspekte (engl. *crosscutting concerns*) wie Verfügbarkeit oder Sicherheit auf mehrere funktionale Anforderungen einzelner Stakeholder abgebildet werden. Dieser wie auch ähnliche Ansätze [BC04, SSR+05, BVMR07, MA09] sehen nur syntaktische Verbindungen zwischen Aspekten und Anforderungen vor – eine Kompositionssemantik wird nicht definiert. „Despite composition being central to AORE, most AORE techniques do not provide any formal composition semantics“ [RC08] (siehe die Arbeit von Rashid und Chitchyan [RC08] für eine Übersicht). Eine Ausnahme bilden wenige formale Ansätze, die im Folgenden erläutert werden.

In der Arbeit von Araujo et al. [AWK04] werden Szenarien und Aspekte in separaten

UML-Sequenzdiagrammen modelliert. Zur Integration von Aspekten zu einer szenarienbasierten Spezifikation werden diese Diagramme in endliche Automaten transformiert und anschließend kombiniert. Im Gegensatz zur vorliegenden Arbeit erfolgt die Kombination von Automaten manuell, was den Ansatz praxisuntauglich macht.

In [KR04] präsentieren Katz und Rashid einen Ansatz zur Verwaltung von Konflikten zwischen Aspekten und Anforderungen. Im Gegensatz zu Anforderungen, die immer nur zu einer UML-Klasse gehören, werden Aspekte über mehrere Klassen verteilt. Um die richtige Implementierung von Aspekten zu überprüfen, werden Aspekte in LTL-Formeln übersetzt. Die Erfüllung dieser Formeln wird anschließend in den an der Realisierung dieser Aspekte beteiligten Klassen überprüft. Allerdings wird in diesem Ansatz nicht erklärt, wie aus Anforderungen und Aspekten ein Klassendiagramm erzeugt wird.

Im Ansatz von Mehner et al. [MMT06] werden Basisanwendungsfälle (UML-Use-Cases) um Aspekte (*crosscutting* Use-Cases) durch die Graphentransformation erweitert. Dieser Ansatz ist stark an die Programmiersprache AspectJ[1] angelehnt. In den Basisanwendungsfällen werden die so genannten *pointcuts* definiert, an deren Stelle die Anwendungsfälle modifiziert werden. Aspekte werden in diesem Ansatz nur sequentiell komponiert – die parallele Kombination überlappender Aspekte ist nicht vorgesehen.

Ein weiterer Ansatz zur formalen Spezifikation von Aspekten ist in der Arbeit von Weston et al. [WCR09] zu finden. Aspekte werden in diesem Ansatz in der Prädikatlogik formalisiert. Obwohl die Semantik der Kombination von Aspekten formal definiert ist, sieht dieser Ansatz die automatische Konsistenzanalyse von Aspekten nicht vor.

Die Gemeinsamkeit aller aspektorientierten Ansätze liegt in einer klaren Trennung Querschnittsaspekte von der Basisfunktionalität. Im Gegensatz dazu werden in der vorliegenden Arbeit alle Nutzerfunktionen als gleichberechtigte Aspekte des Systemverhaltens betrachtet. Allerdings werden im dienstbasierten Ansatz bis jetzt nur funktionale Aspekte modelliert – Aspekte wie Sicherheit oder Verfügbarkeit liegen nicht im Fokus der Betrachtung. Viele dieser nichtfunktionalen Anforderungen können jedoch mittels Szenarien (vgl. [BCK98, Kapitel 4]) bzw. Zustandsautomaten (vgl. [JLHM91]) modelliert werden. Das bedeutet, dass auch nichtfunktionale Querschnittsaspekte durch überlappende Dienste spezifiziert werden können. Die Kombination „funktionaler“ und „nichtfunktionaler“ Dienste durch dieselben Operatoren ergibt das Gesamtverhalten des Systems. Dadurch können für nichtfunktionale Aspekte dieselben Analyseverfahren eingesetzt werden wie für funktionale. Dies bringt einen wesentlichen Vorteil gegenüber den aspektorientierten Ansätzen: „Most AORE techniques lack sufficient support for ensuring consistency of the separated aspects with other requirements and even fewer have any formal underpinning for such consistency checking“ [RC08].

[1] http://www.eclipse.org/aspectj/

Service-oriented Computing

Service-oriented Computing (SOC) ist ein Paradigma zur Modellierung und Kombination von Diensten, die von verteilten Systemen erbracht werden. Eine besondere Rolle spielt dabei die Orientierung an Geschäftsprozessen, deren Abstraktionsebenen die Grundlage für die Zusammensetzung (Orchestrierung) von Diensten sind: durch Zusammensetzen von Diensten niedriger Abstraktionsebene können Dienste höherer Abstraktionsebenen geschaffen werden. Die technische Implementierung einzelner Dienste wird hinter standardisierten Schnittstellen verborgen.

Im Forschungsprojekt SENSORIA [WBC+09] wurden formale Grundlagen sowie Modellierungstechniken erarbeitet, die es ermöglichen, dienstbasierte Modelle der Geschäftslogik einer Anwendung von ihrer technischen Implementierung (d.h. einer Infrastruktur von Web-Diensten [STK02]) zu trennen. Mit der eingeführten Sprache SRML [BFL08] wird die Geschäftslogik auf einem hohen Abstraktionsgrad modelliert. Bausteine eines SRML-Modells sind Module, die mehrere Komponenten sowie eine „bereitgestellte“ und mehrere „erforderliche“ Schnittstellen umfassen. Die Komponenten sind an der Erbringung des vom Modul bereitgestellten Dienstes beteiligt. Die erforderlichen Schnittstellen definieren externe Dienste, auf die das Modul angewiesen ist und die von anderen Modulen erbracht werden müssen. Die bereitgestellte Schnittstelle definiert den vom Modul bereitgestellten Dienst. Die Verbindung der internen Komponenten mit den Komponenten der externen Module wird durch die darunter liegende Middleware zur Laufzeit errichtet. Ein Dienst wird durch Interaktion der internen Komponenten und der dynamisch verbundenen externen Komponenten erbracht. Die Kombination formal definierter Module und Interaktionsmuster („interaction protocols“) zwischen diesen ergibt die Spezifikation eines dienstbasierten Modells (siehe die Arbeiten von Fiadeiro et al. [FLB07, ABFL07]).

Im Allgemeinen unterscheiden sich die Problemstellungen der SOC-Ansätze von denen der vorliegenden Arbeit (siehe die Arbeiten von Bocchi et al. [BFL08] und Krüger et al. [KMM06] für eine Gegenüberstellung). Während sich unser Ansatz mit der Komplexität in der Entwicklung eines Systems befasst, ist die Laufzeit-Komplexität die Herausforderung für die SOC-Ansätze. Dienste, die im Fokus der vorliegenden Arbeit liegen, sind „crosscutting elements of the system under consideration, describing partial views on the set of components in the system under consideration“ [BFL08]. Im Gegensatz dazu existiert das „Gesamtsystem“ als solches im SOC-Paradigma nicht: „services in the sense of SOC get combined at run time and redefine the way they are orchestrated as they execute; no ‘whole’ is given a priori and services do not compute within a fixed configuration of a ‘universe’“ [BFL08]. Im Falle des SRML-Ansatzes liegt ein weiterer Unterschied zur vorliegenden Arbeit darin, dass der Ansatz Netzwerke logischer Komponenten voraussetzt, die in etwa der logischen Architektur aus Kapitel 6 entsprechen. Dabei steht die dynamisch errichtete Interaktion zwischen diesen Netzwerken im Mittelpunkt der Betrachtung. Dienste aus der vorliegenden Arbeit abstrahieren von der logischen Architektur und definieren eine reine Black-Box-Spezifikation eines Systems. Das Hauptaugenmerk liegt auf der Integration von Interaktionsmustern desselben Systems mit einer statischen Schnittstelle.

Ein weiterer SOC-Ansatz ist in der Arbeit von Knapp et al. [KMWZ10] zu finden, in der dienstbasierte Modelle deklarativ spezifiziert werden. Genau wie in der vorliegenden Arbeit betrachten Knapp et al. Dienste als (zielorientierte) Anforderungen an die Implementierung. Das Verhalten einzelner Dienste wird durch eine lokale und deren Orchestrierung durch eine globale Logik modelliert. Durch die globale Logik können dynamische Konfigurationen von potentiell unendlich vielen Diensten formal spezifiziert werden. Im Gegensatz zur vorliegenden Arbeit liegt die dynamische Interaktion zwischen Diensten im Fokus der Betrachtung. Jedoch sieht der Ansatz die automatische Analyse dienstbasierter Modelle nicht vor. Außerdem wird in [KMWZ10] auf den Zusammenhang zwischen einer dienstbasierten Spezifikation und ihrer Implementierung nicht eingegangen. Es ist nicht offensichtlich, wie die Einhaltung einer deklarativen Spezifikation in einer Implementierung überprüft wird.

8.2. Konsistente Spezifikationen

In diesem Abschnitt stehen Arbeiten mit Schwerpunkt Konsistenzprüfung von Anforderungsspezifikationen im Mittelpunkt. Die Arbeit von Spanoudakis und Zisman [SZ01] gibt eine Übersicht über Ansätze zur Verwaltung von Inkonsistenzen.

Konsistenzprüfung

Der SCR-Ansatz von Heitmeyer et al. [HJL96, HKLB98] (SCR für *Software Cost Reduction*) setzt auf dem 4-Variablen-Modell [PM95] auf. Die Relationen zwischen Variablen werden in SCR mittels verschiedener Tabellen spezifiziert. Die wichtigsten davon sind *Mode-Transition*- und *Condition*-Tabellen. Eine *Mode-Transition*-Tabelle beschreibt ein Transitionssystem, dessen Zustände Betriebsmodi sind und dessen Transitionen durch Ereignisse ausgelöst werden. Eine *Condition*-Tabelle definiert Ereignisse einer oder mehrerer Variablen in bestimmten Zeitintervallen. Nach Heitmeyer et al. muss jede *Condition*-Tabelle total sein. Dies hat zur Folge, dass im Gegensatz zu den vorgestellten Diensten, das Verhalten an einer Variable ausschließlich in einer Tabelle definiert werden muss – überlappende Modelle sind dadurch ausgeschlossen. Ein weiterer Vorteil des dienstbasierten Ansatzes im Vergleich zu SCR ist die Möglichkeit einer hierarchischen Strukturierung der Systemfunktionalität: SCR sieht nur eine flache Unterteilung vor, was die Modellierung umfangreicher Systeme erschwert.

In [HT09] konzentrieren sich Hummel und Thyssen auf die Spezifikation einer einzigen Komponente durch mehrere Anforderungen, statt auf die Dekomposition eines Systems in mehrere Komponenten. Anforderungen werden in Form strombasierter I/O-Tabellen spezifiziert. Eine I/O-Tabelle besteht aus mehreren Segmenten, die in etwa dem Begriff des Dienstes in der vorliegenden Arbeit entsprechen. Ein Segment umfasst mehrere Zeilen, die mögliche Ein- und Ausgabemuster an der syntaktischen Schnittstelle der Komponente beschreiben. Das Gesamtverhalten der Komponente ergibt sich aus der parallelen Komposition der Segmente. Die Tabellen können auf Inkonsistenzen überprüft werden,

die in etwa den Definitionen aus Abschnitt 5.2.2 entsprechen. In [HT09] können nur Dienste mit derselben syntaktischen Schnittstelle kombiniert werden – alle Segmente einer Tabelle werden auf denselben Ein- und Ausgabeports definiert. Diese Tatsache erschwert die Modellierung von Anforderungen aus verschiedenen Benutzerperspektiven. Weiterhin sieht der tabellenbasierte Ansatz keine Wechselwirkungen zwischen Diensten vor, was in der Domäne multifunktionaler Systeme einen Mangel darstellt. Im Gegensatz zur polynomialen Laufzeit des Algorithmus aus Abschnitt 5.2.3, ist die Laufzeit des in [HT09] eingeführten Algorithmus zur Konsistenzprüfung doppelexponentiell.

Im PlayEngine-Ansatz [HKMP02] schlagen Harel et al. eine Kette von Modellen vor, beginnend mit „eingespielten“ Szenarien bis hin zum Design. Mit Hilfe der so genannten *PlayEngine* werden Szenarien interaktiv validiert, indem das Werkzeug eine Menge von Live Sequence Charts [DH01] (LSCs) einliest und den Benutzer mit dem dadurch spezifizierten System „spielen“ (d.h. dessen Verhalten simulieren) lässt. Auf jede Eingabe des Benutzers reagiert das Werkzeug mit einer Ausgabe, die von einem der LSCs definiert ist. Im Falle einer unerwünschten Reaktion modelliert der Benutzer eines der Interaktionsmuster um. Der Ansatz ist für interaktive Validierung von Interaktionsmustern geeignet, stellt aber, im Gegensatz zur vorliegenden Arbeit, keine automatische Konsistenzprüfung zur Verfügung.

Das gleiche Prinzip hat das Tool RAT von Bloem et al. [BCP+07]. Mit dem Werkzeug können Systemabläufe validiert werden, indem dem Benutzer mögliche Abläufe angezeigt werden, welche die eingegebenen Anforderungen (aussagenlogischen Formeln) erfüllen. Widerspricht ein Ablauf den Erwartungen des Benutzers, werden die Anforderungen entsprechend angepasst. Außerdem können die Formeln auf logische Widersprüche untersucht werden. Das Tool ist nur für Analyse einer kleinen Menge von Anforderungen geeignet. Es sieht weder hierarchische Strukturierung von Anforderungen noch explizite Modellierung von Wechselwirkungen zwischen diesen vor.

In [AEY03] befassen sich Alur et al. mit implizitem (engl. *implied*) Verhalten in szenarienbasierten Spezifikationen (siehe auch ähnliche Ansätze von Muccini [Muc03] und de Sousa et al. [dSMUK07]). Die Autoren argumentieren, dass durch die Kombination mehrerer MSCs implizites Verhalten auftreten kann, da ein MSC nur eine lokale Sicht auf das Gesamtverhalten modelliert. Implizites Verhalten ist ein Interaktionsmuster, das in jeder möglichen Implementierung der Spezifikation auftritt, jedoch von keinem MSC erfasst wird. Der in [AEY03] eingeführte Algorithmus entdeckt diese impliziten Szenarien automatisch. Außerdem überprüft der Algorithmus, ob die Spezifikation vollständig ist, d.h. ob es für jede Eingabe eine definierte Ausgabe gibt. In der vorliegenden Arbeit wird die Synthese impliziter Dienste nicht betrachtet, stattdessen liegt das Hauptaugenmerk auf der Synthese von Priorisierungsdiensten, welche die Widerspruchsfreiheit zwischen Interaktionsmustern sicherstellen.

In vielen komponentenbasierten Ansätzen (wie z.B. von de Alfaro und Henzinger [dAH01], Carrez et al. [CFN03], Gössler und Sifakis [GS05], Gössler et al. [GGMC+07] oder Hennicker et al. [HJK10]) stehen funktionale Eigenschaften der Komposition logischer Komponenten im Mittelpunkt der Betrachtung. Diese Art der Komposition entspricht in

etwa der Komposition aus Abschnitt 4.3.5. Im Gegensatz dazu wird in der vorliegenden Arbeit ausschließlich auf Eigenschaften der Kombination von Interaktionsmustern desselben Systems eingegangen.

In der Domäne des dienstbasierten Computings existieren mehrere Ansätze, wie z.B. von Fantechi et al. [FGL+08]) zur Überprüfung funktionaler Eigenschaften in einer dienstbasierten Architektur. Genau wie in der vorliegenden Arbeit wird in [FGL+08] die Semantik von Diensten durch Transitionssysteme definiert. Mittels eines Model Ckeckers wird die Erfüllung funktionaler Eigenschaften automatisch überprüft. Die Unterschiede zwischen dem SOC-Paradigma und unserem Ansatz wurden bereits in Abschnitt 8.1 erläutert.

Feature Interaction

Der Kombination von Diensten wurde in der Telekommunikation viel Aufmerksamkeit gewidmet (siehe [CKMRM03] für eine Übersicht). In dieser Domäne sind Dienste (*Features* genannt) modulare Erweiterungen der Basisfunktion, die sequentiell zusammengestellt werden können. Ein Telekommunikationsdienst setzt sich danach aus der Basisfunktion und beliebig vielen Features zusammen. Die Kombination von Features und die damit verbundene Problematik der *Feature Interaction* ist Gegenstand einer Reihe von Arbeiten (siehe z.B. die Arbeiten von Keck und Kühn [KK98], Felty und Namjoshi [FN03] oder Nejati et al. [NSC+08]). Ein prominenter Ansatz zur Lösung dieses Problems stammt von Jackson und Zave [JZ98]. Die Autoren schlagen mehrere Simulations- und Analyseverfahren zur Identifizierung ungewollter Feature Interactions vor. Außerdem werden in [Zav03] mehrere Designmuster zur Auflösung dieser Konflikte definiert. Designmuster bedeuten im Allgemeinen, dass in der Spezifikation der Implementierung vorgegriffen wird. Außerdem sind Features, im Gegensatz zu Diensten, total und können damit partielle und überlappende Interaktionsmuster nicht spezifizieren.

Konfliktlösung durch Priorisierung

Die Idee der Synthese einer Steuerkomponente (entspricht einem Priorisierungsdienst in der vorliegenden Arbeit) zur Erfüllung einer Eigenschaft durch ein Modell geht auf die Arbeit von Ramadge und Wonham [RW89] zurück. In diesem Ansatz wird ein Supervisor erzeugt, der eine Anlage gemäß einer Eigenschaft steuert. Das Verhalten des Supervisors ergibt sich aus der Komposition des Automaten der Eigenschaft mit demjenigen der Anlage. Im Gegensatz zu einem generierten Priorisierungsdienst, der nur Eingaben liest und einen Dienst priorisiert, steuert der Supervisor die Eingaben der Anlage und kontrolliert deren Ausgaben (vergleichbar mit der logischen Architektur aus Abschnitt 6.2.2). Obwohl sich die Problemstellungen der beiden Ansätze unterscheiden, setzt der Synthesealgorithmus aus Abschnitt 5.4.2 auf der Idee aus [RW89] auf.

Auf der Idee der Erzeugung von Supervisors zur Steuerung von Systemen setzt auch der Ansatz von D'Souza und Gopinathan [DG08], genau wie der Ansatz von Hay und Atlee [HA00], auf. Die Autoren befassen sich mit der Integration von teilweise wider-

sprüchlichen Features. In diesen Ansätzen besteht ein System aus der Basisfunktionalität und mehreren Features (Supervisors), die dem System „empfehlen“ (d.h. steuern), wie eine Reihe von Eigenschaften zu erfüllen ist. Der vorgeschlagene Algorithmus erzeugt für jede Eigenschaft ein Feature, das die Erfüllung dieser Eigenschaft sicherstellt. Im Gegensatz zu den etablierten Features aus der Telekommunikation [KK98] sind Features aus [DG08] gegenüber Konflikten „tolerant“. Das System wird nur von den Features gesteuert, die nicht in Konflikt mit einem höher priorisierten Feature stehen (für die zu erfüllenden Eigenschaften ist eine Rangordnung definiert). Steht ein Feature nicht mehr im Konflikt zu anderen, darf es das System weiter steuern. Im Gegensatz zu diesem Ansatz erzeugt der Algorithmus aus Abschnitt 5.4.2 auf Seite 97 einen Priorisierungsdienst, der die Erfüllung aller Eigenschaften gleichzeitig sicherstellt. Eine statische Rangordnung von Eigenschaften ist deshalb nicht notwendig.

Die vorgestellte Synthese eines Priorisierungsdienstes ähnelt der Synthese eines Schedulers für nebenläufige Echtzeitsysteme (siehe z.B. die Arbeit von Altisen et al. [AGS00]). Ein Scheduler von Tasks wird in der Regel gemäß zeitlichem Verhalten einzelner Tasks, zeitlichen Randbedingungen bzw. allgemeinen Scheduling-Regeln erzeugt. Der Scheduler aktiviert bzw. deaktiviert einen Task abhängig von definierten Ereignissen (wie z.B. Zeitüberschreitungen) und beeinflusst dabei die Eingaben des Tasks nicht. Im Gegensatz zur vorliegenden Arbeit befassen sich die Scheduling-Algorithmen ausschließlich mit zeitlichen Randbedingungen, die nur eine Unterklasse funktionaler Randbedingungen aus Abschnitt 5.4.2 darstellen.

Im Ansatz von Kugler et al. [KPP09] wird ein Kontroller zur Einhaltung von Eigenschaften, gegeben durch eine Menge von LSCs, generiert. Die Erfüllung von Eigenschaften wird in dieser Arbeit als ein Spiel zwischen dem System und seiner Umgebung definiert. Um das Spiel zu gewinnen, versucht das System alle Eigenschaften zu erfüllen, während die Umgebung als Ziel hat, durch ihre Eingaben das System in einen Fehlerzustand zu führen. Der Algorithmus aus [KPP09] erzeugt den Kontroller (d.h. die Implementierung einer Gewinn-Strategie), indem die Transitionssysteme aller LSCs kombiniert, alle „verlierenden“ Transitionen zur Einhaltung von *safety*-Eigenschaften eliminiert und zusätzliche Wächter zur Einhaltung von *liveness*-Eigenschaften hinzugefügt werden. Folgt das System den Transitionen des Kontrollers, sind alle LSC-Eigenschaften garantiert erfüllt. Die Problemstellung dieses Ansatzes unterscheidet sich von der aus der vorliegenden Arbeit. Während in [KPP09] die Konsistenzprüfung von LSC-Eigenschaften im Fokus der Betrachtung steht, strebt der Algorithmus aus Abschnitt 5.4.2 ein Modell von Anforderungen an, das sich leicht in eine logische Architektur überführen lässt. Der Übergang von einem monolithischen, aus LSCs erzeugten Transitionssystem zu einer logischen Architektur kann dagegen nicht automatisch erfolgen.

8.3. Modellbasierte Entwicklung

In diesem Abschnitt werden verwandte Ansätze zur modellbasierten Entwicklung untersucht (siehe die Arbeit von France und Rumpe [FR07] für eine allgemeine Übersicht). Sie

lassen sich in zwei Klassen einteilen: Ansätze zum Übergang von Anforderungen zu Architekturen und Ansätze, die verschiedene Abstraktionsebenen eines Systems vorsehen. Die Arbeit von Galster et al. [GEM06] gibt eine Übersicht über Ansätze zum Übergang von Anforderungen zu Architekturen.

Von Anforderungen zu Architekturen

Bei der modellbasierten Entwicklung werden unterschiedliche Aspekte eines Systems durch Modelle beschrieben. Ein klassisches Beispiel der modellbasierten Entwicklung ist die Anwendung der Modellierungssprache UML [OMG04]. Während funktionale Anforderungen in UML oft mittels Use-Case- und Sequenzdiagrammen modelliert werden, werden zur Beschreibung von Architekturen Klassen-, Zustands- und Komponentendiagramme verwendet. Allerdings setzen all diese Modellierungstechniken auf unterschiedlichen Formalismen auf. „Die Stärken einzelner in der UML enthaltener Theorien sind nicht integriert und daher auch nicht integriert nutzbar“ [BR07].

Der Synthese komponentenbasierter Modelle aus deklarativen Anforderungen bzw. Szenarien wurde in den letzten Jahren viel Aufmerksamkeit gewidmet (siehe die Arbeiten von Hennicker und Knapp [HK07], Letier et al. [LKMU08], Uchitel et al. [UBC09], oder Harel et al. [HKM10]). Beispielsweise bilden Szenarien und *safety*-Eigenschaften den Ausgangspunkt für den Synthesealgorithmus von Damas et al. [DLvL06]. Der Algorithmus – genau wie die meisten Ansätze zur Synthese von Zustandsautomaten aus Szenarien (siehe [LDD06] für eine Übersicht) – integriert die Szenarien in der Art, dass deren Kombination alle Eigenschaften erfüllt. Der wichtigste Mangel all dieser Ansätze ist das Problem der Zustandsexplosion in den erzeugten Modellen. Im schlechtesten Fall wächst der Zustandsraum des erzeugten Automaten exponentiell oder sogar doppelexponentiell mit der Größe der Eingabe der szenarienbasierten Spezifikation [HK01, KS09]. Im Gegensatz dazu kombinieren Dienste die Vorteile partieller Szenarien und ausführbarer Zustandsautomaten und ermöglichen somit Validierung durch Simulation sowie automatische Analyse von Spezifikationen. Beim Übergang zur logischen Architektur werden Dienste mittels zusätzlicher Steuerkomponenten komponiert, anstatt sie zu einem monolithen Automaten zu integrieren.

In [KM04] bilden Krüger und Mathew Dienste auf ein Netzwerk von Komponenten ab. Den Ausgangspunkt für diesen methodologischen Ansatz bilden Anwendungsfälle, aus denen Rollen und Dienste (d.h. Interaktionsmuster zwischen Rollen) abgeleitet werden. Die Rollen werden zu Komponenten verfeinert, auf welche die gewonnenen Dienste abgebildet werden. Die Komponenten zusammen mit den Diensten bilden dann eine logische Architektur des Systems. Wegen des eingesetzten Algorithmus zur Synthese von Zustandsautomaten aus MSCs (wie in [KGSB99] eingeführt) wird in [KM04] angenommen, dass die erzeugten Dienste total sind. Diese Annahme ist eine wesentliche Einschränkung des Ansatzes, da die meisten Szenarien partiell sind [UC04]. Im Gegensatz dazu sieht unser Ansatz die Kombination partieller Verhalten derselben Komponente vor.

In [vL03, vL09] beschreibt van Lamsweerde den Übergang von zielorientierten Spezifi-

kationen über funktionale Anforderungen zu abstrakten Architekturen. Dieser Ansatz verwendet sehr unterschiedliche Modelle für die drei Abstraktionsebenen. Da die Transformation zwischen diesen Modellen sehr kompliziert ist, wird in [vL03] eine Menge von Transformationsmustern definiert, die den Übergang zur Architektur unterstützen. Diese Transformationsmuster decken nur eine Subklasse von Zielen ab – für die restlichen Ziele erfolgt die Transformation manuell. Im Gegensatz dazu setzt die Diensthierarchie auf derselben theoretischen Grundlage wie die logische Architektur auf. Dies erleichtert einen automatischen Übergang von Anforderungsspezifikationen zur Architektur.

Ein weiterer interessanter Ansatz ist in der Arbeit von Schätz [Sch05] zu finden, in dem logische Komponenten aus Funktionen abgeleitet werden. Die Definition einer Funktion entspricht in etwa der Definition eines Dienstes. Genau wie Dienste müssen Funktionen nicht notwendigerweise total sein. Allerdings dürfen sie keine gemeinsamen Ausgabeports haben. Demzufolge ist es nicht möglich, das Verhalten an einem Ausgabeport durch mehrere Funktionen zu spezifizieren. Unser Ansatz hat diese Einschränkung nicht und ist deswegen für eine Spezifikation aus verschiedenen Benutzerperspektiven besser geeignet.

Eine wichtige Arbeit im Bereich der modellbasierten Entwicklung ist der generative Ansatz von Czarnecki et al. [CE00, Cza04]. Charakteristisch für die generative Software-Entwicklung ist die automatische Erzeugung von Programmcode aus komponentenbasierten Schablonen mit Hilfe eines Generators. Folglich ist dieser Ansatz – wie auch viele andere modellbasierte Ansätze, beispielsweise von Schätz [Sch07] – im Vergleich zu dem Dienstmodell auf einem deutlich niedrigeren Abstraktionsgrad angesiedelt.

Use Case Maps von Buhr [Buh98] kombinieren das Verhalten und die Struktur des Systems in einem Modell, indem Anwendungsfälle auf Komponenten abgebildet werden. Im Gegensatz zum vorgestellten Ansatz erfordert diese Abbildung manuelle Intervention des Benutzers.

Ein weiterer Vertreter der modellbasierten Entwicklung ist der in AutoFocus integrierte Ansatz AutoRAID [SFGP05] zur Strukturierung von Anforderungen. Durch iterative Überprüfungen und Konsolidierungen wird in AutoRAID ein integriertes Modell von Anforderungen und logischen Komponenten erstellt. Anforderungen werden in AutoRAID textuell erfasst und mit Design-Artefakten verknüpft. Die Strukturierung von Anforderungen erfolgt entlang der existierenden logischen Architektur. Im Gegensatz dazu konzentriert sich der dienstbasierte Ansatz auf eine designunabhängige Strukturierung und insbesondere Analysen von Anforderungen.

Verschiedene Abstraktionsebenen

Die EAST ADL[2] (Electronics Architecture and Software Technology – Architecture Definition Language) ist eine Modellierungstechnik zur Beschreibung Software-intensiver Elektrik/Elektronik-Systeme in Automobilen auf fünf unterschiedlichen Abstraktionsebenen beginnend mit informellen Anforderungen und nutzersichtbaren Features bis hin

[2] http://www.atesst.org

zu Implementierungsdetails wie Betriebssystemkonstrukten und elektronischer Hardware. Hierbei liegt das Hauptaugenmerk der EAST ADL auf der Beschreibung der Struktur des Systems, das Verhalten wird zu einem großen Teil in externen Werkzeugen modelliert. Der dienstbasierte Ansatz beschreibt dagegen die Struktur und das Verhalten integriert. Auch liegt bei der EAST ADL eine formale Grundlage für die Modelle nicht im Vordergrund, sondern vielmehr die Aufteilung der System-Entwicklung in Artefakte, die unterschiedlichen Abstraktionsebenen zugeordnet werden können. Der dienstbasierte Ansatz geht hier einen Schritt weiter und definiert einen formalen und automatischen Übergang von der funktionalen Spezifikation zur logischen Architektur.

Die industrielle Entwicklungspartnerschaft AUTOSAR[3] (Automoive Open System Architecture) strebt die Entwicklung einer standardisierten Software-Infrastruktur an, die eine modulare und weitgehend plattformunabhängige Software-Entwicklung erlaubt. Die von AUTOSAR vorgeschlagene Software-Architektur ist eine Schichten-Architektur, die plattform- und anwendungsspezifische Softwareanteile voneinander trennt und dadurch eine Grundlage für eine plattformunabhängige und modulare Entwicklung von Funktionen schafft. Ein durchgängiger Entwicklungsprozess von Anforderungen zu AUTOSAR-Softwarekomponenten existiert jedoch noch nicht. Der in der vorliegenden Arbeit vorgestellte Übergang zur logischen Architektur dient als Basis für einen durchgängigen Entwicklungsprozess und hat zum Ziel, eine Systematik zur Software-Entwicklung entlang von Abstraktionsebenen anzugeben. Als mögliches Ziel ist hier auch die Generierung von AUTOSAR-Software-Komponenten denkbar.

Im Forschungsprojekt SENSORIA[4] wurden drei Abstraktionsebenen eines dienstbasierten Modells erarbeitet (vgl. die Arbeit von Bocchi et al. [BFL+09]). In der bereits erwähnten Modellierungssprache SMRL werden die oberen zwei Abstraktionsebenen modelliert. Auf der abstraktesten der beiden Ebenen wird die Geschäftslogik einer Anwendung durch Aktivitätsdiagramme modelliert, die auf den darunter liegenden und von den technischen Details abstrahierten Diensten aufbauen. Die COWS-Modelle [LPT08] spezifizieren auf dem niedrigsten Abstraktionsgrad die technische Implementierung dieser Dienste. Beispielsweise werden auf dieser Ebene Web-Dienste sowie Such- und Bindungsmechanismen für deren Orchestrierung modelliert. Die Unterschiede in den Problemstellungen zwischen dem SOC-Paradigma und der vorliegenden Arbeit wurden bereits in Abschnitt 8.1 erläutert.

[3] http://www.autosar.org/
[4] http://www.sensoria-ist.eu

KAPITEL 9

Zusammenfassung und Ausblick

In diesem Kapitel werden die wesentlichen Ergebnisse der vorliegenden Arbeit zusammengefasst. In Abschnitt 9.2 werden die Anwendbarkeit und Skalierbarkeit der erzielten Ergebnisse in Form von zwei Fallstudien nachgewiesen. Schließlich werden mögliche Weiterentwicklungen des Ansatzes aufgezeigt.

Inhalt

9.1. Zusammenfassung

Der Beitrag dieser Arbeit ist ein formales Framework für die Spezifikation und Konsistenzprüfung der Funktionalität eines Systems. Durch die vorgestellte Modellierungstechnik wird ein System so spezifiziert, dass sich dessen Verhalten aus der Kombination einzelner Interaktionsmuster zwischen dem System und dessen Umgebung ergibt. Durch diese Technik kann der Stakeholder nur das Black-Box-Verhalten des Systems festlegen und vermeidet somit zwangsweise Vorentscheidungen bezüglich des Systemdesigns.

Zwei Arten der Strukturierung In der vorliegenden Arbeit wurden zwei orthogonale Arten der Systembeschreibung eingeführt sowie ihre Zusammenhänge und Unterschiede erläutert. Die strukturelle Dekomposition unterteilt das Verhalten in ein Netzwerk kollaborierender Komponenten und ist für die Beschreibung von Architekturen somit bestens geeignet. Die funktionale Dekombination strukturiert das Verhalten in eine Hierarchie von Interaktionsmustern, abstrahiert von jeglichen Implementierungsdetails und ist somit besser für die Anforderungsanalyse geeignet. In Abschnitt 3.2 wurde gezeigt, dass sich die beiden Arten der Strukturierung nicht gegenseitig ersetzen sondern ergänzen. Sie bilden die Grundlage für eine zweidimensionale Modellierung der Funktionalität eines Systems (vgl. Abbildung 3.2 auf Seite 22).

Der Schwerpunkt der vorliegenden Arbeit lag auf der funktionalen Dekombination. Es wurde eine Spezifikationstechnik erarbeitet, die es erlaubt, das Systemverhalten aus verschiedenen Benutzerperspektiven zu beschreiben. Im ersten Schritt werden Teilverhalten des Systems aus unterschiedlichen Benutzerperspektiven von zuständigen Stakeholdern in separaten Modellen spezifiziert. Dabei können Aspekte des Verhaltens widersprüchlich oder für eine Benutzergruppe irrelevant (und somit in der Perspektive dieser Benutzergruppe unterspezifiziert) sein. Erst in einem darauffolgenden Schritt werden diese Modelle zu einer Gesamtspezifikation integriert. Somit entsteht eine Spezifikation aus sich gegenseitig beeinflussenden Modellen unterschiedlicher Perspektiven. Die Vorteile dieser Modellierungsart wurden bereits in Abschnitt 3.4.2 auf Seite 44 erläutert.

Operationelle Semantik von Interaktionsmustern Der eingeführten Spezifikationstechnik liegt eine operationelle Semantik zugrunde. Dienste werden mittels I/O-Automaten modelliert und anschließend zu zusammengesetzten Diensten kombiniert. Während jeder der Dienste nur einen Aspekt des Verhaltens modelliert, ergibt ihre Kombination das Gesamtverhalten. Für die Integration von Diensten wurden zwei Operatoren zur einfachen bzw. priorisierten Kombination von Automaten definiert. Im Gegensatz zu klassischen Ansätzen können diese Operatoren Automaten mit gemeinsamen Ausgabevariablen kombinieren. Die Kombination entspricht in etwa der synchronen Nebenläufigkeit von Automaten mit gemeinsamen Variablen.

Die einfache Dienstkombination ist kommutativ, assoziativ und idempotent. Das bedeutet, dass die Reihenfolge, in der Dienste kombiniert werden, sich nicht auf das resultierende Gesamtergebnis auswirkt. Die priorisierte Dienstkombination ist nicht assoziativ,

jedoch kommutativ, distributiv und idempotent. Dies hat zur Folge, dass die Struktur der Diensthierarchie Auswirkung auf das Gesamtverhalten hat, falls die Spezifikation Priorisierungen enthält.

In den Fallstudien (vgl. Abschnitt 9.2) erwiesen sich Interaktionsmuster als eine geeignete Spezifikationstechnik, Nutzerfunktionen so allgemein wie möglich und so einschränkend wie nötig zu formulieren. Diese Technik bietet die notwendige funktionsorientierte Sicht, um komponentenübergreifende Funktionen spezifizieren und analysieren zu können. Die erarbeitete operationelle Semantik dient als Grundlage für die in der Praxis notwendige Werkzeugunterstützung.

Integration in die FOCUS-Theorie Der vorgestellte Ansatz steht in der Vielzahl von Modellierungstechniken nicht isoliert da. Er beruht auf der umfassenden Modellierungstheorie für Anforderungsspezifikationen und den Architekturentwurf multifunktionaler Systeme, eingeführt durch Broy. Die operationelle dienstbasierte Spezifikation aus der vorliegenden Arbeit ist eine Instanz der annotierten Diensthierarchie von Broy.

Konsistenz und Korrektheit In dieser Arbeit wurden zwei Arten von Konflikten in einer funktionalen Spezifikation definiert, Inkonsistenzen zwischen Diensten und Verletzungen von Randbedingungen. Die Ersteren sind anwendungsunabhängig und führen zu einer inkonsistenten Spezifikation, die mehrdeutig und nicht implementierbar ist. Die Letzteren (anwendungsspezifischen) führen zu einer inkorrekten Spezifikation, die den Anforderungen von Stakeholdern nicht entspricht.

Für die beiden Arten von Konflikten wurden automatische Suchalgorithmen erarbeitet. Die Umsetzbarkeit dieser Algorithmen in einem Werkzeug wurde anhand eines kleinen Beispiels in einem Model Checker nachgewiesen. Dadurch wurde auch der Mehrwert einer operationellen Semantik deutlich.

In multifunktionalen Systemen werden die meisten Konflikte durch unberücksichtigte Wechselwirkungen zwischen Nutzerfunktionen verursacht. Die Beseitigung solcher Konflikte durch automatische Generierung von Priorisierungsdiensten ist ein wesentlicher Beitrag dieser Arbeit. Die resultierenden Priorisierungsdienste stellen sicher, dass eine dienstbasierte Spezifikation konsistent und korrekt ist.

Integration zweier Abstraktionsebenen Der vorgeschlagene Ansatz ist in ein 3-Ebenen-Modell entlang den klassischen Phasen des Entwicklungsprozesses integriert. In der vorliegenden Arbeit wurde ein automatischer Übergang von Anforderungen zum Design vorgestellt. Die eingeführte eigenschaftserhaltende Transformation einer Diensthierarchie in ein Netzwerk logischer Komponenten garantiert die Erhaltung aller spezifizierten Eigenschaften in der resultierenden Architektur. Dank der Assoziativität der Komposition ist ein Netzwerk logischer Komponenten einfacher zu restrukturieren als eine Diensthierarchie. Dieses Netzwerk dient als Basis für die Strukturierung der Architektur gemäß weiteren Architekturtreibern. Demzufolge stellt die Transition nur den ersten Schritt in

Richtung einer guten Architektur dar.

Werkzeug und Fallstudien Auf Basis der eingeführten operationellen Semantik wurde im Rahmen dieser Arbeit ein prototypisches Werkzeug für die dienstbasierte Modellierung implementiert, das die theoretischen Ergebnisse der Arbeit praktisch nutzbar macht. In diesem Werkzeug wurden zwei Fallstudien erarbeitet, welche die Anwendbarkeit und Skalierbarkeit der eingeführten Modellierungstechnik nachweisen.

9.2. Fallstudien

Die Anwendbarkeit der eingeführten Modellierungstechnik wurde in der vorliegenden Arbeit durch zwei Fallstudien nachgewiesen. Sie wurden vollständig in der dienstbasierten Erweiterung von AutoFocus aus Kapitel 7 modelliert und simuliert.

Die Fallstudien haben bestätigt, dass die werkzeugunterstützte Modellierung von Interaktionsmustern sowie die Unterteilung der Funktionalität in Systemmodi folgende Vorteile mit sich bringen:

- Formale Spezifikationen werden für den Anforderungsanalytiker verständlich und intuitiv.
- Einzelne Interaktionsmuster eines komplexen multifunktionalen Systems können durch sehr einfache Automaten modelliert und hierarchisch strukturiert werden. Deren Integration zu einer Gesamtspezifikation erfolgt automatisch.

Aus den Fallstudien geht außerdem hervor, dass der dienstbasierte Ansatz skalierbar ist.

ACC-Steuerung

In einer Fallstudie (Anhang C.1 auf Seite 160) wurde die Funktionalität der ACC-Steuerung aus Kapitel 2 vollständig modelliert. Die Steuerung umfasst sieben Nutzerfunktionen, allen voran die Geschwindigkeits- und Abstandsregelung, und kann sich in einem von 21 Systemmodi befinden. Je nach Systemmodus ist das Verhalten der Nutzerfunktionen unterschiedlich. Insgesamt mussten bei der Modellierung 31 mögliche (zusammengesetzte) Interaktionsmuster berücksichtigt werden, beispielsweise das Verhalten der Nutzerfunktion *Abstandsregelung* im folgenden Modus: das ACC ist eingeschaltet, die momentane Geschwindigkeit liegt zwischen 80 und 180 km/h, kein Blinker wurde betätigt, das Fahrzeug befindet sich auf einer Gerade und ein vorausfahrendes Fahrzeug ist erkannt.

In der Fallstudie wurde ein wesentlicher Vorteil der eingeführten Spezifikationstechnik deutlich: die Möglichkeit, die funktions- und modusbasierten Modellierungen zu kombinieren (vgl. Abschnitt 3.4.3 auf Seite 47). Die Funktionalität der ACC-Steuerung wurde zuerst in eine Hierarchie von Systemmodi und Moduswechseln unterteilt. Anschließend wurden für jeden Modus Nutzerfunktionen modelliert, die in diesem Modus den Nutzern

zur Verfügung stehen. Durch diese Art der Strukturierung konnten einzelne Nutzerfunktionen eines komplexen multifunktionalen Systems durch einfache Automaten modelliert werden. Statt alle möglichen Modi in jedem Dienst zu berücksichtigen und dadurch komplexe Automaten zu erzeugen, werden Nutzerfunktionen in primitive Interaktionsmuster unterteilt und Moduswechsel durch mehrere Priorisierungsdienste modelliert. Durch die in AutoFocus automatisierte Kombination der eigenständig modellierten Interaktionsmuster wurde eine Beschreibung des Verhaltens des Gesamtsystems erzeugt.

Siemens Abfüllanlage

Die Fallstudie aus Abschnitt C.2 auf Seite 165 dient zur Überprüfung der Mächtigkeit der dienstbasierten Modellierungstechnik im industriellen Umfeld. Im Kooperationsprojekt FLASCO mit dem Siemens Bereich „Automation and Drives“ wurde eine bereits vorher existierende Anlage zur Abfüllung von Flaschen formal spezifiziert. Den Ausgangspunkt für die Modellierung bildete eine textuelle, ca. 200 Seiten umfassende Spezifikation [Did06], die vom Hersteller der Anlage zur Verfügung gestellt wurde.

Die Diensthierarchie der Anlage umfasst 58 atomare Dienste und sieben Priorisierungsdienste. Dadurch wurden ca. 170 teils sehr komplexe – aus mehreren Mustern zusammengesetzte – Interaktionsmuster modelliert. Aus der Fallstudie geht hervor, dass die dienstbasierte Spezifikationstechnik für die Modellierung real existierender Systeme mit einer Vielzahl interagierender Nutzerfunktionen gut geeignet ist, da die hierarchische Unterteilung von Funktionen in Modi und Interaktionsmuster sowie die automatisierte Dienstkombination den Ansatz skalierbar machen.

Das zweite Ergebnis dieser Fallstudie ist eine Liste von ca. 30 Unstimmigkeiten in der gegebenen textuellen Spezifikation, allen voran Unterspezifikationen und Widersprüchen zwischen Anforderungen, die während der Modellierung und Simulation (manuell) entdeckt wurden. Beispiele der identifizierten Unterspezifikationen und Widersprüche finden sich in Abschnitt C.2.3 auf Seite 173. Aus der Fallstudie geht hervor, dass schon alleine ein formales Modell von Nutzerfunktionen und dessen Simulation (ohne Verifikation) die Qualität einer funktionalen Spezifikation verbessert.

9.3. Ausblick

Aus den Ergebnissen dieser Arbeit ergeben sich viele Anknüpfungspunkte für weitere Themen, die auf dem präsentierten Ansatz aufbauen oder ihn ergänzen.

Traceability Beim Übergang von Anforderungen zum Design wurde nur eine automatische Modelltransformation betrachtet. Häufig ist es aber sinnvoll, ausgehend von einer Spezifikation eine Architektur manuell zu erstellen. In diesem Fall muss nachträglich überprüft werden, ob alle Anforderungen in der Architektur richtig implementiert wurden. Für diese Überprüfung sowie für die Wartbarkeit des Systems sind die so genannten

Traceability-Links zwischen Anforderungen und den an deren Implementierung beteiligten logischen Komponenten essentiell wichtig.

Es ist eine interessante Herausforderung, für eine gegebene Diensthierarchie und eine logische Architektur die Traceability-Links zwischen einzelnen Diensten und logischen Komponenten automatisch festzustellen. Dafür wird eine minimale Menge logischer Komponenten ermittelt, die für die Erfüllung einer spezifizierten Eigenschaft (d.h. eines Dienstes) notwendig ist. Eine Skizze dieser Idee findet sich in den Ansätzen zur kompositionalen Verifikation von Nam et al. [NMA08] und Cobleigh et al. [CGP03].

Kontinuierliches Zeitmodell Der dienstbasierten Modellierungstechnik aus der vorliegenden Arbeit liegt ein diskretes Zeitmodell zugrunde. Will man von eingebetteten Software-Kontrollsystemen abstrahieren und stattdessen physikalische Prozesse modellieren, ist ein kontinuierliches Zeitmodell notwendig. In diesem Fall kann die eingeführte operationelle Semantik auf Timed Automata von Alur [Alu99] oder hybride Automaten von Lynch [LSV03] umgestellt werden. Bei der Umstellung muss vor allem die Synchronisation von Diensten umdefiniert werden, da die Ausführung einer Transition nicht mehr ein konstantes Zeitintervall benötigt.

Echtzeitanforderungen Auf Basis von Diensten mit einem kontinuierlichen Zeitmodell können unter anderem Echtzeitanforderungen modelliert und analysiert werden. Ein typisches Beispiel einer solchen Anforderung ist die Definition der Reaktionszeit einer Nutzerfunktion. Ein Katalog von Echtzeitanforderungen an Kontrollsysteme findet sich in der Arbeit von Jaffe et al. [JLHM91]. Diese Anforderungen können mittels „kontinuierlicher“ Dienste modelliert und durch die modifizierten Kombinationsoperatoren mit den restlichen Anforderungen kombiniert werden.

Nichtfunktionale Anforderungen Der dienstbasierte Ansatz unterstützt bis jetzt nur die Modellierung funktionaler Anforderungen – Anforderungen an die Sicherheit oder Verfügbarkeit eines Systems liegen nicht im Fokus dieser Arbeit. Viele dieser nichtfunktionalen Anforderungen können mittels Szenarien (vgl. [BCK98, Kapitel 4]) bzw. Zustandsautomaten (vgl. [JLHM91]) modelliert werden. Das bedeutet, dass auch nichtfunktionale Anforderungen durch überlappende Dienste spezifiziert werden können. Die Kombination „funktionaler“ und „nichtfunktionaler“ Dienste durch die Operatoren aus Abschnitt 4.3 ergibt das Gesamtverhalten des Systems. Dadurch können für nichtfunktionale Anforderungen dieselben Analyseverfahren eingesetzt werden, wie für funktionale.

ANHANG A

Beweise

Alle in dem vorliegenden Anhang aufgezählten Beweise wurden bereits vollständig oder skizzenhaft in [BH09a] bzw. [BH09c] veröffentlicht.

Satz A.1 (Reduktion auf Erreichbarkeitsproblem)**.** *Für einen Dienst S und eine Eigenschaft φ gilt Gleichung* (5.3) *auf Seite 93 genau dann, wenn Gleichung* (5.4) *gilt.*

Beweis. Der Beweis folgt aus der Feststellung, dass für S, S_φ und $C = \hat{S_\varphi} \parallel S$ folgende Gleichung gilt: $\{\alpha \in \text{Attr}(C) \mid \neg\alpha(e)\} = \{\alpha \mid \alpha \in \text{Attr}(S) \wedge \alpha \in \text{Attr}(S_\varphi)\}$. Die Gleichung wird durch Widersprüche bewiesen.

"$\Rightarrow$" Wegen der Eingabevollständigkeit von S_φ ist $\langle\!\langle C \rangle\!\rangle$ nicht leer. Angenommen, es gibt zwei in C erreichbare Belegungen mit $\beta \in \text{Succ}_C(\alpha)$, so dass folgende Prädikate gelten: $\alpha \vdash \mathcal{I}_S$ und $\alpha(e)$. Aus Gleichung (5.3) folgt $\mathcal{I}_S \Rightarrow \mathcal{I}_\varphi$. Die Definitionen von $\hat{\cdot}$ und $\cdot \parallel \cdot$ sowie das Prädikat $\alpha \vdash \mathcal{I}_C$ implizieren $\alpha \vdash (\mathcal{I}_\varphi \Leftrightarrow \neg\alpha(e)) \wedge \alpha \vdash \mathcal{I}_S$. Daraus folgt, dass das Prädikat $\neg\alpha(e)$ gelten muss, was im Widerspruch zur Annahme steht.

Angenommen, es gilt $\text{En}_S(\alpha)$ und $\alpha(e) \neq \beta(e)$. Da S_φ eingabevollständig ist, muss nur der Fall betrachtet werden, in dem $\beta \in \text{Succ}_S(\alpha)$ und $\beta \in \text{Succ}_{\hat{S}_\varphi}(\alpha)$ gelten. Außerdem kann es in keinem Ablauf von $\hat{S}_\varphi$ vorkommen, dass $\alpha(e)$ und $\neg\beta(e)$ gelten, da es keine Transition gibt, die aus einem Fehlerzustand zu einem normalen Zustand führt. Daraus folgt die Annahme, dass $\neg\alpha(e)$ und $\beta(e)$ gelten. Gemäß der Definition der Dienstkombination muss eine Belegung $\alpha_i \in \text{Attr}(S)$ existieren, so dass entweder $\alpha_i = \alpha$ oder $\langle \ldots \alpha_i \alpha_{i+1} \ldots \alpha_j \rangle \in \langle\!\langle C \rangle\!\rangle$ mit $\alpha = \alpha_j$ und für alle $\forall i \leq k < j : \neg\text{En}_S(\alpha_k)$. Im ersten Fall gilt $\langle \ldots \alpha\beta \rangle \in \langle\!\langle S \rangle\!\rangle$ und gemäß der Annahme $\langle \ldots \alpha\beta \rangle \in \langle\!\langle S_\varphi \rangle\!\rangle$. Daraus folgt $\neg\beta(e)$. Im zweiten Fall gilt $\langle \ldots \alpha_i \rangle \in \langle\!\langle S_\varphi \rangle\!\rangle$. Gemäß der Definition von $\cdot \parallel \cdot$ gilt $\langle \ldots \alpha_i \alpha_{i+1} \ldots \alpha \rangle \in \langle\!\langle S_\varphi \rangle\!\rangle$. Wegen der Eingabevollständigkeit von S_φ gilt $\langle \ldots \alpha_i \alpha_{i+1} \ldots \alpha\beta \rangle \in \langle\!\langle S_\varphi \rangle\!\rangle$. Daraus

folgt $\neg\beta(e)$, was im Widerspruch zur Annahme steht.

"$\Leftarrow$" Angenommen, es gibt eine sowohl in S als auch in S_φ erreichbare Belegung α, so dass $\alpha \vdash \mathcal{I}$ und $\neg\exists\overline{\alpha} \in \Lambda(V_\varphi) : \overline{\alpha} \vdash \mathcal{I}_\varphi \wedge \alpha \overset{V\cap V_\varphi}{=} \overline{\alpha}$. Gemäß dieser Annahme gelten folgende Prädikate: $\neg\alpha(e)$ und $\exists\overline{\alpha} \in \Lambda(V_\varphi) : \overline{\alpha} \vdash \mathcal{I}_\varphi \wedge \alpha \overset{V}{=} \overline{\alpha}$.

Angenommen, es gibt eine sowohl in S als auch in S_φ erreichbare Belegung α, so dass $\neg\exists\overline{\alpha},\overline{\beta} \in \Lambda(V_\varphi) : \overline{\beta} \in \mathrm{Succ}_{S_\varphi}(\overline{\alpha}) \wedge \alpha \overset{V\cap V_\varphi}{=} \overline{\alpha} \wedge \beta \overset{V\cap V_\varphi}{=} \overline{\beta}$. Gemäß der Definition von $. \parallel .$ gelten folgende Prädikate $\alpha \in \mathrm{Attr}(S \parallel \hat{S}_\varphi)$ und $\beta \in \mathrm{Succ}_{S\parallel\hat{S}_\varphi}(\alpha) \setminus \mathrm{Succ}_{S\parallel S_\varphi}(\alpha)$. Wenn $\mathrm{En}_S(\alpha)$ gilt, dann muss auch $\beta(e)$ gelten. Aus Gleichung (5.4) folgt $\alpha(e)$, was im Widerspruch zur Erreichbarkeit von α in S_φ steht.

Aus $\neg\mathrm{En}_S(\alpha)$ folgt $\beta \notin \mathrm{Succ}_S(\alpha)$, was im Widerspruch zu den Annahmen steht. □

Satz A.2 (Korrektheit der Synthese)**.** *Für zwei Dienste S_1, S_2, eine Eigenschaft φ und einen von Algorithmus 1 auf Seite 100 erzeugten Priorisierungsdienst P gelten folgende Aussagen. (a) $S_1 \parallel^P S_2 \models \varphi$ und (b) P ist der schwächste eigenschaftserhaltende Priorisierungsdienst, d.h. für jeden anderen Priorisierungsdienst R, der der Ordnung "0 vor 1,2" folgt, folgende Aussage gilt: wenn $S_1 \parallel^R S_2 \models \varphi$ dann $\langle\!\langle S_1 \parallel^R S_2 \rangle\!\rangle \subseteq \langle\!\langle S_1 \parallel^P S_2 \rangle\!\rangle$.*

Beweis. Zunächst müssen folgende zwei Invarianten für Algorithmus 1 nachgewiesen werden. Es wird dabei folgende Notation benutzt: $\mathrm{Attr}(C, A)$ bezeichnet die Menge aller Zustände, die in der Kombination C von einem Zustand $\alpha \in A$ aus erreichbar sind.

Für alle $\alpha \in A$ gilt

$$\alpha \in Err^n \Leftrightarrow \forall p \in \{0,1,2\} : \mathrm{Attr}(C, \mathrm{Succ}_p(\alpha)) \cap E_C \neq \emptyset \tag{A.1}$$

$$(\alpha,\beta,p) \in Pri^n \Leftrightarrow p \in \{0,1,2\} \wedge \alpha,\beta \notin Err^n \wedge \beta \in \mathrm{Succ}_p(\alpha) \wedge p \neq 0 \Rightarrow \mathrm{Attr}(C, \mathrm{Succ}_0(\alpha)) \cap E_C \neq \emptyset \tag{A.2}$$

Die Erfüllung der beiden Invarianten kann durch strukturelle Induktion bewiesen werden.

(a) Gemäß Satz A.1 muss gezeigt werden, dass Gleichung (5.4) auf Seite 93 für $D = (S_1 \parallel^P S_2) \parallel \hat{S}_\varphi$ gilt. Wegen $\langle\!\langle P \rangle\!\rangle \subseteq \langle\!\langle S_\top \rangle\!\rangle$, $\langle\!\langle D \rangle\!\rangle \subseteq \langle\!\langle (S_1 \parallel^{S_\top} S_2) \parallel \hat{S}_\varphi \rangle\!\rangle$ gilt auch Gleichung (A.2) für D. Aus Gleichung (A.2) folgt, dass $\langle\!\langle D \rangle\!\rangle$ nur diejenigen Abläufe umfasst, in denen $S_1 \parallel^P S_2$ keinen Fehlerzustand erreicht.

(b) $S_\top$ ist die Menge aller möglichen Priorisierungsdienste. Demzufolge gelten folgende Aussagen für jeden beliebigen Priorisierungsdienst R: $\langle\!\langle R \rangle\!\rangle \subseteq \langle\!\langle S_\top \rangle\!\rangle$ und $\langle\!\langle S_1 \parallel^R S_2 \rangle\!\rangle \subseteq \langle\!\langle S_1 \parallel^{S_\top} S_2 \rangle\!\rangle$. Angenommen, es gibt einen Dienst R, der Aussage (b) verletzt. Demzufolge muss es einen Ablauf $\langle \alpha_1 \ldots \rangle \in \langle\!\langle S_1 \parallel^R S_2 \rangle\!\rangle \setminus \langle\!\langle S_1 \parallel^P S_2 \rangle\!\rangle$ geben, der φ erfüllt, und es muss eine Belegung α_i in diesem Ablauf geben, so dass α_i erreichbar von $S_1 \parallel^P S_2$ ist und $\alpha_{i+1} \notin \mathrm{Succ}_{S_1 \parallel^P S_2}(\alpha_i)$. Wenn der Zustand α_{i+1} von dem Algorithmus bereits analysiert wurde, muss er in der Menge der Fehlerzustände sein. Demzufolge gibt es gemäß Gleichung (A.1) eine Eingabesequenz $\alpha_{i+1} \ldots$, die unvermeidlich einen Fehlerzustand erreicht, was wiederum einen Widerspruch ergibt.

Wenn α_{i+1} noch nicht vom Algorithmus analysiert wurde, dann ist gemäß Gleichung (A.2) die Priorität $p(\alpha_i, \alpha_{i+1}) \neq 0$ und $S_1 \parallel^P S_2$ hat eine Transition vom Zustand α_i mit der Priorität 0. Das heißt, entweder missachtet R die Priorisierungsordnung oder α_{i+1} ist nicht erreichbar in $S_1 \parallel^R S_2$. □

Satz A.3 (Modulare Priorisierung)**.** *Für zwei Dienste S_1, S_2 und zwei Eigenschaften φ, ψ mit $S_1 \parallel^{P_\varphi} S_2 \models \varphi$ und $S_1 \parallel^{P_\psi} S_2 \models \psi$ gilt folgende Aussage: $S_1 \parallel^{P_\varphi \| P_\psi} S_2 \models \varphi \wedge \psi$.*

Beweis. Seien folgende Dienste $S_1 = (V_1, \mathcal{I}_1, \mathcal{T}_1)$, $S_2 = (V_2, \mathcal{I}_2, \mathcal{T}_2)$, $P_\varphi = (V_P^\varphi, \mathcal{I}_P^\varphi, \mathcal{T}_P^\varphi, p^\varphi)$, $P_\psi = (V_P^\psi, \mathcal{I}_P^\psi, \mathcal{T}_P^\psi, p^\psi)$, $S_\varphi = (V_\varphi, \mathcal{I}_\varphi, \mathcal{T}_\varphi)$ und $S_\psi = (V_\psi, \mathcal{I}_\psi, \mathcal{T}_\psi)$ gegeben, wobei $S_\varphi \models \varphi$ und $S_\psi \models \psi$.

Für alle $\xi \in \{\varphi, \psi\}$ und alle $\alpha, \beta \in \mathrm{Attr}(S_1 \parallel^{P_\xi} S_2 \parallel \hat{S}_\xi)$ gelten gemäß Satz A.1 folgende Aussagen:

$$\alpha \vdash \mathcal{I}_1 \wedge \mathcal{I}_2 \wedge \mathcal{I}_P^\xi \Rightarrow \neg\alpha(e), \tag{A.3}$$

$$\big(\mathrm{En}_{S_1 \parallel^{P_\xi} S_2}(\alpha) \wedge \beta \in \mathrm{Succ}_{S_1 \parallel^{P_\xi} S_2 \parallel \hat{S}_\xi}(\alpha)\big) \Rightarrow \big(\alpha(e) \Leftrightarrow \beta(e)\big). \tag{A.4}$$

Der Dienst $S_{\varphi,\psi}$ mit $S_{\varphi,\psi} \models \varphi \wedge \psi$ sei wie folgt definiert. $S_{\varphi,\psi} \stackrel{\text{def}}{=} (V_{\varphi,\psi}, \mathcal{I}_{\varphi,\psi}, \mathcal{T}_{\varphi,\psi})$ mit

$$V_{\varphi,\psi} \stackrel{\text{def}}{=} V_\varphi \cup V_\psi, \qquad \mathcal{I}_{\varphi,\psi} \stackrel{\text{def}}{=} \mathcal{I}_\varphi \wedge \mathcal{I}_\psi, \qquad \mathcal{T}_{\varphi,\psi} \stackrel{\text{def}}{=} \mathcal{T}_\varphi \wedge \mathcal{T}_\psi.$$

Es ist offensichtlich, dass der Dienst $S_{\varphi,\psi}$ deterministisch und eingabevollständig ist und folgende Formel gilt: $\langle\!\langle S_{\varphi,\psi} \rangle\!\rangle = \langle\!\langle S_\varphi \rangle\!\rangle \cap \langle\!\langle S_\psi \rangle\!\rangle$. Demzufolge umfasst das Verhalten des Dienstes genau diejenigen Abläufe, die sowohl φ als auch ψ erfüllen.

Sei die Dienstkombination C wie folgt definiert: $C \stackrel{\text{def}}{=} (V_C, \mathcal{I}_C, \mathcal{T}_C) \stackrel{\text{def}}{=} S_1 \parallel^{P_\varphi \| P_\psi} S_2$. Es muss gezeigt werden, dass $\langle\!\langle C \rangle\!\rangle \overset{V_{\varphi,\psi} \cap V_C}{\subseteq} \langle\!\langle S_{\varphi,\psi} \rangle\!\rangle$ gilt. Gemäß Satz A.1 müssen folgende zwei Aussagen bewiesen werden. Für alle $\alpha, \beta \in \mathrm{Attr}(C \parallel \hat{S}_{\varphi,\psi})$ gilt

$$\alpha \vdash \mathcal{I}_C \Rightarrow \neg\alpha(e), \tag{A.5}$$

$$\mathrm{En}_C(\alpha) \wedge \beta \in \mathrm{Succ}_{C \parallel \hat{S}_{\varphi,\psi}}(\alpha) \Rightarrow (\alpha(e) \Leftrightarrow \beta(e)). \tag{A.6}$$

Die Aussage $\alpha \vdash \mathcal{I}_C \Leftrightarrow \alpha \vdash \mathcal{I}_1 \wedge \mathcal{I}_2 \wedge \mathcal{I}_P^\varphi \wedge \mathcal{I}_P^\psi$ gilt. Da Gleichung (5.4) auf Seite 93 für φ und ψ gilt, gilt auch $\neg\alpha(e)$. Da P_φ und P_ψ eingabevollständig sind, gilt auch $\mathrm{En}_C(\alpha) \Rightarrow \mathrm{En}_{P_\varphi}(\alpha) \wedge \mathrm{En}_{P_\psi}(\alpha)$. Die Prämisse von Gleichung (A.6) impliziert, dass es eine gemeinsame Priorisierung von (α, β) in p^φ und p^ψ gibt. Gemäß Satz A.2 erfüllt die Transition (α, β) sowohl φ als auch ψ. □

Satz A.4 (Simulation zwischen Diensten und Komponenten)**.** *Sei eine Dienstkombination $S = S_1 \parallel^{S_P} S_2$ gegeben. Die entsprechende Komposition von Komponenten*

$$C_S \stackrel{\text{def}}{=} \mathit{Prio} \oplus S_1^p[o_1/o]_{o \in O_1} \oplus S_2^p[o_2/o]_{o \in O_2} \oplus \mathit{MUX}$$

simuliert den Dienst S, d.h. es gilt die Relation $\mathit{sim}(C, S)$ aus Definition 6.1 auf Seite 118.

Man beachte, dass der Satz auch für eine unpriorisierte Dienstkombination gilt. In diesem Sonderfall priorisiert die aus der Transformation resultierende Komponente Prio keine der beiden Komponenten.

Beweis. Es wird gezeigt, dass jeder stotternfreie Ablauf von C auch aus der Sprache von S ist. Dafür wird durch die Induktion über $i \in \mathbb{N}$ gezeigt, dass für einen gegebenen Ablauf $\rho_C = \alpha_0\alpha_1 \ldots$ mit

$$\alpha_0 \vdash \mathcal{I}_C \qquad \text{und} \qquad \forall i \in \mathbb{N} : (\alpha_{i+1} \in \mathrm{Succ}_C(\alpha_i) \wedge \forall v \in V_S : \alpha_i(v) \neq \bot)$$

ein Ablauf $\rho_S \in \langle\!\langle S \rangle\!\rangle$ existiert, so dass die Relation $sim(\rho_S, \rho_C)$ gilt.

Die Initialbelegungen der Eingabevariablen sowohl von S als auch von C sind durch das jeweilige Prädikat $\mathcal{I}$ nicht eingeschränkt. Demzufolge erfüllt jede Initialbelegung der Eingabevariablen von C das Prädikat $\mathcal{I}_S$. Die Initialbelegungen der lokalen Variablen sowohl von S als auch von C sind durch dasselbe Prädikat $\mathcal{I}_P \wedge \mathcal{I}_1 \wedge \mathcal{I}_2$ definiert. Die Initialbelegungen der Ausgabevariablen von C sind durch das Prädikat $\mathcal{I}_m = \mathcal{I}_1 \wedge \mathcal{I}_2$ des Multiplexers definiert (die Komponenten *Prio*, S_1^p und S_2^p haben keinen Einfluss auf die Initialbelegungen der Ausgabevariablen von C). Daraus folgt, dass $\alpha_0 \vdash \mathcal{I}_S$.

Induktionsschritt. Angenommen, es gibt eine erreichbare Belegung $\beta \in \Lambda(V_S)$ in S, so dass $\beta \overset{I}{=} \alpha_i$, $\beta \overset{L_S}{=} \alpha_{i+1}$ und $\beta \overset{O}{=} \alpha_{i+2}$ für ein $i \geq 0$. Das heißt, der endliche Ablauf von der Initialbelegung bis zum β ist das Präfix von ρ_S. Es werden zwei Fälle betrachtet:

$\underline{\neg\mathrm{En}(\beta)}$ Wenn β die letzte Belegung in einem endlichen Ablauf ist, dann sind die Belegungen der Eingabevariablen in α_i (und in β) die Eingaben von S_1^p und S_2^p im Zeitintervall $i+1$. In diesem Intervall – gemäß der Annahme – sind die Belegungen der lokalen Variablen aus L_S von C und die der entsprechenden Variablen in S im Zustand β identisch. Die totalen Komponenten S_1^p und S_2^p erzeugen im Zeitintervall $i+2$ entweder (1) den Wert $\bot$ an allen Ausgabeports oder (2) widersprüchliche Werte an ihren gemeinsamen Ausgabeports. Im Zeitintervall $i+1$ erzeugt die Komponente *Prio* am Ausgabeport `pp` entweder (a) den Wert 0 (keine Priorisierung) oder (b) den Wert 1 oder 2 zur Priorisierung einer der Komponenten. In den Fällen (1a) und (1b) erzeugt der Multiplexer auch den Wert $\bot$ im Zeitintervall $i+3$ – dies ergibt einen Stotterschritt, was im Widerspruch zur Annahme über ρ_C steht. In den Fällen (2a) und (2b) ist α_{i+2} die letzte Belegung von ρ_C, was nicht wahr sein kann.

$\underline{\mathrm{En}(\beta)}$ Es wird gezeigt, dass es eine Belegung $\gamma \in \mathrm{Succ}_S(\beta)$ gibt, so dass $\gamma \overset{I}{=} \alpha_{i+1}$, $\gamma \overset{L_S}{=} \alpha_{i+2}$ und $\gamma \overset{O}{=} \alpha_{i+3}$. Die erste Gleichung folgt direkt aus der Tatsache, dass die Eingaben weder in S noch in C eingeschränkt sind. Die Belegungen der Eingabevariablen in α_i (und in β) sind Eingaben von S_1^p und S_2^p im Zeitintervall $i+1$. In diesem Intervall sind die Belegungen lokaler Variablen aus L_S von C und S identisch. Daraus folgt, dass für jeden Nachfolger γ der Belegung β in S_i mit $i \in \{1, 2\}$ ein gleicher Nachfolger in S_i^p existiert. Der Wert am Port `pp` der Komponente *Prio* im Zeitintervall $i+1$ entspricht dem Ergebnis der Funktion $p(t)$ für die Transition t von β zu γ in der Dienstkombination S. Demzufolge

kann höchstens eine Menge der Nachfolger (entweder $\mathrm{Succ}_1(\beta)$ oder $\mathrm{Succ}_2(\beta)$) leer sein. Diejenige Komponente, die durch *Prio* zeitlich deaktiviert ist, erzeugt den Wert $\perp$ an allen Ausgabeports.

$\alpha_{i+1}(pp) = 0$. Keine der Komponenten ist priorisiert. Da die Dienstkombination S im Zustand β ausführbar ist, gibt es mindestens ein Paar der widerspruchsfreien Nachfolger von β in S_1 und S_2 oder wenn einer der Dienste (o.B.d.A. S_1) nicht ausführbar ist, erzeugt die Komponente S_1^p eine $\perp$-Belegung.

o.B.d.A. $\alpha_{i+1}(pp) = 1$. Gemäß der Semantik der Dienstkombination muss eine Transition $t_P \in T_P$ mit $p(t_P) = 1$ existieren. Dann muss es einen Nachfolger der Belegung β in S_1 und einen entsprechenden Nachfolger in S_1^p geben. Die Komponente S_2^p macht in diesem Zeitintervall einen Stotterschritt.

Der Multiplexer kann nur gleiche Werte an seinen entsprechenden Eingabeports verarbeiten und blockiert dadurch widersprüchliche Transitionen der Komponenten S_1^p und S_2^p. Demzufolge entsprechen die Ausgaben des Netzwerks C im Zeitintervall $i + 3$ der Ausgaben der Dienstkombination S. □

ANHANG B

Konsistenzprüfung in NuSMV

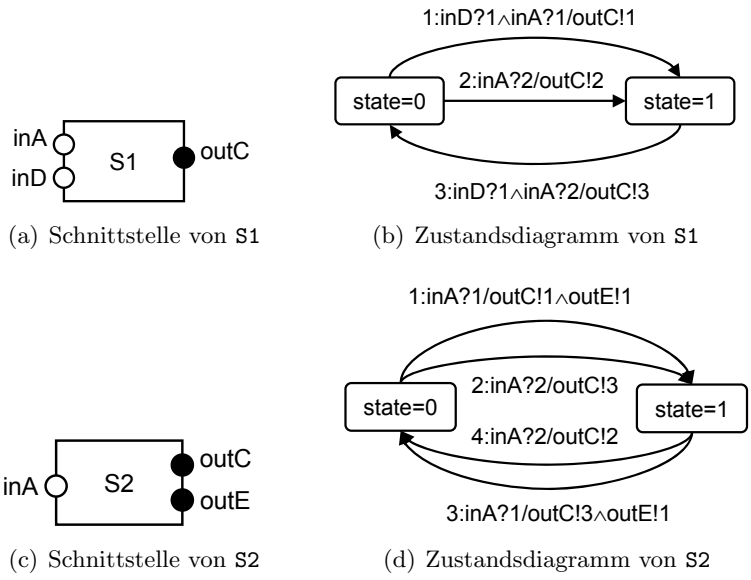
In diesem Anhang wird die Konsistenzprüfung dienstbasierter Modelle im Model Checker NuSMV[1] anhand eines Beispiels vorgeführt. NuSMV ist ein symbolischer Model Checker für die Analyse synchroner und asynchroner endlicher Transitionssysteme. Die zu überprüfenden Eigenschaften können entweder in Computation Tree Logic (CTL) oder Linear Temporal Logic (LTL) ausgedrückt werden. Auf die genaue Einführung der NuSMV-Syntax wird an dieser Stelle verzichtet und stattdessen auf die Internetseite des Modell Chckers verwiesen.

Man beachte, dass die Transformation von Diensten in die Eingabesprache von NuSMV (noch) nicht automatisch erfolgt. Der entsprechende Code-Generator wurde im Rahmen dieser Arbeit nicht erarbeitet. Das folgende Beispiel soll nur die Machbarkeit der automatischen Analyse mittels eines Model Checkers nachweisen. Die theoretischen Grundlagen für diese Art der Konsistenzprüfung wurden in Abschnitt 5.2.3 auf Seite 89 eingeführt.

In der Machbarkeitsstudie wird die Kombination von zwei Diensten aus Abbildung B.1 in die Eingabesprache von NuSMV übersetzt und anschließend auf Konsistenz überprüft. Dabei wird sichergestellt, dass das Verhalten des Modells in NuSMV und das der Dienstkombination identisch sind.

Der Dienst `S1` wird ins Modul `S1` übersetzt (vgl. Zeilen 1 bis 14 im folgenden NuSMV-Code). Das Modul hat zwei Eingabevariablen `inA` und `inD`, eine Ausgabevariable `outC` und eine lokale Variable `state`, in der der Kontrollzustand des Transitionssystems gespeichert wird. In Zeilen 5–6 werden die Initialbelegungen der Variablen definiert. Im Prädikat `enabled` wird definiert, für welche Eingaben und in welchen Zuständen das Transitionssystem nicht ausführbar ist (vgl. Zeilen 7–9). In Zeilen 10–14 werden vier

[1] http://nusmv.irst.itc.it/



(a) Schnittstelle von `S1`

(b) Zustandsdiagramm von `S1`

(c) Schnittstelle von `S2`

(d) Zustandsdiagramm von `S2`

Abbildung B.1.: Dienste in AutoFocus

Transitionen des Systems definiert. Drei davon entsprechen eins-zu-eins den Transitionen aus dem Zustandsdiagramm aus Abbildung B.1(b), die vierte macht eine Schleife, wenn immer das System nicht ausführbar ist – der Kontrollzustand und die Ausgabevariablen des Moduls werden nicht modifiziert. Während sich der Ausdruck der Form $v = w$ auf den aktuellen Zustand bezieht, gilt die Gleichung der Form $next(v) = w$ für den nächsten Zustand. Die Transformation des Dienstes `S2` geht analog (vgl. Zeilen 16–31).

Das Modul `MERGE` stellt die Kombination der beiden Dienste dar (vgl. Zeilen 33–52). Ähnlich wie in AutoFocus (vgl. z.B. Abbildung 7.8 auf Seite 127) muss die von beiden Diensten geteilte Ausgabevariable `outC` im `MERGE` dupliziert werden. In Zeile 38 wird das Transitionssystem um einen Fehlerzustand erweitert: das Prädikat $(state = 0)$ bezeichnet den Fehlerzustand, das Prädikat $(state = 1)$ einen regulären Zustand. Die Kombination `MERGE` ist ausführbar genau dann, wenn mindestens eines der Module `S1` und `S2` ausführbar ist (vgl. Zeile 42). Die Kombination ist konsistent genau dann, wenn die beiden Module an deren gemeinsamem Ausgabeport `outC` denselben Wert erzeugen (vgl. Zeile 43). Die Kombination hat 6 Transitionen (vgl. Zeilen 45–52): (1) der Fehlerzustand darf nie verlassen werden; (2) nur `S1` ist ausführbar; (3) nur `S2` ist ausführbar; (4) beide sind ausführbar und erzeugen denselben Wert am Port `outC`; (5) beide sind ausführbar und erzeugen unterschiedliche Werte am Port `outC`; (6) beide sind nicht ausführbar.

Im Modul `main` werden das Modul `MERGE` und eine Kopie des Moduls `S1` kombiniert (vgl. Zeilen 70–79). Da die Konsistenzprüfung durch eine Simulation zwischen einer Dienstkombination und einem ihrer Teildienste erfolgt, müssen zwei identische aber voneinander unabhängige Module für denselben Dienst erzeugt werden – eins für die Kombination

und eins für die Simulation (auf die analoge Konstruktion einer Kopie für das Modul `S2` wurde im Beispiel verzichtet). Die Konstruktion eines Gegenbeispiels basiert im Model Ckecker auf einer sich unendlich oft wiederholenden Schleife. Ist das Modul `main` partiell, d.h. es gibt einen Zustand und eine Eingabe, für die keine ausführbare Transition existiert, kann kein Gegenbeispiel konstruiert werden. In diesem Fall gilt jede beliebige Eigenschaft als erfüllt. Um dies zu vermeiden, sorgen die Transitionen von `main` dafür, dass das Modul immer ausführbar ist (vgl. Zeilen 76–79).

Die CTL-Formel in Zeile 81 definiert eine Konsistenzbedingung für die Dienstkombination: immer wenn sich die Dienstkombination in keinem Fehlerzustand befindet ($m.state$), ist auch einer der folgenden Zustände kein Fehlerzustand (`EX` $m.state$).

Der Model Checker findet ein Gegenbeispiel für diese Eigenschaft, da die Module `S1` und `S2` auf dieselbe Eingabe ($inA = 2$) unterschiedliche Werte am Port `outC` erzeugen. Die Transition in Zeile 50 führt das Modul `MERGE` in den Fehlerzustand über. In diesem Zustand bildet sich eine unendliche Schleife, die für das Gegenbeispiel ausreichend ist.

```
1  MODULE S1(inA,inD)
2  VAR
3   outC : {0,1,2,3};
4   state : {0,1};
5  INIT
6   outC=0 & state=0
7  DEFINE
8   enabled := (state=0 & inD=1 & inA=1) | (state=0 & inA=2)
9              | (state=1 & inD=1 & inA=2);
10 TRANS
11    state=0 & inD=1 & inA=1 & next(state)=1 & next(outC)=1
12 | state=0 & inA=2 & next(state)=1 & next(outC)=2
13 | state=1 & inD=1 & inA=2 & next(state)=0 & next(outC)=3
14 | !enabled & next(state)=state
15
16 MODULE S2(inA)
17 VAR
18  outC : {0,1,2,3};
19  outE : {0,1,2,3};
20  state : {0,1};
21 INIT
22  outC=0 & state=0
23 DEFINE
24  enabled := (state=0 & inA=1) | (state=0 & inA=2)
25             | (state=1 & inA=1) | (state=1 & inA=2);
26 TRANS
27    state=0 & inA=1 & next(state) = 1 & next(outC)=1 & next(outE)=1
28 | state=0 & inA=2 & next(state) = 1 & next(outC)=3
29 | state=1 & inA=1 & next(state) = 0 & next(outC)=3 & next(outE)=1
30 | state=1 & inA=2 & next(state) = 0 & next(outC)=2
31 | !enabled & next(state)=state
32
33 MODULE MERGE(inA,inD)
34 VAR
35  s1 : S1(inA,inD);
```

```
36   s2 : S2(inA);
37   outC : {0,1,2,3};
38   state : boolean;
39  INIT
40   outC=0 & state=1
41  DEFINE
42   enabled := s1.enabled | s2.enabled;
43   valid := s1.outC=s2.outC;
44  TRANS
45     !state & next(state) = state
46  | state & s1.enabled & !s2.enabled & next(outC)=next(s1.outC) & next(state)
47  | state & s2.enabled & !s1.enabled & next(outC)=next(s2.outC) & next(state)
48  | state & s1.enabled & s2.enabled & next(s1.outC)=next(s2.outC)
49          & next(outC)=next(s1.outC) & next(state)
50  | state & s1.enabled & s2.enabled & next(s1.outC)!=next(s2.outC)
51          & !next(state)
52  | state & !s1.enabled & !s2.enabled & !next(state)
53
54  MODULE S1COPY(inA,inD)
55  VAR
56   outC : {0,1,2,3};
57   state : {0,1};
58   step : boolean;
59  INIT
60   outC=0 & state=0 & step=0
61  DEFINE
62   enabled := (state=0 & inD=1 & inA=1) | (state=0 & inA=2)
63            | (state=1 & inD=1 & inA=2);
64  TRANS
65     state=0 & inD=1 & inA=1 & next(state)=1 & next(outC)=1
66  | state=0 & inA=2 & next(state)=1 & next(outC)=2
67  | state=1 & inD=1 & inA=2 & next(state)=0 & next(outC)=3
68  | !enabled & next(state)=state
69
70  MODULE main
71  VAR
72   inA : {0,1,2,3};
73   inD : {0,1,2,3};
74   s1 : S1COPY(inA,inD);
75   m : MERGE(inA,inD);
76  TRANS
77     s1.enabled & m.enabled & next(m.valid) & next(s1.outC) = next(m.outC)
78  | !s1.enabled & m.enabled & next(m.valid)
79  | s1.enabled & !(m.enabled & next(m.valid))
80  CTLSPEC
81  AG (m.state -> EX m.state)
```

ANHANG C

Fallstudien

Die Anwendbarkeit der eingeführten dienstbasierten Modellierungstechnik wird durch zwei Fallstudien nachgewiesen.

In der Fallstudie aus Abschnitt C.1 wird die Funktionalität der ACC-Steuerung modelliert. Dabei wird ein wesentlicher Vorteil der dienstbasierten Modellierung deutlich: die Möglichkeit, die funktions- und modusbasierte Vorgehensweisen zu kombinieren (vgl. Abschnitt 3.4.3 auf Seite 47). Die Funktionalität wird zuerst in eine Hierarchie von Systemmodi unterteilt. Anschließend werden für jeden Modus Nutzerfunktionen modelliert, die in diesem Modus den Nutzern zur Verfügung stehen. Durch diese Art der Strukturierung können Nutzerfunktionen eines komplexen multifunktionalen Systems durch sehr einfache Automaten modelliert werden.

Die Fallstudie aus Abschnitt C.2 dient zur Überprüfung der Mächtigkeit der dienstbasierten Modellierungstechnik im industriellen Umfeld. Im Kooperationsprojekt FLASCO mit der Siemens AG wurde eine bereits davor existierende Anlage zur Abfüllung von Flaschen formal spezifiziert und auf Inkonsistenzen in Anforderungen überprüft.

Die Fallstudien wurden vollständig in der dienstbasierten Erweiterung von AutoFocus aus Kapitel 7 modelliert und simuliert. Der Schwerpunkt der folgenden Abschnitte liegt in erster Linie auf der Strukturierung von Nutzerfunktionen und weniger auf der Modellierung einzelner Dienste. Deshalb werden nur einzelne Automaten von Diensten aus den Fallstudien präsentiert.

Für die Darstellung von Automaten wurde in diesem Anhang eine tabellarische Notation gewählt. In einer Zeile einer Tabelle wird eine Transition zusammen mit ihren Ausgangs- und Zielzustand definiert (vgl. z.B. Tabelle C.10 auf Seite 169). Besteht ein Automat aus nur einem Kontrollzustand, werden die Spalten mit Zuständen weggelassen.

C.1. Adaptive Cruise Control

In diesem Abschnitt wird die Diensthierarchie der ACC-Steuerung vollständig modelliert. Der informellen Beschreibung in Abschnitt C.1.1 folgt die Diensthierarchie der ACC-Funktionalität in Abschnitt C.1.2.

C.1.1. Informelle Beschreibung

Die Beschreibung der Basisfunktionalität der ACC-Steuerung ist in Kapitel 2 zu finden. Im Folgenden werden zusätzliche, bis jetzt nicht erwähnte Funktionen des ACC erläutert.

- *Überholhilfe.* Bei Geschwindigkeiten oberhalb von 80 km/h wird der Fahrer bei Überholvorgängen durch eine Überholhilfe unterstützt. Basierend auf der Blinkerbetätigung des Fahrers wird eine zusätzliche Beschleunigung eingesteuert.
- *Navigationskopplung.* Durch die Navigationskopplung steht der ACC-Steuerung die Information des aktuellen Straßentyps (Stadt, Landstraße, Autobahn) zur Verfügung, wodurch die Regelung an den jeweiligen Straßentyp angepasst werden kann.
- *Expertenfunktion.* Sollte der Fahrer den Wunsch haben, auch oberhalb der Geschwindigkeit von 180 km/h mit Unterstützung der Geschwindigkeitsregelung zu fahren, ist jederzeit eine Umschaltung in den DCC-Modus möglich. Dieser Modus erschließt zusätzlich den Bereich bis 250 km/h, allerdings steht dem Fahrer hier keine Abstandsregelung zur Verfügung.

C.1.2. Diensthierarchie

Bei der dienstbasierten Strukturierung der Funktionalität wird zuerst eine Hierarchie von Systemmodi aufgebaut, in denen sich die ACC-Steuerung befinden kann. Anschließend werden für jeden Modus Nutzerfunktionen modelliert, die das System in diesem Modus zur Verfügung stellt. Im Wesentlichen besteht die ACC-Steuerung aus zwei Nutzerfunktionen, dem Tempomat und der Abstandsregelung, die abhängig vom aktuellen Modus unterschiedliche Steuerbefehle an den Motor und die Bremse senden.

ACC Die Diensthierarchie der ACC-Steuerung ist in Abbildung C.1 zu finden, die syntaktische Schnittstelle des Systems in Tabelle C.1. Das ACC kann sich im Modus `Ausgeschaltet` oder `Eingeschaltet` befinden. Anhand von Nachrichten an den Eingabeports `cDist` und `cSpeed` entscheidet der Priorisierungsdienst `EinVsAus`, welcher der beiden Dienste aktiviert wird. Im Modus `Ausgeschaltet` erzeugt das System in jedem Zeitintervall den Wert ε an allen seinen Ausgabeports (vgl. Tabelle C.2). Im Modus `Eingeschaltet` stellt das ACC drei Nutzerfunktionen bereit: die `Warnung` zu einer akustischen und optischen Warnung zur Übernahme der Fahrzeugführung durch den Fahrer, `NaviEinAus` zur Anbindung eines Navigationssystems sowie `ExpEinAus` zur Ein- bzw.

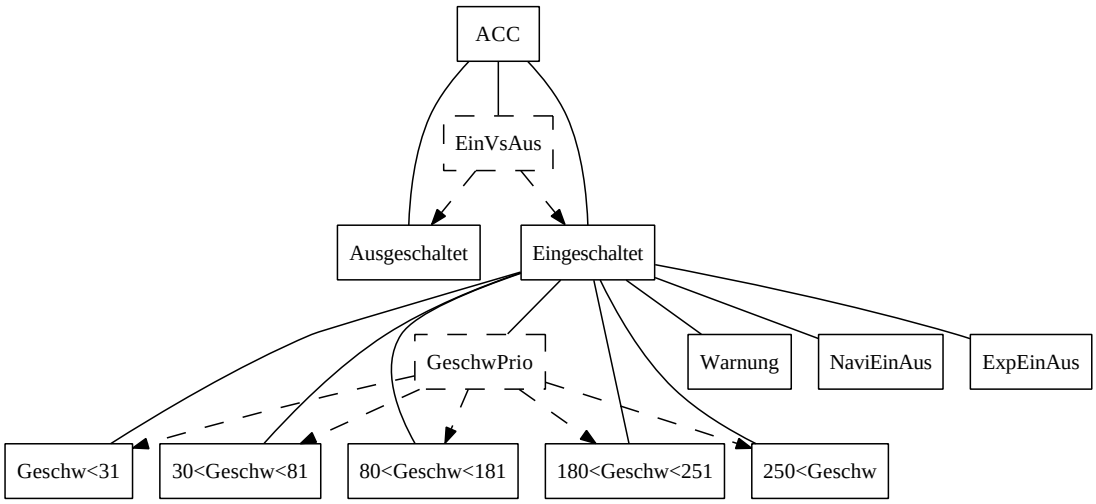


Abbildung C.1.: Dienst `ACC`

I-Port	O-Port
cSpeed	mInstr
rSpeed	bInstr
cDist	speed
rDist	distance
turn	object
rResume	warning
naviInput	
blinker	

Tabelle C.1.: Dienst `ACC`: syntaktische Schnittstelle

ID	Transition
01	/mInstr!ε∧bInstr!ε∧speed!ε∧distance!ε∧object!ε∧warning!ε

Tabelle C.2.: Dienst `ACC/Ausgeschaltet`: Automat

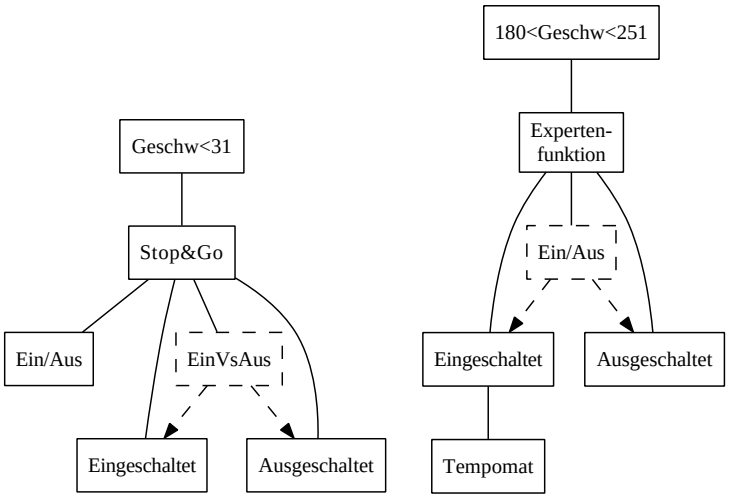
Ausschaltung der Expertenfunktion. Diese Funktionen sind unabhängig von den weiteren Modi des Systems. Der Modus `Eingeschaltet` wird in 5 weitere Modi unterteilt, in denen sich die ACC-Steuerung unterschiedlich verhält:

- `Geschw<31`: Unterhalb von 30 km/h steuert die Nutzerfunktion `Stop&Go` die Geschwindigkeit des Fahrzeugs.
- `30<Geschw<81`: In diesem Modus stehen die Nutzerfunktionen Tempomat und Abstandsregelung dem Fahrer zur Verfügung. In diesem unteren Geschwidigkeitsbereich wird das Fahrzeug dynamischer gebremst und beschleunigt als bei höheren Geschwindigkeiten (d.h. es wird eine höhere Beschleunigung eingesteuert).
- `80<Geschw<181`: Zusätzlich zu den Basisfunktionen wird der Fahrer bei Geschwin-

digkeiten oberhalb von 80 km/h bei Überholvorgängen unterstützt. Basierend auf der Blinkerbetätigung wird eine zusätzliche Beschleunigung eingesteuert.

- `180<Geschw<251`: Oberhalb der Geschwindigkeit von 180 km/h wird die Geschwindigkeitsregelung nur dann aktiviert, wenn die Expertenfunktion eingeschaltet ist. Die Abstandsregelung steht dem Fahrer nicht zur Verfügung.
- `250<Geschw`: Oberhalb von 250 km/h wird das ACC deaktiviert.

Geschw<31 Im Modus `Geschw<31` steht nur die Nutzerfunktion `Stop&Go` zur Verfügung, die ein- und ausgeschaltet werden kann (vgl. Abbildung C.2(a)). Eine ausführliche Beschreibung dieser Funktion findet sich in Kapitel 2. Das Verhalten der Funktion im Modus `Eingeschaltet` ist durch den Dienst aus Abbildung 3.12 auf Seite 37 spezifiziert. Der Dienst `Ausgeschaltet` ist analog zum Dienst aus Tabelle C.2 definiert.



(a) Dienst `Geschw<31` (b) Dienst `180<Geschw<251`

Abbildung C.2.: Teildienste von `ACC`

180<Geschw<251 In diesem Modus steht nur die `Expertenfunktion` zur Verfügung (vgl. Abbildung C.2(b)). Während im Modus `Ausgeschaltet` das ACC deaktiviert ist, steht im Modus `Eingeschaltet` die Nutzerfunktion `Tempomat` zur Verfügung.

80<Geschw<181 Der Priorisierungsdienst `U/N` entscheidet anhand der Nachricht am Eingabeport `blinker`, ob sich das System im Modus `Überholung` oder `Normalverkehr` befindet. Der Letztere wird in weitere Systemmodi unterteilt (vgl. Abbildung C.3). Ganz unten in der Diensthierarchie befinden sich die Dienste `Tempomat` und `Abstand`. Je nach Modus erzeugen diese Dienste unterschiedliche Steuerwerte an den Ausgabeports

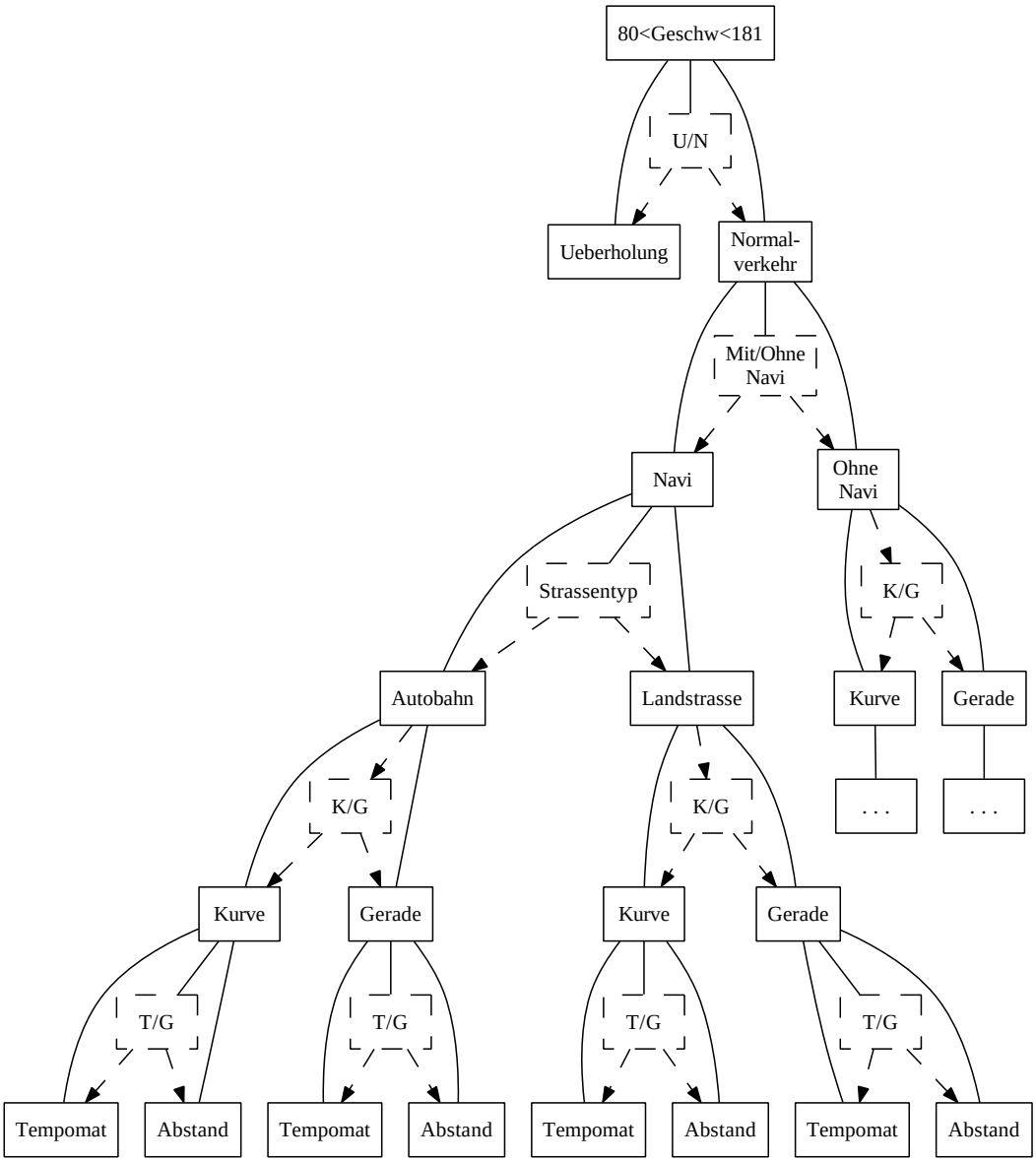


Abbildung C.3.: Dienst `80<Geschw<181`

`mInstr` und `bInstr`. Beispielhaft sind in Tabellen C.3 und C.4 die beiden Dienste im Modus `80<Geschw<181/OhneNavi/Gerade` spezifiziert. Ihre Schnittstellen sind in Abbildungen 3.8(a) auf Seite 33 und 3.11(a) auf Seite 36 zu finden. Für die Berechnung der Steuerbefehle benutzt der Dienst `Tempomat` die Funktionen aus Gleichungen (C.1) und (C.2), die in AutoFocus im *Data Dictionary* definiert wurden. Der Rückgabewert der Funktionen gibt die erforderliche Beschleunigung des Fahrzeugs an. Dabei kann situationsabhängig sowohl mit der Bremse als auch mit dem Motor gebremst werden. Die

Funktionen $l_1(cSpeed, cDist, rDist)$ und $m_1(cSpeed, cDist, rDist)$ berechnen die Steuerwerte im Dienst `Abstand`. Die Funktionen für die restlichen Dienste sind analog definiert und unterscheiden sich nur in der Höhe des erzeugten Beschleunigungswertes.

ID	Transition
01	cSpeed?X∧rSpeed?Y/mInstr!$f_1(X,Y)$∧bInstr!$g_1(X,Y)$

Tabelle C.3.: Dienst `80<Geschw<181/OhneNavi/Gerade/Tempomat`: Automat

ID	Transition
01	cSpeed?X∧rDist?Y∧cDist?Z/mInstr!$l_1(X,Y,Z)$∧bInstr!$m_1(X,Y,Z)$

Tabelle C.4.: Dienst `80<Geschw<181/OhneNavi/Gerade/Abstand`: Automat

$$f_1(cSpeed, rSpeed) = \begin{cases} 4 & \text{falls } (rSpeed - cSpeed) > 70 \\ 2 & \text{falls } 30 < (rSpeed - cSpeed) \leq 70 \\ 1 & \text{falls } 0 < (rSpeed - cSpeed) \leq 30 \\ 0 & \text{falls } (rSpeed - cSpeed) = 0 \vee (cSpeed - rSpeed) > 15 \\ -1 & \text{falls } 0 < (cSpeed - rSpeed) \leq 15 \end{cases} \tag{C.1}$$

$$g_1(cSpeed, rSpeed) = \begin{cases} 1 & \text{falls } 15 < (cSpeed - rSpeed) \leq 30 \\ 2 & \text{falls } 30 < (cSpeed - rSpeed) \leq 50 \\ 4 & \text{falls } 50 < (cSpeed - rSpeed) \\ 0 & \text{falls } (cSpeed - rSpeed) \leq 15 \end{cases} \tag{C.2}$$

30<Geschw<81 Die Funktionalität im Modus `30<Geschw<81` unterscheidet sich von der im Modus `80<Geschw<181` nur geringfügig. In diesem Modus gibt es keine Überholhilfe und statt `Autobahn` den Modus `Stadt`.

250<Geschw In diesem Modus ist das ACC deaktiviert – der Dienst `250<Geschw` ist analog zum Dienst aus Tabelle C.2 definiert.

C.2. SmartAutomation-Modellanlage

Die vorliegende Fallstudie wurde im Rahmen des Kooperationsprojekts FLASCO zwischen der Fachabteilung „Advanced Technologies and Standards“ des Siemens Bereiches „Automation and Drives“[1] und dem Lehrstuhl für Software & Systems Engineering der TU München ausgearbeitet. Das Projekt diente zur Überprüfung der Mächtigkeit der dienstbasierten Modellierungstechnik im industriellen Umfeld sowie zur Konsistenzanalyse einer gegebenen informellen Spezifikation. In der Machbarkeitsstudie wurde eine existierende Modellanlage zur Abfüllung von Flaschen formal spezifiziert. Den Ausgangspunkt für die Modellierung bildete eine textuelle, ca. 200 Seiten umfassende Spezifikation [Did06], die vom Hersteller der Anlage zur Verfügung gestellt wurde.

Im folgenden Abschnitt wird die Funktionalität der Abfüllanlage informell beschrieben. In Abschnitt C.2.2 folgt eine formale Diensthierarchie der Funktionalität. In Abschnitt C.2.3 werden Beispiele identifizierter Inkonsistenzen in der Spezifikation erläutert.

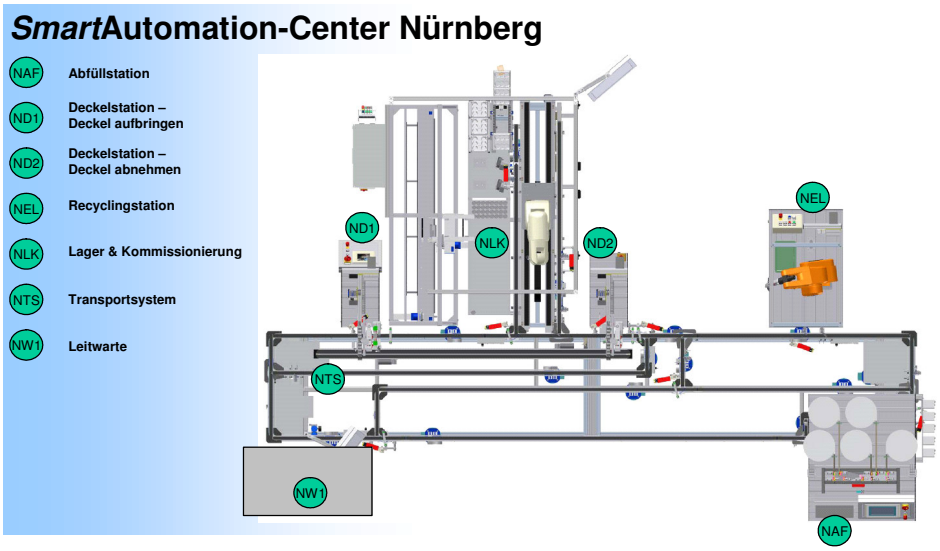


Abbildung C.4.: Anlage zur Abfüllung von Flaschen [Did06]

C.2.1. Informelle Beschreibung

Die Abfüllanlage besteht aus 7 Teilsystemen (vgl. Abbildung C.4).

- *Abfüllstation* zur Befüllung von Flaschen mit unterschiedlichen Feststoffteilen;
- *Deckelstation 1* zum Verdeckeln von Flaschen;
- *Deckelstation 2* zum Entdeckeln von Flaschen;

[1] http://www.automation.siemens.com

- *Recyclingstation* zum Entleeren von Flaschen;
- *Lager* zum Aufbewahren leerer Flaschen und zur Kommissionierung befüllter Flaschen;
- *Transportsystem* zur Beförderung von Flaschen zwischen Stationen;
- *FMS* (Fertigungs Management System) ist das zentrale System zur Steuerung der Stationen, Verwaltung von Kundenaufträgen und Interaktion mit dem Kunden.

Nachdem alle Stationen der Anlage vom FMS gesteuert und koordiniert werden, liegt der Fokus der vorliegenden Fallstudie auf der Modellierung des FMS – die restlichen Stationen werden als Umgebung des FMS betrachtet.

Beschreibung der Prozesse

In der Abfüllanlage werden unterschiedliche Feststoffteile in einer Flasche zusammengebracht. Flaschen werden gemäß einem Kundenauftrag befüllt, verdeckelt, kommissioniert und in Kisten eingelagert. Beim Recycling werden kommissionierte und in Kisten eingelagerte Flaschen aus den Kisten herausgenommen, entdeckelt und entleert.

Jede Flasche im System hat einen Flaschen-Datensatz mit einer eindeutigen ID. In diesem Datensatz sind die Verwendung der Flasche, ihr Status und ihre nächste Zielstation gespeichert. Die Mischung, mit der eine Flasche zu befüllen ist, wird in einem eindeutigen Fertigungsauftrag festgelegt. Die Mischung einer Flasche leitet sich aus einem Kundenauftrag ab, der mehrere Bestellpositionen umfasst. Jede Bestellposition besteht aus der Angabe einer Mischung und der Anzahl von Flaschen, die mit dieser Mischung zu befüllen sind. Das FMS erzeugt aus einem Kundenauftrag mit m Bestellpositionen für jeweils n_m Flaschen $\sum_{0<i\leq m} n_i$ Fertigungsaufträge. Aus n Fertigungsaufträgen erzeugt das FMS nach Abschluss der Fertigung ($n \mod 6$) Kommissionieraufträge, welche die Zusammenstellung von bis zu 6 Flaschen in einer Kiste festlegen.

Ablauf der Befüllung einer Flasche

In diesem Abschnitt ist der Ablauf der Befüllung beispielhaft für eine Flasche beschrieben. Die Zusammenarbeit der Stationen wird vom FMS gesteuert, indem das FMS den Datensatz der Flasche von einer Station empfängt und an die folgende Station mit einem neuen Auftrag sendet – die Kommunikation erfolgt ausschließlich sternpunktförmig.

- Mit einem neu erzeugten Fertigungsauftrag fordert das FMS eine Flasche aus dem Lager an. Der Roboter des Lagers setzt eine leere Flasche auf das Fließband des Transportsystems und übergibt den Datensatz der Flasche (Flaschen-Info genannt) an das FMS. Das FMS legt die Abfüllstation als nächste Zielstation für die Flasche fest und sendet die Flaschen-Info an das Transportsystem zurück. Das Transportsystem fördert daraufhin die Flasche zur Abfüllstation.
- Beim Eintreffen der Flasche in der Abfüllstation wird die Flaschen-Info vom Trans-

portsystem ans FMS übergeben. Das FMS sendet die Flaschen-Info sowie einen Fertigungsauftrag zur Befüllung der Flasche an die Abfüllstation.

- In der Abfüllstation wird die Flasche durch einen Barcode-Scanner identifiziert und dann entsprechend dem Fertigungsauftrag befüllt.
- Nach der Befüllung der Flasche sendet die Abfüllstation die Flaschen-Info zurück an das FMS. Daraufhin trägt das FMS die Deckelstation als Ziel in die Flaschen-Info ein und sendet die Info ans Transportsystem.
- Das Transportsystem fördert die Flasche zur Deckelstation und überträgt die Flaschen-Info zurück ans FMS. Das FMS leitet sie an die Deckelstation weiter und empfängt sie nach der Verdeckelung der Flasche zurück.
- Die Flasche wird nun ins Lager gefördert. Alle befüllten Flaschen werden zunächst im Lager zwischengelagert. Nach der Befüllung aller zu einem Kundenauftrag gehörenden Flaschen erteilt das FMS die benötigte Anzahl von Aufträgen zur Kommissionierung der Flaschen in Kisten.

C.2.2. Diensthierarchie

Im Folgenden wird die Funktionalität der Abfüllanlage durch eine Diensthierarchie formal spezifiziert. Die vollständige Ausarbeitung dieser Fallstudie findet sich in [Har09].

Station	Abfüllung	Transport	Deckel1	Deckel2	Recycling	Lager	HMI
I-Port	Betriebszust FlInfo Notaus Anforderung	Betriebszust FlInfo Notaus Förderweg Anforderung	Betriebszust FlInfo Notaus	Betriebszust FlInfo Notaus	Betriebszust FlInfo Notaus	Betriebszust FlInfo Notaus	Eingabe KAuftrag
O-Port	Kommando FlInfo FAuftrag	Kommando FlInfo	Kommando FlInfo	Kommando FlInfo	Kommando FlInfo	Kommando FlInfo Anforderung FAuftrag KomAuftrag	Status

Tabelle C.5.: Gesamtsystem `FMS`: syntaktische Schnittstelle

Gesamtschnittstelle Die gesamte syntaktische Schnittstelle des FMS ist in Tabelle C.5 definiert. Die Ports sind nach Stationen unterteilt, die durch diese Ports mit dem FMS verbunden sind. In AutoFocus wurden die Namen der Ports um zwei weitere Buchstaben erweitert, um die Zugehörigkeit eines Ports zu einer Station zu kennzeichnen: beispielsweise erfolgt die Kommunikation mit der Abfüllstation durch den Port `BetriebszustAB` und mit dem Transportsystem durch den Port `BetriebszustTR`. Die letzte Spalte der Tabelle definiert die Schnittstelle, über die der Benutzer dem System Anweisungen erteilt und Systemmeldungen angezeigt werden.

Die Datentypen der Ports sind komplexe Records, die im *Data Dictionary* von AutoFocus definiert wurden. Die konkrete Spezifikation der Typen findet sich in [Did06, S. 156-170]. Um das Verständnis zu erleichtern, werden im Folgenden die Typen stark vereinfacht dargestellt.

An- und Abmeldung Die Funktionalität des FMS wird zuerst in Systemmodi unterteilt (vgl. Abbildung C.5). Das FMS kann sich im Modus `Angemeldet` oder `Abgemeldet` befinden. Sind nicht alle sechs Stationen angemeldet, befindet sich das FMS im Modus `Abgemeldet`. In diesem Fall dürfen Flaschen-Infos, Aufträge und Anforderungen an die Stationen nicht gesendet werden. Das Verhalten des Systems im Modus `Abgemeldet` wird durch den Dienst aus Tabellen C.6 und C.7 definiert. Wie bereits erwähnt, werden die Namen der Ports in Automaten um die IDs der Stationen erweitert. Unabhängig von Eingaben, erzeugt der Dienst `Abgemeldet` in jedem Schritt den Wert `Abgemeldet` am Port `Status` und keine Nachricht (d.h. den Wert ε) an seinen restlichen Ausgabeports.

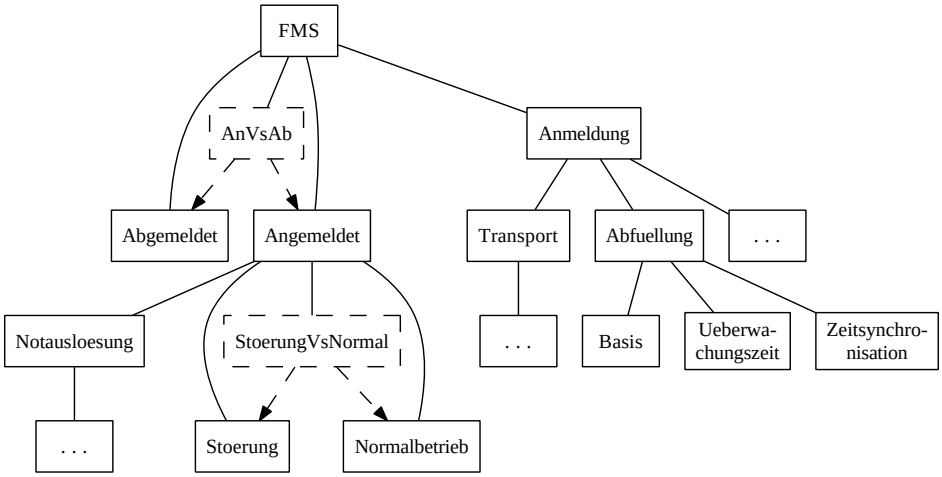


Abbildung C.5.: Diensthierarchie `FMS`

Station	Abfüllung	Transport	Deckel1	Deckel2	Recycling	Lager	HMI
I-Port							
O-Port	FlInfo FAuftrag	FlInfo	FlInfo	FlInfo	FlInfo	FlInfo Anforderung FAuftrag KAuftrag	Status

Tabelle C.6.: Dienst `Abgemeldet`: Schnittstelle

Der Moduswechsel zwischen `Abgemeldet` und `Angemeldet` wird durch den Priorisierungsdienst `AnVsAb` aus Tabelle C.8 definiert. Melden sich alle sechs Stationen mit der Nachricht `Bereit`, gibt der Priorisierungsdienst dem Dienst `Angemeldet` Vorrang, ansonsten

ID	Transition
01	/FlInfoAB!ε∧FAuftragAB!ε∧FlInfoTR!ε∧FlInfoD1!ε∧FlInfoD2!ε∧FlInfoRE!ε∧ FlInfoLA!ε∧AnforderungLA!ε∧FAuftragLA!ε∧KAuftragLA!ε∧Status!Abgemeldet

Tabelle C.7.: Dienst `Abgemeldet`: Automat

dem Dienst `Abgemeldet`.

ID	Transition
01	BetriebszustandAB?Bereit∧BetriebszustandTR?Bereit∧BetriebszustandD1?Bereit∧ BetriebszustandD2?Bereit∧BetriebszustandLA?Bereit[ANG]
02	BetriebszustandAB?¬Bereit∨BetriebszustandTR?¬Bereit∨BetriebszustandD1?¬Bereit∨ BetriebszustandD2?¬Bereit∨BetriebszustandLA?¬Bereit[ABM]

Tabelle C.8.: Priorisierungsdienst `AnVsAb`: Automat

Die Dienste `Angemeldet` und `Abgemeldet` definieren das Verhalten des Systems in den beiden Systemmodi. Der eigentliche Anmeldevorgang der Stationen wird durch den Dienst `Anmeldung` definiert, der wiederum in sechs Teildienste unterteilt ist. Im Folgenden wird nur die Anmeldung der Abfüllstation näher betrachtet. Der Dienst `Abfüllung` ist in drei Teildienste unterteilt: `Basis`, `Überwachungszeit` und `Zeitsynchronisation`. Der Dienst `Zeitsynchronisation` modelliert die Zeitsynchronisation nach der Anmeldung der Station und wird hier nicht näher betrachtet. Der Dienst `Basis`, der die An- und Abmeldung der Station modelliert, ist in Tabellen C.9 und C.10 angegeben. Die Anmel-

I-Port	BetriebszustAB	Eingabe
O-Port	KommandoAB	Status

Tabelle C.9.: Dienst `FMS/Anmeldung/Abfüllung`: Schnittstelle

ID	Quelle	Transition	Ziel
01	ABG	BetriebszustAB?ε∧Eingabe?ANM/KommandoAB!ANM∧Status!ANM	ANM
02	ANM	BetriebszustAB?ε∧Eingabe?ε/KommandoAB!ε∧Status!ANM	ANM
03	ANM	BetriebszustAB?ε∧Eingabe?ε/KommandoAB!ε∧Status!KeineVerb	ABG
04	ANM	BetriebszustAB?Bereit∧Eingabe?ε/KommandoAB!ε∧Status!VerbOK	ANG
05	ANG	BetriebszustAB?¬ε∧Eingabe?ε/KommandoAB!ε∧Status!VerbOK	ANG
06	ANG	BetriebszustAB?¬ε∧Eingabe?ABM/KommandoAB!ABM∧Status!ABM	ABM
07	ABM	BetriebszustAB?ε∧Eingabe?ε/KommandoAB!ε∧Status!KeineVerb	ABG
08	ABG	BetriebszustAB?ε∧Eingabe?ε/KommandoAB!ε∧Status!keineVerb	ABG
09	ABM	BetriebszustAB?¬ε∧Eingabe?ABM/KommandoAB!ABM∧Status!ABM	ABM

Tabelle C.10.: Dienst `FMS/Anmeldung/Abfüllung/Basis`: Automat

dung wird durch die Benutzereingabe `ANM` gestartet (`ABM` für die Abmeldung). Daraufhin sendet das FMS das Kommando `ANM` an die Station und wartet auf eine Rückmeldung. Der aktuelle Status des Vorgangs wird durch den Ausgabeport `Status` angezeigt. Meldet

sich die Station mit der Nachricht `Bereit`, ist sie erfolgreich angemeldet. Wie lange das FMS auf die Rückmeldung der Station warten muss, ist an dieser Stelle nicht spezifiziert. Durch die Transition 02 bleibt der Dienst im selben Kontrollzustand `ANM` stehen und wartet auf die Rückmeldung. Die Transition 03 bricht die Anmeldung ab und führt in den Zustand `ABG` über. Dadurch ist der Dienst nichtdeterministisch definiert.

Man beachte, dass der Dienst partiell definiert ist. Beispielsweise ist die Reaktion auf die Eingabe `ANM` im Zustand `ANG` (d.h. die Station ist bereits angemeldet) nicht definiert. Da der Dienst textuelle Anforderungen aus [Did06] widerspiegelt, wird er absichtlich nicht vervollständigt.

ID	Quelle	Transition	Ziel
01	Idle	/KommandoAB!ANM{t:=0}	Warten
02	Warten	{t<11}BetriebszustAB?ε/KommandoAB!ε{t:=t+1}	Warten
03	Warten	{t>10}BetriebszustAB?ε/Status!KeineVerb	Idle
04	Warten	/KommandoAB!$\neg\varepsilon$	Idle
05	Warten	BetriebszustAB?Bereit/	Idle
06	Idle	BetriebszustAB?ε/KommandoAB!ε	Idle

Tabelle C.11.: Dienst `Überwachungszeit`: Automat

Der Dienst `Überwachungszeit` aus Tabelle C.11 definiert die maximal erlaubte Reaktionszeit einer Station bei deren Anmeldung, bei der Abfüllstation beträgt sie 10 Zeitintervalle. Nach dem (nichtdeterministischen) Versenden der Nachricht `ANM` durch den Port `KommandoAB` wird die lokale Variable t auf 0 gesetzt und der Dienst geht in den Zustand `Warten` über. In diesem Zustand kann der Dienst 10 Zeitintervalle auf eine Nachricht durch den Port `KommandoAB` warten. Nach 10 Zeitintervallen geht der Dienst in den Zustand `Idle` über und erzeugt die Nachricht `KeineVerb` am Port `Status`. Im Zustand `Warten` kann der Dienst (nichtdeterministisch) eine beliebige Nachricht am Port `KommandoAB` erzeugen und in den Zustand `Idle` übergehen. Empfängt der Dienst durch den Port `BetriebszustAB` die Nachricht `Bereit`, geht er in den Zustand `Idle` über und beschränkt seine Ausgaben nicht.

Wie bereits erwähnt, sind die Dienste `Basis` und `Überwachungszeit` nichtdeterministisch. Der Dienst `Basis` modelliert den An- und Abmeldungsvorgang einer Station, ohne die Reaktionszeit der Station zu kontrollieren. Der Dienst `Überwachungszeit` ist nur für das Zählen von Zeitintervallen zwischen dem Versenden der Anmeldeanforderung und der entsprechenden Bestätigung der Station zuständig. Die Kombination der beiden eliminiert den Nichtdeterminismus und ergibt das gewünschte Verhalten.

Angemeldet Jede der Stationen verfügt über einen Notaus-Knopf, durch den die ganze Anlage angehalten werden kann. Voraussetzung für das Wiederstarten der Anlage ist die Behebung aller Störungsursachen und die Quittierung der Störung an der betroffenen Station. Aus diesem Grund wird der Systemmodus `Angemeldet` in Modi `Normalbetrieb` und `Störung` unterteilt, der Moduswechsel wird durch den Priorisierungsdienst `StörungVsNormal` gesteuert. Die Reaktion auf die Betätigung eines Notaus-

Knopfes wird im Dienst `Notauslösung` modelliert.

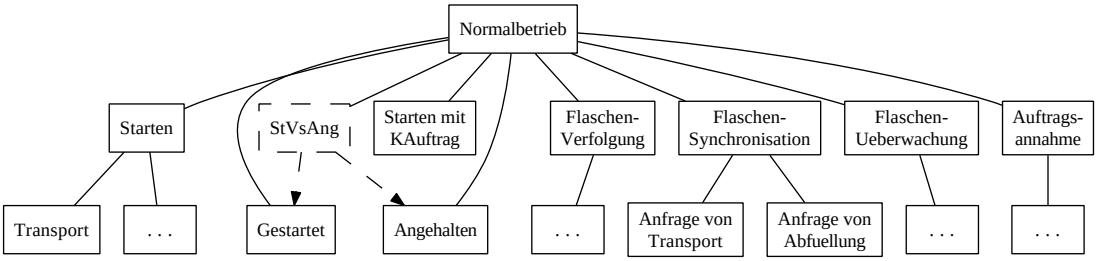


Abbildung C.6.: Diensthierarchie `Normalbetrieb`

Normalbetrieb Der Modus `Normalbetrieb` wird in Modi `Gestartet` und `Angehalten` unterteilt (vgl. Abbildung C.6). Einzelne Funktionen, wie Starten von Stationen, Flaschen-Synchronisation oder Flaschen-Überwachung, die im Modus `Normalbetrieb` zur Verfügung stehen, werden an dieser Stelle nicht näher betrachtet.

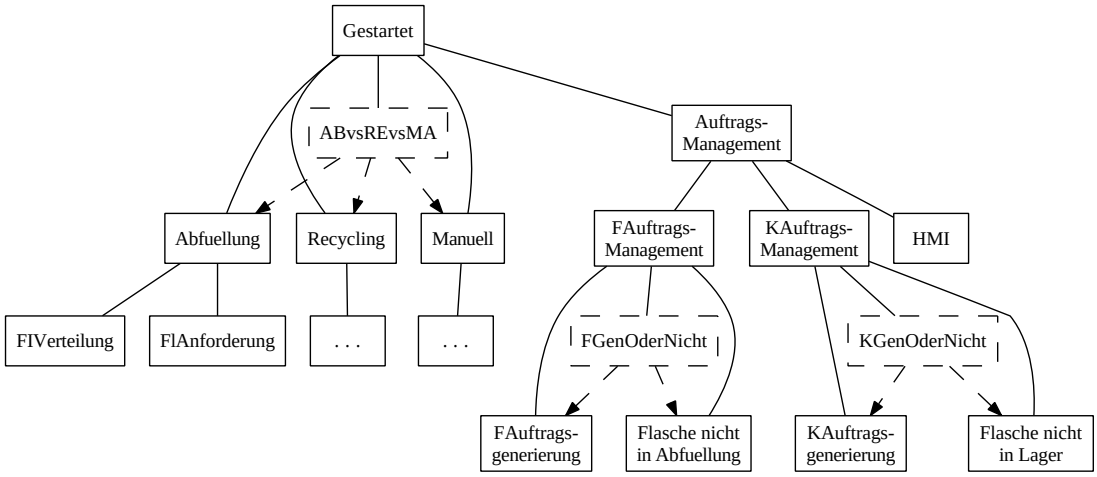


Abbildung C.7.: Diensthierarchie `Gestartet`

Gestartet Sind alle Stationen der Anlage gestartet, können Flaschen gemäß Kundenaufträgen befüllt bzw. entleert werden. Jede Flasche ist einem der drei Prozesse zugeordnet: Abfüllung, Recycling oder dem manuellen Prozess. Abhängig vom zugeordneten Prozess steuert das FMS eine Flasche entlang einer definierten Route. Demzufolge ist der Modus `Gestartet` in Modi `Abfüllung`, `Recycling` und `Manuell` unterteilt (vgl. Abbildung C.7). Außerdem stellt das FMS in diesem Modus mehrere Funktionen zur Verwaltung von Kundenaufträgen bereit. Auf den Dienst `Auftragsmanagement` zusammen mit allen seinen Teildiensten wird an dieser Stelle nicht näher eingegangen.

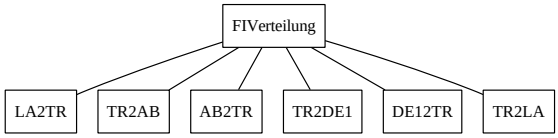


Abbildung C.8.: Diensthierarchie `FIVerteilung`

Flaschen-Info-Verteilung Im Systemmodus `Abfüllung` stellt das FMS mehrere Funktionen zur Steuerung von Flaschen und Stationen bereit. Beispielhaft wird die Funktion zur Verteilung von Flaschen-Infos genauer betrachtet (vgl. Abbildung C.8). Die syntaktische Schnittstelle des Dienstes `FIVerteilung` ist in Tabelle C.12 definiert. Der

Station	Abfüllung	Transport	Deckel1	Deckel2	Recycling	Lager
I-Port	FlInfo	FlInfo	FlInfo	FlInfo	FlInfo	FlInfo
O-Port	FlInfo	FlInfo	FlInfo	FlInfo	FlInfo	FlInfo

Tabelle C.12.: Dienst `FIVerteilung`: Syntaktische Schnittstelle

Dienst ist in sechs Teildienste unterteilt. Jeder dieser Dienste modelliert den Übergang einer Flasche von einer Station zur nächsten. Ihre Kombination modelliert den folgenden Ablauf von Flaschen: Lager – Transportsystem – Abfüllstation – Transportsystem – Deckelstation – Transportsystem – Lager. Im Folgenden wird nur auf drei der Dienste näher eingegangen. Der Dienst `TS2AB` aus Tabelle C.13 ist für die Weiterleitung von

ID	Transition
01	FlInfoTS?ε/FlInfoAB!ε
02	{ziel(X)==AB}FlInfoTS?X/FlInfoAB!X
03	{ziel(X)≠AB}FlInfoTS?X/FlInfoAB!ε

Tabelle C.13.: Dienst `TS2AB`: Automat

Flaschen-Infos vom Transportsystem zur Abfüllstation zuständig. Empfängt der Dienst durch den Port `FlInfoTS` eine Nachricht, deren Feld `Ziel` mit dem Wert `AB` belegt ist, leitet er die Nachricht durch den Ausgabeport `FlInfoAB` an die Abfüllstation weiter. Anderenfalls wird keine Nachricht an diesem Ausgabeport erzeugt. Der Dienst `AB2TS`

ID	Transition
01	FlInfoAB?ε/FlInfoTS!ε
02	FlInfoAB?X/FlInfoTS!erstelleFI(ID(X),DE)

Tabelle C.14.: Dienst `AB2TS`: Automat

aus Tabelle C.14 empfängt eine Flaschen-Info von der Abfüllstation, setzt deren Ziel auf `DE` (für die Deckelstation) und leitet sie ans Transportsystem weiter. Der Dienst `LA2TS` aus Tabelle C.15 empfängt eine Flaschen-Info vom Transportsystem, setzt deren Ziel auf `AB` (für die Abfüllstation) und leitet sie ebenfalls ans Transportsystem weiter.

ID	Transition
01	FlInfoLA?ε/FlInfoTS!ε
02	FlInfoLA?X/FlInfoTS!erstelleFI(ID(X),AB)

Tabelle C.15.: Dienst `LA2TS`: Automat

C.2.3. Beispiele von Inkonsistenzen

Bei der dienstbasierten Modellierung der Abfüllanlage wurden ca. 30 Unstimmigkeiten in der Spezifikation (manuell) entdeckt. Sie lassen sich in drei Klassen unterteilen:

- Unverständliche Anforderungen wegen technischer Details in der Spezifikation;
- Unterspezifikationen;
- Widersprüche zwischen Anforderungen.

Diese Klassen von Unstimmigkeiten werden an kleinen Beispielen kurz erläutert.

Technische Details In der Spezifikation wird die Kommunikation zwischen dem FMS und den Stationen in der Implementierungssprache SIMATIC PCS7[2] beschrieben (vgl. [Did06, Kapitel 2.14.5.1]). Diese Implementierungsdetails sind für einen Anforderungsanalytiker ohne Grundkenntnisse dieser Sprache unverständlich und haben im Laufe des Projekts viele Verständnisfragen verursacht.

Unterspezifikationen In der Spezifikation sind häufig nur Hauptanwendungsfälle (d.h. Anwendungsfälle, die zu erwarten sind) beschrieben. Anwendungsfälle, die vom typischen Verhalten abweichen, werden meistens vernachlässigt. Beispielsweise ist bei der Anmeldung nicht vorgesehen, dass der Benutzer versuchen kann, eine bereits angemeldete Station wieder anzumelden oder den Anmeldungsvorgang abzubrechen (vgl. Tabelle C.10).

Die Reaktionszeiten der Teilsysteme werden in der Spezifikation nicht genau definiert. Es ist z.B. nicht klar, wann das FMS eine Flaschen-Info an die nächste Station senden muss, im nächsten oder in einem beliebigen späteren Zeitintervall (vgl. z.B. Tabelle C.13).

Widersprüche zwischen Anforderungen Die Dienste aus Tabellen C.14 und C.15 können zum selben Zeitpunkt unterschiedliche Nachrichten an ihrem gemeinsamen Ausgabeport `FlInfoTS` erzeugen. Das ist immer der Fall, wenn sie gleichzeitig zwei unterschiedliche Flaschen-Infos durch ihre Eingabeports `FlInfoLA` und `FlInfoAB` empfangen. Das bedeutet, dass die Abfüllstation und das Lager jeweils eine Flasche aufs Transportsystem gleichzeitig ausliefern.

[2] http://www.automation.siemens.com/mcms/topics/en/simatic

Glossar

Architekturtreiber Architekturtreiber sind Anforderungen, die Auswirkungen auf die Gestalt der Architektur haben.

Benutzer Ein Benutzer ist ein Mensch oder ein anderes System, der/das mit dem zu entwickelnden System interagieren soll.

Dienst Ein Dienst ist die Formalisierung eines Interaktionsmusters. Ein Dienst hat eine syntaktische Schnittstelle und eine Verhaltensspezifikation. Die syntaktische Schnittstelle umfasst je eine Menge getypter Ein- und Ausgabeports, durch die der Dienst mit seiner Umgebung kommuniziert. Die Verhaltensspezifikation definiert in Form eines Transitionssystems eine partielle Abbildung von Ein- auf Ausgabewerte an den Ports der Schnittstelle.

Inkonsistenz Zwei Teile einer Spezifikation sind inkonsistent, falls sie eine gegebene Konsistenzbedingung verletzen.

Interaktionsmuster Ein Interaktionsmuster definiert eine Menge möglicher Interaktionen zwischen dem System und seiner Umgebung. Eine Interaktion ist ein zeitlich geordneter Austausch von Nachrichten zwischen dem System und seiner Umgebung.

Konflikt Ein Oberbegriff für *Inkonsistenz* und *Verletzung einer Randbedingung*.

Logische Architektur Die logische Architektur ist ein Netzwerk kommunizierender logischer Komponenten, die durch gerichtete, getypte Kanäle miteinander verbunden sind. Die logische Kommunikation zwischen Komponenten erfolgt ausschließlich über diese Kanäle.

Nutzerfunktion Eine Nutzerfunktion ist ein Dienst, den das System seiner Umgebung über eine Schnittstelle zur Verfügung stellt. Eine Nutzerfunktion charakterisiert die Nutzung des Systems zu einem bestimmten Zweck, d.h. seine

Reaktion auf bestimmte Eingaben und sein Verhalten in bestimmten Situationen.

Stakeholder Stakeholder repräsentieren eine Abstraktion, indem ein Stakeholder jeweils die Zusammenfassung aller Personen mit gleicher Interessenlage und gleicher Sicht auf das System repräsentiert. In der vorliegenden Arbeit sind Stakeholder die Informationslieferanten für Anforderungen an das zu entwickelnde System.

Verletzung einer Randbedingung Eine Diensthierarchie verletzt eine Randbedingung, falls diese Bedingung in der Hierarchie nicht gilt.

Literaturverzeichnis

[ABFL07] João Abreu, Laura Bocchi, José Luiz Fiadeiro, and Antónia Lopes. Specifying and composing interaction protocols for service-oriented system modelling. In *Proceedings of FORTE'07*. Springer-Verlag, 2007.

[AEY03] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Inference of message sequence charts. *IEEE Trans. Softw. Eng.*, 29(7):623–633, 2003.

[AGS00] Karine Altisen, Gregor Gößler, and Joseph Sifakis. A methodology for the construction of scheduled systems. In *Proceedings of FTRTFT'00*, volume 1926 of *LNCS*, pages 106–120. Springer-Verlag, 2000.

[Alu99] Rajeev Alur. Timed automata. In *Proceedings of CAV'99*, volume 1633 of *LNCS*, pages 8–22. Springer-Verlag, 1999.

[AOS08] *AOSD'08: Proceedings of the 7th international conference on Aspect-oriented software development*. ACM, 2008.

[ARS09] Mehmet Aksit, Arend Rensink, and Tom Staijen. A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In *Proceedings of AOSD'09*, pages 39–50. ACM, 2009.

[AWK04] Joao Araujo, Jon Whittle, and Dae-Kyoo Kim. Modeling and composing scenario-based requirements with aspects. In *Proceedings of RE'04*, pages 58–67. IEEE Computer Society, 2004.

[BBG+08] Jewgenij Botaschanjan, Manfred Broy, Alexander Gruler, Alexander Harhurin, Steffen Knapp, Leonid Kof, Wolfgang Paul, and Maria Spichkova. On the correctness of upper layers of automotive systems. *Formal Aspects of Computing*, 20(6):637–662, 2008.

[BBR+07] Andreas Bauer, Manfred Broy, Jan Romberg, Bernhard Schätz, Peter Braun, Ulrich Freund, Nuria Mata, Robert Sandner, Pierre Mai, and Dirk

Ziegenbein. Das AutoMoDe-Projekt. *Inform., Forsch. Entwickl.*, 22(1):45–57, 2007.

[BC04] E. Baniassad and S. Clarke. Theme: an approach for aspect-oriented analysis and design. In *Proceedings of ICSE'04*, 2004.

[BCK98] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison Wesley, 1998.

[BCP+07] Roderick Bloem, Roberto Cavada, Ingo Pill, Marco Roveri, and Andrei Tchaltsev. RAT: A tool for the formal analysis of requirements. In *Proceedings of CAV'07*, volume 4590 of *LNCS*. Springer-Verlag, 2007.

[BFG+08] Manfred Broy, Martin Feilkas, Johannes Grünbauer, Alexander Gruler, Alexander Harhurin, Judith Hartmann, Birgit Penzenstadler, Bernhard Schätz, and Doris Wild. Umfassendes Architekturmodell für das Engineering eingebetteter Software-intensiver Systeme. Technical Report TUM-I0816, TU München, 2008. http://www4.in.tum.de/~harhurin/publications/TUM-I0816.pdf.

[BFL08] Laura Bocchi, José Luiz Fiadeiro, and Antónia Lopes. Service-oriented modelling of automotive systems. In *Proceedings of COMPSAC*, pages 1059–1064. IEEE Computer Society, 2008.

[BFL+09] Laura Bocchi, José Luiz Fiadeiro, Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. From architectural to behavioural specification of services. *Electr. Notes Theor. Comput. Sci.*, 253(1):3–21, 2009.

[BGH+06] Jewgenij Botaschanjan, Alexander Gruler, Alexander Harhurin, Leonid Kof, Maria Spichkova, and David Trachtenherz. Towards Modularized Verification of Distributed Time-Triggered Systems. In *FM'06: Proceedings of Formal Methods*, pages 163–178. Springer-Verlag, 2006.

[BH09a] Jewgenij Botaschanjan and Alexander Harhurin. A Formal Framework for Integrating Functional and Architectural Views of Reactive Systems. Technical Report TUM-I0904, Technische Universität München, 2009. http://www4.in.tum.de/~harhurin/publications/TUM-I0904.pdf.

[BH09b] Jewgenij Botaschanjan and Alexander Harhurin. Integrating Functional and Architectural Views of Reactive Systems. In *Proceedings of CBSE'09*, volume 5582 of *LNCS*. Springer-Verlag, 2009.

[BH09c] Jewgenij Botaschanjan and Alexander Harhurin. Property-Driven Scenario Integration. In *Proceedings of SEFM'09*. IEEE Computer Society, 2009.

[BHK08] Jewgenij Botaschanjan, Alexander Harhurin, and Leonid Kof. Service-Based Specification of Reactive Systems. Technical Report TUM-I0815, Technische Universität München, 2008. http://www4.in.tum.de/~harhurin/publications/TUM-I0815.pdf.

[BHRS08] Manfred Broy, Judith Hartmann, Sabine Rittmann, and Bernd Spanfel-

ner. Positionspapier: Funktions-/Dienstorientierte Softwareentwicklung. Fakultät für Informatik, Technische Universität München, 2008.

[BKM07] Manfred Broy, Ingolf H. Krüger, and Michael Meisinger. A formal model of services. *ACM Trans. Softw. Eng. Methodol.*, 16(1), 2007.

[BKPS07] Manfred Broy, Ingolf H. Krüger, Alexander Pretschner, and Christian Salzmann. Engineering automotive software. *IEEE*, 95(2):356–373, 2007.

[BO92] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Eng. Methodol.*, 1(4), 1992.

[Bot08] Jewgenij Botaschanjan. *Techniques for Property Preservation in the Development of Real-Time Systems.* PhD thesis, TU München, 2008.

[BP99] M. Breitling and J. Philipps. Black box views of state machines. Technical Report TUM-I9916, Technische Universität München, 1999.

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik Spektrum*, 30(1):3–18, 2007.

[Bri97] Douglas Bridges. Constructive mathematics. In Edward N. Zalta, editor, *Stanford Encyclopedia of Philosophy.* CSLI, Stanford University, 1997.

[Bro03] Manfred Broy. Service-oriented systems engineering: Modeling services and layered architectures. In *Proceedings of FORTE'03*, pages 48–61, 2003.

[Bro05a] Manfred Broy. Grundlagen der Programm- und Systementwicklung II: Modellierung verteilter Systeme. TU München, 2005. Vorlesungsskript.

[Bro05b] Manfred Broy. A semantic and methodological essence of message sequence charts. *Sci. Comput. Program.*, 54(2-3):213–256, 2005.

[Bro05c] Manfred Broy. Service-oriented systems engineering: Specification and design of services and layered architectures. In *Engineering Theories of Software Intensive Systems*, pages 47–81. Springer-Verlag, 2005.

[Bro07a] Manfred Broy. A theory for requirements specification and architecture design of multi-functional software systems. *Mathematical Frameworks for Component Software: Models for Analysis and Synthesis*, pages 119–154, 2007.

[Bro07b] Manfred Broy. Two sides of structuring multi-functional software systems: Function hierarchy and component architecture. In *Proceedings of SERA'07*, pages 3–12. IEEE Computer Society, 2007.

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces and Refinement.* Springer-Verlag, 2001.

[Buh98] R. J. A. Buhr. Use case maps as architectural entities for complex systems.

IEEE Trans. Softw. Eng., 24(12), 1998.

[BVMR07] Isabel Sofia Brito, Filipe Vieira, Ana Moreira, and Rita A. Ribeiro. Handling conflicts in aspectual requirements compositions. In *Transactions on Aspect-Oriented Software Development III*, volume 4620 of *LNCS*, pages 144–166. Springer-Verlag, 2007.

[CA07] Betty H. C. Cheng and Joanne M. Atlee. Research directions in requirements engineering. In *Proceedings of FOSE'07*, pages 285–303. IEEE Computer Society, 2007.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley, 2000.

[CFN03] Cyril Carrez, Alessandro Fantechi, and Elie Najm. Behavioural contracts for a sound composition of components. In *Proceedings of FORTE'03*, volume 2767 of *LNCS*, pages 111–126. Springer-Verlag, 2003.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[CGP03] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning assumptions for compositional verification. In *Proceedings of TACAS*, volume 2619 of *LNCS*, pages 331–346. Springer-Verlag, 2003.

[CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[CKMRM03] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Comput. Networks*, 41(1):115–141, 2003.

[Coc03] Alistair Cockburn. *Writing Effective Use Cases*. Addison Wesley, 2003.

[Cza04] Krzysztof Czarnecki. Overview of generative software development. In *In Proceedings of UPP*, pages 313–328. Springer-Verlag, 2004.

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface automata. *SIGSOFT Softw. Eng. Notes*, 26(5):109–120, 2001.

[DG08] Deepak D'Souza and Madhu Gopinathan. Conflict-tolerant features. In *Proceedings of CAV'08*, volume 5123 of *LNCS*, pages 227–239. Springer-Verlag, 2008.

[DH01] Werner Damm and David Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[Did06] Festo Didactic. Detailspezifikation der SmartAutomation-Modellanlage, 2006. Version 1.12.

[Dij72] Edsger Wybe Dijkstra. *Structured programming*, chapter Notes on structured programming. Academic Press Ltd., 1972.

[DLvL06] Christophe Damas, Bernard Lambeau, and Axel van Lamsweerde. Scenarios, goals, and state machines: a win-win partnership for model synthesis. In *Proceedings of SIGSOFT'06/FSE-14*, pages 197–207. ACM, 2006.

[dRdBH+01] Willem-Paul de Roever, Frank de Boer, Ulrich Hanneman, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency verification: introduction to compositional and noncompositional methods.* Cambridge University Press, 2001.

[dSMUK07] Felipe Cantal de Sousa, Nabor C. Mendonça, Sebastián Uchitel, and Jeff Kramer. Detecting implied scenarios from execution traces. In *Proceedings of WCRE*, pages 50–59. IEEE Computer Society, 2007.

[EA009] *Proceedings of the ICSE Workshop on Aspect-Oriented Requirements Engineering and Architecture Design.* IEEE Computer Society, 2009.

[EFKN94] Steve Easterbrook, Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh. Coordinating distributed viewpoints: the anatomy of a consistency check. *Concurrent Engineering: Research and Applications*, 2:209–222, 1994.

[EN95] Steve Easterbrook and Bashar Nuseibeh. Using viewpoints for inconsistency management. *Software Engineering Journal*, 11:31–43, 1995.

[FGH+94] A. C. W. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multiperspective specifications. *IEEE Trans. Softw. Eng.*, 20(8):569–578, 1994.

[FGL+08] Alessandro Fantechi, Stefania Gnesi, Alessandro Lapadula, Franco Mazzanti, Rosario Pugliese, and Francesco Tiezzi. A model checking approach for verifying COWS specifications. In *Proceedings of FASE'08*, volume 4961 of *LNCS*, pages 230–245. Springer-Verlag, 2008.

[FLB07] José Luiz Fiadeiro, Antónia Lopes, and Laura Bocchi. Algebraic semantics of service component modules. In *Recent Trends in Algebraic Development Techniques*, volume 4409 of *LNCS*, pages 37–55. Springer-Verlag, 2007.

[FN03] Amy P. Felty and Kedar S. Namjoshi. Feature specification and automated conflict detection. *ACM Trans. Softw. Eng. Methodol.*, 12(1):3–27, 2003.

[FR07] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *Proceedings of FOSE'07*, pages 37–54. IEEE Computer Society, 2007.

[FU08] Dario Fischbein and Sebastian Uchitel. On correct and complete strong merging of partial behaviour models. In *Proceedings of SIGSOFT/FSE-16*, pages 297–307. ACM, 2008.

[GEM06] Matthias Galster, Armin Eberlein, and Mahmood Moussavi. Transition from requirements to architecture: A review and future perspective. In *Proceedings of SNPD-SAWN'06*, pages 9–16. IEEE Computer Society, 2006.

[GGMC+07] Gregor Gößler, Susanne Graf, Mila E. Majster-Cederbaum, Moritz Martens, and Joseph Sifakis. Ensuring properties of interaction systems. In *Program Analysis and Compilation*, volume 4444 of *LNCS*, pages 201–224. Springer-Verlag, 2007.

[GHH07a] Alexander Gruler, Alexander Harhurin, and Judith Hartmann. Development and configuration of service-based product lines. In *Proceedings of SPLC'07*, pages 107–116. IEEE Computer Society, 2007.

[GHH07b] Alexander Gruler, Alexander Harhurin, and Judith Hartmann. Modeling the functionality of multi-functional software systems. In *Proceedings of ECBS'07*, pages 349–358. IEEE Computer Society, 2007.

[GS05] Gregor Gössler and Joseph Sifakis. Composition for component-based modeling. *Sci. Comput. Program.*, 55(1-3):161–183, 2005.

[HA00] Jonathan D. Hay and Joanne M. Atlee. Composing features and resolving interactions. In *Proceedings of SIGSOFT'00/FSE-8*, pages 110–119. ACM, 2000.

[Har88] David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, 1988.

[Har09] Alexander Harhurin. Dienstbasierte Spezifikation des Fertigungs Management Systems. FLASCO-Ergebnisdokument, 2009. http://www4.in.tum.de/~harhurin/publications/FMS2009Flasco.pdf.

[Hen00] Thomas A. Henzinger. Masaccio: A formal model for embedded components. In *IFIP TCS*, pages 549–563, 2000.

[HH08a] Alexander Harhurin and Judith Hartmann. Service-oriented Commonality Analysis Across Existing Systems. In *Proceedings of SPLC'08*, pages 255–264. IEEE Computer Society, 2008.

[HH08b] Alexander Harhurin and Judith Hartmann. Towards consistent specifications of product families. In *FM'08: Proceedings of Formal Methods*, volume 5014 of *LNCS*. Springer-Verlag, 2008.

[HHR09] Alexander Harhurin, Judith Hartmann, and Daniel Ratiu. Motivation and Formal Foundations of a Comprehensive Modeling Theory for Embedded Systems. Technical Report TUM-I0924, TU München, 2009. http://www4.in.tum.de/~harhurin/publications/TUM-I0924.pdf.

[HJK10] Rolf Hennicker, Stephan Janisch, and Alexander Knapp. On the observable behaviour of composite components. *Electr. Notes Theor. Comput. Sci.*, 260:125–153, 2010.

[HJL96] Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw. Automated consistency checking of requirements specifications. *ACM Trans. Softw. Eng. Methodol.*, 5(3):231–261, 1996.

[HK01] David Harel and Hillel Kugler. Synthesizing state-based object systems

from lsc specifications. In *Proceedings of CIAA'00*. Springer-Verlag, 2001.

[HK07] Rolf Hennicker and Alexander Knapp. Activity-driven synthesis of state machines. In *Proceedings of FASE'07*, LNCS, pages 87–101. Springer-Verlag, 2007.

[HKLB98] Constance L. Heitmeyer, James Kirby, Bruce G. Labaw, and Ramesh Bharadwaj. SCR*: A toolset for specifying and analyzing software requirements. In *Proceedings of CAV'98*, pages 526–531. Springer-Verlag, 1998.

[HKM10] David Harel, Amir Kantor, and Shahar Maoz. On the power of play-out for scenario-based programs. In *Concurrency, Compositionality, and Correctness*, volume 5930 of *LNCS*. Springer-Verlag, 2010.

[HKMP02] David Harel, Hillel Kugler, Rami Marelly, and Amir Pnueli. Smart play-out of behavioral requirements. In *Proceedings of FMCAD'02*, pages 378–398. Springer-Verlag, 2002.

[HN96] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[HN98] Anthony Hunter and Bashar Nuseibeh. Managing inconsistent specifications: Reasoning, analysis and action. *ACM Transactions on Software Engineering and Methodology*, 7:335–367, 1998.

[Hoa85] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1985.

[HP98] David Harel and Michal Politi. *Modeling Reactive Systems with Statecharts: The Statemate Approach*. McGraw-Hill, Inc., 1998.

[HSE97] Franz Huber, Bernhard Schätz, and Geralf Einert. Consistent graphical specification of distributed systems. In *Proceedings of FME'97*, pages 122–141. Springer-Verlag, 1997.

[HT09] Benjamin Hummel and Judith Thyssen. Behavioral Specification of Reactive Systems Using Stream-Based I/O Tables. In *Proceedings of SEFM'09*, pages 137–146. IEEE Computer Society, 2009.

[HU79] John Hopcroft and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

[IT99] ITU-TS. Recommendation Z.120 (11/99): Message Sequence Chart (MSC), 1999.

[JBR99] Ivar Jacobson, Grady Booch, and Jim Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[JLHM91] Matthew S. Jaffe, Nancy G. Leveson, Mats P. E. Heimdahl, and Bonnie E. Melhart. Software requirements analysis for real-time process-control systems. *IEEE Trans. Softw. Eng.*, 17(3):241–258, 1991.

[JM86] Farnam Jahanian and Aloysius K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Trans. Softw. Eng.*, 12(9):890–904, 1986.

[JZ98] Michael Jackson and Pamela Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Trans. Softw. Eng.*, 24(10):831–847, 1998.

[KAAK09] Jörg Kienzle, Wisam Al Abed, and Jacques Klein. Aspect-oriented multi-view modeling. In *Proceedings of AOSD'09*, pages 87–98. ACM, 2009.

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, SEI, CMU, Pittsburgh, 1990.

[KGSB99] Ingolf Krüger, Radu Grosu, Peter Scholz, and Manfred Broy. From MSCs to statecharts. In *Proceedings of the Distributed and Parallel Embedded Systems*. Kluwer Academic Publishers, 1999.

[KK98] Dirk O. Keck and Paul J. Kuehn. The feature and service interaction problem in telecommunications systems: A survey. *IEEE Trans. Softw. Eng.*, 24(10):779–796, 1998.

[KM04] Ingolf H. Krüger and Reena Mathew. Systematic development and exploration of service-oriented software architectures. In *Proceedings of WICSA*, pages 177–187. IEEE Computer Society, 2004.

[KMM06] Ingolf H. Krüger, Reena Mathew, and Michael Meisinger. Efficient exploration of service-oriented architectures using aspects. In *Proceedings of ICSE'06*, pages 62–71. ACM, 2006.

[KMWZ10] Alexander Knapp, Grzegorz Marczynski, Martin Wirsing, and Artur Zawlocki. A heterogeneous approach to service-oriented systems specification. In *Proceedings of SAC'10*. ACM, 2010.

[KPP09] Hillel Kugler, Cory Plock, and Amir Pnueli. Controller synthesis from LSC requirements. In *Proceedings of FASE'09*, volume 5503 of *LNCS*, pages 79–93. Springer-Verlag, 2009.

[KR04] Shmuel Katz and Awais Rashid. From aspectual requirements to proof obligations for aspect-oriented systems. In *Proceedings of RE'04*, pages 48–57. IEEE Computer Society, 2004.

[Krü03] Ingolf Krüger. Capturing overlapping, triggered, and preemptive collaborations using MSCs. In *Proceedings of FASE'03*, pages 387–402. Springer-Verlag, 2003.

[KS09] Hillel Kugler and Itai Segall. Compositional synthesis of reactive systems from live sequence chart specifications. In *Proceedings of TACAS*, volume 5505 of *LNCS*, pages 77–91. Springer-Verlag, 2009.

[Lam83] Leslie Lamport. What good is temporal logic. In R. E. A. Mason, editor, *Proceedings of the IFIP 9th World Congress*, 1983.

[LDD06] Hongzhi Liang, Juergen Dingel, and Zinovy Diskin. A comparative survey of scenario-based to state-based model synthesis approaches. In *Proceedings of SCESM'06*, pages 5–12. ACM, 2006.

[Lev95] Nancy G. Leveson. *Safeware: System Safety and Computers*. ACM, 1995.

[Lev00] Nancy G. Leveson. Intent specifications: An approach to building human-centered specifications. *IEEE Trans. Softw. Eng.*, 26(1):15–35, 2000.

[LKMU08] Emmanuel Letier, Jeff Kramer, Jeff Magee, and Sebastian Uchitel. Deriving event-based transition systems from goal-oriented requirements models. *Automated Software Engineering*, 15(2):175–206, 2008.

[LPT08] Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. Specifying and Analysing SOC Applications with COWS. In *Concurrency, Graphs and Models*, volume 5065 of *LNCS*, pages 701–720. Springer-Verlag, 2008.

[LSV03] Nancy Lynch, Roberto Segala, and Frits Vaandrager. Hybrid I/O automata. *Inf. Comput.*, 185(1):105–157, 2003.

[LT89] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 1989.

[MA09] Gunter Mussbacher and Daniel Amyot. On modeling interactions of early aspects with goals. In *Proceedings of the ICSE Workshop on Aspect-Oriented Requirements Engineering and Architecture Design*, pages 14–19. IEEE Computer Society, 2009.

[Man02] Mike Mannion. Using first-order logic for product line model validation. In *Proceedings of SPLC'02*, pages 176–187. IEEE Computer Society, 2002.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice-Hall, Inc., 1989.

[MMT06] Katharina Mehner, Mattia Monga, and Gabriele Taentzer. Interaction analysis in aspect-oriented models. In *Proceedings of RE'06*, pages 66–75. IEEE Computer Society, 2006.

[MP95] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.

[MSKD07] Rabeb Mizouni, Aziz Salah, Siamak Kolahi, and Rachida Dssouli. Merging partial system behaviours: composition of use-case automata. *IET Software*, 1(4):143–160, 2007.

[Muc03] Henry Muccini. Detecting implied scenarios analyzing non-local branching choices. In *Proceedings of FASE'03*, pages 372–386. Springer-Verlag, 2003.

[NC08] Shiva Nejati and Marsha Chechik. Behavioural model fusion: an overview of challenges. In *Proceedings of MiSE'08*, pages 1–6. ACM, 2008.

[NER00] Bashar Nuseibeh, Steve Easterbrook, and Alessandra Russo. Leveraging inconsistency in software development. *Computer*, 33(4):24–29, 2000.

[NMA08] Wonhong Nam, P. Madhusudan, and Rajeev Alur. Automatic symbolic compositional verification by learning assumptions. *Form. Methods Syst. Des.*, 32(3):207–234, 2008.

[NSC+08] Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Sebastian Uchitel, and Pamela Zave. Towards compositional synthesis of evolving systems. In *SIGSOFT'08/FSE-16*, pages 285–296. ACM, 2008.

[OMG04] Unified Modeling Language 2.0 Superstructure Final Adopted Specification, 2004. Object Management Group Document ptc/03-08-02.

[Par72] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Department of Computer Science, Aarhus University, Denmark, 1981.

[PM95] David Lorge Parnas and Jan Madey. Functional documents for computer systems. *Science of Computer Programming*, 25(1):41–61, 1995.

[PP07] Alexander Pretschner and Wolfgang Prenninger. Computing refactorings of state machines. *Software and Systems Modeling*, 6(4), 2007.

[PSE07] Thiemo Pasenau, Thomas Sauer, and Jörg Ebeling. Aktive Geschwindigkeitsregelung mit Stop&Go-Funktion im BMW 5er und 6er. *ATZ*, pages 900–908, 2007.

[RC08] Awais Rashid and Ruzanna Chitchyan. Aspect-oriented requirements engineering: a roadmap. In *Proceedings of the workshop on Early Aspects*, pages 35–41. ACM, 2008.

[RSMA02] Awais Rashid, Peter Sawyer, Ana M. D. Moreira, and Joao Araújo. Early aspects: A model for aspect-oriented requirements engineerin. In *Proceedings of RE'02*, pages 199–202. IEEE Computer Society, 2002.

[Rup07] Chris Rupp. *Requirements-Engineering und -Management.* Hanser Fachbuchverlag, 2007.

[RW89] P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.

[Sch05] Bernhard Schätz. Building components from functions. In *Proceedings of FACS'05*, volume 160 of *ENTCS*, 2005.

[Sch07] Bernhard Schätz. Combining product lines and model-based development. *Electron. Notes Theor. Comput. Sci.*, 182:171–186, 2007.

[Sch08] Bernhard Schätz. Modular functional descriptions. *Electron. Notes Theor. Comput. Sci.*, 215:23–38, 2008.

[SCK09] Daniel Sinnig, Patrice Chalin, and Ferhat Khendek. LTS semantics for use case models. In *Proceedings of SAC'09*, pages 365–370. ACM, 2009.

[SE06] Mehrdad Sabetzadeh and Steve Easterbrook. View merging in the presence of incompleteness and inconsistency. *Requir. Eng.*, 11(3):174–193, 2006.

[SFG10] Mehrdad Sabetzadeh, Anthony Finkelstein, and Michael Goedicke. *Encyclopedia of Software Engineering*, chapter Viewpoints. Taylor and Francis, 2010.

[SFGP05] Bernhard Schätz, Andreas Fleischmann, Eva Geisberger, and Markus Pister. Model-based requirements engineering with AutoRAID. In *GI Jahrestagung*, volume 68 of *LNI*, pages 511–515. GI, 2005.

[SNEC07] Mehrdad Sabetzadeh, Shiva Nejati, Steve Easterbrook, and Marsha Chechik. A relationship-driven framework for model merging. In *Proceedings of MISE'07*. IEEE Computer Society, 2007.

[SNL+07] Mehrdad Sabetzadeh, Shiva Nejati, Sotirios Liaskos, Steve M. Easterbrook, and Marsha Chechik. Consistency checking of conceptual models via model merging. In *Proceedings of RE'07*, pages 221–230. IEEE Computer Society, 2007.

[Som01] Ian Sommerville. *Software Engineering*. Addison-Wesley, 6th edition, 2001.

[SPHP02] Bernhard Schätz, Alexander Pretschner, Franz Huber, and Jan Philipps. Model-based development of embedded systems. In *Proceedings of OOIS Workshops*, pages 298–312, 2002.

[SPW07] Avik Sinha, Amit Paradkar, and Clay Williams. On generating EFSM models from use cases. In *Proceedings of SCESM'07*. IEEE Computer Society, 2007.

[SSR+05] Arnor Solberg, Devon M. Simmonds, Raghu Reddy, Sudipto Ghosh, and Robert B. France. Using aspect oriented techniques to support separation of concerns in model driven development. In *Proceedings of COMPSAC'05*. IEEE Computer Society, 2005.

[STK02] J. Snell, D. Tidwell, and P. Kulchenko. *Programming Web Services with SOAP*. O'Reilly, 2002.

[STT+05] Joachim Steinle, Thomas Toelge, Sonja Thissen, Andreas Pfeiffer, and Martin Brandstater. Kultivierte Dynamik: Geschwindigkeitsregelung im neuen BMW 3er. *ATZ*, pages 122–131, 2005.

[SZ01] George Spanoudakis and Andrea Zisman. Inconsistency management in software engineering: Survey and open research issues. *Handbook of Software Engineering and Knowledge Engineering*, pages 329–380, 2001.

[SZW05] Jing Sun, Hongyu Zhang, and Hai Wang. Formal semantics and verification for feature modeling. In *Proceedings of ICECCS'05*, pages 303–312, 2005.

[TRS+10] Judith Thyssen, Daniel Ratiu, Wolfgang Schwitzer, Alexander Harhurin, Martin Feilkas, and Eike Thaden. A system for seamless abstraction layers for model-based development of embedded software. In *Proceedings of*

Envision 2020 Workshop, 2010.

[UBC09] Sebastián Uchitel, Greg Brunet, and Marsha Chechik. Synthesis of partial behavior models from properties and scenarios. *IEEE Trans. Software Eng.*, 35(3):384–406, 2009.

[UC04] Sebastian Uchitel and Marsha Chechik. Merging partial behavioural models. *SIGSOFT Softw. Eng. Notes*, 29(6):43–52, 2004.

[vL01] Axel van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proceedings of RE'01*. IEEE Computer Society, 2001.

[vL03] Axel van Lamsweerde. From system goals to software architecture. In *Proceedings of SFM'03*, volume 2804 of *LNCS*. Springer Verlag, 2003.

[vL09] Axel van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[vLLD98] Axel van Lamsweerde, Emmanual Letier, and Robert Darimont. Managing conflicts in goal-driven requirements engineering. *IEEE Trans. Softw. Eng.*, 24(11):908–926, 1998.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of LICS'86*. IEEE Computer Society, 1986.

[WBC+09] Martin Wirsing, Laura Bocchi, Allan Clark, Jose Luiz Fiadeiro, Stephen Gilmore, Matthias Hölzl, Nora Koch, Philip Mayer, Rosario Pugliese, and Andreas Schroeder. *At your service: Service Engineering in the Information Society Technologies Program*, chapter SENSORIA: Engineering for Service-Oriented Overlay Computers. MIT Press, 2009.

[WCR09] Nathan Weston, Ruzanna Chitchyan, and Awais Rashid. Formal semantic conflict detection in aspect-oriented requirements. *Requir. Eng.*, 14(4):247–268, 2009.

[Zav03] Pamela Zave. An experiment in feature engineering. *Programming methodology*, pages 353–377, 2003.