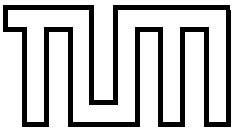
Modellbasierte Adaptierung von Softwarekomponenten am Beispiel Automotive Infotainment

Thomas Pramsohler

Technische Universität München



TECHNISCHE UNIVERSITÄT MÜNCHEN

FAKULTÄT FÜR INFORMATIK

Fachgebiet für Betriebssysteme (F13)

Modellbasierte Adaptierung von Softwarekomponenten am Beispiel Automotive Infotainment

Thomas Pramsohler

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.

Vorsitzende(r): Univ.-Prof. Dr. Jörg Ott

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Uwe Baumgarten
2. Univ.-Prof. Dr. Johann Schlichter

Die Dissertation wurde am 02.10.2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 19.01.2016 angenommen.

Kurzfassung

Ein modernes Fahrzeug enthält mehr als zweitausend Softwarefunktionen, die über verschiedene Bussysteme miteinander kommunizieren und auf bis zu hundert Steuergeräte verteilt sind. Mit der großen Menge an Software und der wachsenden Zahl an Produktvarianten steigt auch der Wunsch nach Wiederverwendung von Softwarekomponenten zur Senkung der Kosten. Eine solche Wiederverwendung setzt stabile Schnittstellen voraus, welche in dieser sich rasch verändernden Umgebung, getrieben von ständigem Wettbewerb, schwer realisierbar sind. Denn es sind gerade die in Software implementierten Funktionen, die den Großteil der Fahrzeug-Innovationen der letzten Jahre ermöglicht haben.

Diese Arbeit stellt ein Konzept zur Adaptierung von Softwareschnittstellen vor und betrachtet dabei die speziellen Anforderungen im Fahrzeug. Dabei ermöglicht ein Software-Adapter die Komposition zweier inkompatibler Softwarekomponenten durch Vermittlung an der Schnittstelle. Der Adapter manipuliert sowohl die Daten als auch das Verhalten der Softwarekomponenten und erhöht somit die Wiederverwendbarkeit einer Softwarekomponente. Das vorgestellte Konzept basiert auf einem Komponentenmodell, welches auf verschiedenen Hierarchieebenen eine klare Trennung zwischen der syntaktischen Beschreibung einer Schnittstelle und dem Verhalten der Softwarekomponente ermöglicht. Der Adapter wird ebenfalls modellbasiert, auf zwei Ebenen beschrieben. Die erste Ebene definiert häufig vorkommende Adaptierungen, welche unabhängig vom aktuellen Zustand der Komponenten ausgeführt werden. Die zweite Ebene adressiert zustandsabhängige Adaptierungen, die durch einen Zustandsautomaten beschrieben werden. Im Vergleich zu einer rein zustandsbasierten Adaptierung ermöglicht diese Trennung die Reduktion der Anzahl der Zustände im Zustandsautomaten und eine robustere Ausführung des Adapters in Bezug auf Abweichungen der Komponenten vom spezifizierten Verhalten. Die in dieser Arbeit vorgestellten Modelle ermöglichen die vollständige Generierung von lauffähigen Adaptern für verschiedene Zielsysteme.

Für die Ausführung des Adapters wird eine auf die Anforderungen im Fahrzeug zugeschnittene Architektur definiert, die besonders für Black-Box Softwarekomponenten geeignete Methoden beinhaltet. Die Arbeit wurde bei der BMW Forschung und Technik GmbH erstellt und die Evaluation erfolgte anhand des in der Serienentwicklung bei BMW eingesetzten Infotainment-Betriebssystems GENIVI.

Vorwort

Die vorliegende Arbeit entstand während meiner Zeit als Doktorand in der BMW Forschung und Technik G.m.b.H. in Zusammenarbeit mit dem Lehrstuhl für Betriebssysteme der Technischen Universität München.

Rückblickend gilt mein Dank an erster Stelle Herrn Prof. Dr. Uwe Baumgarten für die fachliche Betreuung, die wertvollen Diskussionen und die Möglichkeit an seinem Lehrstuhl in einem sehr konstruktiven und angenehmen Umfeld zu promovieren.

Zudem danke ich meinen Kollegen in der BMW Forschung und Technik G.m.b.H. und allen Mitwirkenden im Förderprojekt DANA für die Anregungen und das jederzeit gute Arbeitsklima.

Besonderer Dank gilt meiner Familie und vor Allem meiner Frau Monika für die unentbehrliche Unterstützung und Geduld!

München, im März 2016
Thomas Pramsohler

„... it is not the most intellectual of the species that survives; it is not the strongest that survives; but the species that survives is the one that is able best to adapt and adjust to the changing environment in which it finds itself“
Leon C. Megginson [Meg63]

Inhaltsverzeichnis

Kapitel 1

Einleitung

Das wesentliche Ziel komponentenbasierter Softwareentwicklung ist die Realisierung von Systemen durch das Zusammenfügen von wiederverwendbaren Softwarekomponenten . Ein großer Vorteil, der dadurch erzielt werden soll, ist die Senkung der Kosten durch die Verwendung möglichst vieler Gleichteile in den verschiedenen Produkten. In der Fahrzeug-Softwareentwicklung wird der Begriff des Softwarebaukastens verwendet, der gerade diesen Wunsch nach einer einfachen, oder sogar spielerisch leichten Methode des Zusammenfügens von Softwarekomponenten gut beschreibt. So einfach, wie der Begriff das vielleicht suggeriert, ist die Erstellung eines stabilen Baukastens mit wiederverwendbaren Softwarekomponenten nicht, denn besonders im Bereich der Fahrzeug-Softwarefunktionen erleben wir aktuell die meisten Innovationen und Veränderungen. Wahrscheinlich wäre es deshalb auch etwas naiv zu glauben, dass im Fahrzeugumfeld, getrieben von Konkurrenz und Innovation alle Softwarefunktionen und deren Schnittstellen stabil bleiben. Wenn nun die Schnittstellen nicht stabil gehalten werden können, besteht die Kunst viel mehr darin, die Entwicklungsmethoden und die Softwarearchitektur so flexibel zu gestalten, dass die Interoperabilität von Softwarekomponenten auch bei abweichender Schnittstelle ermöglicht wird.

Die Herausforderung beginnt bereits dabei, festzustellen, ob zwei Softwarekomponenten überhaupt zueinander passen. Idealerweise sollte diese Aussage bereits auf Basis der Spezifikation erfolgen können. Da die Spezifikation trotz aller Qualitätssicherungsmaßnamen manchmal zu abstrakt, unvollständig oder in einzelnen Fällen gar fehlerhaft ist [Bec08], kann die Interoperabilität nur durch umfangreiche Integrationstests ausreichend sichergestellt werden. Nicht nur die in der Spezifikationsphase eingesetzten Methoden verhindern eine kostengünstige Wiederverwendung von Softwarekomponenten, sondern auch die für die Implementierung

eingesetzten Softwarelösungen. Diese führen zu starr gekoppelten Softwarekomponenten, die nur schwer ausgetauscht werden können. Zudem ist es in der Fahrzeugindustrie nicht ungewöhnlich, dass die Software eines Steuergerätes aus den Modulen und Komponenten mehrerer unterschiedlicher Zulieferer besteht [Zee12]. Das Ändern solcher, meist als Blackbox vorliegender Komponenten, ist für den Fahrzeughersteller mit hohem Aufwand und nicht unerheblichen Kosten verbunden.

Diese Dissertation adressiert die genannten Herausforderungen durch ein neues Konzept zur Adaptierung von Softwareschnittstellen. Zentraler Aspekt ist die Modellierung und Generierung von Schnittstellen-Adaptern zur Vermittlung zwischen inkompatiblen Softwarekomponenten. Die Möglichkeit der Adaptierung erhöht die Flexibilität der Softwareschnittstellen und somit auch die Wiederverwendbarkeit der Softwarekomponenten. Auch wenn die Lösungen im Hinblick auf die Anforderungen im Fahrzeug entwickelt wurden, sind sie allgemeingültig für komponentenbasierte Softwaresysteme einsetzbar. Die Konzepte sind modellbasiert und adressieren den Prozess von der Spezifikation der Softwarekomponenten bis hin zur Generierung eines lauffähigen Adapters für ein Zielsystem. Die wesentlichen Bestandteile sind:

- **Hierarchische Schnittstellenmodellierung:** Mit dem hierarchischen Schnittstellenmodell können Softwareschnittstellen auf drei Ebenen spezifiziert werden. Die oberste Ebene beschreibt das *Verhalten* der Softwarekomponente, die mittlere Ebene die über die Schnittstelle ausgetauschten *Events* und die unterste Ebene die *Syntax* der Schnittstelle. Mithilfe dieses Modells kann zum Beispiel auch für das Verhalten jeder Softwarekomponente bereits auf Basis der modellbasierten Spezifikation eine Kompatibilitätsaussage getroffen werden. Das Modell ist so gestaltet, dass aus den drei Ebenen lauffähiger Binärcode für ein Zielsystem generiert werden kann.

- **Geschachtelte Adapter-Modellierung:** Der zweite Beitrag ist die geschachtelte Modellierung von Software-Adaptern. Die wesentliche Neuerung bei dieser Art der Modellierung ist die Trennung des Adapters in einen statischen und einen dynamischen Teil. Der statische beschreibt gängige, meist einfache Schnittstellenänderungen, die global in jedem Zustand angewandt werden. Der dynamische Teil beschreibt eine sehr viel flexiblere zustandsbasierte Adaptierung, die lokal für den jeweilgen Zustand gilt.

- **Architektur zur Ausführung von geschachtelten Adaptern:** Die Trennung zwischen dynamischen und statischen Anteilen erfordert eine eigene Architektur zur Ausführung der Adapter und zur Synchronisation der beiden

Teile.

- **Middlerware-Integration:** Damit der Adapter möglichst reibungslos in ein System bestehend aus Black-Box Softwarekomponenten eingefügt werden kann, wird eine Erweiterung für eine Kommunikationsmiddleware definiert, die das Laden von Adaptern zur Laufzeit ermöglicht. Die Softwarekomponenten selbst müssen für die Adaptierung nicht verändert werden.

Aufbau der Arbeit: Nach dieser Einleitung führt Kapitel 2 den Leser in die Thematik und Terminologie der komponentenbasierten Softwareentwicklung im Fahrzeug ein und beschreibt die Herausforderungen bei der Wiederverwendung von Softwarekomponenten. Darauf aufbauend werden die Anforderungen an eine Lösung zur Adaptierung von Softwarekomponenten abgeleitet und das Ziel der Arbeit definiert. Kapitel 3 stellt aktuelle Ansätze zur Modellierung von Softwareschnittstellen und Adaptern vor und zeigt den aktuellen Stand der Praxis bei der Entwicklung von Fahrzeugsoftware. Schließlich wird in diesem Kapitel auch das Zielsystem GENIVI Linux mit seiner Software-Architektur vorgestellt.

Kapitel 4 stellt den ersten Beitrag dieser Arbeit, die kontrollflusszentrierte Systemmodellierung, vor und zeigt die Modellierung anhand von Beispielen. Dabei wird genauer auf die hierarchische Schnittstellenmodellierung eingegangen und das in dieser Arbeit durchgängig verwendete Fallbeispiel des Parkassistenten eingeführt. Darauf aufbauend beschreibt Kapitel 5 die geschachtelte Adaptierung von Softwareschnittstellen mit statischen und dynamischen Adaptern und zeigt die Anwendung der Modelle am Fallbeispiel. Ferner wird die Architektur zur Ausführung geschachtelter Adapter und ein Algorithmus zur Validierung der Korrektheit eines Adapters vorgestellt. Kapitel 6 zeigt die praktische Realisierung der Konzepte am Beispiel eines aktuellen Infotainment-Betriebssystems. Hier wird nicht nur auf die konkrete Implementierung, sondern auch auf den Prozess der Adapter-Erstellung eingegangen.

Die Arbeit endet schließlich mit einer Zusammenfassung und einem Ausblick.

Kapitel 2

Wiederverwendung und Kompatibilität von Softwarekomponenten im Fahrzeug

Dieses Kapitel definiert die für diese Arbeit zentrale Problemstellung der Wiederverwendung von Softwarekomponenten und zeigt den Bedarf nach einer minimalinvasiven Adaptierung von Softwareschnittstellen im Fahrzeug auf.

Zunächst schafft Abschnitt 2.1 ein grundlegendes Verständnis für Software im Fahrzeug, beschreibt verschiedene Ebenen der Fahrzeugsoftwareentwicklung und ordnet diese Arbeit in die entsprechenden Ebenen ein. Darauf aufbauend definiert Abschnitt 2.2 den Kompatibilitätsbegriff auf verschiedenen Ebenen. Abschnitt 2.3 beschreibt, wieso für einen Softwarebaukasten nicht die gleichen Kriterien angesetzt werden können, wie bei einem mechanischen Baukasten und Abschnitt 2.4 zeigt aktuelle Defizite bei der Wiederverwendung von Softwarekomponenten auf und erklärt diese anhand eines Beispiels. Das daraus resultierende Ziel dieser Arbeit und die davon abgeleiteten Anforderungen an eine Lösung werden in Abschnitt 2.5 formuliert. Schließlich skizziert Abschnitt 2.6 die Lösung der modellbasierten Adaptergenerierung.

Inhaltsverzeichnis

2.1. Ebenen der Automotive Softwareentwicklung

Geprägt durch das in der Automobilindustrie vorherrschende Businessmodell von OEM und Zulieferer, war die Entwicklung von Fahrzeugsoftware lange Zeit auf eine steuergeräteorientierte Architektur ausgerichtet [Bro+08]. Dabei konnte eine Hauptfunktion meist genau einem Steuergerät zugeordnet werden. Durch den wachsenden Bedarf an Softwarefunktionen, die hohe Vernetzung der Funktionen und die zunehmend verteilte Funktionserbringung kann heute oftmals diese Zuordnung von Funktionen zu Steuergeräten nicht mehr eindeutig definiert werden. Um dem hohen Vernetzungsgrad und der verteilten Funktionserbringung Rechnung zu tragen, wurde die steuergeräteorientierte Modellierung von Software um weitere Modellierungsebenen erweitert. Somit wird Fahrzeugsoftware heute auf oberster Ebene als Softwarefunktionsnetz sehr abstrakt modelliert und sukzessive auf konkretere Ebenen bis hin zum lauffähigen Binärcode auf die Ebene der Steuergeräte heruntergebrochen.

Wild et al. [Wil+06] beschreiben ein solches Modell, mit dem domänen- und designspezifisches Wissen in Fahrzeug-Softwareentwicklungsprojekten auf verschiedenen Abstraktionsebenen modelliert werden kann. Dabei beschreibt jede Abstraktionsebene die Sicht auf das Gesamtsystem unter einem bestimmten Abstraktionsgrad. Jede Ebene bietet eigene Modellierungselemente und Sichten, welche unterschiedliche Details spezifizieren oder hervorheben. Das Ebenenmodell der Fahrzeug-Softwareentwicklung wurde in mehreren Beiträgen verwendet oder erweitert [Pre+07; Bro+08]. Zusätzlich bietet der Modellierungsansatz COLA [Hab+10] eine Realisierung des Ebenenmodells mit Werkzeugen zur Analyse und Codegenerierung. Wir verwenden eine Variante des Ebenenmodells für die Definition einer allgemeinen Fahrzeugsoftwarearchitektur und zur Herleitung zweier für die Arbeit zentraler Komponentenbegriffe. Abbildung 2.1 zeigt unsere Interpretation des Ebenenmodells. Im Folgenden werden die einzelnen Ebenen und deren Elemente genauer beschrieben:

2.1.1. Die Featureebene

Auf dieser Ebene werden die vom Kunden sichtbaren Funktionen (*Features*) des Gesamtsystems und deren Abhängigkeiten beschrieben. Synonym zum Feature wird oft auch der Begriff des *Dienstes* auf dieser Ebene verwendet. Ein Formalismus, der für die Beschreibung verwendet werden kann, ist die Feature Oriented Domain Analysis (FODA) [Kan+90]. Die Beschreibungssprache benutzt eine Baumdarstellung zur Beschreibung der Abhängigkeiten zwischen den einzelnen Featu-

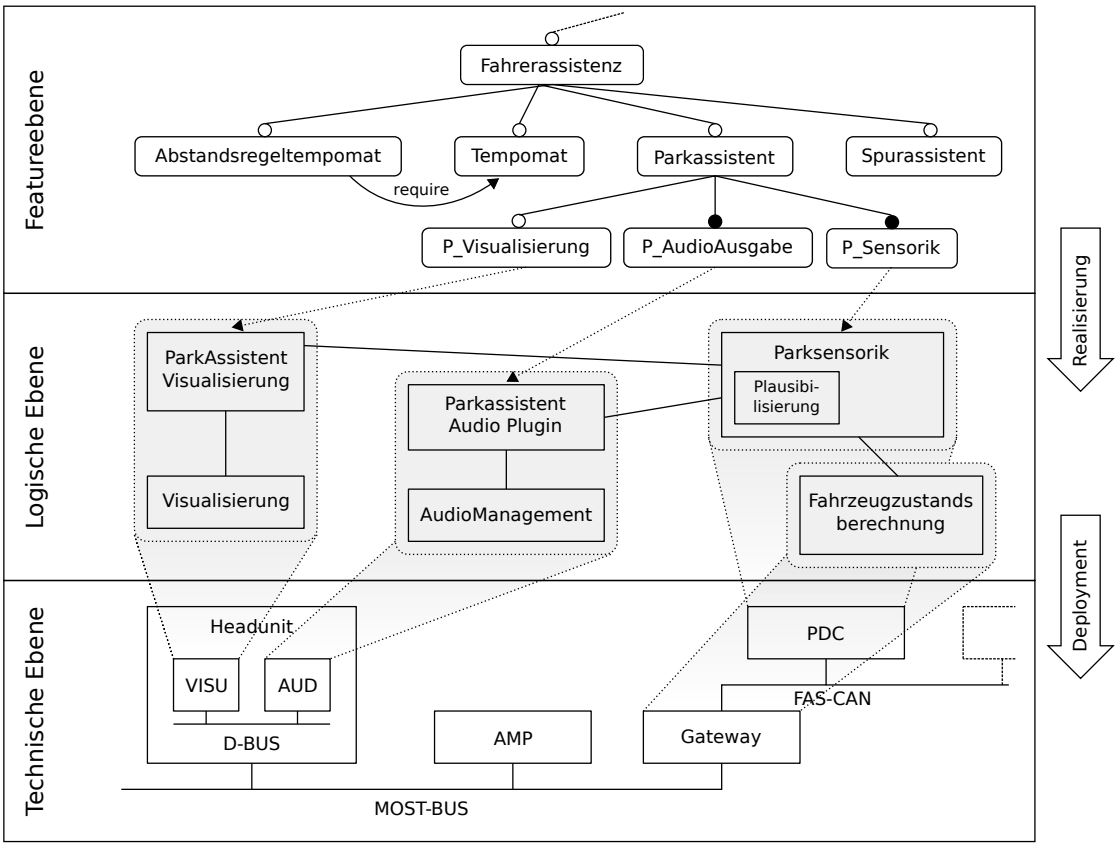


Abbildung 2.1.: Ebenenmodell der Fahrzeug-Softwareentwicklung (angelehnt an [Wil+06]) mit einem fiktiven Beispiel aus dem Bereich Fahrerassistenz.

res. Ein Featurebaum ermöglicht eine hierarchische Dekomposition von Features und die Spezifikation von Zusammenhängen zwischen den Features. Zum Beispiel kann definiert werden, welche Features optional sind, sich gegenseitig ausschließen oder bedingen. Abbildung 2.1 zeigt auf der Featureebene einen Beispielausschnitt eines fiktiven Featurebaumes. Der ausgefüllte Kreis markiert notwendige, der leere Kreis optionale Features. Im Beispiel beinhaltet das Feature *Parkassistent* zwingend die Features *P_AudioAusgabe* und *P_Sensorik* und optional auch das Feature *P_Visualisierung*. Zusätzlich definiert der abgebildete Featurebaum durch eine Assoziation das Feature *Tempomat* als Voraussetzung für das Feature *Abstandsregeltempomat*.

2.1.2. Die logische Ebene

Die Ebene der logischen Architektur definiert eine abstrakte plattformunabhängige Realisierung der einzelnen Features in Form von fachlichen Komponenten. Wir sprechen auf dieser Ebene von *logischen Softwarekomponenten*. Logische Softwarekomponenten sind miteinander über typisierte *Softwareschnittstellen* verknüpft. Zusätzlich kann das Verhalten der einzelnen Softwarekomponenten beschrieben sein. Im Beispiel (siehe Abbildung 2.1) wird das Feature Parkassistentsensorik *P_Sensorik* auf der logischen Ebene durch die Softwarekomponente *Parksensorikverarbeitung* realisiert. Diese ist mit der Komponente *Fahrzeugzustandsberechnung* verknüpft, um beispielsweise beim Einlegen des Rückwärtsganges aktiviert werden zu können. Das *Parkassistent Audio Plugin* ist dafür zuständig, dass die gemessene Entfernung über die Komponente *Audiomanagement* auch auditiv ausgegeben wird. Je nachdem, welcher Zweck mit der Modellierung verfolgt wird, können auf dieser Ebene verschiedene Modellierungssprachen zum Einsatz kommen, zum Beispiel UML Diagramme [OMG10] oder die EAST ADL [Cue+07].

2.1.3. Die technische Ebene

Die Ebene der technischen Architektur beschreibt die Verteilung von Softwarekomponenten auf reale Steuergeräte. Da es sich hier um eine hardwareabhängige Beschreibung handelt, sprechen wir auf dieser Ebene von *technischen Softwarekomponenten*. Dabei werden logische Softwarekomponenten in *Cluster* gruppiert und den Steuergeräten zugeordnet. Neben dieser Komponentenverteilung werden auch Netzwerkverbindungen, Konfigurationsdaten oder die hardwarespezifischen Eigenschaften der Steuergeräte beschrieben. Zusätzlich wird festgelegt, über welche Netzwerke die verschiedenen Daten der verknüpften Funktionen gesendet oder empfangen werden. Im Beispiel aus Abbildung 2.1 werden die Softwarekomponenten *Parkassistent Audio Plugin* und *Parkassistent Video* in einem Cluster gruppiert und dem Steuergeräte Headunit zugeordnet. Sie kommunizieren über den Headunit-internen D-BUS während die Kommunikation mit der Softwarekomponente *Parksensorik Verarbeitung* über den *FAS-CAN*, das Gateway und den *MOST* realisiert werden muss. Je nachdem wie lückenlos die Modellierung auf technischer Ebene realisiert wird, kann aus diesen Modellen lauffähiger Binärcode für die Zielplattform generiert werden. Da eine solche lückenlose Beschreibung schwer realisierbar und oft auch nicht sinnvoll ist, wird meist eine Kombination aus modellbasiert generierter, und von Hand geschriebener Software verwendet.

2.1.4. Deployment von Softwarekomponenten

Basierend auf den Konzepten der untersten beiden Ebenen definieren wir die Begriffe der *logischen* und *technischen Softwarekomponente*.

Definition 2.1.1. ***(Logische Softwarekomponente):*** *Hardwareunabhängiger Teil einer Softwarekomponente mit Unterkomponenten und logischen Schnittstellen zu anderen Softwarekomponenten und einer abstrakten Beschreibung des gewünschten Komponentenverhaltens.*

Die logische Softwarekomponente ist vor allem als Designelement im Entwicklungsprozess relevant. Sie beschreibt zum Beispiel welche Softwarekomponenten miteinander verknüpft sind und welche Daten über die Schnittstellen übertragen werden. Demnach definieren wir die logische Schnittstelle wie folgt:

Definition 2.1.2. ***(Logische Softwareschnittstelle):*** *Die logische Softwareschnittstelle ist der hardwareunabhängige Teil einer Schnittstelle. Sie beschreibt die Struktur der ausgetauschten Daten und deren abstrakte Datentypen. Zusätzlich können Sequenzbedingungen für die Benutzung der Schnittstelle definiert sein.*

Je nachdem auf welchem Zielsystem das Deployment einer Softwarekomponente statt findet, kann die konkrete technische Realisierung anders aussehen. In dieser Arbeit definieren wir den Begriff *Deployment* als die Transformation eines modellierten Systems in ein ausführbares Zielsystem (Definition aus [Hab11]). Für Softwaremodelle beinhaltet das hauptsächlich die Generierung von ausführbarem Binärcode. Das *Deploymentmodell* beschreibt deshalb all jene Informationen, die benötigt werden, um aus den Modellen auf logischer Ebene lauffähigen Binärcode zu generieren. Für logische Softwarekomponenten beschreibt somit das Deployment die Transformation in eine technische Softwarekomponente.

Definition 2.1.3. ***(Technische Softwarekomponente):*** *Die technische Softwarekomponente ist das hardwareabhängige Deployment einer logischen Softwarekomponente. Die technische Softwarekomponente definiert die Eigenschaften der Softwarekomponente in Bezug auf die Zielplattform. Diese Eigenschaften beinhalten zum Beispiel konkrete Programmiersprachen, Zeichencodierungen oder verwendete Betriebssysteme und Hardwarekomponenten.*

Deshalb lässt sich auch eine technische Schnittstelle definiern, denn es geht auf dieser Ebene nun nicht nur um den Austausch von logischen Daten, sondern um eine konkrete Serialisierung der Daten auf einem Kommunikationsmedium. Wir sprechen hier konkret von einer Softwareschnittstelle, da die hardwaretechnische Realisierung der Schnittstelle und deren Eigenschaften wie beispielsweise Modulationstechniken, Stecker, Kabel und Speicherbausteine für diese Arbeit nicht relevant sind.

Definition 2.1.4. ***(Technische Softwareschnittstelle):*** *Die technische Softwareschnittstelle ist die hardwareabhängige Realisierung einer logischen Softwareschnittstelle. Dies betrifft zum Beispiel die verwendete Kommunikationsmiddleware, die konkreten Datentypen oder die Serialisierung der ausgetauschten Daten.*

2.2. Ebenen der Kompatibilität

Die wichtigste Eigenschaft einer Softwarekomponente in Bezug auf ihre Wiederverwendung ist die Kompatibilität mit dem Zielsystem. Deshalb ist der Kompatibilitätsbegriff für diese Arbeit von zentraler Bedeutung. Allgemein werden unter kompatiblen Systemen Systeme verstanden, die “zusammenpassen“ oder “gegeneinander austauschbar“ sind [Bec08]. Je nachdem welche Art der Kompatibilität betrachtet wird, können verschiedene Definitionen sinnvoll sein. Die EN 45020 [DIN98] definiert allgemein Kompatibilität als „die Eignung von Produkten, Prozessen oder Dienstleistungen gemeinsam unter bestimmten Bedingungen benützt werden zu können, um wesentliche Anforderungen zu erfüllen, ohne unannehmbare

gegenseitige Auswirkungen“. In dieser Arbeit wird eine Definition von Kompatibilität verwendet, die etwas mehr die Eigenheiten von mechatronischen Systemen hervorhebt.

Definition 2.2.1. ***(Kompatibilität):*** *Kompatibilität ist die Fähigkeit zweier oder mehrerer Systeme oder Komponenten ihre erforderlichen Funktionen auszuführen, während sie sich eine gemeinsame Hard- oder Softwareumgebung teilen.[IEE90]*

Definition 2.2.1 legt bereits nahe, dass bei der Betrachtung von Kompatibilität verschiedene Aspekte von der logischen Softwarekomponente bis hin zur hardwarespezifischen Realisierung eine Rolle spielen. Dabei definiert die Schnittstelle der Softwarekomponente einen Vertrag (Kontrakt) der für eine korrekte Interaktion mit Nachbarkomponenten eingehalten werden muss. Beugnard [Beu+99] definiert diese Kontrakte auf verschiedenen Abstraktionsebenen. Da Kompatibilität in diesem Sinne die Vertragstreue zwischen den Softwarekomponenten bezeichnet, analog zum Kontrakt auf verschiedenen Ebenen auch von verschiedenen Ebenen der Kompatibilität gesprochen werden. Somit sichert die Einhaltung des Vertrags auf einer Ebene auch die Kompatibilität auf dieser Ebene.

In vielen Arbeiten, in denen Kompatibilität oder die Evolution von Softwarekomponenten [ABP12] behandelt wird, ist die Definition von Kompatibitätsebenen die Basis. Wir nehmen die Arbeiten von Beugnard [Beu+99], Becker et al. [Bec+06] und Bechter [Bec08] als Grundlage für die Definition von verschiedenen Ebenen der Kompatibilität von Softwarekomponenten im Fahrzeug. Abbildung 2.2 ordnet die Ebenen der Kompatibilität in das Ebenenmodell der Fahrzeugsoftwarearchitektur ein. Dabei sind die unteren Kompatibilitätsebenen Voraussetzung für eine Kompatibilität der übergeordneten Ebenen. Die Ebenen der Kompatibilität werden nun genauer beschrieben:

- **Konzeptuelle Kompatibilität:** Konzepte charakterisieren jedes syntaktische Element der ausgetauschten Daten und ordnen es einem realen Objekt zu. Ein Beispiel für ein konzeptuell kompatibles aber syntaktisch inkompatibles Element wäre ein synonym verwendeter Parameter der zwar anders benannt ist, aber im Grunde das gleiche reale Objekt beschreibt (zum Beispiel “Auto“ und “PKW“). Für mehr Informationen zur konzeptuellen Kompatibilität verweisen wir den Leser auf [Bec+06].

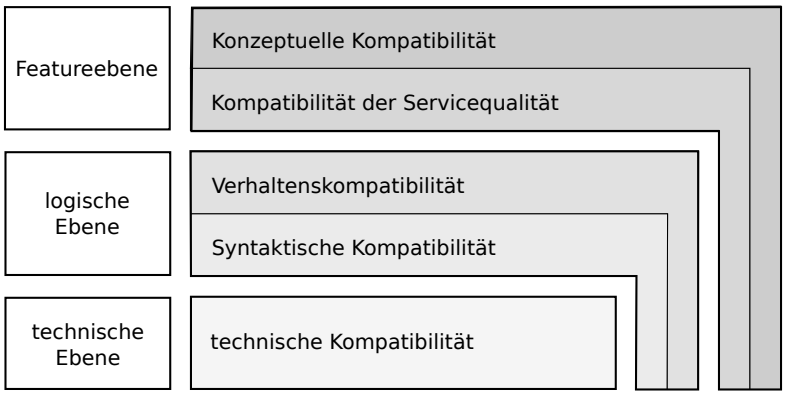


Abbildung 2.2.: Ebenen der Kompatibilität.

- **Kompatibilität der Servicequalität:** Zwei Komponenten sind kompatibel in Bezug auf ihre Servicequalität, wenn die notwendigen Qualitätsattribute (z.B. Effizienz, Verfügbarkeit oder Sicherheit) von beiden eingehalten werden.
- **Verhaltenskompatibilität:** Zwei Komponenten sind miteinander verhaltenskompatibel, wenn jede Komponente die Nachrichten zu den Zeitpunkten sendet, in denen sie von der anderen Komponente verarbeitet werden können oder müssen. Oft wird diese Art der Kompatibilität auch als dynamische Kompatibilität oder Protokollkompatibilität bezeichnet.
- **Syntaktische Kompatibilität:** Zwei Komponenten sind miteinander syntaktisch kompatibel, wenn die Syntax der ausgetauschten Daten übereinstimmt. Diese Syntax wird meist in einer Interfacebeschreibungssprache (IDL) definiert und beinhaltet zum Beispiel Name und Typisierung von Parametern und Methoden oder die Reihenfolge der Parameter einer Methode. Für eine IDL gibt es meist dazugehörende Code-Generatoren, welche bereits ein Grundgerüst für die Software definieren und die Kompatibilität auf syntaktischer Ebene sicher stellen.
- **Technische Kompatibilität:** Technische Kompatibilität betrifft die Interaktion der technischen Softwarekomponente mit ihrer Umgebung in Bezug auf die Zielplattform. Dies beinhaltet die Kompatibilität zur verwendeten Hardware, zum Betriebssystem und zu den Kommunikationstechnologien. Die technische Kompatibilität ist die erste Voraussetzung für die Lauffähigkeit einer Softwarekomponente auf einer Zielplattform.

In dieser Arbeit liegt der Fokus auf der logischen und technischen Ebene und

dem Übergang dieser beiden Ebenen. Deshalb bezieht sich der Begriff der Kompatibilität hier immer auf die technische, syntaktische und Verhaltenskompatibilität.

Zusätzlich zu dieser hierarchischen Klassifizierung, wird auch zwischen *Rückwärts-* und *Vorwärtskompatibilität* unterschieden. Rüchwärtskompatibilität kann durch die Einbindung der älteren Schnittstellenversionen und deren Implementierung realisiert werden. Je nach verwendeter Kommunikationsmiddleware können sogar Schnittstellenänderungen erfolgen, welche die Rückwärtskompatibilität der Softwarekomponente nicht beeinflussen. Zum Beispiel ändert das Hinzufügen einer neuen Methode meist nichts an der Rückwärtskompatibilität der Softwarekomponente. Vorwärtskompatibilität wird meist durch die Unterstützung von *Erweiterbarkeit* realsiert [ABP12]. Wir betrachten in dieser Arbeit beide Arten der Kompatibilität für Black-Box Softwarekomponenten.

2.3. Das Softwarebaukastendilemma

Ziel eines Baukastens ist es, unterschiedliche Produktvarianten aus standardisierten Teilen zusammensetzen zu können. Ein gut strukturierter Baukasten ermöglicht es, differenzierte Produkte für individuelle Kundenwünsche mit relativ geringem Aufwand zu realisieren. Zusätzlich erzielt man in der Automobilindustrie durch einen guten Baukasten - bedingt durch die hohen Stückzahlen - schnell Skaleneffekte bei der Kostenreduktion. Deshalb wird versucht, sowohl für Hard- als auch für Software, möglichst viele Komponenten in den Baukasten zu integrieren.

Grundvoraussetzung für die Realisierung des Baukastens ist die Kompatibilität der Komponentenschnittstellen mit den verschiedenen Zielsystemen. Deshalb wird für aktuelle Baukästen eine einheitliche und stabile Schnittstelle bevorzugt. Bisherige Baukastenkonzepte beziehen sich vorwiegend auf mechanische Komponenten und definieren als Schnittstelle zum Beispiel Schraubpunkte, maximale Kräfte oder Temperaturen, Abmessungen einer Komponente oder Steckverbindungen. Eine Baukastendefinition umfasst einerseits klar definierte Schnittstellen und andererseits einen gewissen Variierungsspielraum. Innerhalb dieses Spielraumes dürfen die Komponenten frei definiert werden. Eine gute mechanische Baukastenschnittstelle, die diesen Variierungsspielraum maximiert, ist deshalb sehr schlank und allgemein gehalten. Bei mechanischen Schnittstellen kann ein geeigneter Abstraktionsgrad beispielsweise so gewählt werden, dass die Schnittstelle so schlank wie möglich definiert wird, bei gleichzeitig korrekter Funktionserbringung der Komponente.

Um einen derartigen Baukasten ebenfalls für Software zu realisieren, müssen die Schnittstellen der Softwarekomponenten standardisiert werden. Für Schnittstel-

len im Infotainment sind Softwarekomponenten mit einer dreistelligen Anzahl an Funktionen und Parametern keine Seltenheit. Zudem besitzt jeder Parameter eine Typdefinition und möglicherweise Sub-Parameter. Zusätzlich zu dieser syntaktischen Definition der Schnittstelle, ist das gewünschte Verhalten der Softwarekomponente in Bezug auf die Aufrufreihenfolge der Deklarationen festgelegt. Darüber hinaus gilt es noch etliche funktionale und nicht-funktionale Anforderungen zu erfüllen, damit die Softwarekomponente mit ihrer Umgebung korrekt interagiert.

Dieses Beispiel macht deutlich, dass bereits eine steuergeräteinterne Schnittstelle im Infotainment durch ihre Komplexität eine sehr große Herausforderung für die Realisierung des Softwarebaukastens darstellt. Jede Abweichung von der Spezifikation der Schnittstelle kann zu Fehlern im System führen. Nun ist es bei Software möglich, die Schnittstelle bis hin zu einem einfachen Zeiger auf eine Speicherzelle (void *) beliebig abstrakt zu definieren. Durch eine Verschlankung der Schnittstelle wird allerdings die Komplexität von der Schnittstelle in die Komponente verlagert und auch in Bezug auf Kompatibilität kein Vorteil erzielt, da die Inhalte der Speicherzelle trotzdem korrekt interpretiert werden müssen. Zusätzlich kann die syntaktische Kompatibilität einer schlanken Schnittstelle mit vielen interpretierten Daten nicht bereits durch den IDL Code-Generator hergestellt werden, sondern muss durch aufwändiges Testen verifiziert werden. Deshalb gilt im Bereich Software die Größe einer Schnittstelle nicht als Maß für ihre Wiederverwendbarkeit.

2.4. Defizite bei der Wiederverwendung von Softwarekomponenten

Der Kernaspekt einer baukastenorientierten Softwareentwicklung ist die *Wiederverwendung* von Softwarekomponenten. Damit der Baukasten effektiv funktioniert, müssen verschiedene Bereiche des Softwareentwicklungsprojektes entsprechende Konzepte für die Realisierung von Wiederverwendung bieten. Beispielsweise muss der Softwareentwicklungsprozess selbst so ausgerichtet sein, dass er Wiederverwendung unterstützt. Auch die Verantwortlichkeiten im Softwareentwicklungsprojekt müssen so definiert sein, dass die Einhaltung der im Baukasten definierten Schnittstellen garantiert werden kann. Zusätzlich müssen die eingesetzten Technologien Konzepte für die Wiederverwendung von Softwarekomponenten bieten. Garlan et al. [GAO95; GAO09] identifizieren als das größte Hindernis bei der Wiederverwendung, die Annahme über bestimmte Eigenschaften der Umgebung, welche oft in einer neuen Umgebung nicht erfüllt werden. Diese Eigenschaften betreffen meist

die unteren Ebenen des in Abschnitt 2.1 vorgestellten Ebenenmodells und somit bergen besonders die Konzepte auf logischer und technischer Ebene noch großes Potential für eine baukastenorientierte Softwareentwicklung. Deshalb stehen diese beiden Ebenen hier besonders im Fokus.

Bei der Softwareentwicklung im Automobil können verschiedene Defizite identifiziert werden. Diese ergeben sich aus den eingesetzten Methoden und Tools oder aus dem im Automobilumfeld vorherrschenden Businessmodell von OEM und Zulieferer. Folgendes Beispiel dient zur Veranschaulichung der Defizite:

Beispiel GUI-update: Durch die Markteinführung eines neuen Fahrzeugmodells erfährt die grafische Benutzeroberfläche (GUI) der Funktion *Parkassistent* innerhalb des Infotainmentsystems eine wesentliche Neuerung. Die neue Komponente wurde von einem Zulieferer erstellt und wird als Black-Box im Binärformat geliefert. Nun ist es der Wunsch, bereits in Produktion befindliche Fahrzeuge zu aktualisieren und auch mit der neuen Benutzeroberfläche auszustatten. Diese Fahrzeuge verwenden dasselbe Betriebssystem und dieselbe Middlewaretechnologie, nur die Schnittstellen der Komponenten sind hardwarebedingt leicht unterschiedlich.

In diesem Szenario gibt es für die Integration der neuen Benutzerschnittstelle mehrere Möglichkeiten. Als erster Schritt muss eine Analyse stattfinden, inwiefern sich die neue Schnittstelle von der vorherigen unterscheidet. Anschließend wird entweder das Zielsystem angepasst, damit die Schnittstellen kompatibel sind, oder eine zweite Version der Benutzerschnittstelle wird speziell für das bereits in Produktion befindliche Fahrzeug erstellt. Die dritte Möglichkeit wäre die Installation einer Vermittlerkomponente (Adapter), welche die Kompatiblität zwischen den inkompatiblen Komponenten herstellt. Die in diesem Szenarion auftretenden Defizite lassen sich grob den einzelnen Phasen des Entwicklungsprozesses zuordnen:

2.4.1. Defizite in der Spezifikationsphase

Beim Fahrzeughersteller, der als Integrator für das Gesamtsystem fungiert, hat die Modellierung von Schnittstellen einen hohen Stellenwert. Denn die Schnittstelle dient als Vertrag zwischen den einzelnen Komponentenverantwortlichen. Eine saubere Spezifikation und Einhaltung der Schnittstelle garantiert deshalb eine möglichst reibungslose Integration der Komponente in das Gesamtsystem.

Rein syntaktische Schnittstellenmodellierung: Aktuell eingesetzte Modelle zur Schnittstellenmodellierung decken hauptsächlich die syntaktischen Eigenschaften einer Softwareschnittstelle ab. Zusätzliche Eigenschaften wie zum Beispiel die gültige Aufrufreihenfolge der Methoden oder zulässige Antwortzeiten werden nur teilweise formal erfasst. Im Beispiel GUI-update könnte ein Client, welcher die Benutzerschnittstelle des Parkassistenten verwendet, zwar dieselbe syntaktische Schnittstelle, jedoch eine unterschiedliche Aufstartreihenfolge implementieren. Auch diese Eigenschaften einer Komponente sind für eine Kompatibilitätsprüfung unerlässlich (siehe Abschnitt 2.2). Häufig sind solche semantischen Aspekte einer Komponente im Lastenheft in natürlicher Sprache spezifiziert und mit grafischen Notationen angereichert. So kann speziell für die Aufstartreihenfolge ein Sequenzdiagramm zur Beschreibung der korrekten Reihenfolge der Nachrichten spezifiziert sein. Aktuell besitzen diese Modelle keine einheitliche Datenbasis oder sind lediglich Grafiken und ermöglichen deshalb keine Analyse der Kompatibilität auf anderen Ebenen außer der syntaktischen.

Uneinheitliche Modelle: Ein weiterer Nachteil ergibt sich durch den Umstand, dass Modellierungswerkzeuge umfangreiche Möglichkeiten zur Erweiterung der Modelle bieten. Dies gibt dem Modellierer zwar die Freiheit, das Modell nach eigenem Wunsch anzupassen, hat allerdings zur Folge, dass Informationen, welche die Schnittstelle betreffen, nicht einheitlich vorliegen. Ein Beispiel für eine Modellerweiterung ist die Möglichkeit in UML [OMG10] für verschiedene Elemente Quelltext in einer Programmiersprache zu definieren. So kann im Beispiel des Parkassistenten ein Testfall in Form eines ein UML Sequenzdiagrammes existieren, welcher den Aufstartprozess beschreibt. Dieses Modell enthält Informationen über das korrekte Aufstartverhalten und definiert in welcher Reihenfolge die Bestandteile der Schnittstelle aufgerufen werden müssen. Da in diesem Fall das Modell vielleicht speziell für den Test erstellt wurde, enthält es Quelltext zur Ausführung des Testfalls. Dieser eingebettete Quelltext und die Freiheiten in der Modellierung machen es schwer automatisch zu identifizieren, welche Anteile des Modells zur Schnittstellenbeschreibung und welche zum Testfall gehören.

2.4.2. Defizite in der Implementierungsphase

Für die Vernetzung von verteilten Softwarefunktionen wird im Fahrzeug auf Kommunikationsmiddleware zurückgegriffen. Dabei existieren je nach Anforderungen verschiedene Middlewaretechnologien. Grundsätzlich folgen die technischen Lösungen dem gleichen Schema: Die syntaktische Schnittstelle der Komponente wird in

einer IDL definiert und anschließend wird durch ein Deployment Binärcode für das Zielsystem generiert. Der als Zwischenschritt generierte Quelltext enthält hauptsächlich zwei Bestandteile. Den *Server-Stub*, der das Interface definiert, welches ein Server implementieren muss und den *Client-Proxy*, der eine Implementierung zur Kommunikation mit einem Server verwendet wird. Diese Teile kümmern sich um die Serialisierung der Daten und garantieren die Einhaltung der syntaktischen Schnittstelle zwischen Client und Server.

Fehlende technologische Unterstützung von Wiederverwendung: Ab dem Zeitpunkt der Generierung von Stub und Proxy ist eine Veränderung der Komponentenschnittstelle nur durch eine Veränderung des Schnittstellenmodells und einer Neugenerierung des Quelltextes möglich. Dies hat auch das Anpassen und Neuübersetzen aller mit dem Proxy oder dem Stub verbundenen Bestandteile zur Folge. Im genannten Beispiel des GUI-updates fand eine Evolution der Schnittstelle statt. Entweder müssen für die Integration des neuen Parkassistenten als Server alle verbundenen Client-Proxies neu generiert werden, oder der Server-Stub des Parkassistenten wird neu generiert. Für die Implementierung bedeutet das, dass in beiden Fällen eine Anpassung von Komponenten und somit auch umfangreiche Tests der veränderten Komponenten notwendig sind. Dieses Defizit betrifft nicht nur verteilte Systeme im Fahrzeug, sondern ebenfalls klassische Lösungen wie CORBA [OMG12] oder Web Services.

2.4.3. Defizite in der Integrationsphase

Softwarekomponenten im Fahrzeug können vom Fahrzeughersteller (OEM) selbst oder von Zulieferern entwickelt werden. Meist ist der OEM für systemübergreifende Funktionen zuständig, welche auf vielen Steuergeräten eingesetzt werden, wie zum Beispiel das Fahrzeugzustandsmanagement oder Energiemanagement.

Black-Box Design von Softwarekomponenten: Im Falle der Entwicklung durch den OEM sind Änderungen an den Komponenten einfach umsetzbar, da der Quelltext bekannt ist und die Artefakte für Test, und Codegenerierung vorliegen. Im genannten Beispiel wurde die Komponente bei einem Zulieferer beauftragt und im Binärformat geliefert. Der Integrator beim OEM ist damit selbst nicht in der Lage, Änderungen an der Komponentenschnittstelle vorzunehmen. Nachträgliche Änderungen der Schnittstellen haben deshalb oft eine neue Bestellung zur Folge. So können auch nur kleine Änderungen hohe Kosten verursachen, die sich durch die Beauftragung und den Änderungsprozess ergeben.

Manuelle Erstellung von Vermittlerkomponenten (Adaptern): Eine Lösung beim Einsatz von inkompatiblen Black-Box Komponenten wäre die Erstellung einer Vermittlerkomponente. Die Aufgabe des Vermittlers ist es, die Kompatibilität der Schnittstellen herzustellen. Der Vorteil bei der Verwendung eines Vermittlers ist, dass die Komponenten selbst nicht verändert werden müssen. Aktuell werden solche Vermittler hauptsächlich manuell erstellt und müssen deshalb aufwändig getestet werden.

2.5. Ziel der Arbeit

Die vorhergehenden Abschnitte haben gezeigt, dass der Wunsch nach einer baukastenorientierten Softwareentwicklung mit den aktuellen Entwicklungsmethoden und Kommunikationtechnologien nicht erfüllt werden kann. In dieser Arbeit wird deshalb ein passendes Konzept erarbeitet, welches es erlaubt, die Schnittstellen einer Softwarekomponente individuell auf die Zielumgebung anzupassen, ohne die eigentliche Komponente zu verändern. Das übergeordnete Ziel kann folgendermaßen zusammengefasst werden:

Ziel dieser Arbeit ist die Entwicklung und Evaluierung eines durchgängigen Konzeptes, zur minimal-invasiven Anpassung der Schnittstellen einer Black-Box Softwarekomponente zur Erhöhung ihrer Wiederverwendbarkeit.

Minimal-invasiv bedeutet in diesem Kontext, dass die Schnittstelle einer Komponente verändert wird, ohne die Weiterentwicklung oder Anpassung der Komponente selbst. Für diese übergeordnete Zielsetzung lassen sich in Verbindung mit den zuvor identifizierten Defiziten folgende Anforderungen ableiten:

Unterstützung von Wiederverwendung auf technischer Ebene: Heute eingesetzte Middlewaretechnologien unterstützen nicht die Änderung der Komponentenschnittstellen ohne die Anpassung der Komponente selbst. Um diese starre Kopplung der Komponenten aufzubrechen, sollen die bereits existierenden Konzepte auf technischer Ebene so erweitert werden, dass ein nachträgliches Anpassen der Schnittstelle einer Black-Box Softwarekomponente ermöglicht wird.

Erweiterte Modellierung von Schnittstellen: Viele in der Praxis eingesetzten IDL's werden primär zur Generierung von Quelltext verwendet. Sie beschreiben deshalb hauptsächlich die syntaktischen Elemente der Schnittstelle. Damit die Kompatibilität von Komponenten über mehrere Ebenen (Siehe Abschnitt 2.2)

sichergestellt werden kann, muss diese klassische (syntaktische) Schnittstellenbeschreibung erweitert werden. Dies betrifft besonders die erlaubte Aufrufreihenfolge der Funktionen und Timing-Anforderungen. Eine wichtige Anforderung für diese Erweiterung ist die Fähigkeit, auch für das beschriebene Verhalten, die Generierung von Binärcode zu unterstützen.

Integration bestehender Schnittstellenmodelle: Auf syntaktischer Ebene ist die Beschreibungssprache sehr stark auf die technologischen Eigenschaften der Kommunikationsmiddleware ausgelegt. Deshalb besitzt auch jede Middlewaretechnologie bereits eine dedizierte IDL. Eine sinnvolle Schnittstellenmodellierung sollte deshalb die Möglichkeit bieten, aktuell eingesetzte Modellierungssprachen auf syntaktischer Ebene wiederzuverwenden. Dies ermöglicht die Wiederverwendung der Code-Generatoren, die auf technischer Ebene eine Deployment für verschiedene Zielplattformen ermöglichen.

Minimierung von Quelltext in den Modellen: Für die Definition von syntaktischen Schnittstellen werden modellbasierte Ansätze verbunden mit Codegeneratoren bereits umfassend eingesetzt. Für die Modellierung von Verhalten hingegen, werden Modelle nicht konsequent genutzt. Es existieren Ansätze zur Modellierung von Verhalten, welche eine Anpassung auf die Zieldomäne durch das Anreichern des Modells mit Quelltext erlauben. Diese Konzepte sind sehr mächtig, allerdings auch sehr fehleranfällig und die entstehenden Modelle sind sehr komplex und dadurch schwer validierbar. Deshalb soll eine durchgängig modellbasierten Ansatz definiert werden, der den Anteil von Quelltext im Modell minimiert.

Erleichterung von häufigen Schnittstellenänderungen: Dig und Johnson [DJ05] analysierten die Evolution von drei gängigen Programmierinterfaces. Sie kamen zum Schluss, dass 80% der Änderungen, die eine Applikation inkompatibel zu einem Programmierinterface machen, durch die Evolution der syntaktischen Schnittstelle verursacht werden. Dies sind zum Beispiel das Umbenennen, das Hinzufügen oder das Entfernen von Funktionen oder Parametern. Deshalb sollen speziell diese häufigen Änderungen durch ein einfaches Konzept adressiert werden, welches ohne die Modellierung komplexer Zustandsautomaten auskommt.

Automatisierter Workflow zur Anpassung von Schnittstellen: Eine manuelle Anpassung von Softwarekomponenten auf Quelltextebene ist sehr aufwändig und meist fehleranfällig. Daher muss das Lösungskonzept eine möglichst automati-

sierte Lösung für die Prüfung von Kompatibilität, die Modellierung und die Generierung von lauffähigen Adaptern für inkompatible Softwarekomponenten bieten.

Sparsamer Umgang mit Ressourcen: Hardware im Fahrzeug ist sehr genau an die Anforderungen der darauf laufenden Software abgestimmt. Das bedeutet, dass nicht viel Spielraum für funktionale Erweiterungen besteht. Deshalb muss eine Realisierung sparsam mit den vorhandenen Ressourcen umgehen.

2.6. Lösungsansatz - Modellbasierte Adaptergenerierung mit statischen und dynamischen Adaptern

Das in dieser Arbeit vorgestellte Konzept zur minimal-invasiven Anpassung von Softwareschnittstellen bietet Lösungen für die in Abschnitt 2.4 identifizierten Defizite bei der Wiederverwendung von Softwarekomponenten. Obwohl die geschilderten Defizite domänenunabhängig für alle Softwareschnittstellen im Fahrzeug gelten, adressiert diese Arbeit die Infotainmentdomäne als Zieldomäne. Gleichwohl ist es das Ziel, die Konzepte so allgemein zu definieren, dass sie auf andere Domänen übertragbar sind.

Mateescu et al.[MPS12] identifizieren *Softwareadaptierung* (kurz. *Adaptierung*) (weiterführende Literatur siehe [Bec+06; CMP06]) als vielversprechendes Konzept, um eine Schnittstellenanpassung umzusetzen. Gerade wegen der Eigenschaft der nicht- oder minimal-invasiven Anpassung von Black-Box Komponenten ist die Generierung von *Adaptern* auch die Kernfunktionalität der in dieser Arbeit vorgestellten Lösung. Generell fungiert ein Adapter als Vermittler/Mediator [Wie92; Wie95] zwischen zwei inkompatiblen Komponenten. Dabei kann die Adaptierung auf allen in Abschnitt 2.2 beschriebenen Kompatibilitätsebenen durchgeführt werden. Wir adressieren in dieser Arbeit vor allem die technische, syntaktische und Verhaltensadaptierung. Unser Ziel ist es, den Prozess der Adaptererstellung in hohem Maße zu automatisieren und die Anforderungen der Fahrzeugdomäne zu berücksichtigen (siehe Abschnitt 2.5).

Abbildung 2.3 zeigt einen Überblick über den Lösungsansatz der modellbasierten Adaptergenerierung am Beispiel von zwei inkompatiblen Komponenten *C1* und *C2*. *C2* Bietet die Schnittstelle *s* in Version 2 an, während *C1* die Schnittstelle in Version 1 benötigt. Wir nehmen an, dass für beide Schnittstellen ein entsprechendes Schnittstellenmodell existiert, welches die Syntax der Schnittstelle und

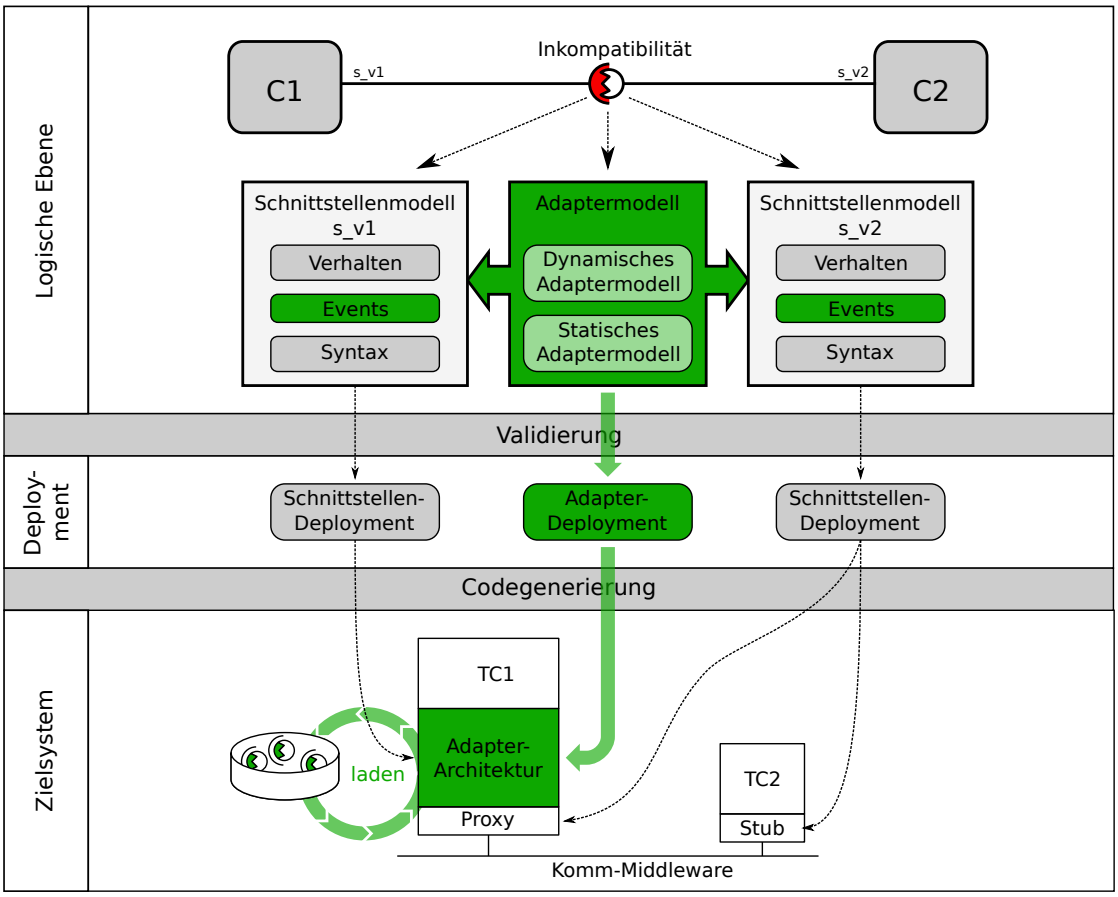


Abbildung 2.3.: Modellbasierte Adaptergenerierung - Überblick.

den über die Schnittstelle erlaubten Kontrollfluss beschreibt. Der Lösungsansatz besitzt folgende Eigenschaften:

Schnittstellenmodellierung: Der Ansatz bietet mit dem *hierarchischen Schnittstellenmodell* die Möglichkeit, Schnittstellen sowohl auf syntaktischer, als auch auf Verhaltensebene zu beschreiben. Die zusätzliche Ebene des *Eventmodells* bildet dabei die Brücke zwischen Syntax und Verhalten und ermöglicht die Abstraktion von der konkreten IDL. Zusätzlich ermöglichen wir die Spezifikation von Zeitbedingungen im Verhaltensmodell.

Adaptermodellierung und Verifikation: Der Adapter wird auf logischer Ebene durch zwei Modelle, dem *statischen* und *dynamischen Adaptermodell*, beschrie-

ben. Wir unterscheiden dabei zwischen häufigen, immer wiederkehrenden Adaptierungsfällen. die mithilfe von einfachen *Mustern* modelliert werden können, und komplexen Verhaltensadaptierungen die mithilfe eines *Schnittstellenautomaten* beschrieben werden können. Zusätzlich können für den Schnittstellenautomaten Zeitbedingungen spezifiziert werden, die eine zeitliche Steuerung von Adapter-Aktionen ermöglichen. Somit bietet der Ansatz ein einfaches Werkzeug zur Behebung von häufigen Inkompatibilitäten und zugleich ermöglicht er die Behebung von komplexen Inkompatibilitäten, welche vom aktuellen Zustand der Kommunikation abhängen. Diese Aufteilung bringt zudem Vorteile, was die Robustheit des Adapters und die Generierung von effizientem Quelltext betrifft.

Durch den Ansatz kann nach der Erstellung des Adaptermodells bereits auf Modellebene verifiziert werden, ob der modellierte Adapter ein korrekter Mediator in Bezug auf den Kotrollfluss zwischen Komponente C1 und Komponente C2 ist.

Adapter-Architektur: Für die Laufzeitumgebung werden Konzepte definiert, die eine minimal-invasive Integration des Adapters in ein Zielsystem ermöglichen. Komponenten können somit selbst eine Inkompatibilität der Schnittstellen erkennen und zur Laufzeit den entsprechenden Adapter laden. Zusätzlich spiegelt sich die Aufteilung des Adapters in einen statischen und dynamischen Teil ebenfalls in der Architektur wieder.

Adapter-Deployment: Das Adaptermodell und ein zusätzliches Adapterdeployment enthalten alle für die Codegenerierung notwendigen Informationen. Der Modellierungsansatz ist so definiert, dass für die syntaktischen Schnittstellen auf die Beschreibungsmittel und Codegeneratoren einer bestehenden Kommunkationsmiddleware zurückgegriffen werden kann. Somit bietet der Lösungsansatz durchgängige Konzepte, vom Systemmodell bis hin zum lauffähigen Adapter für eine Zielplattform, ohne dabei den Binärcode der inkompatiblen Softwarekomponenten zu verändern.

2.7. Zusammenfassung

Dieses Kapitel hat einen generellen Überblick über die Entwicklung von Fahrzeugsoftware gegeben und grundlegende Begriffe zur Wiederverwendung und Kompatibilität von Softwarekomponenten definiert. Darauf aufbauend wurden aktuell vorherrschende Defizite bei der Wiederverwendung von Softwarekomponenten im Fahrzeug aufgezeigt und anhand eines Beispiels erklärt. Ein zentraler Punkt ist dabei, dass die besonderen Anforderungen der Wiederverwendung von Black-Box

Softwarekomponenten im aktuellen Entwicklungsprozess und von den eingesetzten Softwarelösungen nicht ausreichend berücksichtigt werden.

Das Ziel dieser Arbeit ist es daher, ein Konzept zu erstellen, das eine minimalinvasive Adaptierung von Black-Box Softwarekomponenten ermöglicht. Durch die Generierung eines Adapters kann eine Softwarekomponente in einem System installiert werden, obwohl die Schnittstellen nicht exakt zusammenpassen. Dies erhöht die Wiederverwendbarkeit der Softwarekomponente. Insgesamt soll das Konzept sich über den Entwicklungsprozess von der modellbasierten Spezifikation bis hin zum lauffähigen Adapter auf einer Zielplattform erstrecken und dabei die in diesem Kapitel definierten Anforderungen erfüllen.

Kapitel 3

Stand der Technik

Das vorhergehende Kapitel hat den Bedarf einer minimal-invasiven Lösung zur Adaptierung von Softwarekomponenten im Fahrzeug gezeigt. Ein wesentliches Ziel dieser Arbeit ist es, eine praxisorientierte durchgängige Lösung von der Spezifikation der Softwarekomponenten bis hin zum lauffähigen Adapter zu definieren. Deshalb betrachtet dieses Kapitel den Entwicklungszyklus von der modellbasierten Beschreibung bis hin zur Generierung von lauffähigem Binärcode.

Abschnitt 3.1 stellt in der Praxis eingesetzte Methoden zur komponentenbasierten Softwareentwicklung im Fahrzeug vor. Dabei wird der Bereich Infotainment bewusst getrennt vom restlichen Fahrzeug beschrieben, da sich die Anforderungen und eingesetzten Methoden in diesen Bereichen deutlich voneinander unterscheiden. Abschnitt 3.2 stellt Methoden zur Modellierung von Softwareschnittstellen vor. Hier steht vor Allem die Beschreibung des Kontrollflusses Softwarekomponenten im Vordergrund. Abschnitt 3.3 stellt existierende Ansätze zur Adaptierung von Softwareschnittstellen vor und Abschnitt 3.4 zeigt Architekturen zur Umsetzung von Adaptern. Das Kapitel schließt mit einer Zusammenfassung.

Inhaltsverzeichnis

3.1. Komponentenbasierte Softwareentwicklung im Fahrzeug

Das Fahrzeug, sowie die E/E-Architektur[1] werden oft in verschiedene Domänen, wie Antrieb, Fahrerassistenz, Karosserie und Infotainment[2] unterteilt [RKK13]. Diese Klassifizierung ergibt sich zum einen aus den tatsächlichen Funktionen, und zum anderen aus Sicht der Anforderungen an die Hard- und Software solcher Funktionen. Softwarefunktionen im Bereich Karosserie, wie beispielsweise Fensterheber oder Innenraumbeleuchtung, laufen auf kleineren Steuergeräten mit leichtgewichtigen Betriebssystemen [KD14] und sind mit günstigen Eindraht-Bussystemen vernetzt. Im Gegensatz dazu handelt es sich bei Funktionen im Fahrerassistenz- und Antriebsbereich um komplex vernetzte Funktionen mit hohen Echtzeitanforderungen. Diese laufen auf leistungsfähigen Steuergeräten, verwenden breitbandigere Busse und werden zum Großteil mithilfe von AUTOSAR [GbR14a] umgesetzt. Die Funktionen der Infotainment-Domäne hingegen sind charakterisiert durch eine sehr leistungsfähige Hardware, große Datenmengen und geringe Echtzeitanforderungen. Diese domänenspezifischen Anforderungen haben direkte Auswirkungen auf die verwendete Softwarearchitektur und somit auch auf die Konzeption einer Lösung zur Adaptierung von Softwareschnittstellen. Dieser Abschnitt beschreibt exemplarisch die komponentenorientierte Entwicklung mit AUTOSAR und im Infotainment. Damit wird der Großteil der komplexen Softwarefunktionen im Fahrzeug abgedeckt. Besonderer Fokus liegt dabei auf Konzepten zur Wiederverwendung von Softwarekomponenten.

3.1.1. Komponentenbasierte Softwareentwicklung mit AUTOSAR

Das Ziel der AUTomotive Open System ARchitecture (AUTOSAR) ist es, einen Standard für die Entwicklung eingebetteter Software im Fahrzeug zu definieren. Vor Allem soll AUTOSAR die Wiederverwendung von Soft- und Hardwarekomponenten zwischen verschiedenen Plattformen, Fahrzeugherstellern und Zulieferern ermöglichen [Fen+06]. Dabei bietet die AUTOSAR-Architektur [GbR14a] eine klare Trennung zwischen steuergeräteunabhängiger Anwendungssoftware und steuergeräteabhängiger Basissoftware.

[1] Elektrik/Elektronik-Architektur
[2] Kunstwort aus Information und Entertainment

Abbildung 3.1 zeigt die Schichtenarchitektur von AUTOSAR. Das Kernkonzept zur Wiederverwendung von Software ist in AUTOSAR die *Softwarekomponente*. Softwarekomponenten sind in der Applikationsschicht logisch miteinander über ihre Ports verbunden und realisieren somit das in Abschnitt 2.1 beschriebene abstrakte Funktionsnetz auf logischer Ebene. Sie stellen hardwareunabhängige atomare Einheiten dar, welche auf die Steuergeräte verteilt werden. Kommunikationsdienste werden in der AUTOSAR Architektur durch das *Runtime Environment (RTE)* bereitgestellt. Es bietet mit dem abstrakten *Virtual Function Bus (VFB)* ein einheitliches Medium zur Kommunikation. Das RTE ist dabei die Steuergerätespezifische Implementierung des VFB. Sämtliche Kommunikation zwischen Softwarekomponenten findet über das RTE statt, egal ob sich die Komponenten auf demselben, oder auf verschiedenen Steuergeräten befinden. Die Schichten unterhalb des RTE stellen die Basis-Software dar. Diese beinhalten zum Beispiel die Implementierung der Treiber für eine konkrete Bustechnologie. Die gesamte AUTOSAR Infrastruktur basiert dabei auf standardisierten Schnittstellen [Hei+08].

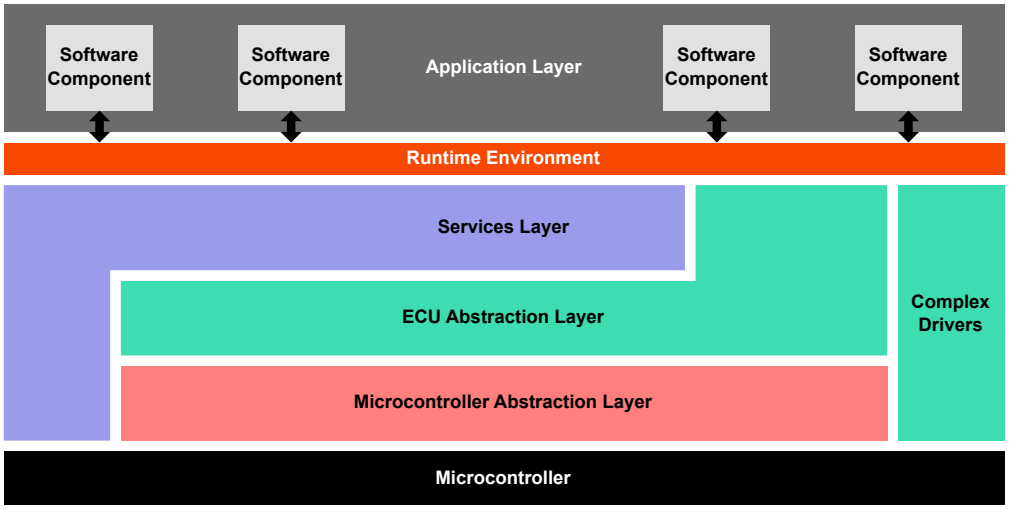


Abbildung 3.1.: Schichtenarchitektur von AUTOSAR [GbR14a].

Nicht nur die Architektur, sondern auch die Entwicklungsmethodik wird durch AUTOSAR standardisiert [GbR14b]. Abbildung 3.2 zeigt die Schritte von einer abstrakten Systembeschreibung links bis hin zum lauffähigen Binärcode für das Steuergerät rechts. Die für die Entwicklung notwendigen Modelle (in der Grafik Rechtecke) sind ebenfalls standardisiert. Die Schnittstelle einer Softwarekomponente wird in der *Software Component Description* beschrieben. Aus diesem Modell wird die Schnittstelle (API) der Softwarekomponente in einer Programmiersprache

generiert (oberer Prozess in Abbildung 3.2). Diese API dient als Grundgerüst für eine Implementierung der Softwarekomponenten. Zudem wird die Software Component Description als Modell für weitere Generierungsschritte benötigt (unterer Prozess in Abbildung 3.2). Der Prozess ist geprägt von verschiedenen halbautomatisierten Konfigurations- und Generierungsschritten. Eine wichtige Rolle spielt dabei der Integrator, der die abstrakten Schnittstellen einer Softwarekomponente in den Kontext des realen Systems bringt. Der Integrator definiert das Deployment einer Softwarekomponente und hat zusätzlich die Möglichkeit die Schnittstelle einer Softwarekomponente zu adaptieren, sodass die Anforderungen des Gesamtsystems erfüllt werden. Diese Adaptierung erfolgt auf syntaktischer Ebene und kann zum Beispiel die Typänderung eines Parameters sein.

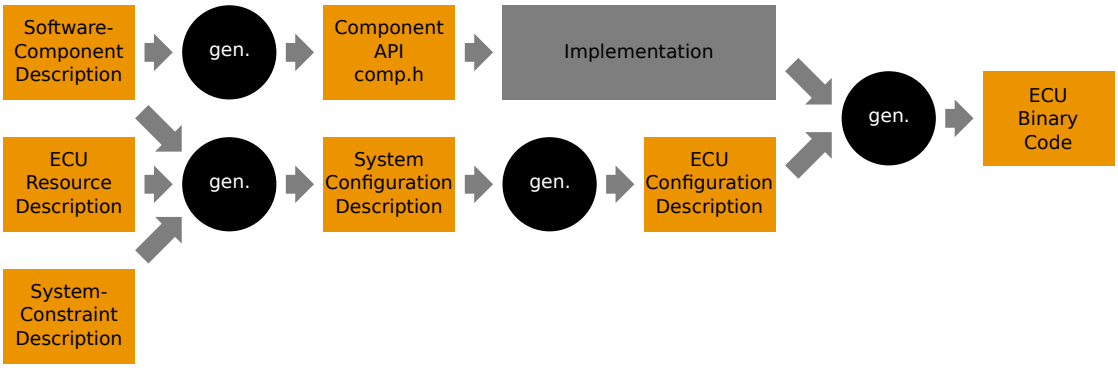


Abbildung 3.2.: Überblick über die AUTOSAR Methodik. Detaillierte Methodik in [GbR14b].

Für die Basissoftware ist in AUTOSAR das Postbuild-Verfahren definiert, um das Verhalten einiger Basissoftware-Module auch nach dem Übersetzen anpassen zu können [Zee12]. So können diese während der Laufzeit neu konfiguriert werden. Dieser, auch als *Parametrierung* bekannte Mechanismus, existiert für Softwarekomponenten der Applikationsebene nicht. Deshalb hat die Veränderung einer solchen Softwarekomponente das neue Übersetzen der gesamten Steuergeräte-Software zur Folge [Zee12].

Bewertung hinsichtlich der Aufgabenstellung

Die nachfolgende Bewertung bezieht sich auf die in AUTOSAR verwendeten Modelle und die AUTOSAR Software-Architektur:

Modellierung: Die AUTOSAR Methodik geht davon aus, dass die Ports der Softwarekomponenten auf logischer Ebene so gestaltet wurden, dass sie zueinander kompatibel sind. Deshalb ist es auch nicht das Ziel des Standards, eine umfangreiche Adaptierung von Softwarekomponenten auf logischer Ebene zu bieten. Die Werkzeuge bieten jedoch die Möglichkeit, Schnittstellen für die technische Ebene zu adaptieren oder durch vordefinierte Parameter anzupassen. Diese Konzepte konzentrieren sich allerdings auf rein syntaktische Schnittstelleneigenschaften. Dies ist dem Umstand geschuldet, dass AUTOSAR Softwarekomponenten meist einen sehr einfachen Kontrollfluss implementieren. Deshalb bietet auch die verwendete Modellierungsmethodik nur eine zu diesen Mustern passende rudimentäre Spezifikation des Komponentenverhaltens.

Software-Architektur: Bis zum jetzigen Zeitpunkt bietet AUTOSAR keine Konzepte, Softwarekomponenten oder Teile davon zur Laufzeit nachzuladen oder zu ersetzen. Eine minimal-invasive Adaptierung von Softwareschnittstellen von AUTOSAR Softwarekomponenten ist somit nur umständlich über das Überschreiben von Adressen im Binärcode [Zee12] zu lösen. Deshalb stellt AUTOSAR für diese Arbeit keine gute Basis zur Evaluierung der Konzepte zur Verfügung. Dennoch ist es für uns wichtig, dass sich die Konzepte auf Modellebene auch mit AUTOSAR in Einklang bringen lassen.

3.1.2. Komponentenbasierte Softwareentwicklung im Infotainment

Die Infotaiment-Domäne im Fahrzeug orientiert sich sehr stark an den technischen Entwicklungen im Bereich der Verbraucher-/Unterhaltungselektronik. Kunden erwarten von der Infotainment-Lösung im Fahrzeug die gleiche Flexibilität und Aktualität wie bei ihrem Mobiltelefon [MTV09]. Deshalb ist in den letzten Jahren im Bereich Infotainment im Vergleich zu den restlichen Fahrzeugdomänen eine erheblich höhere Anzahl an neuen Funktionen mit sehr kurzen Innovationszyklen zu beobachten. Durch die kurzen Innovationszyklen spielt hier die Eigenschaft zur Aktualisierung von Software durch den Kunden eine große Rolle. Deshalb folgt auch die Softwareentwicklung im Bereich Infotainment im Gegensatz zu AUTOSAR einem etwas flexibleren Konzept, welches die Evolution und den Austausch von Softwarekomponenten unterstützt. Moderne Head-Units[3] implementieren deshalb Mechanismen zur Aktualisierung von Software während des

[3]Gängige Bezeichnung für das zentrale Infotainment Steuergerät

Betriebs. Im Bereich Infotainment kommen zunehmend breit eingesetzte Betriebssysteme wie Linux, Windows, QNX oder Android zum Einsatz. Der wesentliche Unterschied zur Unterhaltungselektronik ist die hohe Vernetzung der Head-Unit zu den restlichen Steuergeräten und zur Sensorik und Aktorik im Fahrzeug. Mit der GENIVI Allianz [Gen14b] gibt es Bestrebungen, einen offenen Standard für ein Linux-basiertes Infotainment-Betriebssystem zu schaffen, der sich genau auf diese Fahrzeugspezifika fokussiert. Ähnlich wie AUTOSAR, ist auch dies eine Maßnahme, den Austausch und die Wiederverwendung der Software zu erleichtern und nicht-wettbewerbsdifferenzierende Funktionen herstellerübergreifend zu realisieren.

Ähnlich wie bei AUTOSAR werden auch im Infotainment syntaktische Schnittstellen in einer IDL beschrieben. Für MOST Funktionsblöcke ist das der Funktionskatalog. Für Softwarekomponenten im Bereich GENIVI Linux und der dort verwendeten D-BUS Middleware wird Franca IDL [Bir14] verwendet. Zusätzlich zu dieser syntaktischen Beschreibung, werden UML Sequenzdiagramme [Pra+12] für die Modellierung von einzelnen Abläufen, und UML Zustandsautomaten (siehe Abschnitt 3.2.2) zur Modellierung komplexeren Verhaltens verwendet.

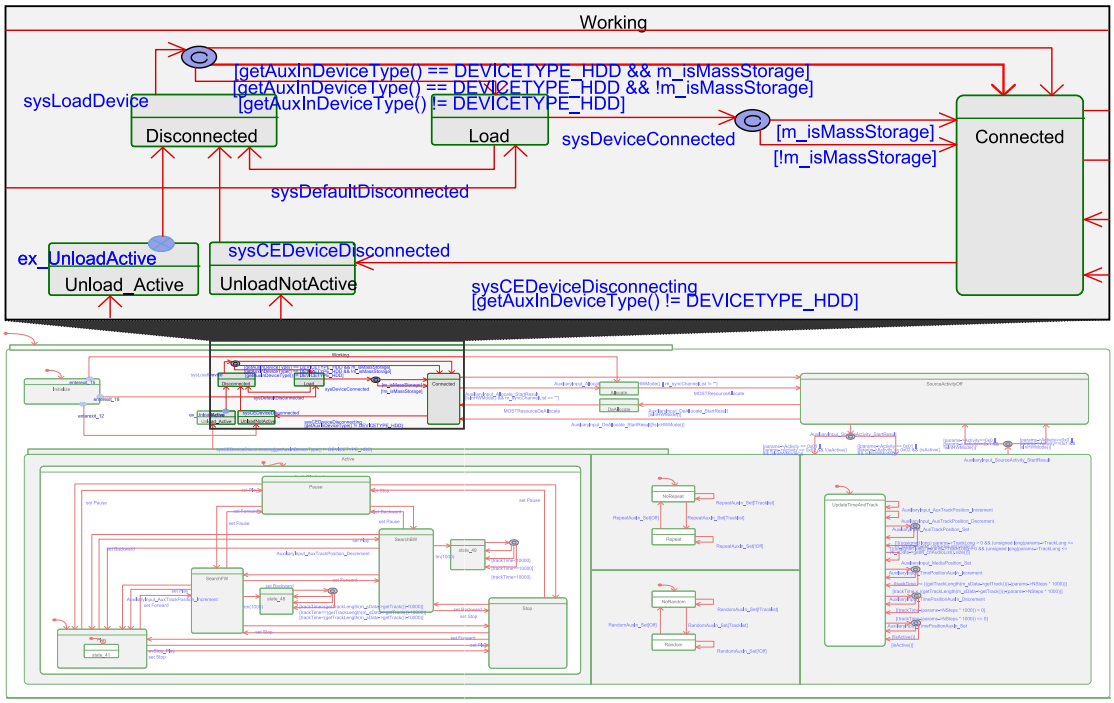


Abbildung 3.3.: Aux-In Softwarekomponente als UML Zustandsdiagramm mit vergrößertem Ausschnitt oben.

Abbildung 3.3 zeigt ein Beispiel für einen Zustandsautomaten, der das Verhalten des Aux-In Funktionsblockes im MOST Verbund spezifiziert. Diese Softwarekomponente regelt die Wiedergabe von Musiktiteln von einem externen Speichermedium wie beispielsweise einem USB-Stick oder einem angeschlossenen Mobiltelefon. Jede Transition besitzt einen Trigger, welcher einem Funktionsaufruf auf der Kommunikationsmiddleware entspricht. Zusätzlich können Bedingungen für einen Zustandsübergang eingegeben werden (zum Beispiel *getAuxInDeviceType() != DEVICETYPE_HDD*). Der vergrößerte Ausschnitt im oberen Bereich der Abbildung zeigt den Verbindungsauf- und -abbau für ein externes Speichermedium. Es wird zum Beispiel zwischen einem Massenspeicher (m_isMassStorage) und einem mp3-Player (!m_isMassStorage) unterschieden. Sobald der Verbindungsaufbau abgeschlossen ist, kann das Gerät zur Musikwiedergabe verwendet werden. Der Zweck für die Erstellung des Modells des Aux-In Funktionsblockes war zum Einen eine frühzeitige Simulation des Verhaltens und zum Anderen die Ableitung von Testfällen. Für die Simulation der Komponente muss der Zustandsautomat in den Bedingungen und Aktionen die Abläufe so genau spezifizieren, dass das Modell ausgeführt werden kann. Der Modellierer hat hier die Möglichkeit, Programmcode einzugeben und so das gewünschte Verhalten bei Ausführung zu erzeugen. Listing 3.1 zeigt ein Beispiel für eine im Modell aus Abbildung 3.3 enthaltene Aktion. Sie definiert die beiden Nachrichten *AuxiliaryInput_AuxDeviceInfo_Status* und *AuxiliaryInput_DeckStatusAuxIn_Status* und deren Parameterbelegung.

```
1  loadDevice();
2
3  CREATEEVENT(AuxiliaryInput_AuxDeviceInfo_Status);
4  BEGIN_PARAMETERS_FOR_COMPARISON;
5  e->pList->DeviceClass=0x02;
6  e->pList->DeviceType=0x0;
7  END_PARAMETERS_FOR_COMPARISON;
8  e->pList->DeviceName.setValue("iPodSerial");
9  e->pList->DeviceName.setM_coding(MOSTString::UTF_16);
10 e->pList->DeviceNum=0x0;
11 SET_PROPERTY();
12
13 m_sData->setM_deckStatus(DS_STOP);
14
15 CREATEEVENT(AuxiliaryInput_DeckStatusAuxIn_Status);
16 BEGIN_PARAMETERS_FOR_COMPARISON;
17 e->pList->DeckStatusAuxIn=DS_STOP;
18 END_PARAMETERS_FOR_COMPARISON;
19 SET_PROPERTY();
```

Listing 3.1: Für eine Transition spezifizierte Aktion, welche sich aus Programmteilen in Programmiersprache C zusammensetzt.

Für die Ausführung des Modells existieren Werkzeuge, die eine nahtlose Integration in die fahrzeugspezifische Middleware erlauben. Zum Beispiel kann ein

UML Modellierungswerkzeug wie Rhapsody [IBM14] mithilfe des MODENA Frameworks [Mat14] an die MOST Middleware angebunden werden und erlaubt somit die Kommunikation mit den realen Softwarekomponenten im Fahrzeug. Was die Spezifikation des Verhaltens betrifft, sind dieser Modellierung kaum Grenzen gesetzt. Jedoch erfordert die Erstellung eines derartigen Modells nicht nur Expertenwissen, was das Verhalten der Softwarekomponenten betrifft, sondern auch fundierte Programmierkenntnisse.

Bewertung hinsichtlich der Aufgabenstellung

Die nachfolgende Bewertung bezieht sich auf die im Infotainment verwendeten Modelle und die vorherrschende Software-Architektur:

Modellierung: Für die Modellierung werden im Infotainment hauptsächlich UML Diagramme verwendet. Durch die Erweiterung mit Programmcode sind diese sehr mächtig und erlauben eine detaillierte Spezifikation des Verhaltens bis hin zur Simulation ganzer Komponenten. Zudem existieren fahrzeugspezifische Erweiterungen, die den direkten Durchgriff vom Modell in die Middleware des Zielsystems erlauben. Die Erstellung solcher Modelle setzt gute Programmier- und Systemkenntnisse voraus und ist deshalb nur Experten vorbehalten. Dadurch, dass das Modell beliebigen Programmcode enthalten kann, wird die Analyse in Bezug auf die Korrektheit des Kontrollflusses wesentlich erschwert.

Software-Architektur: Im Gegensatz zu AUTOSAR sind die im Infotainment Umfeld verwendeten Betriebssysteme für eine einfache Aktualisierung von Software vorbereitet. Deshalb ist auch die Adaptierung von Black-Box Softwarekomponenten einfacher möglich. Zahlreiche IDL's erlauben auch die Definition von Schnittstellenversionen, jedoch bietet bis zu diesem Zeitpunkt keine der Middlewaretechnologien einen Mechanismus zur Adaptierung von Schnittstellen unterschiedlicher Version.

3.1.3. Fazit

Dieser Abschnitt stellte zwei repräsentative Lösungen zur Umsetzung komponentenorientierter Fahrzeugsoftware vor und bewertete diese in Bezug auf die verwendete Modellierungsmethodik und Software-Architektur. AUTOSAR bietet ein durchgängiges Konzept zur Spezifikation und Generierung lauffähiger Steuergerätesoftware. Die Architektur unterstützt allerdings nur minimale Anpassung der

definierten Schnittstellen und ist deshalb nicht für eine umfangreiche Adaptierung von Black-Box Softwarekomponenten geeignet. Zudem erlaubt die eingesetzte Modellierungsmethodik nur eine rudimentäre Spezifikation des Kontrollflusses und ist deshalb nicht als Basis für eine umfangreiche Verhaltensadaptierung verwendbar.

Im Bereich Infotainment hingegen ist der Kontrollfluss der Softwarekomponenten deutlich komplexer. Deshalb werden hier vermehrt Zustandsautomaten zur Spezifikation des Verhaltens eingesetzt. Somit bietet die vorherrschende Modellierungsmethodik die für diese Arbeit notwendige Grundlage um auch komplexere Szenarien für inkompatibles Verhalten zu beschreiben. Im Gegensatz zu AUTOSAR sind die im Infotainment eingesetzten Betriebssysteme so gestaltet, dass sie das Nachladen von Software zur Laufzeit und somit auch den Einsatz von dynamisch geladenen Adaptern ermöglichen. Besonders GENIVI Linux eignet sich mit seinen offenen Schnittstellen gut als Grundlage für diese Arbeit, denn durch den quelloffenen Standard können Erweiterungen leicht umgesetzt und getestet werden. Es kann deshalb festgestellt werden, dass GENIVI Linux sowohl im Bereich der Schnittstellen-Modellierung als auch im Bereich der Software-Architektur eine gute Grundlage für diese Arbeit darstellt. Bisher existiert jedoch keine Lösung für eine durchgängige Generierung von Adaptern ausgehend von einem Modell bis hin zum lauffähigen Binärcode.

3.2. Modellierung von Softwareschnittstellen

Grundlage der meisten Lösungen zur Schnittstellenadaptierung ist eine modellbasierte Beschreibung der Softwarekomponenten auf den unteren in Abschnitt 2.2 beschriebenen Ebenen. Oft wird eine Schnittstelle nicht nur durch ein *syntaktisches Modell* definiert, sondern auch durch ein zusätzliches *Verhaltensmodell*[4]. Dabei definiert dieser Schnittstellenautomat *Sequenz- und Zeitbedingungen* für den Aufruf der syntaktischen Elemente der Schnittstelle und beantwortet somit die Frage, wie eine Schnittstelle benutzt werden muss. In der industriellen Praxis sind solche Verhaltensmodelle oft nicht explizit modelliert, sondern in einer textuellen Spezifikation definiert oder nur implizit durch den Quelltext definiert. Gegenüber der textuellen Spezifikation erlaubt die explizite Beschreibung des gewünschten Verhaltens in einem formalen Modell die Analyse von bestimmten Systemeigenschaften (z.B. Kompatibilität von Schnittstellen [AG97; DH01b; Bec08]) bereits auf Modellebene. Gerade im Bereich der Schnittstellenautomaten gibt es verschiedene Formalismen mit zum Teil sehr unterschiedlicher Notation und Semantik.

[4]Häufig verwendete Begriffe: Schnittstellenverhalten, Behavioral IDL (BIDL), Protokoll, Schnittstellenautomat. In dieser Arbeit wird der Begriff Schnittstellenautomat verwendet.

Einige davon werden in den nächsten Abschnitten vorgestellt und in Bezug auf die Aufgabenstellung aus Abschnitt 2.5 bewertet.

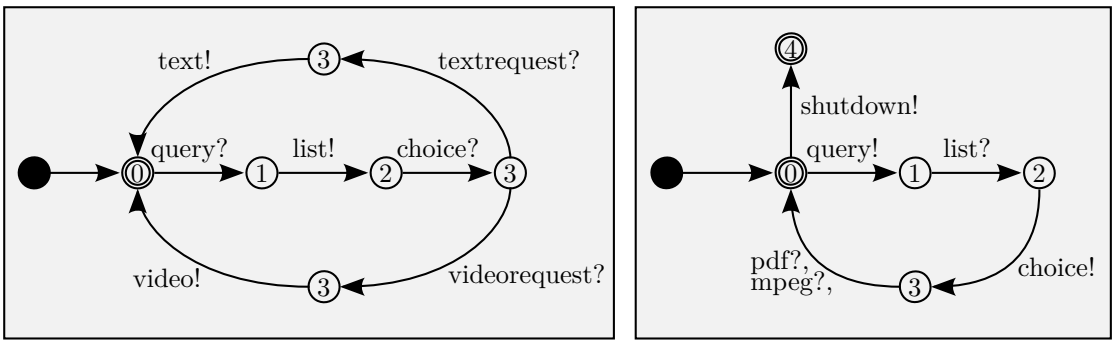
3.2.1. Schnittstellenmodellierung mit endlichen Automaten

Die Verwendung von endlichen Automaten ist eine seit längerem etablierte Methode zur Beschreibung von Kommunikationsprotokollen (erste Arbeiten in [BZ83; GMY84; LT89]). Meist werden für die Modellierung akzeptierende endliche Automaten verwendet. Diese akzeptieren einen Nachrichtenstrom als Eingabe und signalisieren den aktuellen Zustand der Kommunikation. In diesem Abschnitt werden exemplarische Beispiele solcher Automaten und deren Erweiterungen in Bezug auf die Modellierung von Zeitverhalten gezeigt.

Abbildung 3.4 zeigt ein Beispiel zweier Zustandsautomaten. Aus Sicht der in einem Modell beschriebenen Komponente erhalten eingehende Nachrichten das Suffix ? und ausgehende das Suffix !. Das bedeutet, dass in einer Client-Server Beziehung diese Symbole für den Client genau komplementär zu denen des Servers notiert werden müssen. Das Beispiel zeigt die Zustandsautomaten der zwei inkompatiblen Komponenten *ROOM* und *PDA*. ROOM ist der Server und PDA der Client. Da sich diese Automaten auf komplementäre Nachrichten synchronisieren, muss zu dem Zeitpunkt, in dem PDA die Nachricht *query!* sendet, im aktuellen Zustand von ROOM eine entsprechende Kante mit der Nachricht *query?* für den Empfang der Nachricht existieren. Das bedeutet, dass PDA die Nachricht *query* versendet und ROOM auch diese Nachricht erwartet. Nach dem Austausch der Nachricht *choice* befinden sich die Komponenten in einer Verklemmung, denn ROOM erwartet in Zustand 3 entweder die Nachricht *textrequest* oder *videorequest* und PDA erwartet in Zustand 3 *pdf* oder *mpg*. Ein Adapter könnte in diesem Zustand eingreifen und zwischen den beiden inkompatiblen Komponenten vermitteln (siehe Adapter aus 3.4a).

Diese Art der Modellierung beschreibt rein die Sequenzbedingungen für Nachrichten und nicht den Zeitraum, in dem eine Nachricht gesendet oder empfangen werden sollte. Ein erweiterter Modellierungsansatz mit der Berücksichtigung von Zeitbedingungen wurde mit Timed Automata [AD94] definiert. Timed Automata erweitern endliche Automaten mit Zeitvariablen und Zeitbedingungen, welche auf die Transitionen annotiert werden. Der Wert der Zeitvariablen nimmt mit Fortschreiten der Zeit zu. Dabei darf eine Transition nur ausgeführt werden, falls die Zeitbedingung wahr ist.

Ein ähnlicher Formalismus wurde auf Basis der I/O Automata [LT89] mit *(Timed) Interface Automata* definiert [DH01b; DHS02]. Die Modellierung der Zeit



(a) Zustandsautomat für die Komponente ROOM.

(b) Zustandsautomat für die Komponente PDA.

Abbildung 3.4.: Zustandsautomaten der beiden inkompatiblen Komponenten ROOM und PDA (aus [CPS08]).

ähnelt der der Timed Automata. Wesentliche Unterschiede sind die Definition von Kompatibilität und Annahmen über die Umgebung der Komponenten. Demnach verwenden (Timed) Interface Automata einen optimistischen Kompatibilitätsbegriff. Das bedeutet, dass zwei offene Systeme[5], welche zusammen wieder ein offenes System ergeben, kompatibel sind, falls die Eingaben der Nachbarsysteme diese nie in einen inkompatiblen Zustand bringen. Die Überprüfung der Kompatibilität erfolgt durch einen spieltheoretischen Ansatz [DH01a], bei dem ein Spieler *Input* gegen einen Spieler *Output* spielt. Die Züge des Spielers Input sind die eingehenden Nachrichten der Komponente und die des Spielers Output die ausgehenden. Abbildung 3.5 zeigt einen Timed Interface Automaton, der einen zyklischen Ablauf beschreibt. In der grafischen Notation wird sowohl der Automat, als auch die Schnittstelle selbst dargestellt. Der Formalismus arbeitet mit globalen Zeitvariablen (im Beispiel x). Die im Zustand geltenden Invarianten (z.B. $x \leq 10$) beschreiben die in diesem Zustand gültige Belegung von Zeitvariablen und geben somit die obere Grenze für das Fortschreiten der Zeit in diesem Zustand an. Es werden Eingabe-Invarianten (Inv^I) für den Spieler Input von Ausgabe-Invarianten (Inv^O) für den Spieler Output unterschieden. Zusätzlich definieren die Annotationen der Transitionen die Zeitbedingung für die jeweilige Nachricht. Im Beispiel kann der Spieler Output (O) im Zustand p_0 so lange die Zeit vergehen lassen, bis die Bedingung $x = 10$ erfüllt ist. Dann sollte er die Nachricht *js!* senden. Gleichzeitig wird die Zeitvariable x auf 0 gesetzt. Der Spieler Input (I) sollte nach mindestens 6 jedoch maximal 8 Sekunden die Nachricht *if?* senden.

[5]Ein System mit Schnittstellen zu weiteren Systemen

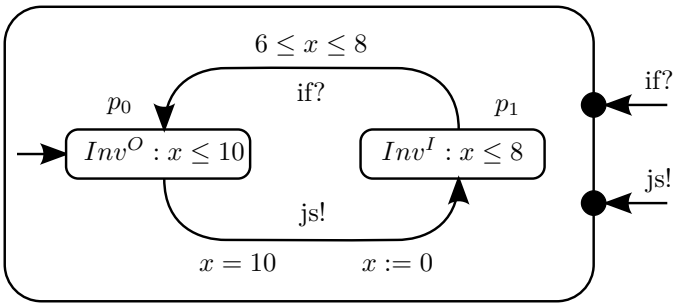


Abbildung 3.5.: Timed Interface Automaton für eine Komponente [DHS02].

Grundsätzlich werden hier flache Zustandsautomaten vorgestellt. Für die übersichtliche grafische Darstellung von großen Automaten können die von Harel [Har87] definierten visuellen Notationen für Hierarchie oder Parallelität verwendet werden, denn sie lassen sich wieder in einen flachen Zustandsautomaten transformieren.

Bewertung hinsichtlich der Aufgabenstellung

Neben diesen hier genannten Modellierungsmethoden für Schnittstellen mit endlichen Automaten, wurden etliche Erweiterungen und Dialekte zusammen mit Methoden zur formalen Analyse definiert. Das bringt den Vorteil mit sich, dass für derartige Modelle bereits Werkzeuge existieren, die bestimmte Eigenschaften der Automaten (z.B. Deadlock-Freiheit oder Kompatibilität) formal überprüfen können.

In Bezug auf die Aufgabestellung aus Abschnitt 2.5 eignen sich endliche Automaten sehr gut zur Beschreibung der geforderten Ablauf- und Zeitbedingungen für Schnittstellen. Besonders für Echtzeitsysteme stellt die Modellierung von Zeitbedingungen eine notwendige Erweiterung dar. Diese Art der Modellierung ist in der Wissenschaft bereits gründlich erforscht, allerdings noch nicht praktisch in der Industrie angewandt. Ein Grund dafür mag die Modellierung von Zeit mithilfe von Zeitvariablen sein. Denn bei größeren Automaten ist es für den Menschen nur noch schwer zu überblicken, welche Teile des Automaten Einfluss auf Zeitvariablen nehmen, oder welche Werte eine Zeitvariable in einem bestimmten Zustand annehmen kann.

Ein weiterer Nachteil ergibt sich aus dem Umstand, dass bei den vorgestellten Arbeiten theoretische Aspekte im Vordergrund standen. Zum Beispiel reicht es für die theoretische Betrachtung aus, Nachrichten mithilfe eines kurzen Strings zu beschreiben. Für die praktische Anwendung in einem realen System ist es notwendig, zusätzlich zum Methodennamen auch Informationen zu den Parametern

anzugeben. Deshalb ist eine Erweiterung notwendig, welche es ermöglicht, eine Nachricht so feingranular zu spezifizieren, dass mit diesen Informationen direkt ein Methodenaufruf durchgeführt werden kann.

3.2.2. Schnittstellenmodellierung mit UML

Die Unified Modelling Language (UML) [OMG10] ist ein herstellerneutraler Standard zur Modellierung, Dokumentation, Spezifikation und Visualisierung komplexer Softwaresysteme, unabhängig von deren Fach- und Realsiierungsgebiet [RQZ07]. Dabei umfasst die UML sowohl die detaillierte Definition von Elementen und deren Zusammenhängen, als auch die Notation der Elemente in den Diagrammen. Heute ist die UML die verbreitetste Notation, um Softwaresysteme zu analysieren und zu entwerfen [RQZ07].

Die UML unterteilt ihre Diagramme grundsätzlich in *Strukturdiagramme* und *Verhaltensdiagramme*. Zusätzlich zu den Diagrammen definiert die UML die Object Constraint Language (OCL) zur Spezifikation von Zusammenhängen, die mit den Diagrammelementen nur sehr umständlich oder gar nicht beschrieben werden könnten. Ein weiterer Aspekt ist die Möglichkeit der Erweiterung von UML durch *UML Profile*. Diese erlauben es, eigene Modellierungssprachen auf Basis der UML zu definieren. UML wird heute nicht nur als rein grafische Spezifikation verwendet, sondern auch, um Artefakte für die Softwareentwicklung und den Test zu generieren (Für eine Übersicht zur Testfallgenerierung wird auf [SK13] verwiesen).

Bei den Strukturdiagrammen werden in dieser Arbeit *Komponentendiagramme* zur Darstellung von Komponenten und deren Schnittstellen verwendet. Im Bereich der Verhaltensdiagramme sind für uns vor Allem *Zustands-* und das *Sequenzdiagramme* relevant. Sequenzdiagramme werden in der Praxis häufig eingesetzt, um ein Szenario der Benutzung einer Schnittstelle zu beschreiben. Sie dienen als Spezifikation, oder als Eingabe für ein Testautomatisierungswerkzeug. Die UML Zustandsdiagramme basieren im Wesentlichen auf den von Harel [Har87] definierten „statecharts“. Sie werden zum Beispiel verwendet um das Verhalten gesamter Komponenten zu beschreiben. Wie in weiteren, im Rahmen dieser Arbeit entstandenen Beiträgen [Pra+12; Dra+13] gezeigt, eignet sich eine mit UML Zustandsdiagrammen modellierte Verhaltensspezifikation zur Absicherung des Kontrollflusses an einer Softwareschnittstelle.

Für die Modellierung von UML Diagrammen existieren mittlerweile eine Vielzahl kommerzieller und freier Lösungen. Diese unterstützen oft auch Code-Generierung aus den Modellen oder die Definition von UML Profilen.

Bewertung hinsichtlich der Aufgabenstellung

Die UML erfreut sich als Standard für die Modellierung von Softwaresystemen einer breiten Akzeptanz. Da die verschiedenen Diagramme der UML ein sehr breites Spektrum an Anwendungsgebieten abdecken, sind sie sehr mächtig, allerdings auch sehr komplex. Um verschiedenste Einsatzszenarien zu ermöglichen, sind manche Elemente bewusst ungenau definiert. Deshalb wird es oft dem Modellierer oder dem Toolentwickler überlassen, welche Bedeutung ein Modellierungselement letztendlich im Softwaresystem hat. Zum Beispiel lassen manche Werkzeuge die Eingabe von beliebigen Programmcode zu. So kann der Zustandsautomat seine positiven Eigenschaften in Bezug auf den automatisierten Nachweis bestimmter Eigenschaften verlieren. Eine dieser Eigenschaften ist zum Beispiel der Nachweis der Kompatibilität zwischen zwei Komponenten.

In Bezug auf die Aufgabenstellung kann festgehalten werden, dass die Diagramme der UML geeignete Methoden zur Spezifikation der syntaktischen und der Verhaltensebene von Schnittstellen bieten. Die Erstellung eines UML Profils erlaubt es zudem die Modellierung anzupassen oder Methoden zur Modellierung von Zeitverhalten hinzuzufügen. Ein Ziel dieser Arbeit ist es, die entwickelten Modellierungskonzepte mit der UML in Einklang zu bringen. Denn der Vorteil einer mit UML konformen Modellierungsmethode ist die Wiederverwendung von UML-Werkzeugen. Für diese Arbeit betrifft dies vor Allem Werkzeuge für die grafische Modellierung und Frameworks zur Ausführung von UML Verhaltensdiagrammen.

3.2.3. Fazit

Die vorherigen Abschnitte haben gezeigt, dass bereits zahlreiche Methoden zur Modellierung einer Schnittstelle, sowohl auf der Syntax- als auch auf Verhaltensebene existieren. Für die Modellierung des Verhaltens eignen sich Zustandsautomaten sehr gut. Die vorgestellten Ansätze reichen allerdings nicht tief genug, um die Ausführung des Automaten als Softwarekomponente in einem Infotainmentsystem zu erlauben. UML Zustandsautomaten, mit der Möglichkeit zur freien Programmierung von Triggern, Guards und Aktionen, würden hierfür eine Lösung bieten. Zum Beispiel kann eine Aktion so programmiert werden, dass sie einen Methodenaufruf mit korrekter Parametrierung durchführt. An dieser Stelle verliert das Modell jedoch die positiven Eigenschaften endlicher Automaten für das Model-Checking. Somit könnte die Korrektheit eines Adapters nur sehr aufwändig nachgewiesen werden. Grundsätzlich kann deshalb festgestellt werden, dass in den vorgestellten Ansätzen ein Bruch zwischen dem Modell des Automaten selbst und den syntaktischen Schnittstellen der Softwarekomponenten in Form von Methoden

und Parametern existiert. Deshalb ist es ein Ziel dieser Arbeit diese Lücke mit einem Modell zu schließen, welches auch auf der Ebene der Methodenaufrufe eine Kompatibilitätsaussage liefern kann.

3.3. Adaptierung von Softwareschnittstellen

Ein aus Softwarekomponenten bestehendes System sollte für eine maximale Wiederverwendung der Softwarekomponenten Mechanismen zur Adaptierung bereitstellen [NM95]. Besonders wenn Black-Box Komponenten involviert sind, deren Quelltext nicht vorliegt und somit nicht angepasst werden kann, ist Schnittstellenadaptierung eine gute Methode, Komponenten in verschiedenen Umgebungen mit unterschiedlichen Schnittstellen wiederverwenden zu können [CMP06; Aut+08]. Zusätzlich entsteht die Notwendigkeit der Adaptierung, wenn sich Softwarekomponenten unabhängig voneinander weiterentwickeln [Bec+06]. Da genau diese Eigenschaften auch in der Fahrzeugdomäne eine Wiederverwendung der Softwarekomponenten erschweren (siehe Abschnitt 2.4), wird auch in dieser Arbeit Adaptierung als die geeignetste Lösung zur Realisierung eines flexibleren Baukastens betrachtet.

Mit einer Adaptierung können unterschiedliche Ziele verfolgt werden. Zum Beispiel die Adaptierung von Software-Architekturen [Gui+11], die Selbst-Adaptierung von Systemen oder die Adaptierung von Softwareschnittstellen, welche hier betrachtet wird. Dafür wird in dieser Arbeit der allgemeine Begriff des *Software-Adapters* verwendet. Generell zielt Software-Adaptierung darauf ab, möglichst automatisiert, lauffähige Adapter zu generieren, welche Inkompatibilitäten zwischen Software-Schnittstellen kompensieren [CPS08; CMP06]. Eine erste Definition des Software-Adapters wird in [YS97] vorgestellt. In Kombination mit dem Kompatibilitätsbegriff aus Abschnitt 2.2 kann der Software-Adapter folgendermaßen definiert werden:

Definition 3.3.1. ***(Software-Adapter):*** *Ein* Software-Adapter *ist eine Softwareeinheit, welche die Interoperabilität funktional kompatibler, jedoch in ihrer Schnittstelle inkompatibler Softwarekomponenten, auf verschiedenen Ebenen der Kompatibilität herstellt.* (angelehnt an [YS97])

Meist werden Schnittstellen und das Verhalten des Adapters durch ein abstraktes Modell beschrieben, welches automatisiert generiert, oder manuell erstellt wird.

Wir sprechen deshalb von *modellbasierter Adaptierung*. Die Vorteile der modellbasierten Adaptierung sind, dass bereits auf Modellebene Aussagen über die Korrektheit des Adapters getroffen werden können und dass das Modell zur Code-Generierung verwendet werden kann.

In diesem Abschnitt werden vorhandene Ansätze zur modellbasierten Adaptierung von Software-Schnittstellen gezeigt. Die wissenschaftlichen Fragestellungen konzentrieren sich meist auf die Adaptierung des Verhaltens und die vollautomatische Erstellung eines Adaptermodells. Allerdings ist bereits die Generierung eines Adaptermodells für einfache Systeme mit abstrakten Nachrichten eine große Herausforderung. Sobald man reale Systeme mit Methoden und Parametern betrachtet, können diese durch bisherige Ansätze nicht automatisiert adaptiert werden. Bestimmte Adaptierungsszenarien, wie z.B. die Berechnung eines neuen Parameters auf Basis anderer Parameter, sind nur durch zusätzliche Informationen eines Experten zu lösen [MPS12]. Dieser Abschnitt konzentriert sich deshalb nicht auf eine vollautomatische Adaptergenerierung, sondern auf eine halbautomatische, bei der das Adaptermodell von einem Modellierer erstellt wird und ein Entwicklungswerkzeug diese Erstellung unterstützt. Die Klassifizierung von datenzentrierter Adaptierung, Adaptierung mit Mappings und der Adaptierung mit endlichen Automaten, ist keine aus der Literatur bekannte Klassifizierung, sondern wurde explizit für diese Arbeit erstellt.

3.3.1. Datenzentrierte Adaptierung

Das Problem der Adaptierung von inkompatiblen Schnittstellen wurde im Bereich der Datenbanken früh diskutiert. Zum Beispiel kann eine Änderung einer Client-Applikation eine Anpassung der Datenbank erfordern und diese Änderung kann wiederum eine Anpassung von anderen Client-Applikationen zur Folge haben. Die logische Evolution solcher Client-Server Architekturen wird bei Wiederhold [Wie95; Wie92] durch Mediator-Architekturen adressiert. In solchen Architekturen wird ein Mediator als zusätzliche Softwareschicht eingefügt und bricht die starre Kopplung zwischen Client und Server auf [Wie95]. Mediation soll in dem Kontext eine Datenbankschnittstelle, durch die Behandlung Repräsentations- und Abstraktionsproblemen, intelligent machen. Ein *Mediator* wird in dem Kontext als „ein Softwaremodul gesehen, welches kodiertes Wissen auswertet, um Informationen für eine höhere Ebene von Applikationen zu generieren“ [Wie92]. Mediatoren haben eine aktive Rolle. Sie enthalten Strukturen, um Daten zu transformieren, zu speichern und einem Client anzubieten. Die praktische Realisierung im Kontext von Datenbanken wird oft durch *Sichten* und *Objekt-Templates* realisiert. Diese

Techniken reorganisieren Daten passend für spezielle Nutzer oder Anwendungen [Wie92]. Die Mediatoren werden dabei von Experten mit dem notwendigen Domänenwissen erstellt. Ein Ziel bei dem Ansatz ist auch die Wiederverwendung der erstellten Mediatoren für verschiedene Client-Applikationen und Datenbanken.

Bewertung hinsichtlich der Aufgabenstellung

Im Falle der vorgestellten Datenadaptierung sind Mediatoren für Client-Applikationen nicht transparent. Das bedeutet, dass eine Applikation explizit mit dem Mediator kommuniziert. Falls nun, durch die angestrebte Wiederverwendung, mehrere Applikationen denselben Mediator verwenden, löst dieser Ansatz nicht die zentrale Problemstellung, dass bei Änderung einer Client-Schnittstelle auch der Mediator, und somit alle Clients, die diesen Mediator verwenden, angepasst werden müssen. Zusätzlich handelt es sich um eine rein syntaktische Adaptierung ohne die Berücksichtigung der Verhaltensebene.

3.3.2. Verhaltensadaptierung mithilfe von Mappings/Mustern

Bei der Adaptierung mit Mappings wird das Verhalten des Adapters als direkte Zuordnung von Eingaben zu Ausgaben realisiert. Ähnliche Klassen von *Mappings* werden oft als Muster (Pattern) bezeichnet, welche wiederkehrende Adaptierungsszenarien beschreiben. Beispiele sind das *Teilen* oder *Zusammenführen* von Nachrichten, Behandlung einer *zusätzlichen* oder *fehlenden Nachricht* oder die Behandlung von *falsch geordneten Nachrichten*. Tabelle 3.1 zeigt ein einfaches Beispiel für mögliche Muster. Zum Beispiel dient das Muster $a \mapsto b$ zum Umbenennen einer Nachricht. Der Adapter wird aufgrund dieser Spezifikation jede Nachricht a in eine Nachricht b transformieren, unabhängig in welchem Zustand sich Komponenten und Adapter befinden.

Brogi et al. [Bro+04] und Bracciali et al. [BBC05] verwenden Mappings zur Realisierung von Adaptern. Dort werden nicht nur Nachrichten, sondern auch Parameterwerte betrachtet. Das Konzept wird sehr abstrakt beschrieben, ohne einen konkreten Bezug zu einem Zielsystem oder einer Middlewaretechnologie. Für die visuelle Notation von Mustern bietet Dumas et al. [DSW06] zudem noch eine visuelle Notation mithilfe von UML Aktivitätsdiagrammen [OMG10].

Benatallah et al. [Ben+05] beschreiben einen auf Mustern basierenden Ansatz im Bereich der Webservices. Ein Vorteil bei der Betrachtung von Webservices ist die einheitliche logische und technische Schnittstelle mit WSDL als Schnittstel-

Beschreibung	**Muster**
Nachricht a kopieren	a $\mapsto$ a, a
Nachricht a löschen	a $\mapsto\perp$
Nachricht a in b transformieren	a $\mapsto$ b
Nachricht a in b c d aufteilen	a $\mapsto$ b, c, d
Nachrichten a b c in d verschmelzen	a, b, c $\mapsto$ d
Rekombiniere a b c nach d e f	a, b, c $\mapsto$ d, e, f

Tabelle 3.1.: Beispiel für die Definition von Mustern (aus [GMW08])

lenbeschreibungssprache und SOAP als Transportmechanismus. Deshalb betrachten Arbeiten in dem Bereich hautpsächich die logische Ebene. Benatallah et al. [Ben+05] ordnen jedem Muster ein *Template* zu, welches Informationen für die Code-Generierung enthält. Für die Modellierung eines Templates wird BPEL eingesetzt und für Datenmanipulationen *XQuery*. Komplexere Adaptierungen können durch die Komposition von Templates beschrieben werden. Das vorgestellte Konzept erlaubt die Beschreibung von beliebig komplexen Adaptern für Syntax und Verhalten. Die Arbeiten definieren nicht, wie die angepasste Version der BPEL genau aussieht oder wie die Code-Generierung funktioniert. Zusätzlich wird leider nicht beschrieben, wie in einer Kommunikation ein bestimmtes Muster erkannt und angewandt werden kann. Ein offensichtlicher Nachteil ist die starre Kopplung von Template und Code-Generator. Zu jedem neuen Muster muss ein separates Template mit entsprechendem Code-Generator erstellt werden.

Jiang et al. [Jia+11] bieten für den musterbasierten Ansatz ein formales Modell. Die Basis für das Modell sind Message Sequence Charts [ITU11], welche einen Teil des Verhaltens von Client und Server spezifizieren. Jiang et al. beschreiben, wie das Modell die Erkennung von Regeln ermöglicht, und wie für die erkannten Inkompatibilitäten ein Adapter für Webservices generiert werden kann.

Auch Gierds et al. [GMW08] verwenden Muster als ersten Schritt in ihrem Adaptierungsansatz. Diese Muster werden manuell definiert und anschließend verwendet, um den Adapter als Petri-Netz zu generieren (siehe Abschnitt 3.3.4).

Bewertung hinsichtlich der Aufgabenstellung

Die Definition von Mustern eignet sich besonders als einfaches Mittel für die Spezifikation von häufigen Adaptierungsfällen. Zum Beispiel kann für den sehr häufigen Fall der Umbenennung einer Nachricht ein entsprechendes Muster definiert werden. Ein weiterer Vorteil ist, dass Muster im Normalfall zustandsunabhängig gelten und performant implementiert werden können. Ein wesentlicher Nachteil von Mustern

ist, dass sie nicht sehr flexibel einsetzbar sind, denn sie bieten nur für die spezifizierten Szenarien eine Adaptierung. Da sich Muster auch manchmal überlappen, ist es zudem schwer zu entscheiden, welches Muster am Besten für eine konkrete Inkompatibilität einzusetzen ist.

Viele der vorgestellten Ansätze beziehen sich auf Webservices. Da bei diesen Systemen der Fokus naturgemäß nicht sehr stark auf Timing und die Einhaltung von Deadlines liegt, werden diese Aspekte in den Ansätzen nicht betrachtet und können auch nicht in Bezug auf die Aufgabenstellung bewertet werden.

Für diese Arbeit wird besonders die Einfachheit der Muster zur manuellen Erstellung eines Adapters und die zustandsunabhängige Ausführung als großer Vorteil gegenüber der nachfolgend beschiebenen Adaptierung mit endlichen Automaten gesehen. Deshalb stellen Muster einen integralen Bestandteil der hier erarbeiteten Lösung dar.

3.3.3. Verhaltensadaptierung mithilfe von endlichen Automaten

Ein weiterer Ansatz für die Adaptierung von Softwareschnittstellen ist die Verwendung von endlichen Automaten (siehe Abschnitt 3.2.1) in verschiedenen Ausprägungen. Gegenüber der Adaptierung mit Mustern, hat bei der Verwendung eines Automaten die Historie der ausgetauschten Nachrichten Einfluss auf das Verhalten des Adapters.

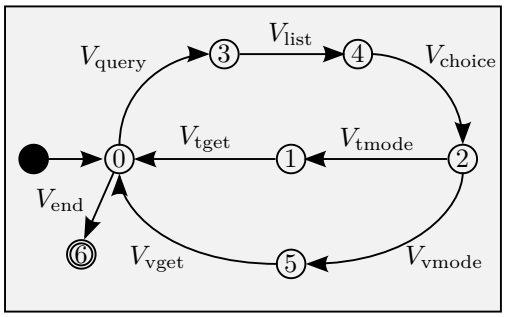
Einer der ersten Ansätze zur Adaptierung von Softwareschnittstellen wurde von Yellin und Strom [YS97] definiert. Sie verwendeten endliche Automaten in Form von Mealy-Automaten [Mea55]. Das bedeutet, dass für eine Transition eine Eingabe und mehrere optionale Ausgaben definiert werden. Das Besondere an diesem Ansatz ist die Betrachtung von Parametern. Parameterwerte können zwischengespeichert, wiederverwendet oder gelöscht werden.

Canal und Poizat [CPS08] stellen einen Ansatz vor, der auf Vektor-LTS basiert. Bei den *Vektoren* handelt es sich im Prinzip um die im vorherigen Abschnitt vorgestellten Mappings. In einem Vektor-LTS werden die Kanten des Automaten mit Vektoren annotiert. Abbildung 3.6 zeigt ein Beispiel für diese Trennung von Vektoren und LTS für das in Abschnitt 3.2.1 beschriebene Beispiel. In der Vektor-LTS muss in jedem Zustand für alle in diesem Zustand möglichen Nachrichten ein entsprechender Vektor existieren. Die ersten drei Vektoren V_{query}, V_{list} und V_{choice} ordnen einer Nachricht ihr gleichnamiges Gegenüber zu. Die nächsten Vektoren beheben die Inkompatibilitäten. Die Nachrichten *videorequest?* und *textrequest?* werden durch die Vektoren V_{vmode} und V_{tmode} durch die Zuordnung der

undefinierten Nachricht „_“ nicht an PDA weitergeleitet. Anschließend werden die letzten zwei Inkompatibilitäten durch die Vektoren V_{vget} und V_{tget} behoben. Der Vorteil ist, dass ein Vektor wiederverwendet werden kann, und der Nachteil, dass der Automat ohne die Betrachtung der Vektoren nicht verständlich ist.

$V_{\text{query}} = \langle$ROOM: query?, PDA: query!$\rangle$
$V_{\text{list}} = \langle$ROOM: list!, PDA: list?$\rangle$
$V_{\text{choice}} = \langle$ROOM: choice?, PDA: choice!$\rangle$
$V_{\text{vmode}} = \langle$ROOM: videorequest?, PDA: _$\rangle$
$V_{\text{tmode}} = \langle$ROOM: textrequest?, PDA: _$\rangle$
$V_{\text{vget}} = \langle$ROOM: video!, PDA: mpeg?$\rangle$
$V_{\text{tget}} = \langle$ROOM: text!, PDA: pdf?$\rangle$

(a) Beispielvektoren.



(b) Beispiel-Vektor-LTS.

Abbildung 3.6.: Adaptierung von ROOM und PDA mit einem Vektor-LTS.

Darauf aufbauend beschreiben Cubo et al. [Scr07] die praktische Anwendung des Ansatzes für Windows® Workflow Foundation (WF)[Scr07], welches zum .NET Framework gehört. Als Verhaltensspezifikation für Softwarekomponenten werden die in WF spezifizierten Workflows verwendet, aus denen das LTS generiert wird. Ansonsten unterscheidet sich der Ansatz kaum von dem in [CPS08] vorgestellten. Am Ende wird das generierte Adapter-Vektor-LTS wieder in einen Workflow transformiert, der die Generierung von Quelltext ermöglicht.

Ein erweiterter Ansatz mit Vektor-LTS wird von Mateescu et al.[MPS12] vorgestellt. Die Erweiterung betrifft zum einen die halbautomatische Generierung von Adaptern mit Prozessalgebra und zum anderen die für uns wichtige Betrachtung von übertragenen Parametern und deren Werten. In dieser Arbeit werden Webservices betrachtet, welche den Vorteil besitzen, dass mit BPEL bereits eine ausführbare Modellierungssprache zur Modellierung des Adapters verwendet werden kann [BP06]. Das Endprodukt der Generierung ist ein in abstrakter BPEL beschriebener Workflow, welcher direkt von einer Workflowmaschine ausgeführt werden kann.

Bewertung hinsichtlich der Aufgabenstellung

Endliche Automaten sind eine intuitive Lösung, das gesamte Verhalten eines Adapters zu beschreiben. Unabhängig von der konkreten Notation, kann die in 2.5 definierte Aufgabenstellung mit den in diesem Abschnitt genannten Ansätzen al-

leine nicht gelöst werden. Mehrere Erweiterungen wären nötig um die geforderte Durchgängigkeit bis hin zur Code-Generierung zu erreichen:

Hohe Abstraktion: Ein Nachteil ist die hohe Abstraktion des realen Systems, denn es werden meist nur die Operationen berücksichtigt. In seltenen Fällen [MPS12; YS97] werden auch Parameter betrachtet, deren Werte für einen anderen Aufruf wiederverwendet werden können. Dabei wird allerdings nicht auf die Transformation der Daten eingegangen. Diese Konzepte gilt es zu erweitern, um auch beliebige Berechnungen mithilfe der Parameter zu erlauben. Des Weiteren sind Parameter auch für das Verhalten des Adapters von entscheidender Bedeutung. Es kommt zum Beispiel oft vor, dass ein Client nach einer fehlgeschlagenen Operation (result = ERROR) ein anderes Verhalten aufweist, wie bei einer korrekten Ausführung (result = OK). Diesem Umstand wird in keinem der Ansätze Rechnung getragen. Letztendlich bietet auch keiner der Ansätze die geforderte Modellierung von Zeitbedingungen, welche bei fahrzeugspezifischen Funktionen eine große Rolle spielt.

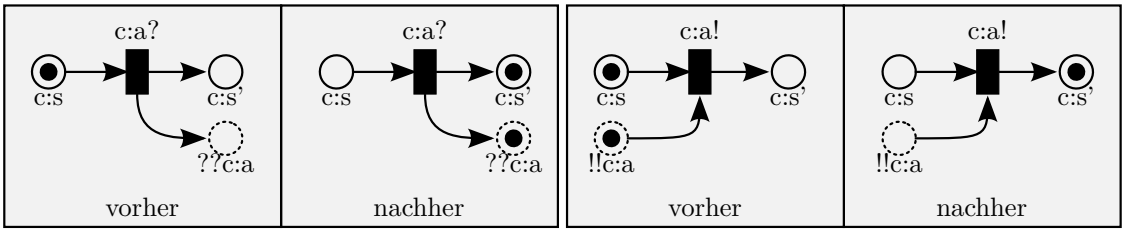
Monolithisches Adaptermodell: Die in den Ansätzen entstehenden Adapter werden durch die Modellierung eines monolithischen endlichen Automaten groß und komplex. Canal et al. [CPS08] sprechen bei einem einfachen Beispiel bereits von mehr als 100 Zuständen und über 200 Transitionen. Für einen Menschen ist ein derartiges Modell kaum zu überblicken und ohne weitere Optimierung würde die Ausführung des Modells einen großen Overhead für die Kommunikation bedeuten.

Zielsystem: Die Grundidee der Modellierung mit endlichen Automaten ist generisch und kann auf beliebige Zielsysteme übertragen werden. Die Methoden zur Generierung von lauffähigen Adaptern sind für WF und Webservices vorhanden und müssen verallgemeinert und auf die Fahrzeugdomäne angepasst werden.

Grundsätzlich wird der Ansatz zur Modellierung mit endlichen Automaten positiv bewertet, da er im Gegensatz zur Adaptierung mit Mustern allgemeingültig für verschiedene Szenarien anwendbar ist. Es liegt vor allem an den Details der Modellierungssprache, wie zum Beispiel der Annotation der Kanten und der Modellierung von Timing, ob ein Ansatz geeignet ist, die in dieser Arbeit gestellten Anforderungen zu erfüllen.

3.3.4. Verhaltensadaptierung mithilfe von Petri-Netzen

Petri-Netze sind eine Erweiterung endlicher Automaten zur Beschreibung nebenläufiger Schaltvorgänge. Einige der vorgestellten Methoden zur Adaptermodellierung rein mit endlichen Automaten unterstützen zum Beispiel nicht die Wiederverwendung einer empfangenen Nachricht für zukünftige Aktionen. Deshalb wird in [CPS08] ein Ansatz zur Erweiterung des Vector-LTS mithilfe von Petri-Netzen vorgestellt. Dieser erlaubt es, Nachrichten in Form von Token in Stellen zu speichern und später die gespeicherten Nachrichten aus den Stellen zu konsumieren. Dieser Mechanismus ermöglicht zum Beispiel das Umsortieren von Nachrichten. Dabei wird das LTS der zu adaptierenden Komponenten zuerst in ein Petri-Netz transformiert (mehr Informationen zu dieser Transformation in [Cor+98; BD98]) und anschließend mit den zusätzlichen Stellen zur Speicherung von Nachrichten angereichert. Abbildung 3.7 zeigt die zwei dafür notwendigen Petri-Netze. Die Stelle zum Speichern einer Nachricht ist mit gepunkteter Linie gekennzeichnet. Um die Diagramme nicht zu überladen, wird angenommen, dass Speicher-Stellen mit demselben Bezeichner (z.B. ??c:a und !!c:a) dieselbe Stelle im Modell sind. Das bedeutet, sobald in eine derartige Stelle geschrieben wird, erscheint in allen gleichnamigen Stellen im Modell ein Token. Abbildung 3.7a zeigt das Speichern der von einer Komponente gesendeten Nachricht *c:a* im Adapter in der Stelle *??c:a*. Diese Stelle *??c:a* kann nun als Eingabe für beliebige Transitionen verwendet werden. Abbildung 3.7b zeigt die dafür notwendigen Stellen und Transitionen. Die Ausgabe der Nachricht kann erst erfolgen, sobald ein Token in der Stelle *!!c:a* liegt.



(a) Speichern einer eingehenden (Symbol ? an der Transition) Nachricht *c:a* in der Stelle *??c:a* vor (links), und nach dem Empfang der Nachricht (rechts).

(b) Ausgabe (Symbol ! an der Transition) der Nachricht *c:a* mithilfe der gespeicherten Nachricht in der Stelle *!!c:a*, vor (links) und nach der Ausgabe (rechts).

Abbildung 3.7.: Einsatz von Petri-Netzen zum Zwischenspeichern von Nachrichten und zum Ausgeben von gespeicherten Nachrichten (aus [CPS08]).

Zusätzlich zu dieser Erweiterung zur Speicherung von Nachrichten präsentieren Gierds et al. [GMW08; GMW12] eine weitere Methode zur Schnittstellenadap-

tierung mit *Open Petri-Nets* [Kin97], welche spezielle Stellen im Petri-Netz als Schnittstelle der Komponente ausweisen. Der Adapter besteht zunächst aus manuell erstellten *elementaren Aktivitäten*, welche auf eingehende Nachrichten angewendet werden: Create, Copy, Delete, Transform, Split, Merge und Recombine. Diese Aktivitäten entsprechen den aus Abschnitt **??** bekannten Mustern. Anschießend wird das Adaptermodell auf Basis dieser Aktivitäten, in Form eines Petri-Netzes, generiert und algorithmisch reduziert. Der Ansatz resultiert bereits für kleine Softwarekomponenten in sehr umfangreichen Petri-Netzen.

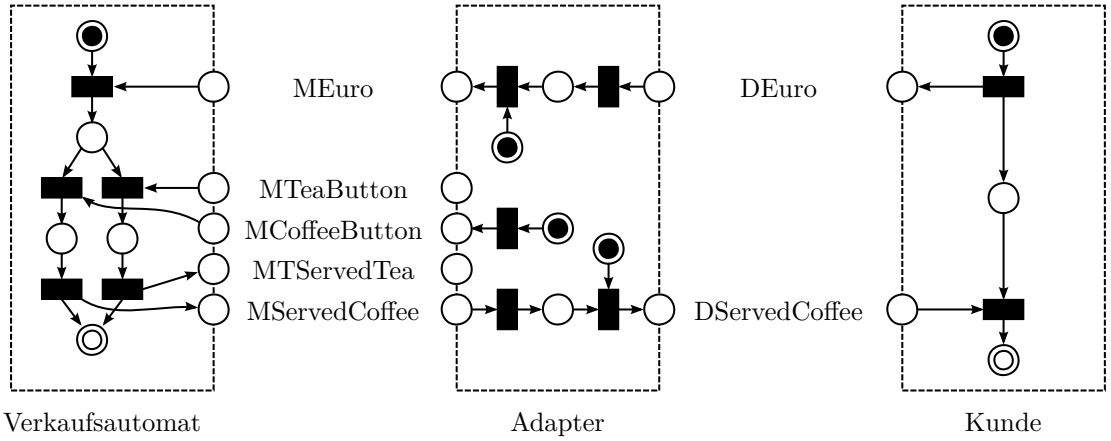


Abbildung 3.8.: Adaptierung mit Petri-Netzen (aus [GMW08])

Abbildung 3.8 zeigt ein Beispiel für die Adaptierung mit Open Petri-Nets. Der Verkaufsautomat erwartet die Eingabe eines Euro und die Auswahl von Kaffee oder Tee und gibt das entsprechende Getränk aus. Der Kunde auf der rechten Seite erwartet nach Eingabe eines Euro bereits die Ausgabe eines Kaffee. Die Stellen auf dem Rahmen des Diagramms bilden die Schnittstelle. Open Petri-Nets werden an ihren Schnittstellen verbunden und so kann ein Token vom Petri-Netz *Kunde* über den Adapter bis hin zum Petri-Netz *Verkaufsautomat* wandern. In diesem Fall ist der Adapter so konstruiert, dass der Verkaufsautomat nur die Ausgabe eines Kaffee unterstützt, denn die Schnittstelle *MTeaButton* ist im Adapter nicht mit einer Transition verbunden und kann somit auch nicht mit einem Token befüllt werden. Wenn man nun annimmt, dass jede Transition, die feuern kann, auch feuert, dann erhält der Kunde nach Eingabe des Euro durch die Adaptierung ohne weitere Aktion einen Kaffee.

Petri-Netze werden auch von Popescu et al. [Pop+12] für die Adaptierung von geschichteten Webservices verwendet. Der Formalismus wird in einer komplexen Adapter-Architektur ebenfalls zur Adaptierung von Verhalten verwendet.

Bewertung hinsichtlich der Aufgabenstellung

Petri-Netze sind ein sehr bekannter Formalismus zur Beschreibung von nebenläufigen Prozessen oder Schaltvorgängen. Durch ihre formale Natur eignen Sie sich ebenfalls für das Beweisen bestimmter Eigenschaften, wie zum Beispiel der Deadlockfreiheit eines Systems. Zusätzlich erlauben sie die Modellierung von nebenläufigen Aktivitäten im Adapter.

Bei Verwendung von Petri-Netzen ist zu beachten, dass sich die möglichen Zustände nicht aus den modellierten Stellen, sondern aus der Menge an erreichbaren Belegungen ergibt. Deshalb sind Petri-Netze auch bei kleinen Modellen für den Menschen nicht einfach zu durchschauen und für die manuelle Modellierung nicht geeignet.

Unabhängig von diesem Defizit, lässt sich die in Abschnitt 2.5 definierte Aufgabenstellung mit Petri-Netzen alleine nicht lösen. Es gilt (wie bei den Ansätzen mit endlichen Automaten) auch hier die Einschränkung, dass die abstrakte Modellierung keine Generierung von lauffähigem Binärcode für eine konkrete Middleware ermöglicht. Das liegt einerseits an der abstrakten Definition von Nachrichten und andererseits an der fehlenden Adaptierung von Funktionsparametern.

3.3.5. Fazit

In diesem Abschnitt wurden verschiedene Methoden zur Realisierung von Software-Adaptern vorgestellt, welche eine gute Grundlage für die Adaptierung von Softwarekomponenten im Fahrzeug bieten.

Der erste Ansatz der *datenzentrierten Adaptierung* bezieht sich hauptsächlich auf die Adaptierung von Datenbanken. Auch wenn aktuelle Kommunikationsmedien im Fahrzeug ohne Datenbanken arbeiten, ist die Adaptierung von Nutzdaten eine notwendige Eigenschaft, um eine Transformation von Parameterwerten durch den Adapter zu erlauben. Die vorgestellte *Adaptierung mit Mustern*, eignet sich besonders für einfache, und häufige Schnittstellenänderungen, wie zum Beispiel das Umbenennen einer Methode oder eines Parameters. Muster sind meist so konstruiert, dass sie einfach vom Modellierer spezifiziert werden können und global für den gesamten Zustandsraum der Softwarekomponente gelten. Als allgemeingültiger Ansatz für komplexe Adaptierungen eignen sich die vorgestellten *graphenbasierten Ansätze*, die auch Konzepte zur Modellierung von Zeitbedingungen bieten.

Das Ziel dieser Arbeit ist es nun, die positiven Eigenschaften der gezeigten Ansätze zu verbinden. Das bedeutet, dass für häufige Adaptierungen Muster verwendet werden sollen und für komplexe Adaptierungen ein allgemeingültiger graphenbasierter Ansatz. Zusätzlich soll die Adaptierung von Nutzdaten und die Spezifi-

kation von Zeitbedingungen ermöglicht werden.

3.4. Adapter-Architekturen

In der objektorientierten Programmierung ist der Adapter [Gam+95] ein oft verwendetes strukturelles Entwurfsmuster, welches zur Übersetzung einer Schnittstelle in eine andere verwendet wird. Grundsätzlich kann zwischen *interner* und *externer Adaptierung* unterschieden werden. Bei der internen Adaptierung werden interne Abläufe der Komponente adaptiert und bei der externen die Schnittstellen der Komponente. Da in dieser Arbeit die Adaptierung von Black-Box Softwarekomponenten im Mittelpunkt steht, deren internes Verhalten schwer modifiziert werden kann, ist besonders die externe Adaptierung interessant. Dort gibt es grundsätzlich zwei Möglichkeiten einen Adapter für ein verteiltes System zu realisieren: Eine *zentrale* oder eine *dezentrale* Adapter-Architektur [Aut+08]. Abbildung 3.9 skizziert eine zentrale (System), und zwei dezentrale (Komponente und Schnittstelle) Adapter-Architekturen.

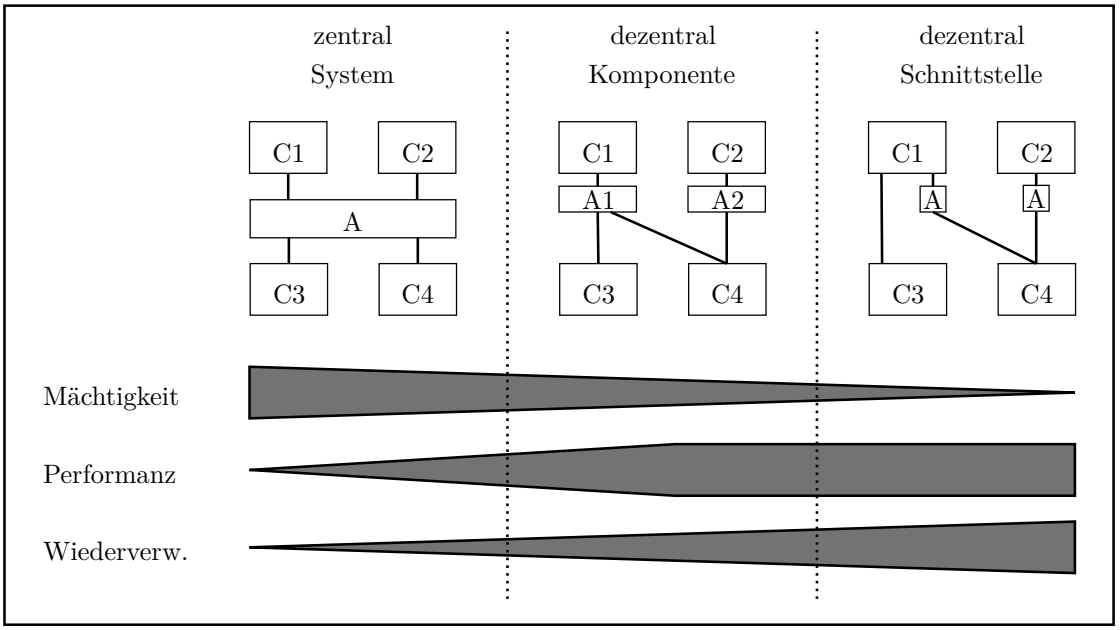


Abbildung 3.9.: Drei verschiedene Adapter-Architekturen und deren Vor-/Nachteile: zentrale Architektur (links), dezentrale Architektur auf Komponentenebene (mittig dargestellt), dezentrale Architektur auf Schnittstellenebene (rechts)(Angelehnt an [Aut+08]).

Zentrale Adapter-Architektur: Bei der zentralen Adapter-Architektur sind alle Softwarekomponenten mit dem Adapter verbunden, der eine Brücke für die gesamte Kommunikation zwischen den Komponenten bildet. Vorteil dieser Architektur ist die Mächtigkeit der Adaptierung, denn der Adapter kann das Verhalten nicht nur auf Basis der Kommunikation zweier Komponenten, sondern auf Basis der gesamten Kommunikation im System spezifizieren. Der Nachteil dieser Lösung ist der große Overhead an Kommunikation, denn jede Nachricht, auch wenn sie keine Adaptierung zur Folge hat, muss über den Adapter gesendet werden. Dies beinhaltet zweimaliges Serialisieren und Deserialisieren der Daten. Ein Beispiel einer zentralen Adapter-Architektur wird von Gui et al. [Gui+11] gezeigt.

Dezentrale Adapter-Architektur - Komponenten-Adapter: Beim Komponenten/=Adapter kann einer Komponente ein dedizierter Adapter zugeordnet werden. Der Nachteil gegenüber der zentralen Lösung ist, dass der Adapter weniger mächtig ist, denn er kann nur Entscheidungen auf Basis der Daten und Abläufe der eigenen Komponente und nicht des gesamten Systems treffen. Ein wesentlicher Vorteil ist, dass der Komponenten-Adapter effizient an die Komponente angebunden werden kann, im Idealfall durch sehr performante interne Funktionsaufrufe. Ein weiterer Vorteil ist die höhere Wiederverwendbarkeit des Adapters, da er für ähnliche Komponenten wiederverwendet werden kann.

Dezentrale Adapter-Architektur - Schnittstellen-Adapter: Eine zweite Variante der dezentralen Adapter-Architektur ist die Schnittstellen-Adaptierung. Der Vorteil ist eine nochmals gestiegene Wiederverwendbarkeit des Adapters, denn er kann für jede Komponente, welche dieselbe Schnittstelle benötigt oder anbietet, verwendet werden. Im Beispiel aus Abbildung 3.9 kann der Adapter A zur Kommunikation mit der Komponente C4 von den Komponenten C1 und C2 verwendet werden. Der Nachteil ist, dass der Adapter weniger mächtig ist, da er nur Einfluss auf Abläufe an der adaptierten Schnittstelle nehmen kann. Szenarien, die eine Adaptierung über mehrere Schnittstellen der Softwarekomponente erfordern, sind deshalb nicht umsetzbar.

Bewertung hinsichtlich der Aufgabenstellung

Für eine minimal-invasive Adaptierung von Softwareschnittstellen ist für uns die externe Adaptierung von Bedeutung. Sie ermöglicht es, die Schnittstelle von Softwarekomponenten anzupassen, ohne die Softwarekomponente selbst zu verändern. Bei der Wahl zwischen einer zentralen und einer dezentralen Adapter-Architektur muss zwischen Mächtigkeit und Performanz abgewogen werden. Grundsätzlich

spielt es für die in dieser Arbeit vorgestellten Konzepte keine große Rolle mit welcher Architektur sie implementiert werden. Um die im Fahrzeugumfeld bereits gut ausgelasteten Kommunikationskanäle nicht mit zusätzlichen Nachrichten für den Adapter zu belasten, wählen wir dafür einen dezentralen Ansatz.

3.5. Zielsystem GENIVI Linux

Ein wesentliches Ziel der GENIVI Initiative ist es, Open-Source auf die Fahrzeug-Headunit zu bringen. Grundlage dafür ist das Linux Betriebssystem mit seinen zahlreichen Open-Source Projekten und Derivaten. In den Bereichen, in denen noch keine Open-Source Lösungen für das Fahrzeug-Infotainment existieren, werden in der GENIVI Initiative entsprechende Open-Source Projekte initiiert und unterstützt. Ähnlich wie AUTOSAR [GbR14a; GbR14b] definiert GENIVI einen herstellerübergreifenden Standard, der eine gemeinsame Architektur definiert und die Schnittstellen für herstellerübergreifend genutzte Komponenten standardisiert.

Abbildung 3.10 zeigt eine einfache Darstellung der Architektur für Interprozesskommunikation in GENIVI. Die syntaktischen Schnittstellen werden mit Franca IDL beschrieben. Aus dieser textuellen Schnittstellenbeschreibung wird Quelltext für die Kommunikations-Middleware generiert. Für Interprozesskommunikation auf der Head-Unit wird in diesem Fall D-Bus verwendet. Um den Applikationscode möglichst unabhängig vom verwendeten Middleware-Backend zu halten, wurde im GENIVI-Konsortium mit CommonAPI C++ eine zusätzliche Abstraktionsschicht hinzugefügt. Diese bietet für den Programmierer applikationsseitig eine einheitliche Schnittstelle unabhängig von der verwendeten Middleware. Wir beschreiben nun die einzelnen Teile genauer.

Franca IDL (FIDL) Franca IDL (FIDL) ist die im GENIVI Projekt verwendete Sprache zur Beschreibung der syntaktischen Schnittstellen. Die Sprache bietet Konstrukte zur Beschreibung von Methoden, Broadcasts und komplexen Datentypen. Zusätzlich zu der syntaktischen Beschreibung der Schnittstelle erlaubt Franca IDL auch die Definition von Protokollen auf Basis von Zustandsautomaten in textueller Form. Diese Funktion ist in einem experimentellen Stadion und wird aktuell nicht angewandt. Um eine Übersicht aller Sprachkonstrukte zu erhalten, verweisen wir auf [Bir14].

CommonAPI: CommonAPI ist ein GENIVI Teilprojekt [Gen14b], welches die Verwendung einer Kommunikationsmiddleware in der Programmiersprache C++

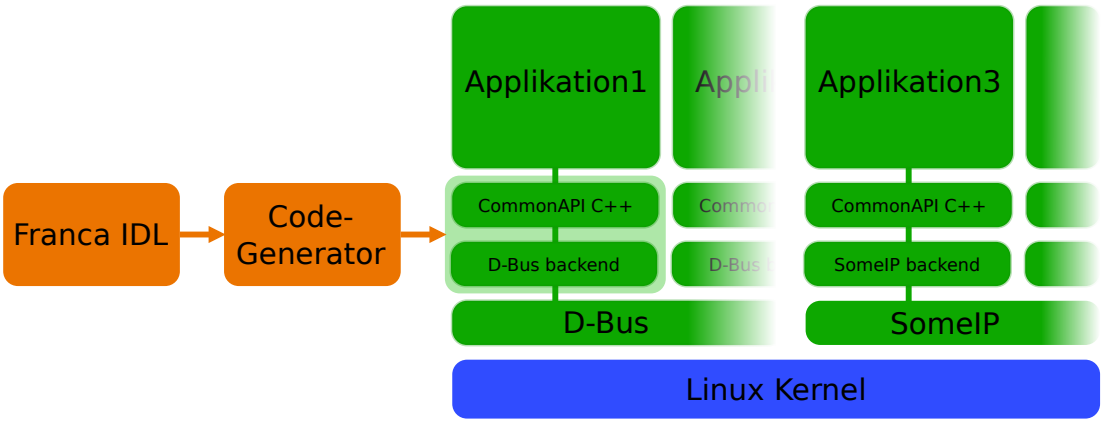


Abbildung 3.10.: GENIVI Architektur mit CommonAPI C++ und verschiedenen Middleware-Technologien.

standardisiert. Diese Schnittstelle ist sehr abstrakt gehalten, sodass sie den Austausch der Kommunikationsmiddleware erlaubt, ohne den Quelltext einer Applikation zu verändern. Für diese Arbeit ist dies ein entscheidender Vorteil, denn eine Lösung zur Adaptierung auf Ebene der CommonAPI ermöglicht es uns, unabhängig von der verwendeten Middleware sämtliche in C++ geschriebene Applikationen zu adaptieren, vorausgesetzt sie verwenden CommonAPI. Zusätzlich zu den eigentlichen Schnittstellen stellt CommonAPI auch Funktionen zur Erzeugung der Stub- und Proxy-Objekte zur Verfügung.

Franca CommonAPI (D-BUS) Code-Generator: Der Franca CommonAPI D-BUS Code-Generator übersetzt eine Franca IDL Schnittstellenbeschreibung in C++ Quelltext. Der besteht aus dem middlewareunabhängigen CommonAPI-Teil und dem D-Bus spezifischem. Für Methoden wird jeweils eine Deklaration für synchrone und eine für asynchrone Aufrufe generiert. Im synchronen Fall blockiert der Client nach dem Methodenaufruf solange, bis die Rückgabe vom Server gesendet wird. Im asynchronen Fall fährt der Client unmittelbar nach dem Methodenaufruf mit der Berechnung fort und erhält eine eventuelle Rückgabe durch einen Callback vom Server. Zahlreiche Middlewarelösungen realisieren diese Aufrufsemantiken. Bei Corba [OMG12] zum Beispiel, wird ein asynchroner Aufruf ohne Rückgabe durch das *oneway*-Schlüsselwort definiert [YS97].

3.6. Zusammenfassung

Dieses Kapitel beleuchtete verschiedene Aspekte im Bezug auf die Adaptierung von Softwareschnittstellen im Fahrzeug. Zum Einen waren dies im Fahrzeugumfeld eingesetzte Methoden der komponentenbasierten Softwareentwicklung und zum Anderen aktuelle Ansätze zur Modellierung und Adaptierung von Softwareschnittstellen.

Wir haben gezeigt, dass die aktuell eingesetzten Methoden zur komponentenbasierten Softwareentwicklung im Bereich Infotainment bereits Möglichkeiten bieten, dynamisch Teile der Software nachzuladen oder auszutauschen. Diese grundlegende Voraussetzung für eine minimal-invasive Adaptierung von Softwarekomponenten fehlt in der AUTOSAR Methodik. Für die modellbasierte Beschreibung von Softwarekomponenten werden in beiden Bereichen bereits Methoden angewandt. In der AUTOSAR Methodik sind die Modelle ein integraler Bestandteil und werden für den Generierungsprozess verwendet. Im Infotainment, beispielsweise im GENIVI Projekt, wird vor allem die UML zur Modellierung verwendet. Hier ist die Modellierungsmethodik nicht so stark mit dem Entwicklungsprozess verwoben und wird vorwiegend zur Spezifikation verwendet.

Anschließend wurden verschiedene Methoden zur Spezifikation von Schnittstellen verglichen. Für die Modellierung der syntaktischen Schnittstellen existieren bereits zahlreiche etablierte Modellierungsmethoden, die wir in dieser Arbeit übernehmen und nicht anpassen möchten. Bei der Modellierung von Verhalten ist ein deutlicher Unterschied zwischen den praktisch eingesetzten und den wissenschaftlichen Ansätzen zu erkennen. Zum Beispiel ergibt sich der Vorteil der Modellierung von Schnittstellenautomaten in der Praxis erst, wenn dadurch die Qualität oder die Wiederverwendbarkeit einer Softwarekomponente erhöht werden kann. Dazu ist es meist notwendig, das Modell so konkret zu gestalten, dass es in eine ablauffähige Form gebracht werden kann. Das Modell wird dadurch so komplex, dass aus der Wissenschaft etablierte Methoden des Model-Checkings nicht mehr sinnvoll einsetzbar sind. Auf der anderen Seite sind die vorgestellten wissenschaftlichen Ansätze mit Petri-Netzen oder endlichen Automaten so abstrakt gehalten, dass sie in ihrer heutigen Form keine direkte Generierung von Adaptern für die im Fahrzeugumfeld eingesetzten Kommunikationssysteme erlauben. Deshalb ist es ein Ziel dieser Arbeit, die vorgestellten Konzepte anzupassen und zu erweitern um die Ziele in Abschnitt 2.5 zu erreichen.

Zudem wurde gezeigt, dass bereits zahlreiche Ansätze zur Adaptierung von Softwarekomponenten existieren. Die Datenadaptierung schafft die Grundlage für eine Transformation von Parameterwerten und ist für ein praxistaugliches Konzept unerlässlich. Muster eignen sich vor Allem für häufig wiederkehrende Adaptierungen

und sind für den Modellierer einfach anzuwenden. Endliche Automaten hingegen eignen sich zur Modellierung von komplexen Abläufen. Auch hier konnte festgestellt werden, dass eine Kombination der Methoden am besten zur Erreichung der Ziele geeignet ist.

Im nachfolgenden Kapitel wird ein neuer Ansatz zur hierarchischen Modellierung von Schnittstellen vorgestellt, der eine klare Trennung zwischen Syntax und Verhalten spezifiziert. Dieser Ansatz enthält im Wesentlichen zwei Erweiterungen für die hier vorgestellten Schnittstellenautomaten. Die erste Erweiterung ist eine zusätzliche Modellierungsebene, welche die Übersetzung der String-basierten Nachrichten in konkrete Funktionsaufrufe und umgekehrt beschreibt und somit sowohl das Model-Checking des Automaten, als auch die Generierung von Binärcode für ein Zielsystem ermöglicht. Zusätzlich erreichen wir durch die Trennung von Syntax und Verhalten, dass auf syntaktischer Ebene mit einer etablierte IDL gearbeitet werden kann und der Automat unabhängig erstellt und wiederverwendet werden kann. Die zweite Erweiterung ist eine vereinfachte Modellierung von Zeitbedingungen. Im Gegensatz zu der hier vorgestellten Modellierung mithilfe globaler Zeitvariablen können nur einfache zeitliche Abläufe basierend auf der aktuellen Zeit im aktuellen Zustand beschrieben werden. Dies beinhaltet die Beschreibung von Maximal- und Minimal-Wartezeiten (Timeouts) und deckt somit die häufigsten Anwendungsfälle ab. Nicht beinhaltet ist das Zusammensetzen einer Zeitbedingung aus mehreren Zeitvariablen oder das Verändern einer Zeitbedingung in jedem beliebigen Zustand des Automaten.

Kapitel 4

Kontrollflusszentrierte Systemmodellierung mit DANA

Grundgedanke der komponentenbasierten Softwareentwicklung ist, dass eine Softwarekomponente durch ihre Schnittstelle wiederverwendet werden kann, was in der Praxis die gesamte Schnittstelle auf allen in Abschnitt 2.2 beschriebenen Ebenen betrifft. Bei modellbasierter Entwicklung der Softwarekomponenten bestimmen die Charakteristiken und die Ausdruckskraft des Schnittstellenmodells, welcher Grad der Interoperabilität bereits auf Modellebene nachgewiesen werden kann [CPS08]. Die beiden vorherigen Kapitel haben gezeigt, dass aktuell eingesetzte Schnittstellenmodelle dafür nur teilweise geeignet sind. In der Industrie verbreitete Modelle decken hauptsächlich die syntaktische Ebene ab, während die in Abschnitt 3.2.1 beschriebenen Automaten die höheren Ebenen betrachten. Der in diesem Abschnitt vorgestellte Modellierungsansatz schließt diese Lücke zwischen syntaktischer Ebene und Verhaltensebene und ermöglicht somit eine durchgängige Beschreibung von Schnittstellen. Dieses Kapitel baut auf existierenden Vorarbeiten des Autors auf [PSB13; Dra+13; Pra+14].

Abschnitt 4.1 gibt einen Überblick über die Abstraktionsebenen des Modellierungsansatzes. Zentrales Element für die spätere Schnittstellenadaptierung ist das hierarchische Schnittstellenmodell, welches in Abschnitt 4.2 formal beschrieben wird. Darauf aufbauend beschreibt Abschnitt 4.3 die Kompatibilitätseigenschaften des hierarchischen Modellierungsansatzes und Abschnitt 4.4 zeigt die Anwendung des Modellierungsansatzes anhand der in dieser Arbeit verwendeten Fallstudie.

Inhaltsverzeichnis

4.1. Abstraktionsebenen im Modellierungsansatz

Der DANA Modellierungsansatz wird zur Modellierung des Kontrollflusses komponentenbasierter Softwaresysteme verwendet. Die Modelle werden dabei in der Spezifikationsphase manuell durch einen Softwarearchitekten erstellt. Eine wesentliche Eigenschaft des Ansatzes ist die durchgängige Modellierung einer Schnittstelle von der Syntax bis hin zum Verhalten. Der Modellierungsansatz ist sehr abstrakt definiert und lässt sich somit auf verschiedene komponentenbasierte Systeme anwenden. Die Modelle wurden für folgende Einsatzzwecke konzipiert:

- **Absicherung von Schnittstellen:** Eines der wichtigsten Ziele bei der Entwicklung des Modellierungsansatzes war die automatisierte Absicherung des Kontrollflusses zwischen Softwarekomponenten in der Integrationsphase. Zum Beispiel soll die Frage beantwortet werden, ob die Interaktion der Softwarekomponenten der Spezifikation entspricht. Dies beinhaltet auch die Zeitbedingungen. Für diesen Zweck wird der Kontrollfluss mithilfe von Zustandsautomaten beschrieben, welche anschließend in eine ausführbare Form transformiert werden. Die Automaten überwachen das Kommunikationsmedium und protokollieren alle Abweichungen vom spezifizierten Verhalten. Ein Synchronisationsmechanismus sorgt dafür, dass Kommunikation und Zustandsautomat im Falle eines Fehlers nicht auseinanderdriften. Für eine genauere Beschreibung des Ansatzes wird auf Drabek et al. [Dra+13] verwiesen.

- **Emulation von Softwarekomponenten:** Ein weiteres Ziel war die Emulation von noch nicht existenten Softwarekomponenten in einer frühen Entwicklungsphase. Diese Emulation wird benötigt, falls ein System getestet werden soll, bei dem noch nicht alle Softwarekomponenten vorhanden sind. In diesem Fall wird der Kontrollfluss der fehlenden Softwarekomponenten modelliert. Anschließend wird ausführbarer Binärcode für das Zielsystem generiert. Die so generierten Softwarekomponenten bedienen die spezifizierte Schnittstelle, implementieren allerdings keine weitere Logik.

- **Model-Checking:** Der Ansatz eignet sich zur Überprüfung der Kompatibilität von Softwarekomponenten in der Spezifikationsphase und zur Berechnung von Wirkketten im Gesamtsystem [Pra+13].

- **Adaptierung von Softwarekomponenten:** Der Modellierungsansatz bildet die Basis für den in dieser Arbeit vorgestellten Ansatz zur Adaptierung von Softwarekomponenten.

Der Modellierungsansatz gliedert sich in verschiedene Abstraktionsebenen. Abbildung 4.1 zeigt die Bestandteile und ordnet sie in das Ebenenmodell der Fahrzeugsoftwareentwicklung ein. Auf der logischen Ebene beschreibt die *Systemstruktur* das Gesamtsystem als flaches Netzwerk von Softwarekomponenten, die über ihre Schnittstellen miteinander kommunizieren. Abbildung 4.1 zeigt als Beispiel die zwei Softwarekomponenten C1 und C2. Eine Komponente kann mehrere *Schnittstellen* besitzen. Alle Schnittstellen zwischen zwei Komponenten bilden die *Kommunikationsbeziehung*. Zusätzlich zu dieser strukturellen Sicht bietet die logische Ebene Modelle zur Spezifikation des gewünschten Verhaltens. Im Gegensatz zur

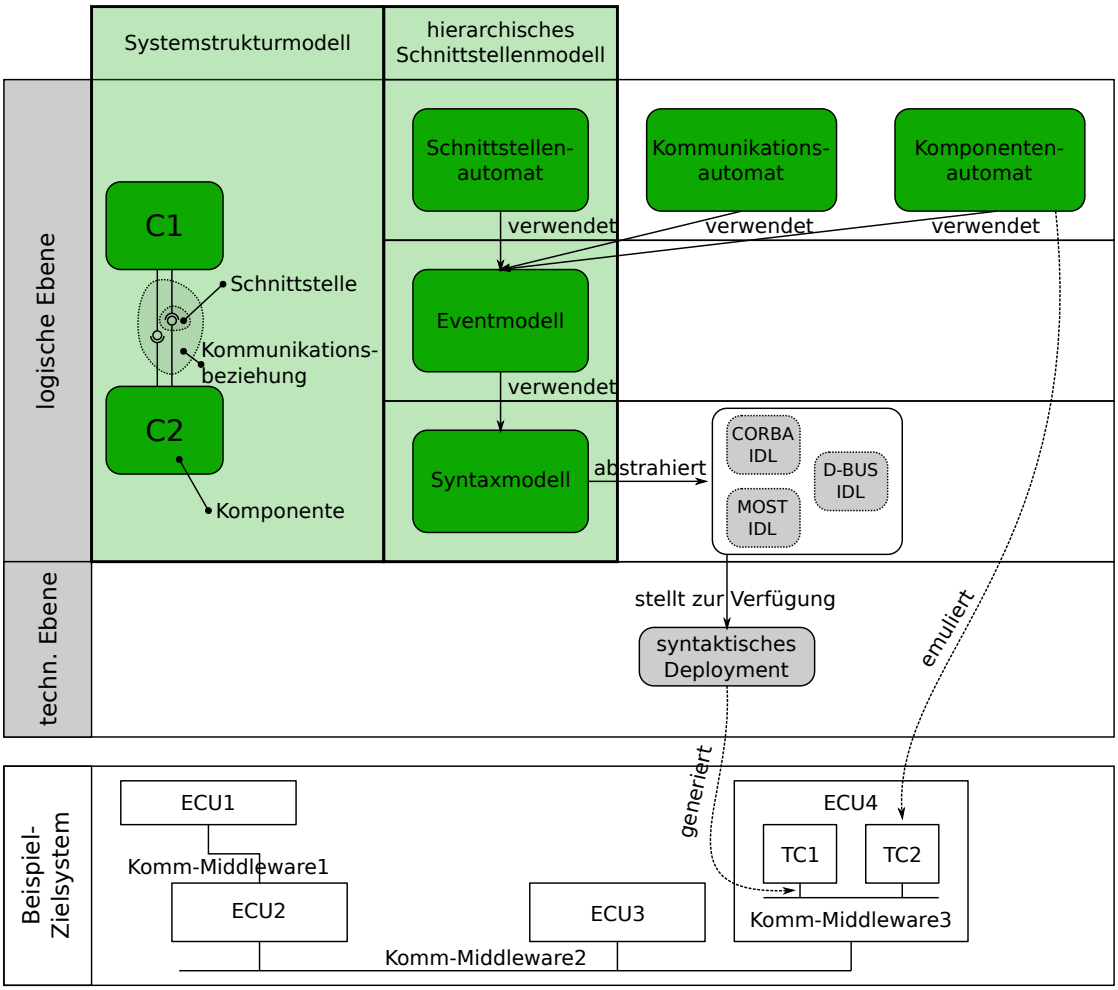


Abbildung 4.1.: Überblick über die DANA Modellierungsmethodik (Die wesentlichen Bstandteile von DANA sind grün schattiert).

Modellierungssprache COLA [Hab+10], welche den Datenfluss betrachtet, liegt der Fokus von DANA auf der Modellierung des Kontrollflusses. Der Kontrollfluss wird dabei durch die Abfolge von Funktionsaufrufen der syntaktischen Schnittstelle definiert. Die internen Abläufe innerhalb einer Softwarekomponente werden nicht betrachtet, sondern ausschließlich das an der Komponentenschnittstelle sichtbare Verhalten. Je nachdem, in welchem Kontext (*Komponente*, *Kommunikation* oder *Schnittstelle*) das Verhalten spezifiziert wird, unterscheiden wir die drei Verhaltensmodelle *Komponentenautomat*, *Kommunikationsautomat* und den *Schnittstellenautomat*.

Ein zentrales Ziel bei der Systemmodellierung mit DANA ist die Generierung von lauffähigem Binärcode für ein Zielsystem. Da bestehende Kommunikationsmiddlewaretechnologien bereits Schnittstellenbeschreibungssprachen und dazugehörige Werkzeuge zur Analyse der Modelle und Generierung von Quelltext bieten, setzt der Modellierungsansatz auf syntaktischer Ebene auf Wiederverwendung dieser Modelle. In Abbildung 4.1 sind zum Beispiel die drei möglichen Middlewaretechnologien MOST [Coo10], CORBA [OMG12] oder D-BUS [Pen+14] skizziert. Jede dieser Technologien stellt eigene Modelle und Code-Generatoren für verschiedene Zielsysteme zur Verfügung. Durch die Wiederverwendung dieser Modelle kann ein möglichst reibungsloses Deployment der Softwarekomponenten erfolgen. Die einzelnen Modelle werden nun genauer beschrieben.

4.1.1. Das Systemstrukturmodell

Das *Systemstrukturmodell* beschreibt die Verknüpfung von Softwarekomponenten durch ihre Schnittstellen. Auf der Ebene der Systemstruktur wird eine Untermenge des UML Komponentendiagrammes verwendet, welches Komponenten und deren Verknüpfung durch Schnittstellen beschreibt. Die Modellierung erfolgt dabei ohne die Betrachtung der in UML möglichen Ports. Eine Komponente kann somit direkt Schnittstellen *anbieten* oder *benötigen*.

Abbildung 4.3 zeigt auf der linken Seite ein Komponentenmodell für das Beispiel *Audiomanagement*, angelehnt an den im GENIVI-Projekt [Gen14b] spezifizierten Audiomanager (AM). Ein *CommandInterface* (CI) nimmt vom Benutzer initiierte Anfragen entgegen und stellt dann über den Audiomanager mithilfe von *Routing-Plugins* (RP) eine Audioverbindung zwischen einer Quelle und einer Senke her.

4.1.2. Verhaltensmodelle zur Modellierung des Kontrollflusses

Je nachdem, welcher Aspekt einer Softwarekomponente modelliert werden soll, werden drei Modelle zur Modellierung des Kontrollflusses unterschieden. Abbildung 4.2 zeigt diese Modelle und deren Überlappung am Beispiel der Komponenten *CI* und *AM*.

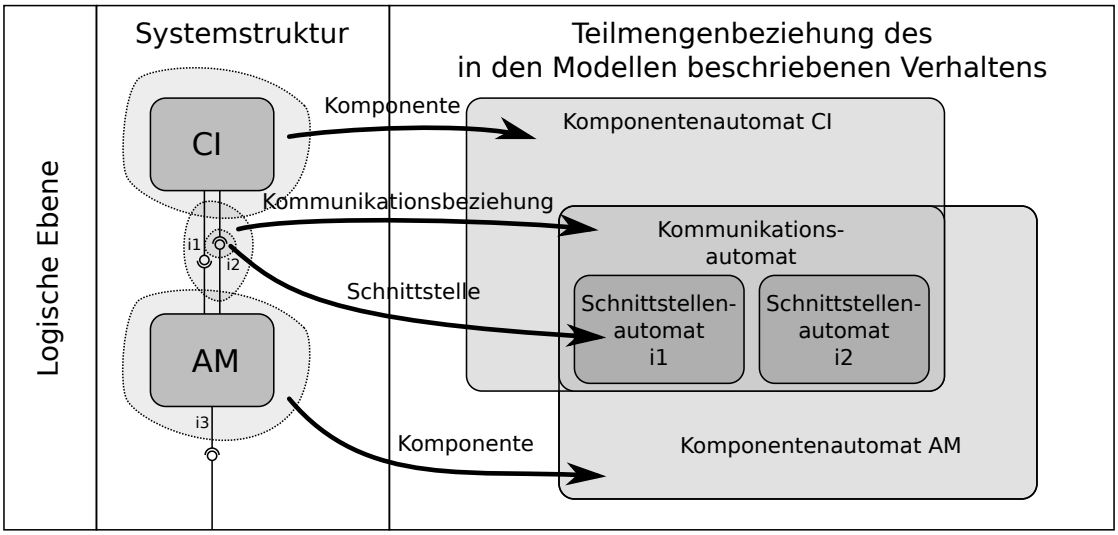


Abbildung 4.2.: Modellierung des Kontrollflusses im Kontext von Schnittstellen, Kommunikationsbeziehungen und Komponenten. Dabei ist das im Schnittstellenautomaten beschriebene Verhalten eine Untermenge des im Kommunikationsautomaten beschriebenen Verhaltens und dieses wiederum eine Untermenge des im Komponentenautomaten beschriebenen Verhaltens.

Die *Komponentenschnittstelle* definiert die Kommunikation einer Softwarekomponente mit allen Nachbarkomponenten und umfasst deshalb alle von einer Softwarekomponente angebotenen und benötigten Schnittstellen. Somit definiert die Komponentenschnittstelle den für eine Komponente relevanten und an ihren Schnittstellen sichtbaren Kontrollfluss. In Abbildung 4.2 umfasst die Komponentenschnittstelle von AM die Schnittstellen i1, i2 und i3. Der Kontrollfluss an der Komponentenschnittstelle wird im *Komponentenautomaten* spezifiziert.

Die *Kommunikationsbeziehung* beschreibt die Beziehung von genau zwei Softwarekomponenten und beinhaltet deshalb nur die Menge der Schnittstellen zwischen diesen beiden. Zum Beispiel ist die Kommunikationsbeziehung zwischen AM und

CI in Abbildung 4.2 durch die Schnittstellen i1 und i2 definiert. Der Kontrollfluss einer Kommunikationsbeziehung wird im *Kommunikationsautomaten* spezifiziert.

Letztendlich wird der auf eine *Schnittstelle* bezogene Kontrollfluss im *Schnittstellenautomaten* beschrieben. Eine Beschreibung auf dieser Ebene ist notwendig, um den Automaten für verschiedene Komponenten, die dieselbe Schnittstelle anbieten oder benötigen, wiederverwenden zu können. Wenn man eine Komponente betrachtet, ist der schnittstellenbezogene Kontrollfluss einer ihrer Schnittstellen Teil des kommunikationsbezogenen Kontrollflusses, welcher wiederum ein Teil des komponentenbezogenen Kontrollflusses ist. Somit können aus dem vollständigen Komponentenautomaten die anderen Modelle generiert werden.

Schnittstellen- und Kommunikationsautomat

Zur Beschreibung des Kontrollflusses zwischen zwei Komponenten werden die in Abschnitt 3.2.1 beschriebenen akzeptierenden Automaten (Akzeptoren) verwendet. Die auf den Transitionen annotierten Nachrichten werden als *Trigger* oder *Events* bezeichnet. Bei einem *Schnittstellenautomaten* wird der Kontrollfluss für eine Schnittstelle beschrieben. Konkret bezieht sich das auf die Kommunikation zwischen einem Server, der eine Schnittstelle anbietet, und einem Client, der diese Schnittstelle benötigt. Im *Kommunikationsautomaten* wird der Kontrollfluss einer Kommunikationsbeziehung beschrieben.

Abbildung 4.3 zeigt ein Beispiel für einen Schnittstellenautomaten. Er beschreibt die gültige Aufrufreihenfolge für die Schnittstelle i3. In diesem Beispiel ist AM der Client und RP der Server. Zu Beginn der Kommunikation muss der Client *connect?* zum Server senden. Bei einer negativen Antwort (*connectERROR!*) wird wieder in Zustand *1* gesprungen und bei einer positiven (*connectOK!*) in Zustand *3*. Anschließend muss der Client *establishConnection?* senden und erhält die Antwort *establishConnection!*. Canal et al. [CPS08] definieren die Notation anhand der betrachteten Komponente und deren Ein- und Ausgaben. Im Gegensatz dazu, orientiert sich die hier vorgestellte Notation an den Rollen Client und Server. Deshalb ist eine Nachricht mit Suffix ? immer eine Nachricht vom Client zum Server und das Suffix ! markiert die Gegenrichtung. Der Vorteil dieser Notation ist, dass ein Zustandsautomat gleichermaßen für Client und Server gilt. Diese Notation wird ab hier für den Rest der gesamten Arbeit verwendet. Da der Kommunikationsautomat auf mehreren Schnittstellen basieren kann, muss zum Event und der Richtung des Events noch zusätzlich die betroffene Schnittstelle annotiert werden.

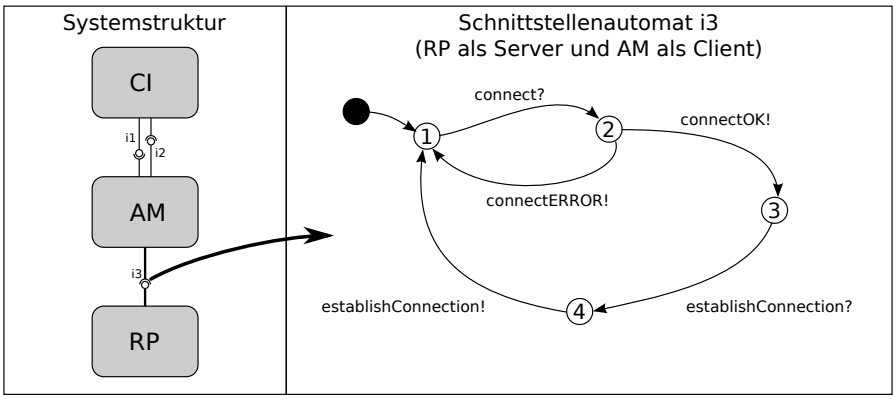


Abbildung 4.3.: Schnittstellenautomat am Beispiel Audiomanagement. Im Gegensatz zu der in Abschnitt 3.2 vorgestellten Notation beschreibt das Suffix ? eine Nachricht vom Client zum Server und das Suffix ! die Gegenrichtung.

Komponentenautomat

Die Beschreibung des Komponentenautomaten basiert auf Mealy Automaten [Mea55]. Im Gegensatz zu den für die Schnittstelle und die Kommunikationsbeziehung verwendeten Akzeptoren, ist dieser Automat ein Transduktor der Ausgaben in Abhängigkeit des aktuellen Zustands und der aktuellen Eingabe generiert. Abbildung 4.4 zeigt einen Mealy-Automaten, der den Kontrollfluss an einer Komponentenschnittstelle beschreibt. Er besteht aus *Zuständen* (Knoten), *Transitionen* (gerichteten Kanten), *Triggern* und *Aktionen*. Dabei stellen die Trigger das Eingabealphabet und die Aktionen das Ausgabealphabet dar. Eine Kantenbeschriftung a/b bedeutet, dass bei Eingabe des Triggers a zusätzlich zum Wechsel des Zustands die Aktion b ausgeführt wird. Aus Sicht der Softwarekomponente sind Trigger eingehende und Aktionen ausgehende Nachrichten. Abbildung 4.4 zeigt den Komponentenautomaten von AM.

AM nimmt von CI Anforderungen zum Wechseln der Audioquelle entgegen (*connectRequest?*) und leitet diese als Aktion an RP weiter (*RP:connect?*). Falls RP signalisiert, dass eine Verbindung möglich ist (*connectOK!*) sendet AM eine Nachricht an CI (*connectRequestOK!*) und initiiert die Verbindung mit RI *establishConnection?*. Nach erfolgreicher Antwort sendet AM noch eine Nachricht, dass sich der Status der Verbindung geändert hat (*connectionStateChanged!*)

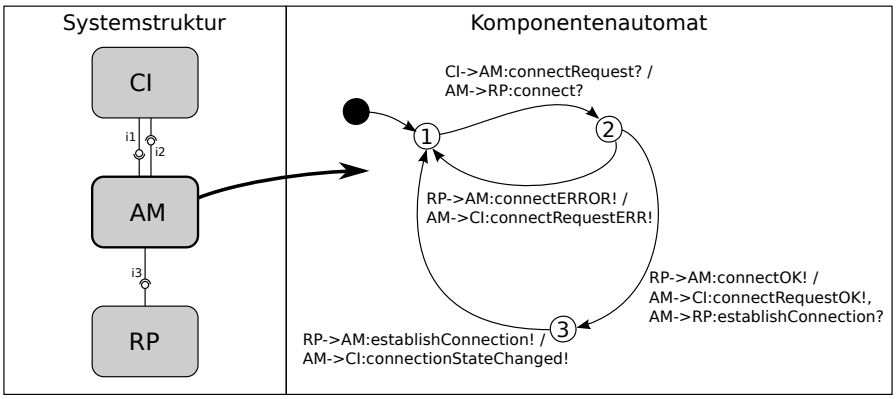
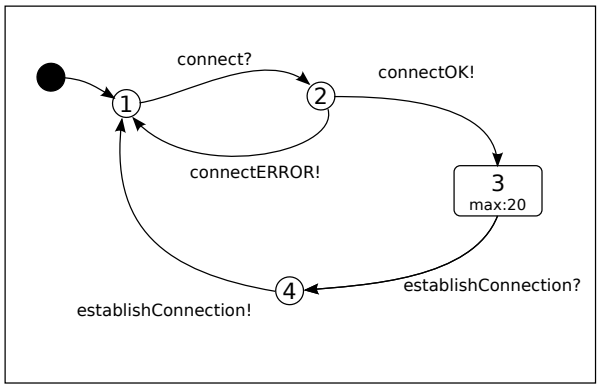


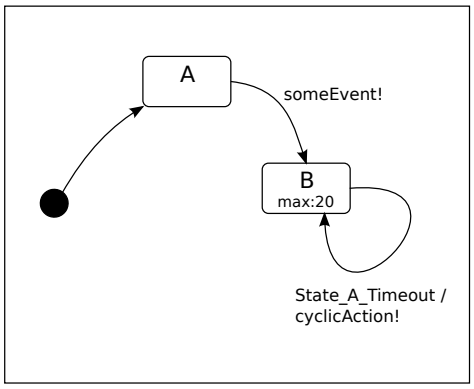
Abbildung 4.4.: Komponentenbezogenes Verhaltensmodell am Beispiel Audiomanagement.

Modellierung von Zeitverhalten

Für die Modellierung von Zeitverhalten werden die in Abschnitt 3.2.1 vorgestellten Invarianten für Zustände verwendet. Im Gegensatz dazu werden nicht beliebige logische Ausdrücke zugelassen, sondern lediglich die Zustände des Automaten mit einem zusätzlichen Attribut versehen. Dieses *max-Attribut* gibt die maximale Verweildauer in einem Zustand in Millisekunden an. Abbildung 4.5a zeigt ein Bei-



(a) Modellierung einer Zeitbedingung im Schnittstellenautomaten am Beispiel eines Verbindungsaufbaues.



(b) Modellierung von Zeitverhalten im Komponentenautomaten mit dem Timeout-Event als Trigger.

Abbildung 4.5.: Modellierung von Zeitverhalten.

spiel für die Modellierung von Zeitverhalten für den Zustand 3. Das max-Attribut

wird auf 20 Millisekunden gesetzt. In diesem Beispiel muss nach dem Eintreten in Zustand 3 spätestens nach 20 Millisekunden der Trigger *establishConnection?* ausgelöst worden sein.

Das max-Attribut ist fest mit einem dazugeörigen *Timeout-Event* verknüpft. Deshalb kann neben der Beschreibung der maximalen Verweildauer, das max-Attribut als Trigger im Zustandsautomaten verwendet werden. Dabei wird angenommen, dass beim Ausführen des Zustandsautomaten mit Eintritt in einen Zustand mit gesetztem max-Attribut ein Timer gestartet wird, der nach Ablauf das entsprechende Timeout-Event ausgibt. Abbildung 4.5b zeigt diese Art der Modellierung am Beispiel eines zyklischen Ablaufes. In diesem Beispiel wird im Zustand B im Zyklus von 20 ms die Ausgabe *cyclicAction!* erzeugt.

Ohne Zweifel erlauben die in Abschnitt 3.2.1 vorgestellten Timed (Interface) Automata durch die Verwendung einer globalen Zeit eine präzisere Beschreibung des zeitlichen Verhaltens. Die hier vorgestellte Art der Modellierung bietet allerdings eine wesentlich einfachere Beschreibung von Zeitbedingungen auf Basis des aktuellen Zustands und adressiert die häufigsten Anwendungsfälle, wie beispielsweise Deadlines und zyklische Aktionen. Wir nehmen dabei an, dass zwei Komponenten nach dem Austausch einer Nachricht zeitgleich den nächsten Zustand betreten.

4.1.3. Ausführungssemantik der DANA Kontrollflussautomaten

Ein Ziel bei der Modellierung des Kontrollflusses ist die spätere Ausführung der Modelle. Zum Beispiel kann aus dem Komponentenautomaten eine Komponente generiert werden, die den spezifizierten Ablauf implementiert. Ein solches Verfahren wird während des Entwicklungsprozesses verwendet, um ein noch unvollständiges System trotzdem mit allen Komponenten testen zu können. Dabei werden die Ausgaben der noch fehlenden Komponenten durch den Automaten generiert. In der Praxis sind solche Komponenten mit einer Warteschlange für eingehende Nachrichten ausgestattet. Diese *asynchrone Ausführungssemantik* ist einfach zu implementieren, birgt allerdings Probleme für eine Analyse des Systems, da zum Beispiel Eigenschaften, wie mögliche *Deadlocks*, auf Modellebene nicht entscheidbar sind [YS97; BZ83].

Deshalb wird in dieser Arbeit die *synchrone Ausführungssemantik* ohne Warteschlange verwendet. Das bedeutet, dass ein Zustandsautomat nur eine Nachricht versenden kann, wenn sich der empfangende Zustandsautomat in einem Zustand befindet, der auch diese Nachricht als Eingabe akzeptiert. Deshalb schalten zum Beispiel zwei Schnittstellenautomaten beim Austausch einer Nachricht synchron

weiter, sodass das Senden und Empfangen einer Nachricht als atomare Aktion gesehen werden kann. Für mehr Informationen zum Thema synchroner und asynchroner Ausführungssemantik wird auf Yellin und Strom [YS97] verwiesen.

4.2. Das hierarchische Schnittstellenmodell

Die bisher beschriebenen Modelle erlauben die Beschreibung von Kontrollfluss und Zeitbedingungen. Die auf den Transitionen annotierten Trigger und Aktionen sind dabei als Strings angegeben und beinhalten noch keine Information, wie ein Trigger auf Basis einer eingehenden Nachricht erkannt, oder wie eine Aktion ausgeführt werden kann. Denn meist ist dafür nicht nur die Betrachtung eines Methodenaufrufs, sondern auch die Betrachtung der übermittelten Parameter notwendig. Diese Lücke zwischen Kontrollflussautomat und Syntaxmodell wird mit dem hierarchischen Schnittstellenautomat geschlossen. Auch wenn in diesem Abschnitt nur der Schnittstellenautomat betrachtet wird, gilt diese Art der Abstraktion auch für Komponenten- und Kommunikationsautomaten.

Wir fassen alle Modelle, die eine Schnittstelle auf logischer Ebene charakterisieren, unter dem Begriff *Schnittstellenmodell* zusammen (siehe Abbildung 4.1). Dazu gehören das *Syntaxmodell*, das *Eventmodell* und der *Schnittstellenautomat*. Für diese drei Modelle definieren wir eine formale Basis, auf der die spätere Konstruktion der Schnittstellenadapter aufbaut.

Definition 4.2.1. ***(Schnittstellenmodell):*** *Ein* Schnittstellenmodell $\mathcal{I} = (\mathcal{I}^S, \mathcal{I}^E, \mathcal{I}^B)$ *ist definiert durch:*

- $\mathcal{I}^S$ *ist das* Syntaxmodell
- $\mathcal{I}^E$ *ist das* Eventmodell
- $\mathcal{I}^B$ *ist der* Schnittstellenautomat

Abbildung 4.6 zeigt das Schnittstellenmodell mit den drei Untermodellen und deren Zusammenhängen. Auf unterster Ebene befindet sich das Syntaxmodell ($\mathcal{I}^S$). Es definiert eine Schnittstelle mithilfe von *Deklarationen* (d) und *Parametern* (p). Das Eventmodell ($\mathcal{I}^E$) auf der zweiten Ebene wird verwendet, um eine einheitliche Abstraktion für den Schnittstellenautomaten durch *Events* zu definieren. Dabei kann jedem Event (e) eine Constraint (c) zugewiesen werden. Der Schnittstellenautomat $\mathcal{I}^B$ auf oberster Ebene, spezifiziert die gültige Aufrufreihenfolge der syntaktischen Deklarationen mithilfe der in $\mathcal{I}^E$ definierten Events. Dabei kann das Eventmodell $\mathcal{I}^E$ als eine Art *Übersetzer* zwischen den in $\mathcal{I}^S$ definierten Deklarationen und den in $\mathcal{I}^B$ verwendeten Events gesehen werden. Die konkrete Ausprägung des Syntaxmodells hängt stark von der verwendeten Middleware (MW) ab,

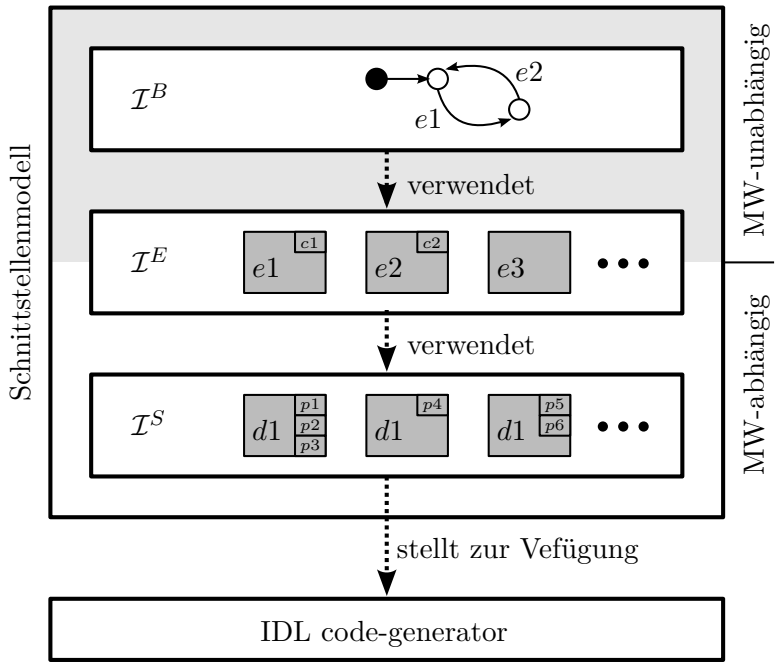


Abbildung 4.6.: Aufbau des hierarchischen Schnittstellenmodells mit drei Untermodellen [Pra+14].

denn aus ihm wird der Code für Server und Client Stubs generiert. Das Eventmodell ist teilweise middlewareabhängig, denn es verwendet die Elemente des darunter liegenden Syntaxmodells. Der Schnittstellenautomat hingegen enthält keinerlei Abhängigkeiten zur verwendeten Middleware, denn er greift auf die einheitliche Abstraktion der Events zu.

4.2.1. Das Syntaxmodell

Das Syntaxmodell ($\mathcal{I}^S$) beschreibt die über die Schnittstelle ausgetauschten Daten. Dabei abstrahiert das Syntaxmodell existierende Interfacebeschreibungssprachen (IDLs). Der primäre Zweck einer IDL ist es, eine maschinell verarbeitbare abstrakte Beschreibung der syntaktischen Schnittstelle zu bieten, die eine Generierung von Binärcode für eine Zielplattform ermöglicht. Gängige IDLs bieten für diesen Zweck bereits Codegeneratoren für verschiedene Middlewarelösungen oder Programmiersprachen. Das Ziel bei der Wiederverwendung der IDLs auf syntaktischer Ebene ist die Wiederverwendung der ensprechenden Codegeneratoren. So wird sichergestellt, dass die Konzepte auf eine große Anzahl von Zielplattformen mit unterschiedlichen

Kommunikationstechnologien anwendbar sind.

Die Erfahrung hat gezeigt, dass oft mit einer neuen Kommunikationsmiddleware Anpassungen an einer IDL oder gar die Neudefinition einer für die Kommunikationsmiddleware speziellen IDL einhergehen. Deshalb wird das Syntaxmodell sehr abstrakt definiert und muss für eine konkrete Anwendung für eine spezielle IDL konkretisiert werden. Dies garantiert eine möglichst universelle Einsetzbarkeit des Syntaxmodells. Die Hauptelemente von $\mathcal{I}^S$ sind *Deklarationen*. Deklarationen sind die zentralen syntaktischen Schnittstellenelemente, welche zum Versandt oder Empfang von Daten genutzt werden.

Definition 4.2.2. ***(Syntaxmodell):*** *Ein* Syntaxmodell $\mathcal{I}^S = (D, P, T, m^D, m^T)$ *ist definiert durch:*

- $D = D^{\text{in}} \dot{\cup} D^{\text{out}}$ *ist die Menge von Deklarationen, die aus disjunkten Mengen von Eingabe- (D^{in}) und Ausgabedeklarationen (D^{out}) besteht.*
- $P = P^{in} \dot{\cup} P^{\text{out}}$ *ist die Menge von Parametern, die aus disjunkten Mengen von Eingabe- (P^{in}) und Ausgabeparametern (P^{out}) besteht.*
- T *ist die Menge der Datentypen.*
- $m^D : P \to D$ *ist die Funktion, die jedem Parameter eine Deklaration zuordnet. Eingabeparameter können nur Eingabedeklarationen, und Ausgabeparameter nur Ausgabedeklarationen zugeordnet werden:*

$$\forall d \in D : \left\{ \begin{array}{l} \Big(\exists p \in P^{in}, m^D(p) = d\Big) \Rightarrow \Big(d \in D^{in}\Big) \\ \Big(\exists p \in P^{out}, m^D(p) = d\Big) \Rightarrow \Big(d \in D^{out}\Big) \end{array} \right\}$$

  Die Parameter, die zu einer Deklaration d gehören, werden mit $P_d = (m^D)^{-1}(d)$ definiert.
- $m^T : P \to T$ *ist die Funktion, die jedem Parameter einen Datentyp zuordnet.*

Diese abstrakte Definition erlaubt es, das Syntaxmodell für verschiedene Aufrufsemantiken zu verwenden. Syntaxmodelle im Infotainment setzen zum Beispiel auf eine methodenbasierte und im AUTOSAR [GbR14a] Umfeld auf eine signalbasierte Beschreibung. Zum Beispiel erfordert ein Methodenaufruf als Deklaration mit

Eingabe- und Ausgabeparametern sowohl eine Deklaration für die Nachricht zum Server und eine für die Antwort zum Client. AUTOSAR Signale hingegen können mit einer einzelnen Ausgabedeklaration modelliert werden und besitzen nur Ausgabeparameter. Diese Beispiele zeigen, dass das allgemein gehaltene $\mathcal{I}^S$ auf unterschiedliche Aufrufsemantiken der jeweils konkreten Schnittstellenbeschreibungssprache angepasst werden kann.

Für eine Schnittstelle wird die Richtung von Ein- und Ausgabedeklarationen aus Sicht der Softwarekomponente, welche die Schnittstelle anbietet (Server), definiert. Deshalb sind für den weiteren Verlauf der Arbeit Eingabedeklarationen Nachrichten zum Server und Ausgabedeklarationen Nachrichten, die der Server sendet. Für eine auf Methodenaufrufen basierende Kommunikation bedeutet das, dass der Aufruf einer Methode eine Eingabe, und die Antwort eine Ausgabe darstellt. Bevor die nächste Modellebene betrachtet werden kann, wird nun der Begriff des *Deklarationsaufrufs* definiert.

Definition 4.2.3. ***(Deklarationsaufruf):*** *Ein Aufruf der Deklaration d ist definiert durch die Funktion:*

$$\phi_d : P_d \rightarrow \hat{P}_d$$

Diese ordnet jedem Deklarationsparameter $p \in P_d$ einen Wert aus seiner Zielmenge, $\hat{p}$, zu. Dabei ist $\hat{P}_d = \hat{p}_1 \cup \ldots \cup \hat{p}_k, p_n \in P_d$ die Menge aller möglichen Parameterbelegungen. Φ_d bezeichnet die Menge aller möglichen Deklarationsaufrufe für die Deklaration d und Φ die Menge aller Deklarationsaufrufe einer Schnittstelle.

4.2.2. Das Eventmodell

Das Hauptziel bei der Komponentenmodellierung mit DANA ist die Definition des Kontrollflusses auf Basis von Deklarationsaufrufen. In der Praxis angewandte Ansätze, wie zum Beispiel UML State Machines [OMG10], beschreiben den Kontrollfluss direkt auf Basis der syntaktischen Schnittstelle. Für eine Transition kann in UML ein *Trigger* in Form eines *Signal Events* definiert werden, der mithilfe einer *Guard Condition* noch genauer spezifiziert werden kann. Der Trigger kann beispielsweise auf eine aufgerufene Methode verweisen, und die Guard Condition überprüft zusätzliche Variablen auf die Einhaltung einer spezifizierten Bedingung. Besonders für die Guard Conditions gelten kaum Einschränkungen und der Model-

lierer kann oft beliebige Programme definieren. Die Erstellung und Analyse eines solchen Modells ist sehr zeitintensiv und ressourcenaufwändig.

Deshalb wird mit dem Eventmodell $\mathcal{I}^E$ eine Ebene zwischen Syntaxmodell und Schnittstellenautomat definiert, welche es erlaubt, Deklarationsaufrufe mithilfe von *Events* zu klassifizieren. Diese Art der Modellierung hat zwei große Vorteile: Erstens, können so definierte Events in einem Kontrollflussautomaten ohne Angabe von zusätzlichen Bedingungen referenziert und wiederverwendet werden. Zweitens, definiert das Eventmodell, wie ein Event wieder in einen Deklarationsaufruf übersetzt werden kann. Dies ist insbesondere für die Ausführung der Automaten wichtig. Somit kann ein Eventmodell als surjektive Funktion betrachtet werden, welche Deklarationsaufrufe in Events übersetzt.

$$\mathcal{I}^E : \Phi \to E$$

Die Umkehrfunktion $(\mathcal{I}^E)^{-1}$ erlaubt es, für ein Event Deklarationsaufrufe zu erhalten. Die Übersetzung eines Deklarationsaufrufs in ein Event wird durch eine *Constraint* und die Gegenrichtung durch eine *Emulation* beschrieben.

Definition 4.2.4. ***(Eventmodell):*** *Sei* $\mathcal{I}^S = (D, P, T, m^D, m^T)$ *ein* Syntaxmodell*, dann ist das* Eventmodell $\mathcal{I}^E = (E, C, X, e^D, e^C, e^X, e^H)$ *definiert durch:*

- $E = E^{\mathcal{I}} \dot{\cup} E^{\mathcal{T}}$ *ist die Menge von* Events*. Sie besteht aus disjunkten Mengen von* Schnittstellen- *(*$E^{\mathcal{I}}$*) und* Timeoutevents *(*$E^{\mathcal{T}}$*). Schnittstellenevents lassen sich in* Ein- *und* Ausgabeevents *unterteilen (*$E^{\mathcal{I}} = E^{\text{in}} \dot{\cup} E^{\text{out}}$*).*
- $e^D : E^{\mathcal{I}} \to D$ *ordnet jedem Schnittstellenevent eine Deklaration aus* $\mathcal{I}^S$ *zu.*
- C *ist die Menge der* Constraints*. Für ein Schnittstellenevent* $e \in E^{\mathcal{I}}$ *mit* $e^D(e) = d$ *ist die Constraint* $c_e \in C$ *definiert durch einen logischen Ausdruck* f *mit den Parametern* P_d*:*

$$c_e := f(P_d)$$

- X *ist die Menge von Emulationen. Für ein Schnittstellenevent* $e \in E^{\mathcal{I}}$ *mit* $e^D(e) = d$ *definiert die Emulation* $x_e \in X$*, wie alle Deklarationsparameter (*P_d*) gesetzt werden müssen, um die entsprechende Deklaration* d *aufzurufen.* x_e *definiert deshalb für jeden Parameter aus* P_d *eine Konstante (*g^{const}*) und einen optionalen arithmetischen Ausdruck (*g^{func}*) zur Berechnung des Parameters mithilfe von beliebigen Schnittstellenparametern* P*:*

$$x_e : P_d \to \left\{ (g^{\text{const}}, g^{\text{func}}) : \begin{matrix} g^{\text{const}} & \textit{ist eine Konstante,} \\ g^{\text{func}} & \textit{ist ein optionaler arithmetischer} \\ & \textit{Ausdruck über } P \end{matrix} \right\}$$

- $e^C : C \to E^{\mathcal{I}}$ *ist die injektive Funktion, die Constraints auf Schnittstellenevents abbildet.*
- $e^X : X \to E^{\mathcal{I}}$ *ist die injektive Funktion, die Emulationen auf Schnittstellenevents abbildet.*
- $e^H : E^{\mathcal{I}} \to \{E^{\mathcal{I}} \bigcup \bot\}$ *ist die Funktion, die Events ein Elternelement zuordnet, um eine hierarchische Event-Struktur zu schaffen.* $\bot$ *ist als Wurzel des Baumes das undefinierte Elternelement und wird für Events auf oberster Ebene verwendet. Dementsprechend gibt die Umkehrfunktion* $(e^H)^{-1}$ *alle Kinder eines Events zurück.*

Die hierarchische Definition von Events mit der Funktion e^H erlaubt es, zu be-

rechnen, welche Events kompatibel zueinander sind, und welche nicht. Zur Berechnung dieser Eigenschaft wird die reflexiv-transitive Hülle der Hierarchiefunktion (e^{H*}) oder die der Umkehrfunktion($((e^H)^{-1})^*$) verwendet. Falls ein Event $a \in E^I$ von einer Komponente als Eingabe erwartet wird, sind beim aktuellen Zustand der Nachbarkomponente alle Kinder des Events und das Event selbst $e' \in ((e^H)^{-1})^*(a)$ als Ausgabe erlaubt. Abbildung 4.7 zeigt ein Beispiel für ein hierarchisches Eventmodell. Für jedes Eventmodell gilt folgende Bedingung für die Funktion e^H:

$$\forall e1, e2 \in E^I : \left\{ \begin{array}{c} \Big(e^H(e1) = e^H(e2) = \perp \Big) \Rightarrow \Big(e^D(e1) \neq e^D(e2)\Big) \\ \Big(((e^{H*}(e1) \cap e^{H*}(e2)) \setminus \perp \neq \emptyset\Big) \Rightarrow \Big(e^D(e1) = e^D(e2)\Big) \end{array} \right\}$$

Somit müssen alle Events im selben Ast des Baumes auf dieselbe Deklaration verweisen und Events in unterschiedlichen Ästen auf verschiedene Deklarationen. Abbildung 4.7 zeigt ein Beispiel für ein hierarchisches Eventmodell. Um die Bedingung für die Relation e^H zu erfüllen, müssen die Events *e3* und *e4* auf dieselbe Deklaration verweisen wie das Event *e1*: $e^D(e1) = e^D(e3) = e^D(e4)$. Alle Events auf oberster Ebene müssen auf eine unterschiedliche Deklaration verweisen. Deshalb gilt: $e^D(e1) \neq e^D(e2)$.

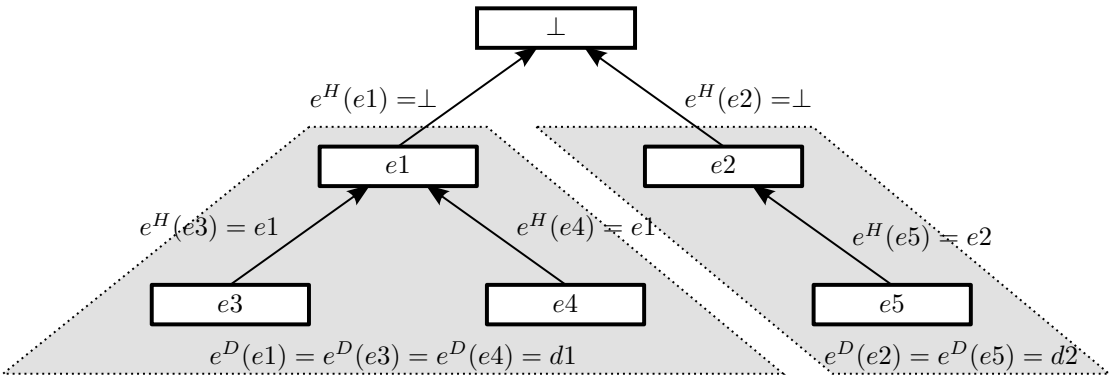


Abbildung 4.7.: Beispiel für die Hierarchie-Beziehung im Eventmodell.

Die Richtung von Ein- und Ausgaben wird analog zu den in Abschnitt 4.2.2 definierten Deklarationen festgelegt. Deshalb sind für den weiteren Verlauf der Arbeit Eingabeevents Nachrichten vom Client zum Server und Ausgabeevents Nachrichten, die der Server sendet.

4.2.3. Der Schnittstellenautomat

Der *Schnittstellenautomat* $\mathcal{I}^B$ beschreibt gültige Abfolgen von Events zwischen einer Komponente, die eine Schnittstelle anbietet, und einer, die sie benötigt. Wie in Abschnitt 4.1.2 beschrieben, werden dafür akzeptierende endliche Automaten verwendet. Die Automaten besitzen keinen Endzustand und bleiben somit nach dem Start aktiv.

Definition 4.2.5. ***(Schnittstellenautonmat):***
Sei $\mathcal{I}^E = (E, C, X, e^D, e^C, e^X, e^H)$ *ein Eventmodell, dann ist der* Schnittstellenautomat $\mathcal{I}^B = (S, s^0, T, \delta, \tau)$ *definiert durch:*

- S *ist die Menge von* Zuständen.
- $s^0 \in S$ *ist der Startzustand.*
- $T \subseteq E$ *ist die Menge von Triggern.*
- $\delta : S \times T \rightsquigarrow S$ *ist die partielle Transitionsfunktion.*
- $\tau : S \rightsquigarrow \mathbb{N} \times E^{\mathcal{T}}$ *ist die partielle Funktion, die einem Zustand eine natürliche Zahl (Timeout) und ein Timeoutevent zuordnet.*

Das Eingabealphabet des Automaten ist eine Untermenge der in $\mathcal{I}^E$ spezifizierten Events. Zeitverhalten wird mit dem in Abschnitt 4.1.2 beschriebenen Mechanismus modelliert, indem einem Zustand eine natürliche Zahl zugeordnet werden kann, die die maximale Verweildauer in diesem Zustand beschreibt. Zusätzlich wird durch die Funktion τ auch ein entsprechendes Timeoutevent zugeordnet, das ausgegeben wird, falls die Verweildauer überschritten wird. Wie jedes andere Event, kann auch dieses als Trigger für Transitionen verwendet werden.

Abbildung 4.8 zeigt ein Beispiel für einen Schnittstellenautomaten. In Zustand 1 wird ein Timeout defninert. Diesem Zustand wurde auch das Timeoutevent *1_timeout* zugeordnet. Dieses Event wird in der Ausgangstransition als Trigger verwendet. Deshalb wird beim Ausführen des Modells der Zustand 2 genau nach 12 Millisekunden betreten. Als nächstes Event wird erwartet, dass der Client an den Server *a?* sendet und darauf vom Server *a!* als Antwort erhält.

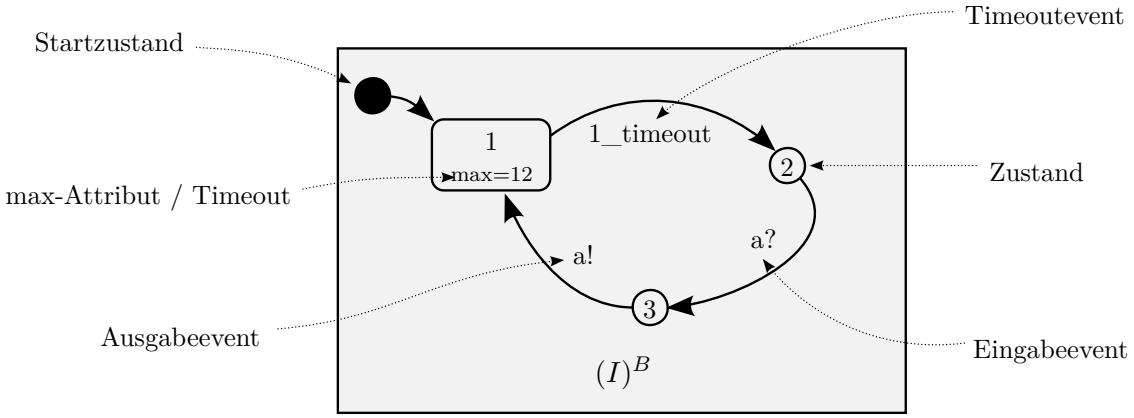


Abbildung 4.8.: Elemente und Notation des Schnittstellenautomaten.

4.2.4. Code-Generierung aus dem Syntaxmodell

Der hier vorgestellte Modellierungsansatz zielt darauf ab, bestehende IDLs als Syntaxmodelle wiederverwenden zu können. Ein Vorteil ist die Wiederverwendung der für die Modellierungssprache bereits existierenden und auf die Zielplattform abgestimmten Code-Generatoren. Der Code-Generator erstellt üblicherweise zwei Stubs, einen für den Server und einen für den Client. Dieser Generierungsschritt stellt sicher, dass die Schnittstelle auf syntaktischer Ebene von beiden Komponenten eingehalten wird.

Alle Informationen, die zusätzlich zum Syntaxmodell für die Generierung von Quelltext notwendig sind, werden im *Deploymentmodell* beschrieben. Franca IDL [Bir14] stellt für den Bereich Infotainment bereits ein textuelles Modell zur Verfügung, welches zusätzlich zur syntaktischen Schnittstellenbeschreibung ein separates Deploymentmodell spezifiziert. Mit diesem Modell kann definiert werden, wie die Ausprägung einer Schnittstelle in Bezug auf die Zielplattform aussieht. Für das Deployment einer Schnittstelle auf eine Zielplattform mit Media Orientend Systems Transport (MOST) [Coo10] muss zum Beispiel zusätzlich zu dem Namen einer Methode oder eines Parameters auch eine numerische ID vergeben werden, während das Deployment auf Linux D-Bus [Pen+14] dies nicht verlangt.

Abbildung 4.9 zeigt ein Beispiel für die Code-Generierung für drei verschiedene Zielplattformen. Im Syntaxmodell wird die Schnittstelle sehr abstrakt mit Deklarationen, Parametern und Datentypen beschrieben. Auf der nächsten Ebene folgt eine Konkretisierung mithilfe einer konkreten IDL. Diese Sprache bildet die syntaktischen Elemente einer konkreten Kommunikationsmiddleware ab. Im Falle von D-Bus wird zum Beispiel Franca IDL verwendet, im Falle von MOST den Funk-

tionskatalog und im Falle von ETCH die Network Service Definition Language (NSDL). Meist lässt sich direkt aus dem konkreten Syntaxmodell Quelltext generieren. Manchmal ist von der IDL ein zusätzliches Deploymentmodell vorgesehen, wie im Falle von Franca IDL.

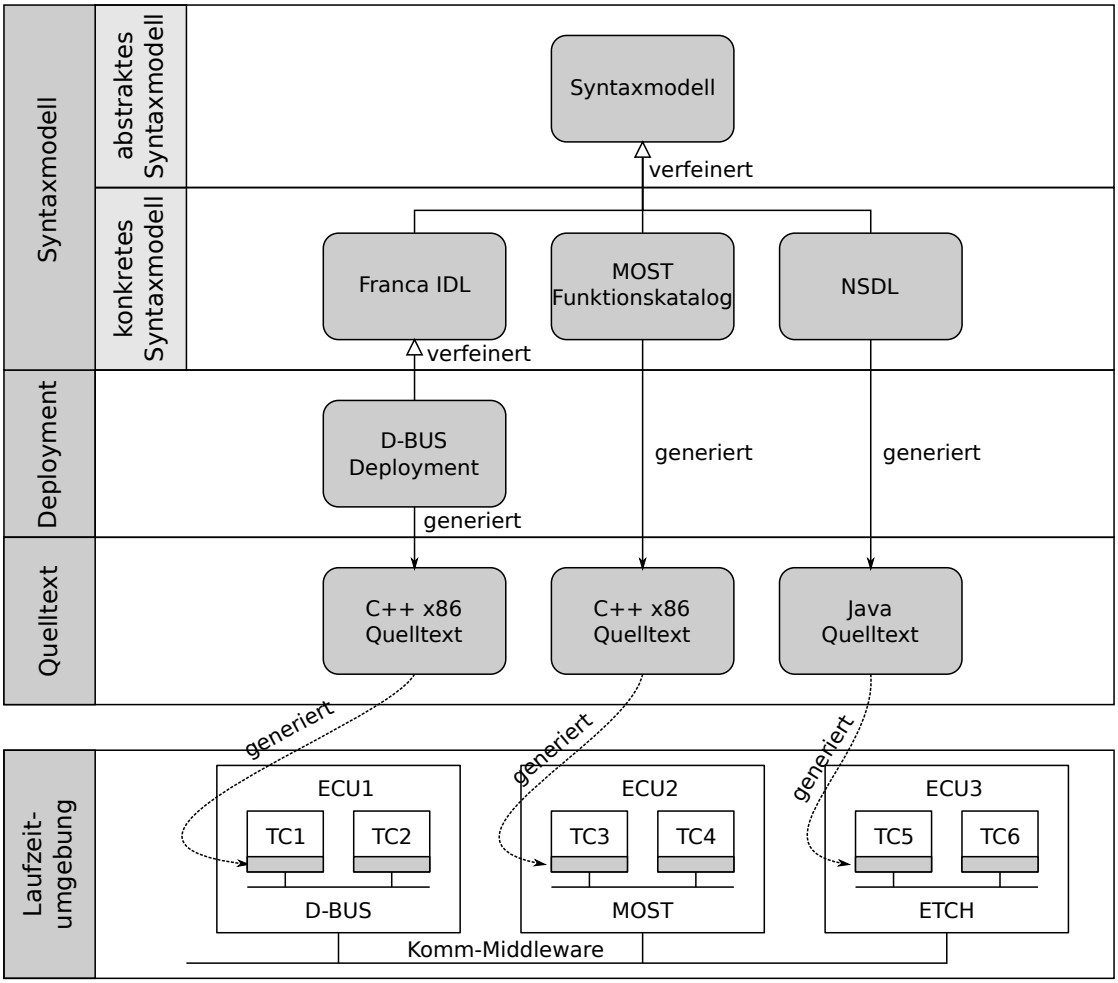


Abbildung 4.9.: Code-Generierung aus dem Syntaxmodell durch die Verwendung der IDL-Code-Generatoren.

4.3. Kompatibilität von hierarchischen Schnittstellenmodellen

Durch die hierarchische Definition des Schnittstellenmodells lassen sich die Ebenen gut in die in Abschnitt 2.2 vorgestellten Kompatibilitätsebenen einordnen. Bechter [Bec08] definiert Kompatibilität für einen den *Timed Safety Automata* [Hen+94] ähnlichen Formalismus. Timed Safety Automata sind ein sehr mächtiger Formalismus, der eine präzisere Definition der zeitlichen Abläufe ermöglicht, als die hier vorgestellten Schnittstellenautomaten. Da diese in den Formalismus der Timed Safety Automata transformiert werden können, gelten ähnliche Kompatibilitätseigeschaften. In dieser Arbeit werden die von Bechter definierten Kompatibilitätseigenschaften um die Ebenen der syntaktischen und der technischen Kompatibilität erweitert und die darüber-liegenden Ebenen konkretisiert. Abbildung 4.10 zeigt verschiedene Fehler, die eine Inkompatibilität hervorrufen können, und ordnet sie dem jeweiligen Modell zu. Die dargestellten Fehlerklassen werden in den folgenden Abschnitten genauer beschrieben.

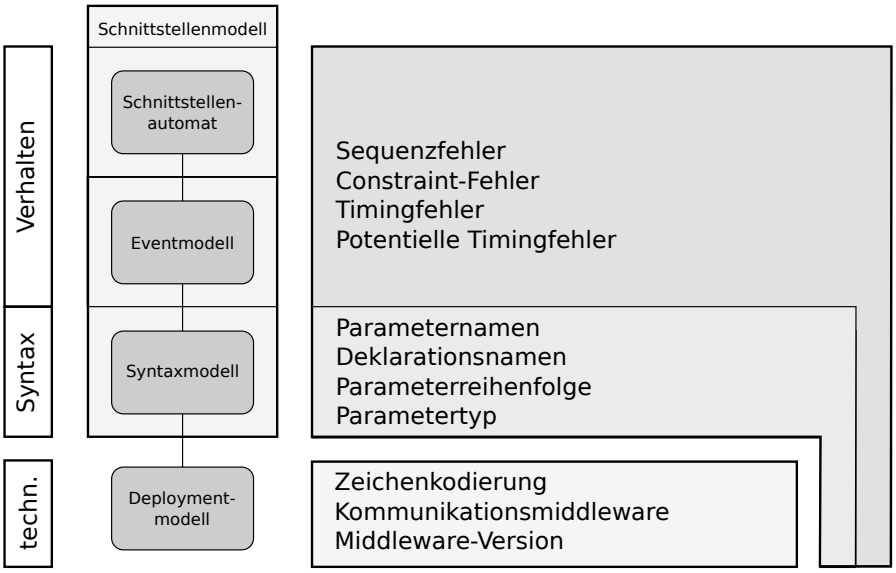


Abbildung 4.10.: Zuordnung von möglichen Fehlern zu den Ebenen des Schnittstellenmodells.

Technische Fehler: Technische Fehler sind all jene Fehler, die sich bei kompatiblen Syntaxmodellen erst nach der Code-Generierung und Übersetzung des

Quelltextes ergeben. Dazu gehören zum Beispiel unterschiedliche Zeichencodierungen, Übertragungsgeschwindigkeiten, Middleware-Technologien oder Middleware-Versionen. Viele dieser Fehler können durch die Verwendung des richtigen Code-Generators und Compilers mit den richtigen Einstellungen ausgeschlossen werden.

Syntaktische Fehler: Syntaktische Fehler ergeben sich zum Beispiel, wenn dieselbe Deklaration oder derselbe Parameter mit unterschiedlichem Namen definiert wurden. Bei einigen Middleware-Lösungen wird zum Beispiel der Namen von Parametern nicht direkt mit der Nachricht versandt. Falls sich die Anzahl und die Reihenfolge der Parameter nicht unterscheidet, hat in diesem Fall eine syntaktische Inkompatibilität nicht zwangsläufig einen Kommunikationsfehler zur Folge. Zusätzlich zählen zu den syntaktischen Fehlern auch eine falsche Parameterreihenfolge innerhalb einer Deklaration.

Typfehler: Ein Typfehler liegt vor, wenn der Datentyp eines Ausgabeparameters nicht zum Datentyp eines Eingabeparameters passt oder umgekehrt. Auch hier gilt: Je nach verwendeter Kommunikationsmiddleware müssen die verwendeten Datentypen nicht dieselben sein, um kompatibel zu sein.

Sequenzfehler: Sequenzfehler liegen vor, falls die Reihenfolge der gesendeten Events nicht mit der erwarteten Reihenfolge der empfangenden Komponente übereinstimmt. Für Schnittstellenautomaten bedeutet das konkret, dass in einem Zustand nicht alle in diesem Zustand möglichen Eingabeevents als Trigger spezifiziert sind. Abbildung 4.11 zeigt ein Beispiel für einen Sequenzfehler. Der Server erwartet in Zustand *a* eine Sequenz aus den Events *b?*. Der Client versendet aber bereits als zweite Nachricht das Event *a?*. Die Erkennung einer derartigen Inkompatibilität kann bereits auf Modellebene durch die Berechnung des synchronen Produktes zweier Kontrollflussautomaten [CPS08] erfolgen.

Constraint-Fehler: Der Constraint-Fehler gehört zur Menge der Sequenzfehler. Diese Fehlerklasse wird eigenständig betrachtet, denn das DANA Eventmodell besitzt durch die Hierarchie eine besondere Eigenschaft zur Erkennung dieser Fehlerklasse. Ein Constraint-Fehler liegt dann vor, wenn ein Event *e1* erwartet wird, ein Event *e2* versandt wird und folgende Bedingung nicht erfüllt ist: $e1 \in e^{H*}(e2)$. Das bedeutet, dass ein gesendetes Event mit einem erwarteten kompatibel ist, wenn es das erwartete Event selbst ist, oder in der Hierarchie unterhalb steht.

Timingfehler: "Ein Timingfehler existiert, wenn die Sequenz der Nachrichten korrekt ist, diese aber zum falschen Zeitpunkt gesendet werden" [Bec08]. Grund-

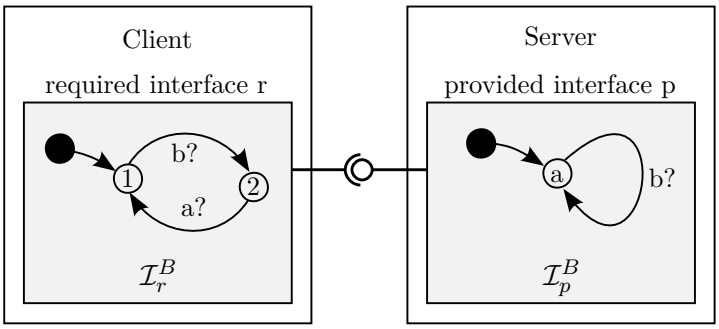


Abbildung 4.11.: Sequenzfehler zwischen einem Client und einem Server.

sätzlich existiert bei der Modellierung von Schnittstellenautomaten ein Timingfehler, falls bei der Ausführung des Automaten ein Timoutevent nicht konsumiert wird. Das passiert, falls ein Zustand mit Timeout über die im Timeout spezifizierte Zeit hinaus aktiv bleibt, und anschließend der emittierte Timeout keine ausgehende Transition triggert. Abbildung 4.12 zeigt ein Beispiel für einen Timingfehler. Der Client versendet das Event *b?* im Zustand *2* erst nach dem abgelaufenen Timout aus State *1*, welcher *20 ms* beträgt. Der Server erwartet aber bereits nach spätestens *10 ms* dieses Event. Wie in Abschnitt 4.1.3 beschrieben, wird von einer synchronen Ausführungssemantik ausgegangen, in der die Zustände 1 und a zeitgleich betreten und die Timer zeitgleich gestartet werden.

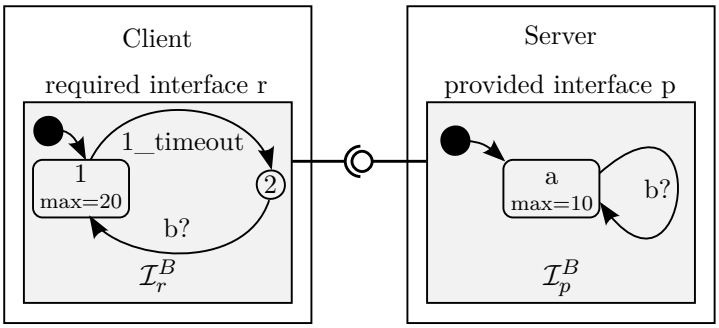


Abbildung 4.12.: Timingfehler zwischen einem Client und einem Server.

Potentieller Timingfehler: „Ein potentieller Timingfehler liegt vor, wenn die Schnittstellenspezifikationen Zeiträume für das Senden und Empfangen von Nachrichten spezifizieren, die sich nur teilweise überlappen“ [Bec08]. Abbildung 4.13 zeigt ein Beispiel für einen potentiellen Timingfehler. Der Server erwartet spätestens nach *10 ms* das Event *b?*. Für den Client gilt eine Deadline von *12 ms*. So

kann es passieren, dass der Client die Nachricht zu spät an den Server versendet.

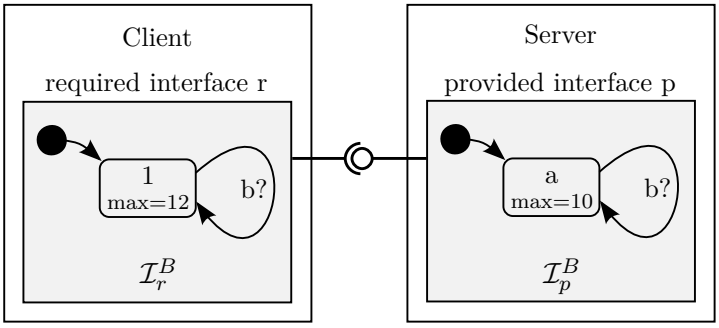


Abbildung 4.13.: Potentieller Timingfehler zwischen einem Client und einem Server.

4.4. Fallbeispiel Parkassistent: Schnittstellenmodellierung

Dieser Abschnitt beschreibt die Anwendung des Modellierungsansatzes an zwei Beispielschnittstellen. Die Zieldomäne ist die Interprozesskommunikation in einer Fahrzeug-Headunit. Das hier beschriebene Beispiel ist eine erweiterte Variante des in [PB13; Pra+14] vorgestellten *Parkassistenten* und wird in der gesamten Arbeit zur Erklärung der Konzepte verwendet. Das Beispiel wurde konstruiert, um die Konzepte dieser Arbeit zu veranschaulichen und entspricht nicht den Softwarekomponenten realer Fahrzeuge.

Der Parkassistent hilft dem Fahrer, Hindernisse im Umfeld des Fahrzeuges mithilfe von Sensoren wahrzunehmen. Abbildung 4.14 zeigt die technischen Softwarekomponenten. Der Server ist dafür zuständig, dass aktuelle Daten über Ethernet von den Sensoren eingelesen und den Clients über den D-Bus zur Verfügung gestellt werden. Je nachdem, welches SENSORS-Steuergerät im Fahrzeug verbaut ist, ist ein anderer ParkA Server installiert, der HEADUNIT-intern auch eine andere Schnittstelle zur Verfügung stellt. Somit müssen die Client-Applikationen *Client1* und *Client2* eine andere Schnittstelle ansprechen, um Sensorwerte zu erhalten.

Das Beispiel besteht aus zwei sich unterscheidenden Versionen der Schnittstelle (`ParkA1` und `ParkA2`). Beide Schnittstellen ermöglichen das Einlesen von Sensorwerten, unterscheiden sich aber in ihrer Syntax und dem Kontrollfluss. `ParkA1` stellt sechs Sensorwerte zur Verfügung und definiert ein einfaches Protokoll für

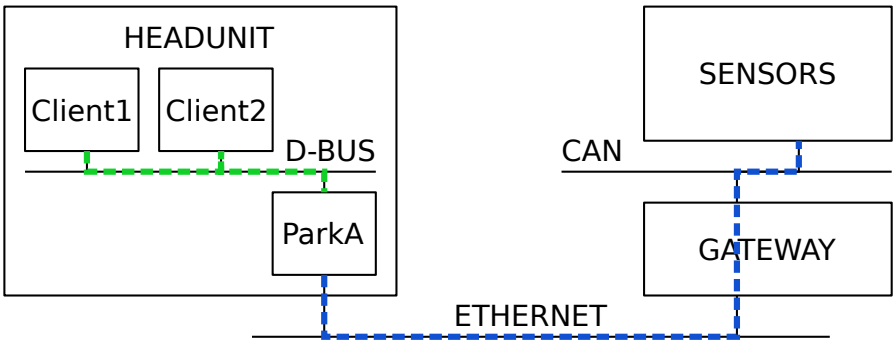


Abbildung 4.14.: Beispiel Parkassistent. Der ParkA Server holt und plausibilisiert Sensorwerte vom *SENSORS*-Steuergerät (blaue Verbindung) und stellt diese innerhalb der *HEADUNIT* für verschiedenen Clients zur Verfügung (grüne Verbindung).

das Initialisieren und Beenden einer Verbindung. `ParkA2` stellt acht Sensorwerte zur Verfügung und besitzt erweiterte Mechanismen zur Fehlerbehandlung beim Initialisieren. Es wird angenommen, dass Methodenaufrufe synchron erfolgen und deshalb der Client bis zum Erhalt der Antwort blockiert.

Die erste Schnittstelle, `ParkA1`, spezifiziert zwei Methoden und zwei Broadcasts. Methoden besitzen jeweils eine Deklaration für den Aufruf und eine für die Antwort. Die einzige Deklaration mit zugeordneten Parametern ist der `sensorStatus` Broadcast. Er gibt zyklisch die Sensorwerte zusammen mit einem Parameter, welcher die Qualität der Sensorwerte anzeigt, zurück. Die `startup`- und die `shutdown` Methode werden verwendet, um den Server zu initialisieren und zu beenden und besitzen keine Parameter. Das Modell 4.4.1 beschreibt das Syntaxmodell für `ParkA1`. Datentypen werden in diesem Beispiel nicht betrachtet. Das Eventmodell für `ParkA1` wird in Modell 4.4.2 gezeigt.

Modell 4.4.1. ***(Syntaxmodell von ParkA1):***

$$\mathcal{I}^S_{\text{ParkA1}} = (D_{\text{ParkA1}}, P_{\text{ParkA1}}, m^D_{\text{ParkA1}}, m^T_{\text{ParkA1}})$$

$$\begin{aligned}
D^{\text{in}}_{\text{ParkA1}} &= \{\mathit{startupCall}, \mathit{shutdownCall}\} \\
D^{\text{out}}_{\text{ParkA1}} &= \{\mathit{startupResp}, \mathit{shutdownResp}, \mathit{sensorStatusCB}, \mathit{startedCB}\} \\
P^{\text{in}}_{\text{ParkA1}} &= \{\} \\
P^{\text{out}}_{\text{ParkA1}} &= \{\mathit{front_left}, \mathit{front_mid}, \mathit{front_right}, \mathit{rear_left}, \mathit{rear_mid}, \mathit{rear_right}, \mathit{quality}\} \\
m^D_{\text{ParkA1}} &= \{(\mathit{front_left}, \mathit{sensorStatusCB}), (\mathit{front_mid}, \mathit{sensorStatusCB}), \\
&\quad (\mathit{front_right}, \mathit{sensorStatusCB}), (\mathit{rear_left}, \mathit{sensorStatusCB}), \\
&\quad (\mathit{rear_mid}, \mathit{sensorStatusCB}), (\mathit{rear_right}, \mathit{sensorStatusCB}), \\
&\quad (\mathit{quality}, \mathit{sensorStatusCB})\}
\end{aligned}$$

Modell 4.4.2. ***(Eventmodell von ParkA1):***

$$\mathcal{I}^E_{\text{ParkA1}} = (E_{\text{ParkA1}}, C_{\text{ParkA1}}, X_{\text{ParkA1}}, e^D_{\text{ParkA1}}, e^C_{\text{ParkA1}}, e^X_{\text{ParkA1}}, e^H_{\text{ParkA1}})$$

$$\begin{aligned}
E^{\text{in}}_{\text{ParkA1}} &= \{\mathit{startup?}, \mathit{shutdown?}\} \\
E^{\text{out}}_{\text{ParkA1}} &= \{\mathit{startup!}, \mathit{shutdown!}, \mathit{sensorStatus!}, \mathit{started!}\} \\
E^{\mathcal{T}}_{\text{ParkA1}} &= \{\mathit{sensorsTimer}\} \\
C_{\text{ParkA1}} &= \{\} \\
X_{\text{ParkA1}} &= \{\} \\
e^D_{\text{ParkA1}} &= \{(\mathit{startup?}, \mathit{startupCall}), (\mathit{startup!}, \mathit{startupResp}), (\mathit{shutdown?}, \mathit{shutdownCall}), \\
&\quad (\mathit{shutdown!}, \mathit{shutdownResp}), (\mathit{sensorStatus!}, \mathit{sensorStatusCB}), \\
&\quad (\mathit{started!}, \mathit{startedCB})\} \\
e^C_{\text{ParkA1}} &= \{\} \\
e^X_{\text{ParkA1}} &= \{\} \\
e^H_{\text{ParkA1}} &= \{(\mathit{startup}?, \bot), (\mathit{shutdown}?, \bot), (\mathit{startup}!, \bot), \\
&\quad (\mathit{shutdown}!, \bot), (\mathit{sensorStatus}!, \bot), (\mathit{started}?, \bot)\}
\end{aligned}$$

Modell 4.4.3 zeigt den Schnittstellenautomaten von `ParkA1` und Abbildung 4.15 die grafische Repräsentation des Modells.

Modell 4.4.3. ***(Schnittstellenautomat von ParkA1):***

$$\mathcal{I}^{B}_{\text{ParkA1}} = (S_{\text{ParkA1}}, s^{0}_{\text{ParkA1}}, T_{\text{ParkA1}}, \delta_{\text{ParkA1}}, \tau_{\text{ParkA1}})$$

$$\begin{aligned}
S_{\text{ParkA1}} &= \{1, 2, 3, 4\} \\
s^{0}_{\text{ParkA1}} &= 1 \\
T_{\text{ParkA1}} &= E_{\text{ParkA1}} \\
\delta_{\text{ParkA1}} &= \{(1, \mathit{startup}?, 2), (2, \mathit{startup}!, 3), (3, \mathit{started}!, 4), (4, \mathit{sensorStatus}!, 4), \\
&\quad (4, \mathit{shutdown}?, 5), (5, \mathit{shutdown}!, 1)\} \\
\tau_{\text{ParkA1}} &= \{(4, 20, \mathit{sensorsTimer})\}
\end{aligned}$$

Um Sensorwerte vom ParkA Server zu empfangen, muss ein Client zuerst die Verbindung mit der `startup`-Methode herstellen. Sobald die Verbindung erstellt wurde, ruft der ParkA Server den `started!` Callback beim Client auf. Anschließend versendet der Parkassistent mit einem Zyklus von maximal 20 Millisekunden die Sensorwerte an den Client durch die `sensorStatus!` Deklaration. Die Verbindung wird mit der Methode `shutdown` beendet.

Die zweite Variante der Schnittstelle, `ParkA2`, besitzt ebenfalls eine Methode zur Intitialisierung (`connect`) und eine zur Beendung einer Verbindung `disconnect`. Der Broadcast, welcher eine erfolgreiche Verbindung signalisiert, wird vom ParkA2 Server nicht gesendet. Stattdessen wird das erfolgreiche Initialisieren der Verbindung direkt mit der Rückgabe eines entsprechenden Parameters signalisiert. Modell 4.4.4 zeigt das Syntaxmodell von ParkA2.

Modell 4.4.4. ***(Syntaxmodell von ParkA2):***

$$\mathcal{I}^S_{\text{ParkA2}} = (D_{\text{ParkA2}}, P_{\text{ParkA2}}, m^D_{\text{ParkA2}}, m^T_{\text{ParkA2}})$$

$$\begin{aligned}
D^{\text{in}}_{\text{ParkA2}} &= \{\mathit{connectCall}, \mathit{disconnectCall}, \mathit{getSensorsCall}\}\\
D^{\text{out}}_{\text{ParkA2}} &= \{\mathit{connectResp}, \mathit{disconnectResp}, \mathit{getSensorsResp}\}\\
P^{\text{in}}_{\text{ParkA2}} &= \{\mathit{retry}, \mathit{retryCount}, \mathit{connectionIDDisconnect}\}\\
P^{\text{out}}_{\text{ParkA2}} &= \{\mathit{connectionIDConnect}, \mathit{status}, \mathit{fl}, \mathit{fml}, \mathit{fmr}, \mathit{fr}, \mathit{rl}, \mathit{rml}, \mathit{rmr}, \mathit{rr}\}\\
m^D_{\text{ParkA2}} &= \{(\mathit{fl}, \mathit{getSensorsResp}), (\mathit{fml}, \mathit{getSensorsResp}), (\mathit{fmr}, \mathit{getSensorsResp}),\\
&\quad (\mathit{fr}, \mathit{getSensorsResp}), (\mathit{rl}, \mathit{getSensorsResp}), (\mathit{rml}, \mathit{getSensorsResp}),\\
&\quad (\mathit{rmr}, \mathit{getSensorsResp}), (\mathit{rr}, \mathit{getSensorsResp}),\\
&\quad (\mathit{retry}, \mathit{connectCall}), (\mathit{retryCount}, \mathit{connectCall}),\\
&\quad (\mathit{connectionIDConnect}, \mathit{connectResp}),\\
&\quad (\mathit{status}, \mathit{connectResp}),\\
&\quad (\mathit{connectionIDDisconnect}, \mathit{disconnectCall})\}
\end{aligned}$$

Im Gegensatz zur `ParkA1` Schnittstelle, ist das `ParkA2` Eventmodell etwas kom-

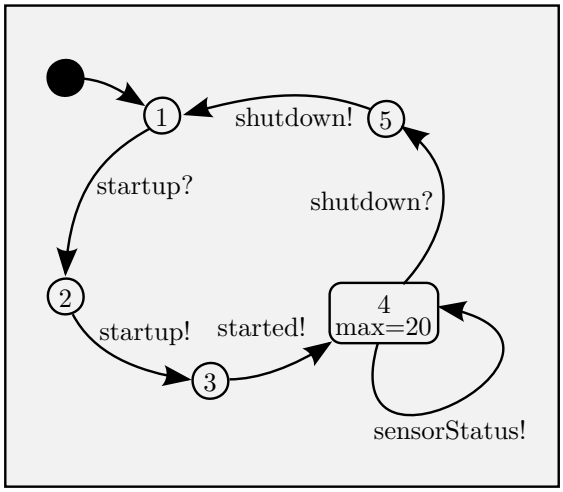


Abbildung 4.15.: Grafische Darstellung des Schnittstellenautomaten von `ParkA1`.

plexer. Zum Beispiel werden zwei Events für die Rückgabe der `connect`-Methode modelliert. Eines signalisiert einen korrekten Verbindungsaufbau und das andere einen Fehler. Zusätzlich ist es möglich die `connect`-Methode als mehrfacher Verbindungsversuch (`reconnect`) auszuführen. Abbildung 4.16 zeigt das hierarchische Eventmodell von ParkA2. Für den Methodenaufruf von `connectCall` wird der allgemeine Aufruf `connect?` und der spezielle `reconnect?` definiert - für die Antwort ein Event `connectOK!` für eine gelungenen, und `connectERR!` für eine fehlgeschlagenen Verbindungsversuch. Modell 4.4.5 zeigt das komplette Eventmodell.

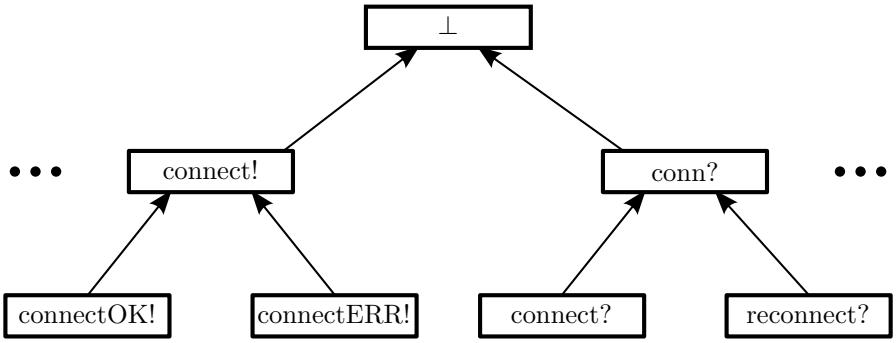


Abbildung 4.16.: Hierarchie-Eigenschaft des Eventmodells der `ParkA2` Schnittstelle.

Modell 4.4.5. ***(Schnittstellen-Eventmodell für ParkA2):***

$$\mathcal{I}^{E}_{\text{ParkA2}} = (E_{\text{ParkA2}}, C_{\text{ParkA2}}, X_{\text{ParkA2}}, e^{D}_{\text{ParkA2}}, e^{C}_{\text{ParkA2}}, e^{X}_{\text{ParkA2}}, e^{H}_{\text{ParkA2}})$$

$$\begin{aligned}
E^{\text{in}}_{ParA2} &= \{\textit{conn?}, \textit{connect?}, \textit{reconnect?}, \textit{disconnect?}, \textit{getSensors?}\}\\
E^{\text{out}}_{\text{ParkA2}} &= \{\textit{connect!}, \textit{connectOK!}, \textit{connectERR!}, \textit{disconnect!}, \textit{getSensors!}\}\\
C_{\text{ParkA2}} &= \{\textit{c1, c2, c3, c4}\}\\
&\quad c1 : (\textit{status} = OK), c2 : (\textit{status} \neq OK)\\
&\quad c3 : (\textit{retry} = false), c4 : (\textit{retry} = true \wedge \textit{lastErr} \neq null)\\
X_{\text{ParkA2}} &= \{\}\\
e^{D}_{\text{ParkA2}} &= \{(\textit{conn?}, \textit{connectCall}), (\textit{connect?}, \textit{connectCall}), (\textit{reconnect?}, \textit{connectCall}),\\
&\quad (\textit{connect!}, \textit{connectResp}), (\textit{connectOK!}, \textit{connectResp}),\\
&\quad (\textit{connectERR!}, \textit{connectResp}), (\textit{getSensors?}, \textit{getSensorsCall}),\\
&\quad (\textit{getSensors!}, \textit{getSensorsResp})\}\\
e^{C}_{\text{ParkA2}} &= \{(\textit{c1}, \textit{connectOK!}), (\textit{c2}, \textit{connectERR!}), (\textit{c3}, \textit{connect?}), (\textit{c4}, \textit{reconnect?})\}\\
e^{X}_{\text{ParkA2}} &= \{\}\\
e^{H}_{\text{ParkA2}} &= \{(\textit{conn}?, \perp), (\textit{connect}?, \textit{conn}?), (\textit{reconnect}?, \textit{conn}?),\\
&\quad (\textit{disconnect}?, \perp), (\textit{getSensors}?, \perp), (\textit{connect}!, \perp),\\
&\quad (\textit{connectOK}!, \textit{connect}!), (\textit{connectERR}!, \textit{connect}!),\\
&\quad (\textit{disconnect}!, \perp), (\textit{getSensors}!, \perp)\}
\end{aligned}$$

Modell 4.4.6 zeigt den Schnittstellenautomaten von ParkA2 und Abbildung 4.17 die grafische Repräsentation. Die Verbindung mit dem ParkA2-Server wird durch den Aufruf der `connect`-Methode initiiert. Falls die Verbindung erstellt werden konnte, wird als `status`-Parameter `OK` zurückgegeben und im Fehlerfall `ERROR`. In diesem Fall muss der Client connect erneut aufrufen. Sobald die Verbindung aufgebaut wurde, können die Sensorwerte mit der Methode `getStatus` abgerufen werden. Heruntergefahren wird der Dienst mit der `disconnect`-Methode.

Modell 4.4.6. ***(Schnittstellenautomat von ParkA2):***

$$\mathcal{I}^{B}_{\text{ParkA2}} = (S_{\text{ParkA2}}, s^{0}_{\text{ParkA2}}, T_{\text{ParkA2}}, \delta_{\text{ParkA2}}, \tau_{\text{ParkA2}})$$

$$S_{\text{ParkA2}} = \{a, b, c, d, e, f\}$$
$$s^{0}_{\text{ParkA2}} = a$$
$$T_{\text{ParkA2}} = E_{\text{ParkA2}}$$
$$\delta_{\text{ParkA2}} = \{(a, connect?, b), (b, connectERR!, c), (c, reconnect?, b), (b, connectOK!, d),$$
$$(d, getSensors?, e), (e, getSensors!, d), (d, disconnect?, f), (f, disconnect!, a)\}$$
$$\tau_{\text{ParkA2}} = \{\}$$

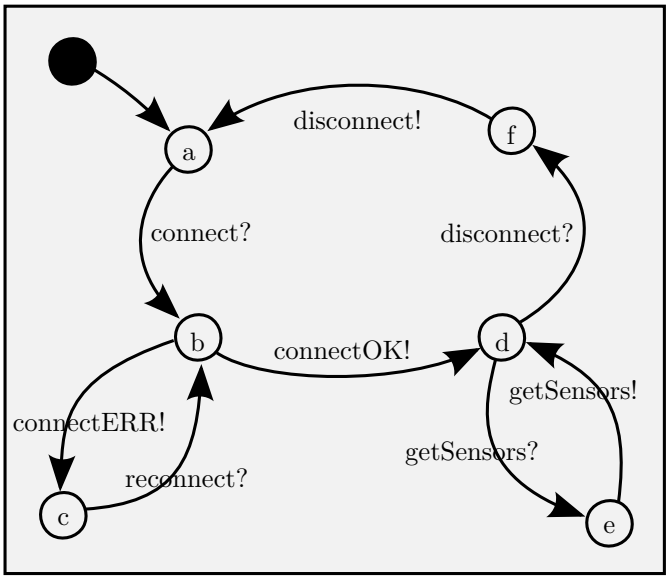


Abbildung 4.17.: Grafische Darstellung des Schnittstellenautomaten von `ParkA2`.

Der Vergleich der beiden ParkA Schnittstellen zeigt, dass mehrere Inkompatibilitäten sowohl in der Schnittstellensyntax, als auch im Kontrollfluss existieren. Tabelle 4.1 listet diese Unterschiede auf.

Fehlerart	**Beschreibung**
Syntaktischer Fehler	`startup`, `shutdown` und `getSensors` umbenannt.
Syntaktischer Fehler	`quality`-Parameter für ParkA2 unbekannt.
Syntaktischer Fehler	Sensorwerte: Parameternamen und Anzahl verschieden
Sequenzfehler	`connectERR!` und `connectOK!` werden von `ParkA1` nicht verarbeitet.
Sequenzfehler	`started!` wird von `ParkA2` nicht versandt.
Sequenzfehler	`ParkA1` erwartet Sensorwerte in Zustand 4 während `ParkA2` diese erst nach einer Anfrage `getSensors?` versendet.
Timingfehler	`ParkA1` erwartet die Sensorwerte im Takt von 10ms und ParkA2 definiert keinen Timeout.

Tabelle 4.1.: Inkompatibilitäten der beiden Schnittstellen `ParkA1` und `ParkA2` auf Basis der Klassifizierung aus Abschnitt 4.3

4.5. Zusammenfassung

Das in diesem Kapitel vorgestellte Modellierungskonzept beschreibt Softwarekomponenten und deren Schnittstellen auf mehreren Ebenen. Die Besonderheit dabei ist die hierarchische Natur des Modells, welche es ermöglicht, den Schnittstellenautomaten einfach zu halten und alle Abhängigkeiten zum Syntaxmodell im Eventmodell zu kapseln. Die sehr abstrakt definierte Ebene des Syntaxmodells kann für verschiedene IDL's konkretisiert werden, damit auch die zur IDL gehörenden Code-Generatoren wiederverwendet werden können. So kann zusammen mit dem Eventmodell und dem Schnittstellenautomaten lauffähiger Code für verschiedene Middleware-Systeme generiert werden. Zusätzlich wurden die Kompatibilitätseigenschaften des hierarchischen Schnittstellenmodells diskutiert, welche für den Nachweis der Korrektheit eines Adapters durch Model-Checking relevant sind. In diesem Zusammenhang ermöglicht die hierarchische Definition von Events eine einfache Analyse der Kompatibilität von Deklarationsaufrufen.

Das nächste Kapitel stellt die zentralen Konzepte dieser Arbeit zur Adaptierung von Softwareschnittstellen vor. Dabei wird davon ausgegangen, dass jede Version einer Schnittstelle durch ein hierarchisches Schnittstellenmodell modelliert wurde. Der Ansatz für die Adaptierung ist ebenfalls hierarchisch aufgebaut und zeichnet sich dadurch aus, dass er die in Abschnitt 3.3 vorgestellte Modellierung mit Mustern, mit der der endlichen Automaten vereint. Somit können einfache Adaptierungen durch ein global geltendes Mapping und komplexe Adaptierungen durch einen endlichen Automaten beschrieben werden.

Kapitel 5

Geschachtelte Adaptierung von Softwareschnittstellen

Dieses Kapitel stellt den neuen Ansatz zur Modellierung von geschachtelten Adaptern vor. Die wesentliche Neuerung gegenüber bekannten Ansätzen ist die Aufteilung des Adaptermodells in einen statischen und einen dynamischen Teil. Dies bringt Vorteile in Bezug auf die Automatengröße, die Robustheit der Adapter-Ausführung und die einfache Modellierung häufiger Adaptierungen.

Abschnitt 5.1 beschreibt die beiden Adaptermodelle des statischen und dynamischen Adapters. Abschnitt 5.2 definiert die für die Ausführung von geschachtelten Adaptern notwendige Architektur. Die Auftrennung des Adapters in die zwei Sub-Modelle wird in Abschnitt 5.4 am Beispiel des Parkassistenten veranschaulicht. Abschnitt 5.5 definiert die Algorithmen zur Validierung eines geschachtelten Adaptermodells auf Korrektheit des Kontrollflusses und Abschnitt 5.6 zeigt, wie aus dem Adaptermodell lauffähiger Code generiert werden kann. Abschließend bewertet Abschnitt 5.7 die Lösung in Bezug auf die Aufgabenstellung.

Inhaltsverzeichnis

5.1. Geschachtelte Adaptermodellierung mit statischen und dynamischen Adaptern

Das Ziel bei der Modellierung von Adaptern ist die Beschreibung einer korrekten Mediation bei der Verknüpfung von inkompatiblen Schnittstellen. In dem vorgestellten Fallbeispiel aus Abschnitt 4.4 ist das ein System mit einem Client, der die `ParkA1`-Schnittstelle benötigt, und einem Server, der die `ParkA2`-Schnittstelle anbietet. In diesem Fall muss das Adaptermodell eine Mediation für die in Tabelle 4.1 gezeigten Inkompatibilitäten bieten.

Ein Ziel bei der Entwicklung der geschachtelten Schnittstellenadaptierung ist es, eine möglichst strikte Trennung zwischen Syntax- und Verhaltensadaptierung auf logischer Ebenen zu erreichen. Abbildung 5.1 zeigt das geschachtelte Adaptermodell a für eine angebotene (p) und eine benötigte Schnittstelle (r). Die Schnittstellen werden jeweils durch ihre hierarchischen Schnittstellenmodelle beschrieben.

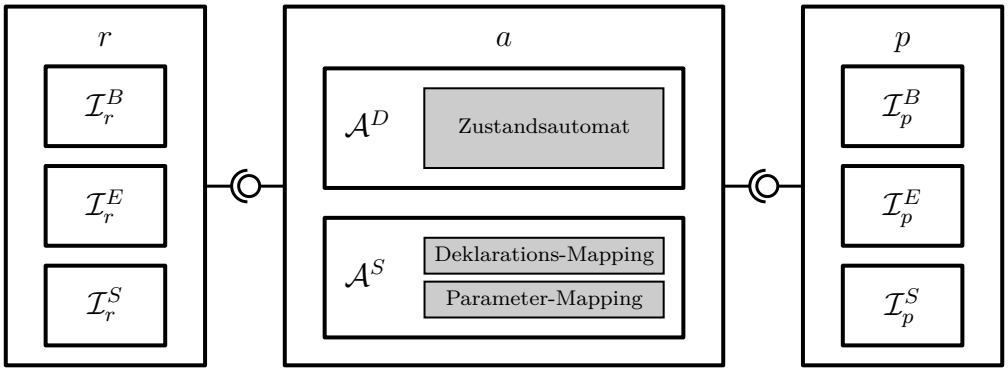


Abbildung 5.1.: Das geschachtelte Adaptermodell a setzt sich aus zwei Sub-Modellen zusammen: Das statische Adaptermodell $\mathcal{A}^S$ beschreibt ein globales Mapping zwischen Deklarationen und Parametern und das dynamische Adaptermodell $\mathcal{A}^D$ einen Zustandsautomaten für kontextabhängige Adaptierungen [Pra+14].

Das Adaptermodell kann dabei aus zwei Untermodellen bestehen. Das *statische Adaptermodell* ($\mathcal{A}^S$) beschreibt Adaptierungen, welche unabhängig vom internen Zustand der adaptierten Komponenten durchgeführt werden. Die statische Adaptierung erfolgt durch ein Mapping zwischen syntaktischen Elementen der beiden inkompatiblen Schnittstellen. Das bedeutet, dass das statische Adaptermodell rein auf Basis der Schnittstellen-Syntaxmodelle beschrieben wird. Das zweite Untermodell, das *dynamische Adaptermodell* ($\mathcal{A}^D$), beschreibt kontextabhängige Adap-

tierungen, welche von der Sequenz der Events zwischen den beiden inkompatiblen Komponenten abhängen. Bei dieser Art der Modellierung muss keine strikte Trennung von Syntax und Verhalten zwischen $\mathcal{A}^S$ und $\mathcal{A}^D$ erfolgen, denn auch $\mathcal{A}^D$ kann syntaktische Adaptierungen beschreiben. Deshalb wurden die Begriffe „statisch“ und „dynamisch“ gewählt, da sie besser die Eigenschaften der beiden Modelle charakterisieren.

Definition 5.1.1. ***(Adaptermodell):*** *Ein* Adaptermodell $\mathcal{A} = (\mathcal{A}^S, \mathcal{A}^D)$ *ist definiert durch:*

- $\mathcal{A}^S$ *ist das* statische Adaptermodell
- $\mathcal{A}^D$ *ist das* dynamische Adaptermodell

Für die formale Definition des Adaptermodells a zwischen einer benötigten (r) und angebotenen Schnittstelle (p) werden folgende Mengennotationen verwendet:

- $\mathcal{I}_r = (\mathcal{I}_r^S, \mathcal{I}_r^E, \mathcal{I}_r^B)$ ist das Schnittstellenmodell der benötigten Schnittstelle. Subelemente werden analog notiert.
- $\mathcal{I}_p = (\mathcal{I}_p^S, \mathcal{I}_p^E, \mathcal{I}_p^B)$ ist das Schnittstellenmodell der angebotenen Schnittstelle. Subelemente werden analog notiert.
- $D_a = D_r \cup D_p$ sind Adapter-Deklarationen.
- $D_a^{\text{in}} = D_r^{\text{in}} \cup D_p^{\text{out}}$ sind Adapter-Eingabedeklarationen.
- $D_a^{\text{out}} = D_r^{\text{out}} \cup D_p^{\text{in}}$ sind Adapter-Ausgabedeklarationen.
- $P_a = P_r \cup P_p$ sind Adapter-Parameter.
- $P_a^{\text{in}} = P_r^{\text{in}} \cup P_p^{\text{out}}$ sind Adapter-Eingabeparameter.
- $P_a^{\text{out}} = P_r^{\text{out}} \cup P_p^{\text{in}}$ sind Adapter-Ausgabeparameter.
- $E_a = E_r \cup E_p$ sind Adapter-Events.
- $E_a^{\text{in}} = E_r^{\text{in}} \cup E_p^{\text{out}}$ sind Adapter-Eingabeevents.
- $E_a^{\text{out}} = E_r^{\text{out}} \cup E_p^{\text{in}}$ sind Adapter-Ausgabeevents.

5.1.1. Das statische Adaptermodell

Dig und Johnson [DJ05] untermauern durch ihre Studie, dass mehr als 80% der Änderungen, die eine Schnittstelle inkompatibel zu ihrer Vorgängerversion machen, strukturelle Änderungen sind, bei denen das Verhalten der Komponente erhalten bleibt. Diese Art der Inkompatibilität entsteht, wenn das Syntaxmodell der Schnittstelle angepasst wird, z.B. durch das Umbenennen oder Löschen von Methoden oder Parametern. Eine derartige Schnittstellenänderung wirkt sich zustandsunabhängig auf die gesamte Kommunikation aus. Genau diese, meist syntaktischen Unterschiede, zwischen Schnittstellen werden durch das *statische Adaptermodell* adressiert. Wegen der Häufigkeit solcher Schnittstellenänderungen wird für dieses Modell der in Abschnitt 3.3.2 beschriebene sehr einfache Ansatz zur Adaptierung mithilfe von Mustern verwendet. Somit besteht eine Adaptierung im statischen Adaptermodell aus einem Mapping, welches Adapter-Eingabedeklarationen Adapter-Ausgabedeklarationen zuordnet.

Definition 5.1.2. ***(statisches Adaptermodell):*** *Sei a das* Adaptermodell *für zwei Schnittstellen, dann ist das* statische Adaptermodell $\mathcal{A}^S = (s^D, s^P)$ *definiert durch:*

- $s^D : D_a^{\text{in}} \rightsquigarrow D_a^{\text{out}} \times \cdots \times D_a^{\text{out}}$ *ist das Deklarations-Mapping, welches Adapter-Eingabedeklarationen eine Sequenz von Adapter-Ausgabedeklarationen zuordnet. Dabei beschreibt* $s_1^D : d := \perp$ *das undefinierte Mapping.*
- $s^P : P_a^{\text{out}} \rightsquigarrow \left\{ (s^{\text{const}}, s^{\text{func}}) : \begin{array}{rl} s^{\text{const}} & \textit{ist eine Konstante,} \\ s^{\text{func}} & \textit{ist ein arithmetischer Ausdruck über } P_a \\ & \textit{oder der undefinierte Ausdruck } \perp \end{array} \right\}$

  ist das Parameter-Mapping, welches Adapter-Ausgabeparametern einen Wert zuordnet. Dieser Wert ist definiert durch die Konstante s^{const} *und einen optionalen arithmetischen Ausdruck* s^{func}*, der die Berechnung des Ausgabeparameters auf Basis von anderen Parametern definiert.*

Falls eine Adapter-Ausgabedeklaration in einem Deklarations-Mapping $s^D(d)$ verwendet wird, muss für jeden Adapter-Ausgabeparameter der Deklaration d ein Parameter-Mapping existieren:

$$\exists d_a^{\text{in}} : s^D(d_a^{\text{in}}) = (d_1, \ldots, d_n) \Rightarrow$$
$$\left(\exists p, i : 1 \leq i \leq n, m^D(p) = d_i, p \in P_a^{\text{out}} \Rightarrow \left(s^P(p) \text{ ist definiert}\right)\right)$$

Ein Deklarations-Mapping $s^D(d_a^{\text{in}})$ beschreibt, welche Aktion ausgeführt werden soll, falls eine Adapter-Eingabedeklaration d_a^{in} von einer der beiden Softwarekomponenten aufgerufen wird. Dabei wird als Aktion nur der Aufruf von Adapter-Ausgabedeklarationen erlaubt. So kann zum Beispiel die Umbenennung der Methode d_a^{in} in d' durch das Deklarations-Mapping $s^D(d_a^{\text{in}}) := (d')$ modelliert werden. Das Aufteilen einer Methode d_a^{in} in zwei Methoden d_1 und d_2 wird durch $s^D(d_a^{\text{in}}) := (d_1, d_2)$ als 2-Tupel modelliert. Das bedeutet, dass nach der Adaptierung die beiden Methoden d_1 und d_2 nacheinander aufgerufen werden. Jedes Mal, wenn der statische Adapter dieses Deklarations-Mapping ausführt, werden die zu den Deklarationen ($d_1$und d_2) gehörenden Adapter-Ausgabeparameter mit den im statischen Adaptermodell definierten Parameter-Mappings gesetzt und anschließend die Deklarationen aufgerufen. Dieses Parameter-Mapping besteht aus einer obligatorischen Konstante und einem optionalen funktionalen Parameter-Mapping. Die Konstante definiert die Standardbelegung des Parameters, denn es kann Fälle geben, in denen das funktionale Parameter-Mapping nicht berechnet werden kann. Zum Beispiel kann es bei einem funktionalen Parameter-Mapping $s^{\text{func}}(p_a^{\text{out}}) := ((p_1 + p_2)/2))$ vorkommen, dass nicht alle in dem Ausdruck verwendeten Parameter (im Beispiel $p_1$und p_2) vom Adapter zum Aufrufzeitpunkt bereits empfangen wurden. In diesem Fall wird die obligatorische Konstante $s^{\text{const}}(p_a^{\text{out}})$ als Parameterwert gesetzt.

Das statische Adaptermodell gilt global, unabhängig vom Adapterzustand. Zur Beschreibung eines zustandsabhängigen Verhaltens hingegen muss das dynamische Adaptermodell verwendet werden. Dieses wird im folgenden Abschnitt vorgestellt.

5.1.2. Das dynamische Adaptermodell

Das *dynamische Adaptermodell* ist ein Zustandsautomat, der das gewünschte Adapterverhalten spezifiziert. Im Gegensatz zum statischen Adaptermodell ist das Verhalten des dynamischen Adapters zu einem Zeitpunkt abhängig von der Sequenz der bereits erfolgten Deklarationsaufrufe. Der Automat erlaubt die Beschreibung umfangreicher Adaptierungen inklusive der Adaptierung von Zeitverhalten.

Definition 5.1.3. ***(dynamisches Adaptermodell):*** *Sei a das Adaptermodell für eine benötigte (r) und eine angebotene Schnittstelle (p). Das* dynamische Adaptermodell $\mathcal{A}^D = (S^D, s_0^D, T^D, X^D, A^D, \delta^D, \tau^D, \text{active}, m_a^X)$ *ist definiert durch:*

- S^D *ist eine endliche Menge von* Zuständen.
- $s_0^D \in S^D$ *ist der* Startzustand.
- $T^D \subseteq E_a \cup E_\mathcal{T}$ *ist eine endliche Menge von* Triggern*, welche sich aus den Adapter-Eingabeevents und einer Menge von Timeoutevents definiert.*
- X^D *ist die Menge von Adapter-Emulationen. Für ein Schnittstellenevent* $e \in E_a^{out}$ *definiert die Emulation* $x_e \in X^D$ *wie alle für das Event relevanten Parameter* (P^e) *gesetzt werden müssen, um die entsprechende Deklaration* $e^D(e)$ *aufzurufen.* x_e *definiert deshalb für jeden Parameter aus* P^e *eine Konstante* (g^{const}) *und einen optionalen arithmetischen Ausdruck* (g^{func}) *zur Berechnung des Parameters. Im Unterschied zur Schnittstellen-Emulation (siehe Abschnitt 4.2.2), können hier beliebige Adapter-Parameter* (P_a) *von beiden Schnittstellen r und p verwendet werden:*

$$x_e : P^e \rightarrow \left\{ \begin{array}{rl} g^{\text{const}} & \textit{ist eine Konstante,} \\ (g^{\text{const}}, g^{\text{func}}) : \; g^{\text{func}} & \textit{ein optionaler arithmetischer} \\ & \textit{Ausdruck über } P_a \end{array} \right\}$$

- $A^D = X_a \times \cdots \times X_a$ *ist eine endliche Menge von* Aktionen*, welche aus Sequenzen von Emulationen besteht.* $X_a = X_r^{\text{out}} \cup X_p^{\text{in}} \cup X^D$.
- $\delta^D : S^D \times T^D \rightsquigarrow A^D \times S^D$ *ist die partielle* Transitionsfunktion.
- $\tau^D : S^D \times \mathbb{N} \rightsquigarrow E^\mathcal{T}$ *ist die partielle Funktion, die einem Zustand eine natürliche Zahl (Timeout) und ein Timeoutevent zuordnet.*
- a*ctive* : $\{S^D \bigcup \delta^D\} \rightarrow \{\text{true}, \text{false}\}$ *ist eine Funktion die jedem Zustand und jeder Transition einen booleschen Wert zuordnet. Zustände z mit* $\text{active}(z) = \text{true}$ *werden* aktive Zustände *und Transitionen t mit* $\text{active}(t) = true$ aktive Transitionen *genannt.*
- $m_a^X : X_D \rightarrow E_a^\mathcal{I}$ *ist die Funktion, die Adapter-Emulationen Events zuordnet.*

Ähnlich wie der Schnittstellenautomat, spezifiziert das dynamische Adaptermodell den Kontrollfluss auf Basis der in den Eventmodellen definierten Events. Für die Modellierung wird eine angepasste Variante des in Abschnitt 4.1.2 vorgestellten Komponenten-Verhaltensmodells verwendet. Somit realisiert dieser Automat einen Transduktor, dessen Ausgaben (Aktionen) von seinem aktuellen Zustand und der aktuellen Eingabe (Trigger) bestimmt werden. Trigger können alle Events der benötigten oder angebotenen Schnittstelle sein (E_a). Das bedeutet, dass der dynamische Adapter nach jedem Event auf Client- oder Serverseite zusätzliche Aktionen ausführen kann. Eine Aktion ist durch eine Sequenz von Emulationen definiert. Erlaubte Elemente einer Aktion sind nur jene Emulationen, die vom Adapter ausgeführt werden können (X_a). Eine Aktion besteht nun darin, die Sequenz von Emulationen nacheinander durch den Aufruf der entsprechenden Deklaration auszuführen. Wie die Parameter für den Aufruf gesetzt werden sollen, ist in der Emulation selbst definiert. Da eine Emulation immer genau einem Event zugeordnet ist, produziert der Adapter durch die Emulation genau dieses Event.

Die active-Funktion wird zur Synchronisation des statischen und dynamischen Adapters verwendet. Sie spezifiziert, ob der dynamische Adapter beim Eintreten in einen Zustand aktiv bleiben soll und somit eine eventuelle statische Adaptierung verzögert wird. So kann zum Beispiel mit $active(s_1) = true$ definiert werden, dass der dynamische Adapter im Zustand s_1 verweilt, solange bis der nächste passive Zustand erreicht wird. Die genaue Funktionsweise dieses Synchronisationsmechanismus ist in Abschnitt 5.2.2 definiert.

5.2. Adapter-Architektur und Ausführungssemantik

Die Architektur zur Ausführung der geschachtelten Adapter orientiert sich an der Struktur der Modelle und spiegelt deshalb auch die Trennung zwischen statischen und dynamischen Anteilen wieder. Abbildung 5.2 zeigt die verschiedenen Module in der Architektur und den Informationsfluss zwischen ihnen. Der Adapter wird dabei als Schicht zwischen Client und Server, unabhängig von einer konkreten Kommunikationsmiddleware, definiert.

5.2.1. Adapter-Module

Die beiden zentralen Module sind der *statische Adapter*, der mit den Deklarationen der beiden Schnittstellen arbeitet, und der *dynamische Adapter*, der Events als

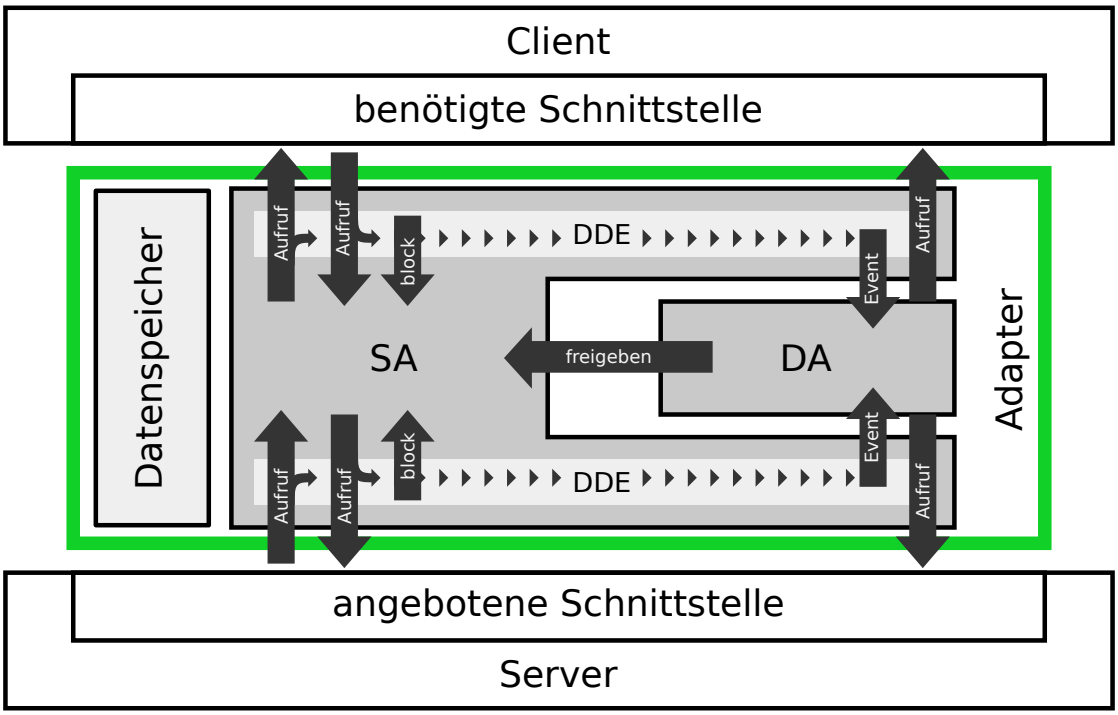


Abbildung 5.2.: Architektur zur Ausführung von statischen und dynamischen Adaptern (angelehnt an [Pra+14]).

Eingabe benötigt. Der *Datenspeicher* wird verwendet, um die zwischen Client und Server ausgetauschten Parameter adapterintern zur Verfügung zu stellen und den Zugriff auf Parameter zu synchronisieren. Die *Deklarations-De-/Encoder-Module (DDE)* regeln die Übersetzung zwischen Deklarationen und Events. Im Folgenden werden die einzelnen Module und deren Interaktion genauer beschrieben.

Statischer Adapter (SA)

Der statische Adapter wird durch Deklarationsaufrufe von Client und Server gesteuert. Er implementiert das im statischen Adaptermodell, $\mathcal{A}^S$, beschriebene Mapping. Dieses Mapping beschreibt eine Zuordnung von Adapter-Eingabedeklarationen zu Adapter-Ausgabedeklarationen. Falls nun eine Deklaration d aufgerufen wird und ein Mapping $s^D(d) \mapsto (d_1, \cdots, d_n)$ existiert, werden vom statischen Adapter die Ausgabedeklarationen $d_1, \cdots, d_n$ der Reihe nach aufgerufen. Die für die Aufrufe benötigten Parameter werden mithilfe des Parameter-Mappings berechnet. Das bedeutet, dass der statische Adapter jeden Parameter entweder mithilfe des funktionalen Parametermappings s^{func} oder dem konstanten Parametermappings

s^{const} berechnet.

Der statische Adapter wird nach jedem Deklarationsaufruf blockiert und muss vom dynamischen Adapter wieder freigegeben werden. Dieser Mechanismus ist notwendig, um die beiden Adapterkomponenten zu synchronisieren. Denn nach dem Aufruf einer Deklaration muss der dynamische Adapter entscheiden, ob sein aktueller Zustand und das von den DDE-Modulen berechnete Event eine Aktion zur Folge hat. Sobald der dynamische Adapter keine Aktion mehr ausführt und sich nicht in einem aktiven Zustand befindet, wird der statische Adapter wieder freigegeben und kann die statische Adaptierung fortsetzen.

Dynamischer Adapter (DA)

Der dynamische Adapter implementiert den im dynamischen Adaptermodell, $\mathcal{A}^D$, definierten Zustandsautomaten. Als Eingabe für den Automaten dienen die von den DDE-Modulen übersetzten Deklarationsaufrufe in Form von Events. Jede in einer der Schnittstellen aufgerufene Deklaration kann somit eine Transition im dynamischen Adapter triggern, egal ob sie nun vom Client, vom Server oder vom statischen Adapter aufgerufen wurde.

Falls nun eine Transition $trans := ((s_{\text{source}}, t), (a, s_{\text{target}}))$ im dynamischen Adapter getriggert wird, wird die ensprechende Aktion $a = (x_1, \cdots, x_n)$ ausgeführt. Dabei ist eine Aktion eine Reihe von Emulationen, die von den De-/Encodermodulen in Deklarationsaufrufe übersetzt werden.

Grundsätzlich wird der statische Adapter blockiert, sobald der dynamische Adapter aktiviert wird. Der dynamische Adapter entscheidet auf Basis des aktuellen Zustandes und des eintreffenden Events, ob der statische Adapter wieder freigegeben werden kann.

Deklarations-De-/Encoder

Die DDE-Module implementieren die Übersetzung von Deklarationsaufrufen und Events. Diese Transformation ist notwendig, um den mit Events arbeitenden dynamischen Adapter an die Schnittstellen von Client und Server anzubinden. Dabei bezeichnen wir die Transformation eines Deklarationsaufrufes in ein Event als *Dekodieren* und umgekehrt als *Enkodieren*.

Falls eine Deklaration d aufgerufen wird, analysiert das entsprechende DDE-Modul die Menge aller zu dieser Deklaration gehörigen Events $(m^A)^{-1}(d) = E^e$. Anhand der im Event definierten Constraint $m^C(e), e \in E^e$ und der im Deklarationsaufruf übergebenen Parameter kann das DDE-Modul feststellen, welches Event zu diesem konkreten Deklarationsaufruf gehört. Das entsprechende Event

wird dem dynamischen Adapter weitergeleitet. Durch den hierarchischen Aufbau der Events kann die Suche nach dem passenden Event effizient erfolgen.

Um eine Aktion des dynamischen Adapters in einen konkreten Deklarationsaufruf übersetzten zu können, wird direkt die übergebene Emulation ausgeführt. Diese beschreibt, wie die Parameter gesetzt werden müssen, und welche Deklaration aufgerufen werden muss, um ein Event zu emulieren.

Datenspeicher

Das Datenspeicher-Modul speichert die Parameter der Deklarationsaufrufe. Parameter werden vom statischen Adapter benötigt, um das funktionale Parametermapping s^{func} zu berechnen und von den DDE-Modulen um ein Event e mit der entsprechenden Emulation $m^X(e) = x$ zu enkodieren.

5.2.2. Ausführungssemantik

In diesem Abschnitt definieren wir den Kontrollfluss zwischen den Modulen im Adapter. Besonders in Situationen, in denen sowohl eine statische, als auch eine dynamische Adaptierung für eine Deklaration modelliert wurde, muss definiert werden, ob zuerst die statische oder die dynamische ausgeführt wird.

Grundsätzlich werden in der vorgestellten Architektur alle Aufrufe von Client- oder Serverseite durch die DDE-Module zum statischen Adapter geleitet. Die DDE Module entscheiden aufgrund der Dekodierung, ob der statische Adapter blockiert wird oder nicht. Falls ein Event vom DDE Modul an den dynamischen Adapter gesendet wird, muss der statische Adapter blockiert werden. Dieser Vorgang verhindert, dass die statische Adaptierung stattfindet bevor die dynamische abgeschlossen ist. Erst wenn der dynamische Adapter seine Aktionen durchgeführt hat, gibt er den statischen Adapter wieder frei. Nicht nur Aufrufe von Client- oder Serverseite, sondern auch die Deklarationsaufrufe des statischen Adapters werden dekodiert und gegebenenfalls als Event an den dynamischen Adapter gesendet. Damit der dynamische Adapter sich durch seine Aktionen nicht selbst triggern, und somit Endlosschleifen verursachen kann, werden Deklarationsaufrufe des dynamischen Adapters nicht von den DDE-Modulen dekodiert.

Beispielablauf

Abbildung 5.3 zeigt den Kontrollfluss im Adapter für den Aufruf einer Deklaration auf Clientseite. Wir gehen in dem Beispiel davon aus, dass die aufgerufene Deklaration ein statisches Mapping besitzt und sowohl der Deklarationsaufruf selbst, als

auch der Aufruf des statischen Mappings eine Transition im dynamischen Adapter triggern. Die Aktionen im dynamischen Adapter rufen jeweils Deklarationen beim Server auf. Zuerst wird der Aufruf im DDE-Modul dekodiert. Da das dekodierte

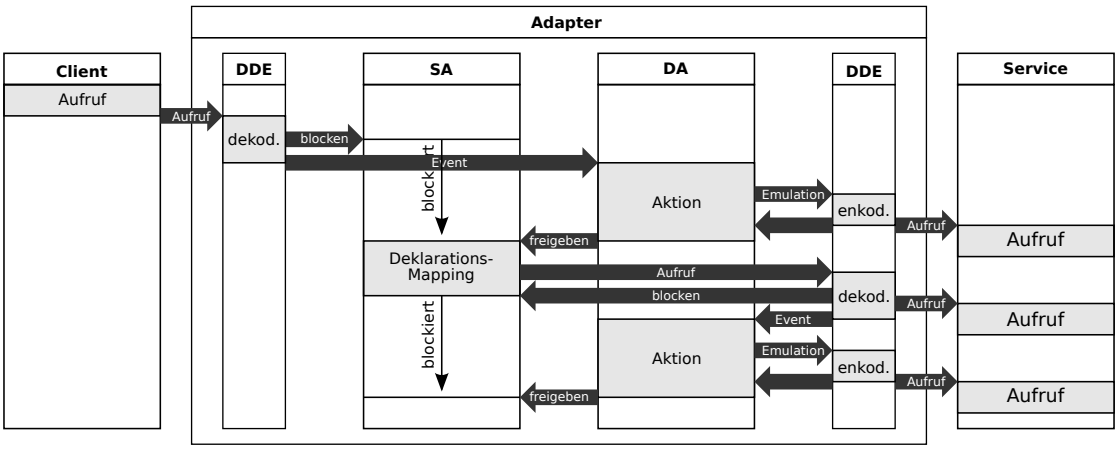


Abbildung 5.3.: Beispiel für einen Adapter-Kontrollfluss im statischen und dynamischen Adaptierungsfall für den Aufruf einer Deklaration.

Event an den dynamischen Adapter gesendet wird, signalisiert das DDE Modul dem statischen Adapter davor, dass er warten soll bevor das statische Mapping ausgeführt wird. Anschließend wird der dekodierte Deklarationsaufruf als Event an den dynamischen Adapter gesendet. Dieser triggert eine Transition und führt die zugehörige Aktion aus, indem er sie zur Enkodierung an das DDE-Modul sendet. Der dynamische Adapter entscheidet nun selbst, ob der statische Adapter freigegeben wird. In folgenden drei Fällen würde der statische Adapter nicht freigegeben: (1) Andere Aktionen sind auf der aktuellen Transition spezifiziert. (2) Die getriggerte Transition ist eine aktive Transition. (3) Der Zustandsautomat erreicht einen aktiven Zustand. Nachdem der statische Adapter freigegeben wurde, führt er das statische Mapping mit dem spezifizierten Deklarationsaufruf aus. Der Aufruf wird vom DDE-Modul an den Server geschickt, dekodiert, und als Event an den dynamischen Adapter gesendet. Nach dem Ausführen der nächsten Aktion ist die Adaptierung abgeschlossen, und der statische Adapter wird wieder freigeben.

Alle Deklarationsaufrufe werden von den DDE-Modulen ausgeführt. Deshalb übernehmen diese Module auch die Speicherung der Parameter im Datenspeicher.

Aktive Zustände und Transitionen

Der Kontrollfluss von statischen und dynamischen Adaptern ist so definiert, dass der dynamische Adapter nach jedem Deklarationsaufruf Aktionen durchführen kann. Ob Aktionen durchgeführt werden, hängt vom dekodierten Event als Eingabe und dem aktuellen Zustand des dynamischen Adapters ab. Während der Ausführung des dynamischen Adapters wird der statische Adapter blockiert. Dabei entscheidet der dynamische Adapter, wann der statische Adapter wieder freigegeben wird.

Abbildung 5.4 zeigt die vier verschiedenen Szenarien bei der Modellierung von aktiven und passiven Zuständen und Transitionen. Nur im Szenario 1 wird der statische Adapter bei Erreichen des Zustandes b freigegeben. Sobald entweder die aktuelle Transition eine aktive Transition oder der Zielzustand ein aktiver Zustand sind, bleibt der dynamische Adapter aktiv. Dieser Mechanismus ist notwendig, falls die nächsten Adaptierungsschritte rein vom dynamischen Adapter bestimmt werden sollen. Zum Beispiel ist das der Fall, wenn der dynamische Adapter ohne Störung eine Sequenz von Aktionen ausführen soll, in der zwischenzeitlich Eingaben von Client oder Server erwartet werden. Meist reicht die Verwendung einer aktiven Transition aus. Ein aktiver Zustand kann dann verwendet werden, wenn ein Zustand unabhängig von der eingehenden Transition aktiv bleiben soll. Ein aktiver Zustand entspricht einem passiven Zustand bei dem alle eingehenden Transitionen aktiv sind.

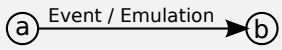
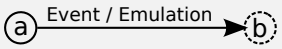
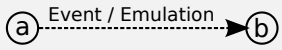
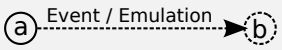
	Automat	Beschreibung	Aktion
1		passive Transition und passiver Zielzustand	Statischen Adapter freigeben
2		passive Transition und aktiver Zielzustand	Statischen Adapter blockieren
3		aktive Transition und passiver Zielzustand	Statischen Adapter blockieren
4		aktive Transition und aktiver Zielzustand	Statischen Adapter blockieren

Abbildung 5.4.: Aktionen des dynamischen Adapters beim feuern einer aktiven oder passiven Transition. Aktive Elemente werden durch gestrichelte Linien dargestellt.

5.3. Elementare Schnittstellenänderungen

Die Verwendung von statischen und dynamischen Adaptern wird nun anhand verschiedener elementarer Aktivitäten, die bei der Evolution von Schnittstellen auftreten, veranschaulicht. Teilweise können diese sowohl mit dem statischen, als auch mit dem dynamischen Adapter realisiert werden. Für die Adaptierung von zeitlichem Verhalten wird ausschließlich der dynamischen Adapter verwendet.

5.3.1. Deklaration löschen

Beim Löschen einer Deklaration muss weder im statischen, noch im dynamischen Adapter ein Modell erstellt werden, denn jede Deklaration, für die kein Mapping im statischen Adapter oder kein Trigger im dynamischen Adapter existiert, wird nicht adaptiert und somit auch keine Aktion ausgeführt.

5.3.2. Deklaration umbenennen

Die Umbenennung einer Deklaration hat zur Folge, dass jeder Aufruf der Deklaration an die umbenannte Deklaration weitergereicht werden muss. Eine Umbenennung gilt somit global, und sollte deshalb im statischen Adapter durch ein entsprechendes Mapping beschrieben werden. Modell 5.3.1 beschreibt die Umbenennung einer Methode *a* nach *b*. Die Methode besteht aus dem Aufruf *aCall* und der Antwort *aRes*. Neben der Deklaration, muss im statischen Adaptermodell auch jeder Parameter der Deklaration zugewiesen werden. Im Beispiel ist das der Parameter *param1*. Falls *param1* zum Zeitpunkt der Adaptierung noch nicht empfangen wurde, wird die Konstante 0 als Wert verwendet.

Modell 5.3.1. ***(Statisches Adaptermodell zum Umbenennen einer Deklaration):***

$$\begin{aligned}\mathcal{A}^S =& (s^D, s^P)\\ s^D =& \{(\mathit{required.aCall}, \mathit{provided.bCall}),\\ & (\mathit{provided.bRes}, \mathit{required.aRes})\}\\ s^P =& \{(\mathit{required.aCall.param1}, (0, \mathit{provided.bCall.param1}))\}\end{aligned}$$

Die Umbenennung könnte auch mit einem dynamischen Adapter modelliert werden. Abbildung 5.5 zeigt den entsprechenden Zustandsautomaten. Für die beiden Adapter-Ausgabeevents *bCall?* und *aRes!* muss eine entsprechende Emulation manuell modelliert werden. Die dafür notwendige Emulation wird in Modell 5.3.2 gezeigt. Sie ähnelt dem statischen Mapping aus Modell 5.3.1:

Modell 5.3.2. ***(Emulation als Teil des dynamischen Adaptermodells zum Weiterleiten eines Parameters):***

$$\begin{aligned}X^D =& \{x_{\mathrm{bCall}}\}\\ x_{\mathrm{bCall}} =& \{(\mathit{required.a.param1}, (0, \mathit{provided.b.param1}))\}\end{aligned}$$

Bei der Modellierung mit dem dynamischen Adapter wird die Adaptierung nur korrekt durchgeführt, falls sich der Adapter im Zustand α befindet. Falls die Adaptierung in jedem Zustand gelten soll, können die von Mealy [Mea55] definierten parallelen Regionen verwendet werden.

5.3.3. Deklaration aufteilen

Falls eine Deklaration in mehrere Deklarationen aufgeteilt wird, kann dies ebenfalls sowohl durch den statischen als auch durch den dynamischen Adapter erfolgen. Modell 5.3.3 zeigt das für die Aufteilung von Deklaration a in b und c notwendige statische Adaptermodell. In diesem Beispiel wird *param1* für den Aufruf der Deklaration *b* verwendet und *param2* für den Aufruf von *c*.

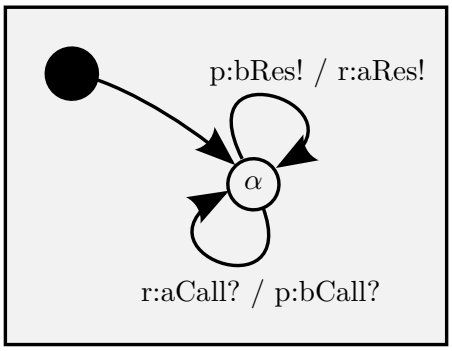


Abbildung 5.5.: Dynamisches Adaptermodell zum Umbenennen einer Methode a in b. r (Client-Seite) und p (Server-Seite) beziehen sich dabei auf das zu adaptierende Interface.

Modell 5.3.3. ***(Statisches Adaptermodell zum Aufteilen einer Deklaration):***

$$\begin{aligned}\mathcal{A}^S =& (s^D, s^P)\\ s^D =& \{(\mathit{required.aCall}, (\mathit{provided.bCall}, \mathit{provided.cCall})),\\ & (\mathit{provided.cResp}, \mathit{required.aResp})\}\\ s^P =& \{(\mathit{required.bCall.param1}, (0, \mathit{provided.aCall.param1})),\\ & (\mathit{required.cCall.param2}, (0, \mathit{provided.aCall.param2}))\}\end{aligned}$$

Der dynamische Adapter für die gleiche Adaptierung ist in Abbildung 5.6 dargestellt. Der Zustand α wird mit dem Aufruf von $aCall$ verlassen und die Emulation $bCall$ beim Server aufgerufen. Der Zustand β ist als aktiver Zustand modelliert. Deshalb unterbindet der dynamische Adapter in diesem Zustand jegliche Ausführung des statischen Adapters. Dies ist zum Beispiel wichtig, falls für die Deklaration $cCall$ auch im statischen Adapter ein Mapping existiert. Der aktive Zustand β wird erst dann verlassen, sobald das Event $cRes$ vom Server erhalten wurde. Für jede Ausgabe des Zustandsautomaten muss eine entsprechende Emulation existieren.

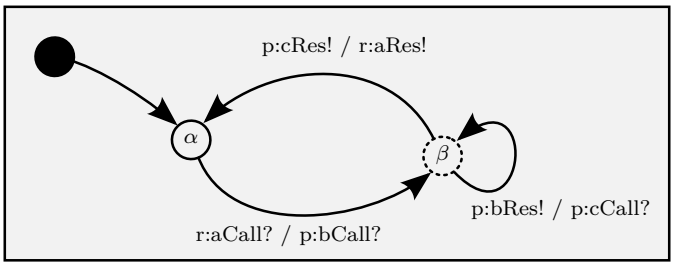


Abbildung 5.6.: Dynamisches Adaptermodell zum Aufteilen einer Deklaration a in b und c.

5.3.4. Deklarationsaufruf verzögern

Im geschachtelten Adaptermodell unterstützt lediglich der dynamische Adapter die Betrachtung von Zeit. Grundsätzlich wäre es auch möglich, das statische Adaptermodell so zu erweitern, dass auch für zeitliche Adaptierungen entsprechende Modellierungselemente existieren. Dies wäre sinnvoll, falls bestimmte zeitliche Adaptierungen einen immer wiederkehrenden Anwendungsfall darstellen.

In diesem Beispiel wird eine Kombination aus statischer und dynamischer Adaptierung zur Verzögerung des Aufrufs *aCall* verwendet. Der statische Adapter aus Modell 5.3.4 wird verwendet, um den Deklarationsaufruf weiterzuleiten. Das dynamische Adaptermodell in Abbildung 5.7 wird dann verwendet, um den Deklarationsaufruf *aCall* zu verzögern.

Modell 5.3.4. ***(Statisches Adaptermodell zum Weiterleiten einer Deklaration):***

$$
\begin{aligned}
\mathcal{A}^S =& (s^D, s^P) \\
s^D =& \{(\mathit{required.aCall}, \mathit{provided.aCall}), \\
& \quad (\mathit{provided.aResp}, \mathit{required.aResp})\} \\
s^P =& \{(\mathit{required.aCall.param1}, (0, \mathit{provided.aCall.param1}))\}
\end{aligned}
$$

Sobald die Deklaration *aCall* aufgerufen wird, betritt der dynamische Adapter Zustand β und verweilt dort für 10 Millisekunden. Sobald der Timer abgelaufen ist (β_timeout), wechselt der dynamische Adapter in Zustand α und ermöglicht

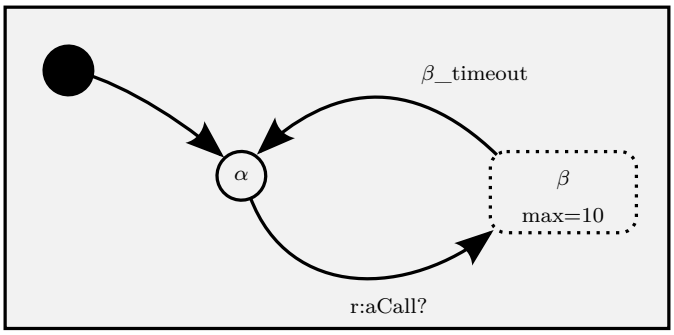


Abbildung 5.7.: Dynamisches Adaptermodell zum Verzögern eines Deklarationsaufrufs.

dem statischen Adapter den Aufruf an den Server weiter zu leiten. Ohne die Verwendung eines aktiven Zustandes würde der statische Adapter direkt nach dem Betreten von β das statische Mapping ausführen. Die Verzögerung eines Deklarationsaufrufs könnte auch komplett durch den dynamischen Adapter erfolgen, indem entsprechende Trigger und Aktionen hinzugefügt werden.

5.3.5. Zyklischer Aufruf einer Deklaration

Für die Modellierung eines zyklischen Aufrufs wird der dynamischen Adapter verwendet. Abbildung 5.8 zeigt das dynamische Adaptermodell für den zyklischen Aufruf der Deklaration *bCall*. Der dynamische Adapter betritt durch den Aufruf von *aCall* den Zustand β. Der dort definierte Timeout von 10 Millisekunden wird zyklisch ausgelöst, da die Transition mit dem Trigger *β_timeout* immer wieder den Zustand β betritt. Der zyklische Aufruf von *bCall* erfolgt solange, bis der Client *cCall* aufruft.

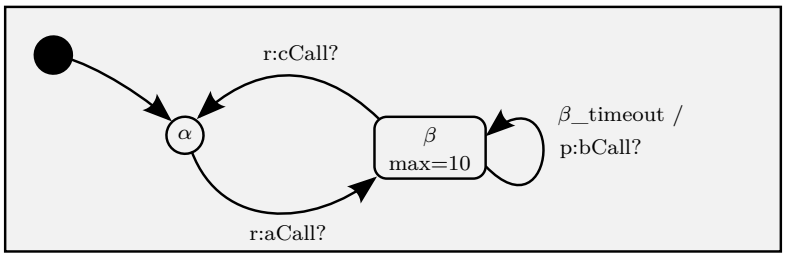


Abbildung 5.8.: Dynamisches Adaptermodell für einen zyklischen Deklarationsaufruf.

5.4. Fallstudie Parkassistent: Adaptermodellierung

Dieser Abschnitt zeigt die Adaptierung von Schnittstellen mit dem geschachtelten Adaptermodell am Beispiel des in Abschnitt 4.4 vorgestellten Parkassistenten. Betrachtet wird ein Client, der die Schnittstelle *ParkA1* benötigt, und ein Server, der die Schnittstelle *ParkA2* zur Verfügung stellt. Abbildung 5.9 stellt das Adaptierungsszenario grafisch dar. Neben den syntaktischen Unterschieden der Schnittstellen muss auch das Verhalten adaptiert werden (siehe Tabelle 4.1). Teilweise können Adaptierungen sowohl mit dem statischen als auch mit dem dynamischen Adapter umgesetzt werden. In diesen Fällen ist es dem Modellierer überlassen wie die Adaptierung umgesetzt wird. Hier wird eine Kombination aus statischer und dynamischer Adaptierung verwendet.

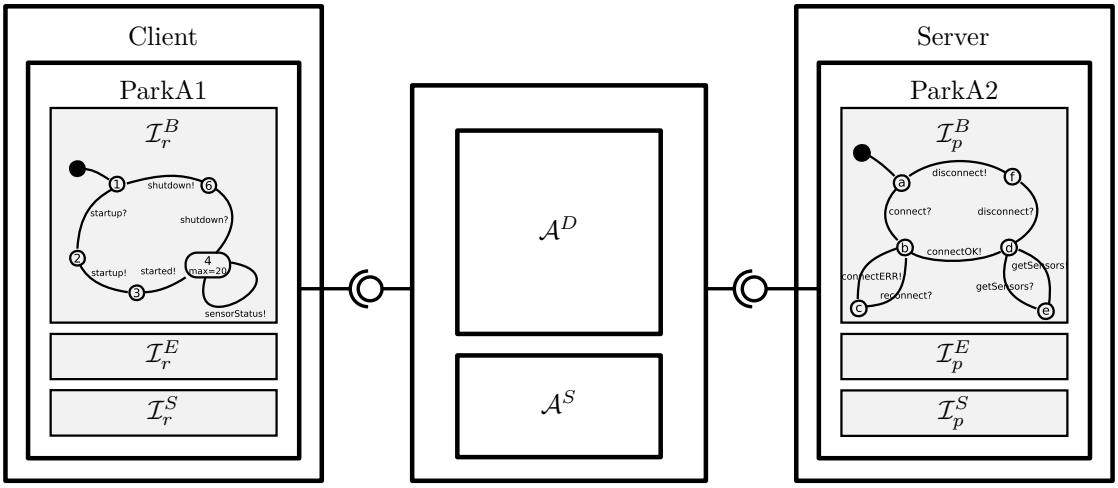


Abbildung 5.9.: Adaptierungsszenario Parkassistent.

5.4.1. Statischer Adapter

Grundsätzlich sind die Methoden zum Starten und Beenden der Schnittstellen *ParkA1* und *ParkA2* in ihrer Funktion sehr ähnlich und können statisch adaptiert werden. Modell 5.4.1 zeigt das statische Adaptermodell. Die Deklarationen *startupCall* und *startupResp* werden *connectCall* und *connectResp* zugeordnet. Analog werden auch die Deklarationen *disconnectCall* und *disconnectResp* gemappt. Zusätzlich wird die Deklaration *getSensorsResp* der Deklaration *sensorStatusCB*

zugeordnet. Die Parameter der gemappten Ausgabe-Deklarationen müssen ebenfalls gemappt werden. Parameter *retry*, *retryCount* werden mit einem konstanten Parameter-Mapping definiert. Die Funktionsweise der ParkA2 Schnittstelle ist so definiert, dass der Server dem Client bei Erstellung der Verbindung den Parameter *connectionID* sendet, der später für den Abbau der Verbindung wieder verwendet werden muss. Deshalb wird zusätzlich ein funktionales Parameter-Mapping für den Parameter *connectionIDDisconnect* definiert, welches den Wert des *connectionID* Parameters der letzten von diesem Client aufgerufenen *connect*-Deklaration zuweist. Abschließend werden alle Sensorwerte mit einem konstanten und einem funktionalen Parameter-Mapping modelliert.

Modell 5.4.1. ***(Parkassistent: Statisches Adaptermodell [1]):***

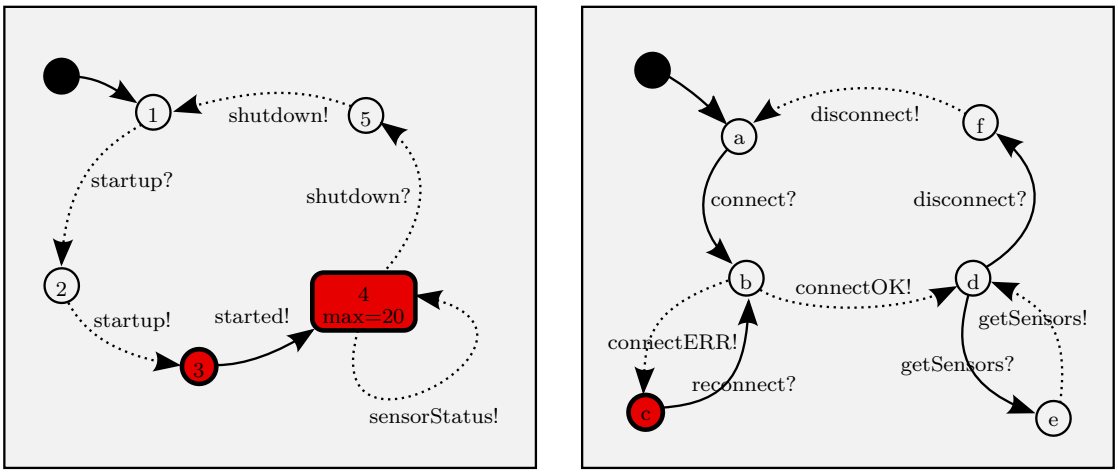
$\mathcal{A}^S = (s^D, s^P)$

$s^D = \{$(*P1.startupCall*, *P2.connectCall*),
(*P2.connectResp*, *P1.startupResp*),
(*P1.shutdownCall*, *P2.disconnectCall*),
(*P2.disconnectResp*, *P1.shutdownResp*),
(*P2.getSensorsR*, *P1.sensStatCB*)$\}$
(*P2.connect.retry*, $(0, \epsilon)$), (*P2.connect.retryCount*, $(0, \epsilon)$),
(*P1.sensStatCB.front_left*, (0, *(P2.getSensorsR.fl + P2.getSensorsR.fml)/2*)),
(*P1.sensStatCB.front_mid*, (0, *(P2.getSensorsR.fml + P2.getSensorsR.fmr)/2*)),
(*P1.sensStatCB.front_right*, (0, *(P2.getSensorsR.fmr + P2.getSensorsR.fr)/2*)),
(*P1.sensStatCB.rear_left*, (0, *(P2.getSensorsR.rl + P2.getSensorsR.rml)/2*)),
(*P1.sensStatCB.rear_mid*, (0, *(P2.getSensorsR.rml + P2.getSensorsR.rmr)/2*)),
(*P1.sensStatCB.rear_right*, (0, *(P2.getSensorsR.rmr + P2.getSensorsR.rr)/2*)),
(*P2.sensStatCB.quality*, $(100, \epsilon)$)$\}$

Abbildung 5.10 markiert in den Schnittstellen-Verhaltensmodellen welche Transitionen vom statischen Adaptermodell 5.4.1 abgedeckt sind. Zum Beispiel wird der erste `startup?` Aufruf vom Client bereits korrekt vom statischen Adapter in

[1] Einige Bezeichner (getSensorsResp, sensorStatusCB, ParkA1 und ParkA2) wurden gekürzt, um eine übersichtlichere Darstellung zu erhalten. ϵ ist der undefinierte Ausdruck.

einen `connect?`-Aufruf, der vom Server in Zustand a erwartet wird, übersetzt. Die beiden möglichen Antworten `connectERR!` und `connectOK` können ebenfalls vom statischen Adapter übersetzt werden. Die erste Verklemmung tritt auf, sobald der Server sich im Zustand c befindet. Dort erwartet er einen nochmaligen Aufruf der `connectCall`-Deklaration, welchen der Client in Zustand 3 nicht bereit stellt. Der Client befindet sich ebenfalls in einer Verklemmung, da er in Zustand 3 auf ein `started!`-Event wartet. Die dritte Inkompatibilität tritt in Zustand 4 auf, in dem der Client spätestens alle 20 Millisekunden `sensorStatus!` erwartet. Diese verbleibenden Inkompatibilitäten werden durch ein dynamisches Adaptermodell abgedeckt, welches im nächsten Abschnitt beschrieben wird.



$(I)_r^B$ $(I)_p^B$

Abbildung 5.10.: Fallstudie Parkassistent: Abdeckung des statischen Adaptermodells 5.4.1. Statisch adaptierte Transitionen sind mit gepunkteten Linien markiert. Die verbleibenden Inkompatibilitäten sind hervorgehoben.

5.4.2. Dynamischer Adapter

Für die Auflösung der beiden Verklemmungen und das Auslesen der Sensorwerte wird der dynamische Adapter verwendet. Abbildung 5.11 zeigt den dafür notwendigen Zustandsautomaten. Der dynamische Adapter betritt durch das Event `connect?` den aktiven Zustand β. Hier bleibt der dynamische Adapter solange aktiv, bis vom Server die Antwort `connectOK!` gesendet wird. Im Falle von `connectERR!` triggert der dynamische Adapter die Aktion `reconnect?`. Dieser Zustand muss als aktiver Zustand modelliert werden, damit der Deklarationsaufruf `connectResp` im Falle eines Fehlers nicht vom statischen Adapter adaptiert wird. In Zustand γ wartet der dynamische Adapter auf das Event `startup!`, welches vom statischen Adapter an den Client gesendet wird, und führt danach die Emulation `started!` aus. Im nachfolgenden Zustand δ triggert der dynamische Adapter nach 10 Millisekunden die Emulation `getSensors?`. Die Antwort auf diesen Aufruf wird anschließend vom statischen Adapter bearbeitet. Letztendlich wird der dynamische Adapter mit einem Aufruf von `shutdown?` wieder in den Startzustand α versetzt.

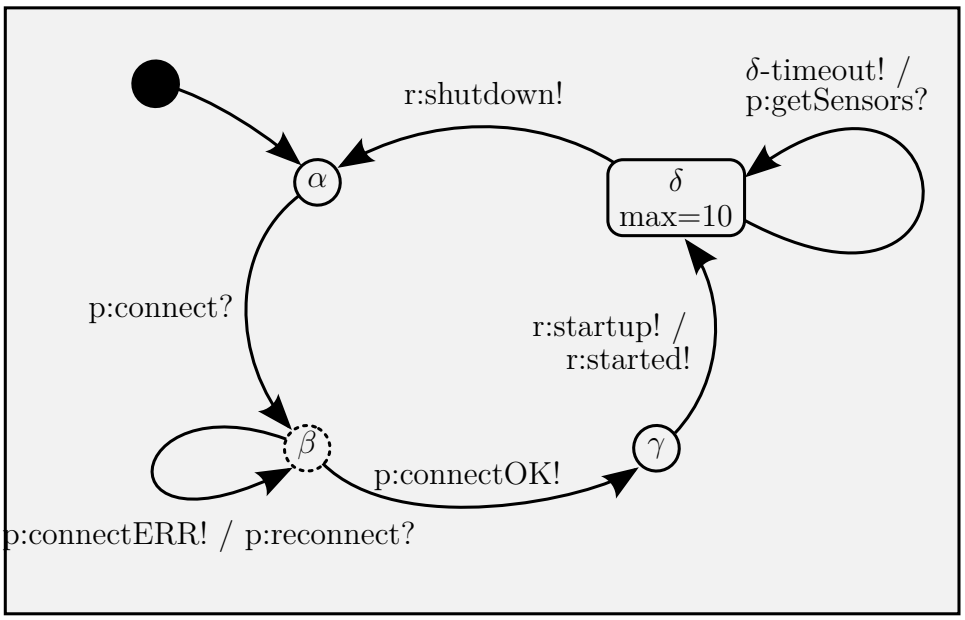


Abbildung 5.11.: Fallstudie Parkassistent: dynamisches Adaptermodell.

5.4.3. Adapter-Kontrollfluss im Beispiel ParkA

Um das Zusammenspiel des geschachtelten Adapters im Beispiel besser zu verdeutlichen, zeigt Abbildung 5.12 exemplarisch den Ablauf für den Verbindungsaufbau. Der Client ruft zur Initialisierung der Verbindung die Deklaration `startupCall`

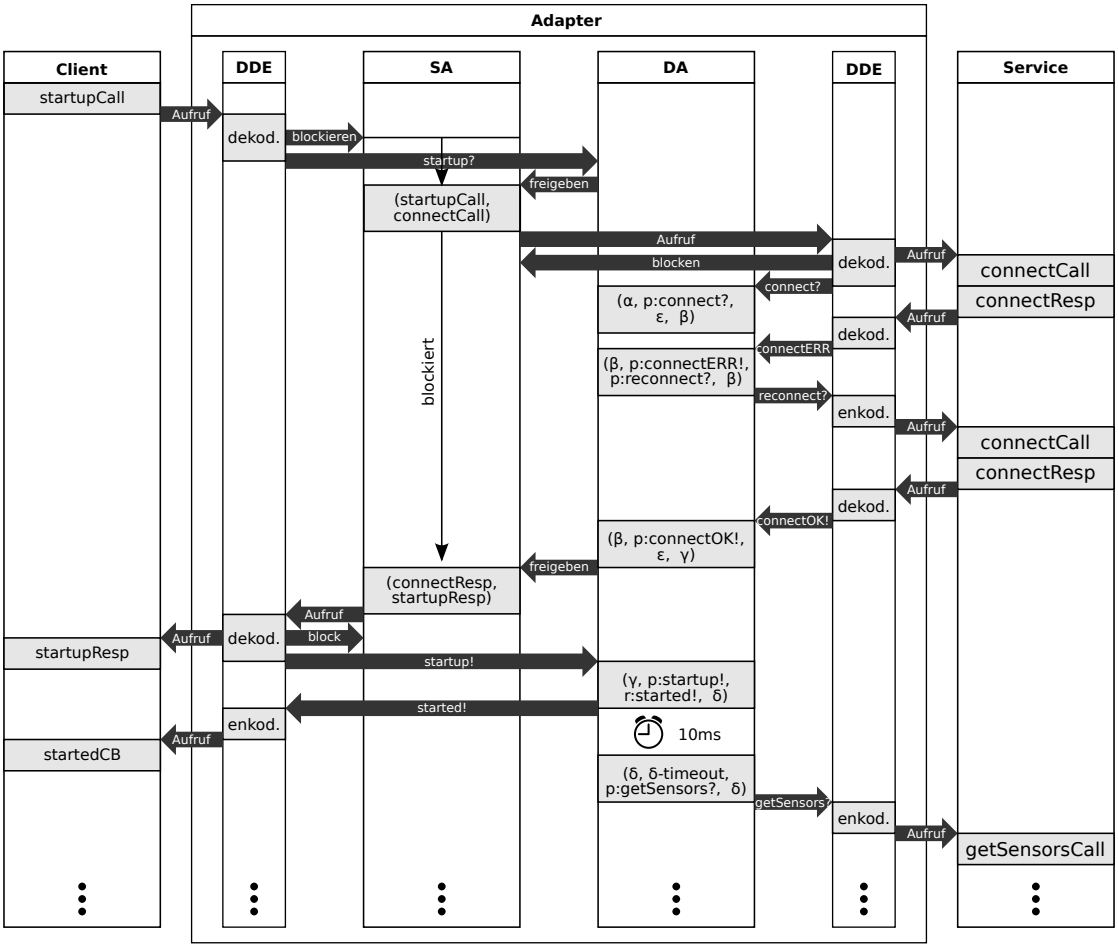


Abbildung 5.12.: Fallstudie Parkassistent: Sequenz der adapterinternen Abläufe beim Verbindungsaufbau.

auf. Der Aufruf wird im DDE-Modul dekodiert und an den dynamischen Adapter weitergeleitet. Dieser beschreibt für den Zustand α keine entsprechende Transition und gibt die Kontrolle an den statischen Adapter weiter[2]. Daraufhin führt der stati-

[2]Für die weitere Beschreibung des Beispiels wird dieser Mechanismus nur dann beschrieben, falls ein dekodiertes Event auch tatsächlich den dynamischen Adapter triggert.

sche Adapter das Deklarationsmapping durch und ruft beim Server die Deklaration `connectCall` mit der im statischen Adapter festgelegten Parameterbelegung auf. Dieser Aufruf wird vom DDE-Modul dekodiert (`p:connect?`) und dem dynamischen Adapter weitergeleitet, welcher die erste Transition triggert und den aktiven Zustand β betritt. Dies hat zur Folge, dass der statische Adapter nicht freigegeben und der aktuelle Deklarationsaufruf nicht statisch adaptiert wird. Die Antwort des Servers `connectResp` wird ebenfalls dekodiert und an den dynamischen Adapter weitergeleitet. Für die erste Rückgabe wird das Event `p:connectERR!` dekodiert und der dynamische Adapter führt die Aktion `p:reconnect?` durch. Dabei bleibt der dynamische Adapter im selben Zustand. Bei der darauffolgenden Rückgabe von `p:connectResp` wird das Event `p:connectOK` dekodiert und der statische Adapter wird somit freigegeben. Dieser führt für den letzten Deklarationsaufruf das statische Mapping durch und ruft deshalb die Deklaration `startupResp` auf. Auch dieser Deklarationsaufruf wird vom DDE-Modul dekodiert und hat ein Triggern des dynamischen Adapters zur Folge. Dieser führt die Aktion `r:started!` durch und startet mit der zyklischen Abfrage der Sensorwerte (`p:getSensors?`).

5.5. Validierung von statischen und dynamischen Adaptern

Durch die einheitliche Modellierung von Schnittstellen und Adaptern auf Grundlage der Schnittstellen-Verhaltensmodelle und der Schnittstellen-Eventmodelle, kann bereits auf Modellebene eine Validierung des Adapters in Bezug auf die Korrektheit des Kontrollflusses erfolgen. In diesem Kontext ist ein System korrekt, wenn es Deadlockfrei ist und nur spezifizierte Interaktionen durchführt [BCT04; YS97]. Für den Adapter bedeutet dies, dass er alle in einem Zustand mögliche Eingaben verarbeiten kann und nur in diesem Zustand spezifizierte Ausgaben produziert. Für die Definitionen von Deadlockfreien Zuständen und Graphen-Kompatibilität sei an dieser Stelle auf Canal et al .[CPS08] verwiesen. Für den hier vorgestellten Anwendungsfall des geschachtelten Adaptermodells können existierende Model-Checking Ansätze nicht verwendet werden, da sie die Trennung von statischem und dynamischem Adapter nicht berücksichtigen.

Abbildung 5.13 zeigt ein einfaches Beispiel für die drei Verhaltensmodelle. Das Verhaltensmodell der benötigten Schnittstelle ($\mathcal{I}_r^B$) definiert zwei mögliche Events, welche im Zustand 1 vom Client zum Server gesendet werden können. Der Server bietet lediglich die Methode `call_a`, welche im Zustand `a` aufgerufen werden kann. Das dynamische Adaptermodell realisiert ein Mapping von `call_1` auf `call_a`.

Ein Model-Checking Algorithmus muss nun prüfen, ob das Adaptermodell einen korrekten Mediator zwischen angebotener und benötigter Schnittstelle darstellt.

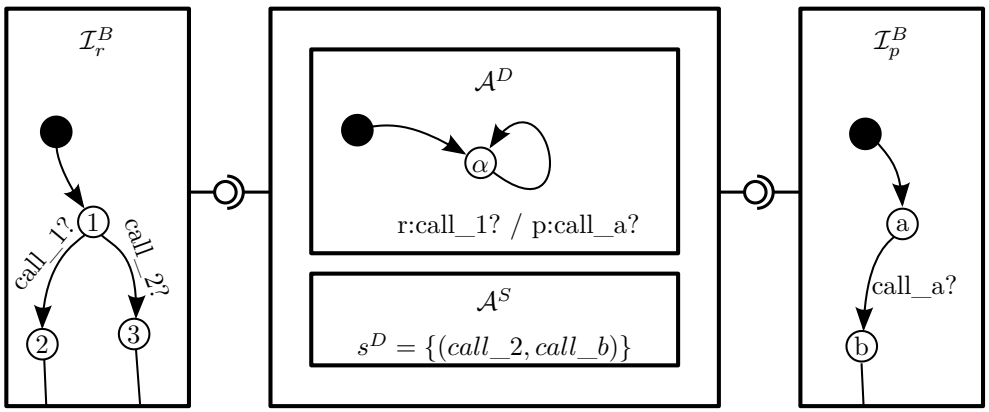


Abbildung 5.13.: Beispiel eines Adapters als Grundlage für die Validierung.

Der Model-Checking Algorithmus implementiert eine Tiefensuche durch die Verhaltensmodelle $\mathcal{I}_r^B, \mathcal{I}_p^B$ und $\mathcal{A}^D$. Während der Tiefensuche wird ein Graph aufgebaut, um auszuschließen, dass ein bereits validierter Zustand mehrfach validiert wird. Ein *Validierungsknoten* im Graphen wird dabei eindeutig durch den jeweiligen Zustand der beiden Schnittstellen-Verhaltensmodelle ($\mathcal{I}_r^B$ und $\mathcal{I}_p^B$) und dem Zustand des dynamischen Adapters ($\mathcal{A}^D$) definiert. Sobald einer der drei Zustandsautomaten seinen Zustand wechselt, handelt es sich um einen neuen Validierungsknoten. Zusätzlich zu den Zuständen der Automaten werden in jedem Validierungszustand die gefundenen Fehler gespeichert, welche das Validierungsergebnis beschreiben. Abbildung 5.14 zeigt die ersten Schritte des Algorithmus für das Beispielmodell aus Abbidlung 5.13. Im ersten Schritt wird ein Validierungsknoten mit den Initialzuständen der drei Automaten erstellt (1, α, a). Anschließend werden Kanten für alle in diesem Validierungszustand möglichen Adapter-Eingabeevents erstellt. Auf der Kante wird die entsprechende Adaptierung annotiert. Im Fall eines Aufrufs von `call_1` findet eine Adaptierung durch den dynamischen Adapter statt, welcher dem Server ein `call_a`-Event sendet. Dabei wechseln Client und Server die Zustände und der Adapter bleibt im selben Zustand. Bei einem Aufruf von `call_2` findet eine statische Adaptierung statt indem das Event `call_b` an den Server gesendet wird. Dieses Event wird in `a` nicht akzeptiert. Deshalb wird dieser Zweig der Validierung abgebrochen und der Fehler im Validierungszustand gespeichert.

Listing 5.1 zeigt die Schritte für die Vorbereitung der Adapter-Validierung in der Programmiersprache Xtend [**_xtend_2015**]. Xtend ist eine ausdrucksstar-

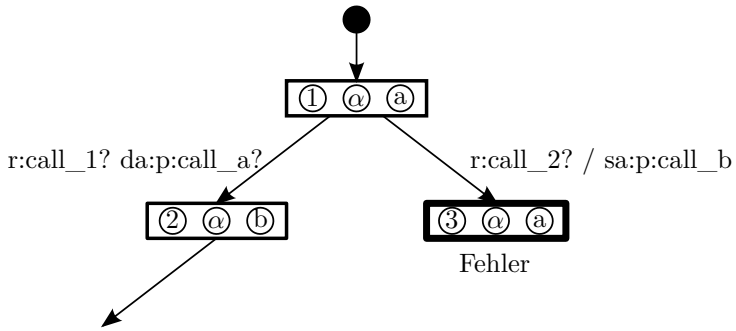


Abbildung 5.14.: Schrittweise Validierung des Adapters mit Validierungsknoten. Der Knoten mit dem Validierungsfehler ist fett eingerahmt.

ker und flexibler Java-Dialekt aus dem ausführbarer Java-Code generiert wird [**_xtend_2015**]. In den Zeilen 1 bis 10 wird der erste Validierungsknoten mit den Initialzuständen und ein Kontext für die Validierung erstellt. Dieser Kontext stellt sicher, dass während der gesamten Validierung auf alle Modellelemente zugegriffen werden kann.

```
1  var requiredBehaviorModel = adapter.requiredInterfaceBehaviorModel
2  var providedBehaviorModel = adapter.providedInterfaceBehaviorModel
3
4  var reqFirst = requiredBehaviorModel.firstState
5  var provFirst = providedBehaviorModel.firstState
6  var daFirst = dynamicAdapter.firstState
7
8  var context = new ValidationContext(adapter)
9  var firstNode = new ValidationNode(null, reqFirst, provFirst, daFirst)
10 firstNode.setContext(context)
11
12 validateAdapter(firstNode);
```

Listing 5.1: Vorbereitung der Adapter-Validierung

Die im letzten Schritt aufgerufene Validierungsmethode implementiert das Verhalten, welches in Beispiel 5.14 gezeigt wurde. Der rekursive Model-Checking Algorithmus durchläuft beginnend vom ersten Validierungsknoten alle möglichen Abläufe zwischen Client und Server auf Basis der Schnittstellen-Verhaltensmodelle. Listing 5.2 zeigt die Implementierung des rekursiven Algorithmus in der Programmiersprache Xtend.

```
1  def void validateAdapter(ValidationNode node){
2     // Markiere den aktuellen Validierungsknoten als bereits analysiert
3     node.setAlreadyAnalyzed
4
```

```
5     // Validiere jedes Adapter-Eingabeevent
6     for (transition : node.getPossibleAdapterInputTransitions){
7        var ValidationNode newNode
8        var ValidationEdge newEdge
9
10       // Falls der aktuelle Validierungszustand das Event dynamisch adaptiert,
11       // führe den dynamischen Adapter aus
12       if (node.adaptsDynamically(transition.trigger)){
13          newEdge = new ValidationEdge(node, transition.trigger)
14          newNode = node.executeDynamicAdapter(transition.trigger, newEdge)
15          newEdge.setTarget(newNode)
16       }
17       // Falls der aktuelle Validierungszustand das Event statisch adaptiert,
18       // führe den statischen Adapter aus
19       else if (node.adaptsStatically(transition.trigger) && !node.isActive){
20          newEdge = new ValidationEdge(node, transition.trigger)
21          newNode = node.executeStaticAdapter(transition.trigger, newEdge)
22          newEdge.setTarget(newNode)
23       }
24       // Falls das Event nicht adaptiert wurde, füge einen Fehler zum Validierungsergebnis hinzu
25       else
26          node.addDeadlockValidationError(transition)
27
28       // Rekursiver Aufruf der Validierung falls der neue Knoten noch nicht validiert wurde und
29       // im aktuellen Knoten kein Fehler gefunden wurde
30       if (newNode != null && !newNode.alreadyAnalyzed && !node.errorDetected)
31          newNode.validateAdapter
32    }
33 }
```

Listing 5.2: rekursiver Model-Checking Algorithmus

Die Funktion simuliert die Ausführung des Adapters auf Grundlage der in einem Validierungsknoten möglichen Adapter-Eingabeevents (Zeile 6). Für jedes Adapter-Eingabeevent wird entweder der dynamische (Zeile 12) oder der statische Adapter(Zeile 19) ausgeführt. Falls einer der beiden Adaptierungsfälle auftritt, wird eine neue Kante in den Validierungsgraphen eingefügt. In dieser Kante werden die einzelnen Adaptierungsschritte für eine spätere Analyse gespeichert. Falls sowohl der statische als auch der dynamische Adapter die aktuelle Eingabe nicht adaptieren, befindet sich der Adapter in einem Deadlock, welcher protokolliert wird (Zeile 26). Dieser Zustand wird als Fehler gespeichert und anschließend die Validierung mit den noch ausbleibenden Adapter-Eingabeevents fortgesetzt. Wird während der Adaptierung ein neuer Validierungsknoten erreicht, der noch nicht validiert wurde, wird die Funktion rekursiv mit diesem Knoten als Eingabe aufgerufen. Die Rekursion wird abgebrochen, sobald ein bereits durchlaufener Validierungsknoten erreicht wird oder im aktuellen Knoten ein Fehler entdeckt wurde (Zeile 30). Die Implementierung der Methoden `executeDynamicAdapter` und `executeStaticAdapter` wird in den nächsten Abschnitten gezeigt.

5.5.1. Applikation des dynamischen Adapters

Die Applikation des dynamischen Adapters wird nur aufgerufen, wenn bereits geprüft wurde, ob das aktuelle Adapter-Eingabeevent im aktuellen Validierungszustand eine dynamische Adaptierung zur Folge hat. Listing 5.3 zeigt die Schritte zur Ausführung des dynamischen Adapters. Im ersten Schritt in Zeile 3 wird das eingehende Event in allen drei Zustandsautomaten (Client, Adapter und Server) getriggert. In Zeile 6 wird die triggernde Transition des dynamischen Adapters vom aktuellen Validierungsknoten ermittelt. Anschließend werden in Zeile 7 alle Aktionen auf der gerade ausgeführten Transition des dynamischen Adapters in den drei Zustandsautomaten getriggert. Nach der Ausführung aller Aktionen wird in Zeile 12 geprüft, ob das ursprüngliche Event auch statisch adaptiert werden kann, und ob es sich beim aktuellen Zustand im dynamischen Adapter nicht um einen aktiven Knoten handelt. In diesem Fall wird der statische Adapter ebenfalls ausgeführt (Zeile 13). Anschließend wird der neue Validierungsknoten zurückgegeben. Dabei kann es sich auch um einen bereits betrachteten Knoten handeln.

```
1  def ValidationNode executeDynamicAdapter(SignalEvent event, ValidationEdge currentEdge) {
2     // Schritt 1 - Triggern aller Zustandsautomaten mit dem eingehenden Event
3     var newNode = triggerEvent(event)
4
5     // Schritt 2 - Dynamischen Adapter ausführen
6     val Transition daTrans = getDynamicAdapterTransition(event)
7     for (action: daTrans.actions){
8        if (!newNode.errorDetected)
9           newNode = newNode.triggerDynamicAdapter(action, currentEdge)
10    }
11    // Schritt 3 - Prüfen ob das ursprüngliche Event auch vom statischen Adapter adaptiert wird
12    if (newNode.adaptsStatically(event) && !newNode.errorDetected && !newNode.isActive)
13       newNode = newNode.executeStaticAdapter(event, currentEdge)
14
15    // Rückgabe des neuen Validierungsknotens
16    return newNode
17 }
```

Listing 5.3: Ausführung des dynamischen Adapters während der Validierung.

5.5.2. Applikation des statischen Adapters

Die Applikation des statischen Adapters wird nur aufgerufen, wenn bereits geprüft wurde, ob die aktuelle Adapter-Eingabedeklaration eine statische Adaptierung zur Folge hat. Die Applikation des statischen Adapters auf eine Deklaration erfolgt in drei Schritten. Listing 5.4 zeigt die Funktion in der Programmiersprache Xtend. Zuerst verarbeiten alle Zustandsautomaten das Adapter-Eingabeevent,

welches die statische Adaptierung auslöst (Zeile 3). Anschließend wird das statische Mapping angewandt, welches möglicherweise einen neuen Validierungszustand zur Folge hat (Zeile 6). Die Methode `triggerStaticMapping` triggert dabei nur die Zustandsautomaten der Schnittstellen, denn das ausgegebene Event muss eventuell im Anschluss noch vom dynamischen Adapter verarbeitet werden. In Zeile 11 wird deshalb geprüft, ob der dynamische Adapter beim Event `mappedEvent` feuert und ob im aktuellen Validierungsknoten nicht bereits ein Fehler festgestellt wurde. In diesem Fall wird zusätzlich der dynamische Adapter ausgeführt (Zeile 12). Anschließend wird der neue Validierungsknoten zurückgegeben. Dabei kann es sich auch um einen bereits betrachteten Knoten handeln.

```
1  def ValidationNode executeStaticAdapter(SignalEvent event, ValidationEdge currentEdge) {
2     // Schritt 1 - Triggern aller Zustandsautomaten mit dem eingehenden Event
3     var newNode = triggerEvent(event)
4
5     // Schritt 2 - Statisches Mapping ausführen
6     newNode = newNode.triggerStaticMapping(event, currentEdge)
7     val mappedEvent = context.sa.getMappedEvent(event, context)
8
9     // Schritt 3 - Prüfen ob das durch den statischen Adapter verursachte Event
10    // den           dynamischen Adapter triggert
11    if (newNode.adaptsDynamically(mappedEvent) && !newNode.errorDetected){
12       newNode = newNode.executeDynamicAdapter(mappedEvent, currentEdge)
13    }
14    // Rückgabe des neuen Validierungsknotens
15    return newNode
16 }
```

Listing 5.4: Ausführung des statischen Adapters während der Validierung.

5.5.3. Ergebnisgraph und Validierungsfehler

In Listing 5.2 wurde bereits gezeigt, wie erkannt wird, ob ein Adapter-Eingabeevent vom Adapter verarbeitet werden kann. Falls es nicht verarbeitet werden kann, muss ein entsprechender Validierungsfehler ausgegeben werden. Da der aktuelle Validierungsknoten genaue Information über den aktuellen Zustand der drei Automaten besitzt, kann der Fehler auch genau einem Zustand in den Schnittstellen-Verhaltensmodellen zugeordnet werden.

Zusätzlich zu den Adapter-Eingabeevents müssen auch die Ausgaben des Adapters validiert werden. Während der Validierung wird beim Auslösen jeder Transition geprüft, ob die Schnittstellen-Verhaltensmodelle der angebotenen und benötigten Schnittstelle das vom Adapter ausgegebene Event im aktuellen Validierungszustand akzeptieren. Falls nicht, wird ein entsprechender Fehler für den aktuellen

Validierungsknoten gespeichert. Listing 5.5 zeigt die Implementierung dieser Validierung. Falls das Event der angebotenen Schnittstelle zugeordnet werden kann und es sich um ein `InputEvent` handelt, muss der aktuelle Zustand im Interface-Verhaltensmodell der angebotenen Schnittstelle auch eine Transition besitzen, welche dieses Event triggert (Zeile 5). Dasselbe gilt für Ausgaben vom Adapter in Richtung Client (Zeile 10).

```
1  def validateEventCompatibility(EventTrigger trigger) {
2     var event = trigger.event
3     // Eingabeevents der angebotenen Schnittstelle sollten im aktuellen Zustand des Server
4     // eine Transition mit entsprechendem Trigger besitzen
5     if (event.isProvided && event.isInputEvent && !providedState.triggers(event)){
6        addProvidedInterfaceError(trigger, event)
7     }
8     // Ausgabeevents der benötigten Schnittstelle sollten im aktuellen Zustand des Client
9     // eine Transition mit entsprechendem Trigger besitzen
10    if (event.isRequired && event.isOutputEvent && !requiredState.triggers(event)){
11       addRequiredInterfaceError(trigger,event)
12    }
13 }
```

Listing 5.5: Erkennung von Validierungsfehlern.

5.5.4. Fallstudie Parkassistent: Adaptervalidierung

Die Adaptervalidierung wird nun mithilfe des Fallbeispiels aus Abschnitt 5.4 veranschaulicht. Abbildung 5.15 zeigt den Validierungsgraphen als Ergebnis der Adaptervalidierung. Für die Beschriftung der Kanten werden zusätzlich zu der in Abschnitt 5.1.2 eingeführten Notation noch die Präfixe *sa* und *da* verwendet. Sie definieren, ob eine Aktion vom statischen oder vom dynamischen Adapter durchgeführt wird.

An diesem Beispiel wird auch das Verhalten des Adapters in einem aktiven Zustand sichtbar. Im aktiven Zustand $(2, \beta, b)$ gibt es die möglichen Adapter-Eingabeevents `p:connectOK!` und `P:connectERR!`. Im Falle von `P:connectERR!` ist der Zielzustand des dynamischen Adapters der aktive Zustand β und deshalb wird nach der Aktion `da:p:reconnect?` kein statisches Mapping ausgeführt. Konkret bedeutet dies, dass der Adapter solange im Zustand β verweilt, bis der Server mit `connectOK!` antwortet. Dann verlässt der dynamische Adapter den aktiven Zustand β und betritt den passiven Zustand δ. Deshalb wird das statische Mapping `sa:r:startup!` durchgeführt. Somit ist der Deklarationsaufruf `startup?` mit der entsprechenden Antwort `startup!` quittiert worden.

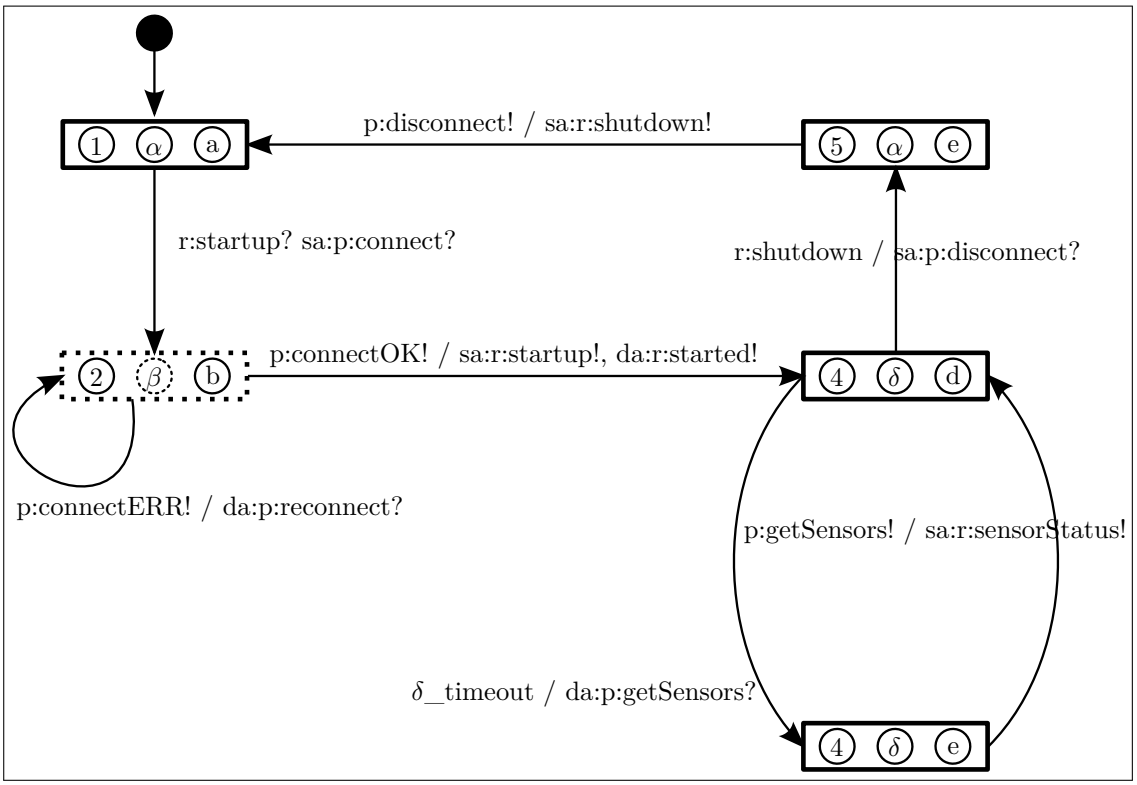


Abbildung 5.15.: Validierungsgraph für das Fallbeispiel Parkassistent. Aktive Zustände und daraus folgende aktive Validierungszustände sind mit einer gepunkteten Umrandung gekennzeichnet.

5.6. Automatisches Adapter-Deployment

Ein Ziel bei der Entwicklung des Adaptermodells war die Generierung eines lauffähigen Adapters direkt aus den Modellen. Deshalb kann jedes der in Abschnitt 5.2 gezeigten Module aus den Modellen vollständig generiert werden. Abbildung 5.16 zeigt die Zuordnung von Modellen zu Modulen.

Die Basis für die korrekte Implementierung des Adapters ist die Generierung der Client- und Server-Stubs. Wie in Abschnitt 4.2 beschrieben, werden diese aus den syntaktischen Schnittstellenmodellen generiert. Für eine konkrete IDL erledigt diesen Generierungsschritt der dazugehörende Code-Generator. Der Client-Proxy wird aus dem Schnittstellen-Syntaxmodell der benötigten Schnittstelle $\mathcal{I}_r^S$ und der Server-Proxy aus dem Schnittstellen-Syntaxmodell der angebotenen Schnittstelle $\mathcal{I}_p^S$ generiert. Dieser Generierungsschritt stellt die technische Kompatibilität sicher.

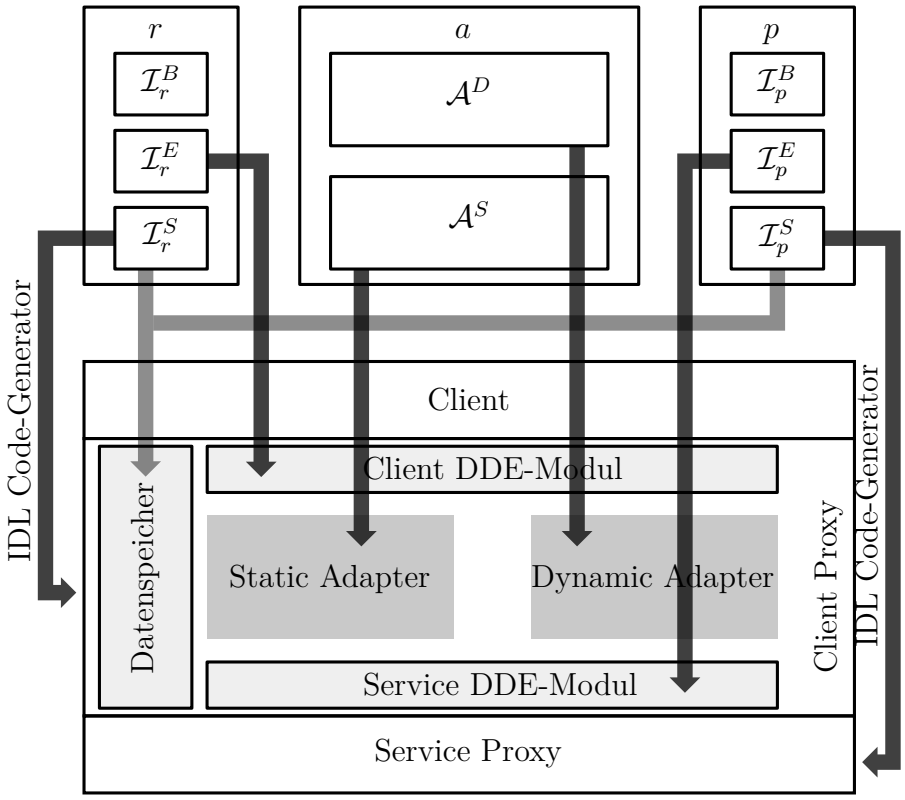


Abbildung 5.16.: Generierung der Adaptermodule aus den Modellen [Pra+14].

Die hier vorgestellte Adapterarchitektur wird in den Client-Proxy integriert und verwendet den generierten Server-Proxy für die Kommunikation mit dem Server.

Das Datenspeicher-Modul wird aus den Schnittstellen-Syntaxmodellen ($\mathcal{I}_r^S$ und $\mathcal{I}_p^S$) generiert. Für jeden Parameter, der von den DDE-Modulen oder dem statischen Adapter benötigt wird, wird ein Feld generiert, das sowohl geschrieben als auch gelesen werden kann.

Die DDE-Module werden aus dem jeweiligen Schnittstellen-Eventmodell $\mathcal{I}^E$ generiert. Es enthält die für die Übersetzung von Deklarationsaufrufen und Events notwendigen Definitionen. Aus den Constraints wird die Dekodierung der Deklarationsaufrufe generiert und aus den Emulationen die Enkodierung. Die Performance des Adapters kann dadurch verbessert werden, dass nur die im dynamischen Adaptermodell tatsächlich verwendeten Events verarbeitet werden.

Letztendlich kann der statische Adapter aus dem statischen Adaptermodell $\mathcal{A}^S$, und der dynamische Adapter aus dem dynamischen Adaptermodell $\mathcal{A}^D$ generiert werden.

5.7. Bewertung

In diesem Abschnitt wird das Konzept in Bezug auf die in Abschnitt 2.5 definierte Aufgabenstellung bewertet. Der Vergleich erfolgt dabei mit der aktuell gängigsten, der rein zustandsbasierten Adaptierung mit endlichen Automaten (siehe Abschnitt 3.3.1). Der Ansatz enthält dabei Anteile aus allen drei in Abschnitt 3.3.1 vorgestellten Modellierungsansätzen und kombiniert dadurch deren Vorteile.

- **Datenzentrierte Adaptierung:** Der Ansatz unterstützt die Adaptierung von Daten durch die Berechnung der Deklarationsparameter.
- **Verhaltensadaptierung mithilfe von Mappings:** Das statisches Adaptermodell bietet Muster um häufige Adaptierungen einfach zu beschreiben.
- **Verhaltensadaptierung mithilfe von endlichen Automaten:** Der Ansatz beschreibt das dynamische Adaptermodell mit einem endlichen Automaten und verwendet diesen auch unverändert für die Ausführung.

5.7.1. Einfachheit

Verschiedene Adaptierungen, wie zum Beispiel die Neuberechnung von Parametern, erfordern in der Praxis die Expertise eines Spezialisten. Auch für die Zukunft kann davon ausgegangen werden, dass verschiedene Aufgaben bei der Modellierung eines Adapters manuell durchgeführt werden müssen. Deshalb ist es wichtig, dass das Modell für den Menschen verständlich bleibt. Ein Kompatibilitätsbruch zwischen zwei Schnittstellenversionen wird sehr häufig durch syntaktische, verhaltensneutrale Schnittstellenänderungen verursacht [DJ05]. Diese Änderungen betreffen somit die syntaktische Schnittstelle und sind globaler Natur. Genau solche Änderungen werden mit dem statischen Adaptermodell adressiert. Zum Beispiel muss für die Umbenennung einer Deklaration nur ein Mapping im statischen Adapter definiert werden. Ähnlich einfach können Parameter umbenannt, entfernt oder neu berechnet werden.

5.7.2. Ausführungsrobustheit

Der statische Adapter ist global gültig und wird, solange er nicht vom dynamischen Adapter explizit blockiert wird, in jedem Zustand ausgeführt. Um die gleiche Robustheit mit einer rein zustandsbasierten Modellierung zu erreichen, müssten die globalen Adaptierungen an jeden Zustand geheftet werden, was den Automaten unnötig kompliziert machen würde.

5.7.3. Modellgröße

Die Anzahl der Elemente im dynamischen Adapter wird dadurch reduziert, dass einfache, global gültige Mappings in den statischen Adapter verlagert werden. Zur Veranschaulichung wird hier das ParkA-Beispiel aus zwei Perspektiven betrachtet.

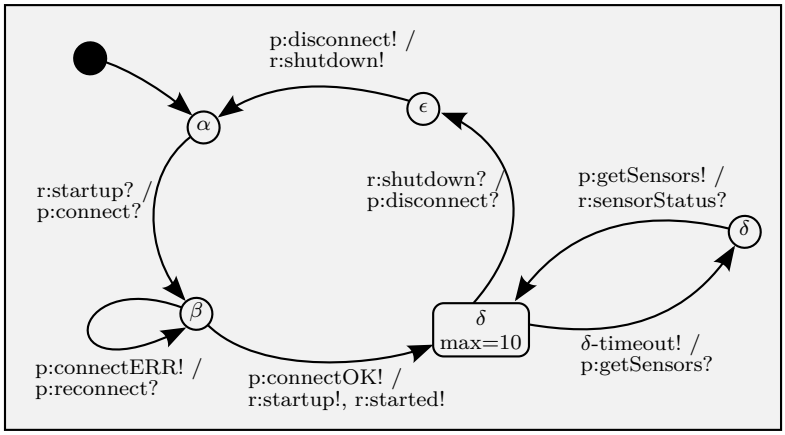


Abbildung 5.17.: Rein dynamisches Adaptermodell zur Vermittlung zwischen einer benötigten Schnittstelle ParkA1 und einer angebotenen ParkA2.

Abbildung 5.17 zeigt die rein dynamische Adaptierung eines ParkA1 Clients, um mit einem ParkA2 Server zu kommunizieren. Ein derartiges Modell benötigt keinen statischen Adapter, um korrekt zu arbeiten. Die Verwendung des statischen Adapters (siehe Modell 5.4.1) resultiert im dynamischen Adapter aus Abbildung 5.11. Wenn man die Graphen aus Abbildung 5.17 und 5.11 vergleicht, wird eine Reduzierung der Zustände von fünf auf vier und der Transitionen von acht auf sieben erreicht. Die Anzahl der Aktionen wird von acht auf drei reduziert. Dieses Beispiel zeigt, dass nicht für jeden Anwendungsfall eine drastische Reduzierung der Elemente zu erwarten ist.

Falls die Client- und Serverversionen getauscht werden ist es möglich, eine weitaus deutlichere Reduktion des Zustandsautomaten zu erreichen. Modell 5.7.1 zeigt ein rein statisches Adaptermodell, welches die gesamte Adaptierung für diesen Anwendungsfall beschreibt. Ein zusätzliches dynamisches Adaptermodell ist nicht nötig, denn alle Adapter-Ausgabedeklarationen sind einer Eingabedeklaration zugewiesen.

Modell 5.7.1. ***(Statisches Adaptermodell für eine benötigte Schnittstelle ParkA2 (P2) und eine angebotene ParkA1 (P1)):***

$$\mathcal{A}^S = (s^D, s^P)$$

$$s^D = \{(P2.connectCall, P1.startupCall), (P1.startupResp, P2.connectResp), (P2.disconnectCall, P1.shutdownCall), (P1.shutdownResp, P1.disconnectResp), (P2.getSensorsCall, P2.getSensorsResp)\}$$

$$\begin{aligned} s^P = \{&(P2.connectResp.connectionIDConnect, (0, (P2.connectResp.connectionIDConnect) + 1)),\\ &(P2.connectResp.status, (OK, \epsilon)),\\ &(P2.getSensorsResp.fl, (0, P1.sensorStatusCB.front\_left)),\\ &(P2.getSensorsResp.fml, (0, (P1.sensorStatusCB.front\_left + P1.sensorStatusCB.front\_mid)/2)),\\ &(P2.getSensorsResp.fmr, (0, (P1.sensorStatusCB.front\_mid + P1.sensorStatusCB.front\_right)/2)),\\ &(P2.getSensorsResp.fr, (0, P1.sensorStatusCB.front\_right)),\\ &(P2.getSensorsResp.rl, (0, P1.sensorStatusCB.rear\_left)),\\ &(P2.getSensorsResp.rml, (0, (P1.sensorStatusCB.rear\_left + P1.sensorStatusCB.rear\_mid)/2)),\\ &(P2.getSensorsResp.rmr, (0, (P1.sensorStatusCB.rear\_mid + P1.sensorStatusCB.rear\_rignt)/2)),\\ &(P2.getSensorsResp.rr, (0, P1.sensorStatusCB.rear\_right)),\\ &(P2.getSensorsResp.rr, (0, P1.sensorStatusCB.rear\_right))\} \end{aligned}$$

Für die restlichen Eingabedeklarationen wird angenommen, dass ein leeres Mapping definiert wurde. Der Aufruf von *connect* wird an die Deklaration *startup* weitergeleitet. Für die Antwort (*connectResp*) wird der Parameter Status immer mit *OK* verknüpft. So wird vermieden, dass der Client den Fehlerpfad betritt und die Verbindung nochmals aufbauen möchte. Die Sensorwerte erhält der Adapter nach dem Verbindungsaufbau automatisch vom Server durch den *sensorStatusCB* Callback und leitet diese an den Client weiter, sobald dieser die Methode *getSensors* aufruft. Falls noch keine Sensorwerte empfangen wurden, wird *0* zurückgegeben. Das Abbrechen der Verbindung erfolgt ebenfalls durch ein statisches Mapping. Somit erreichen wir in diesem Fall durch das hierarchische Schnittstellenmodell eine komplette Reduktion des Zustandsautomaten. Als Vergleich wird in Abbildung 5.18 das funktional äquivalente rein dynamische Adaptermodell mit vier Zuständen, sechs Transitionen und fünf Aktionen gezeigt.

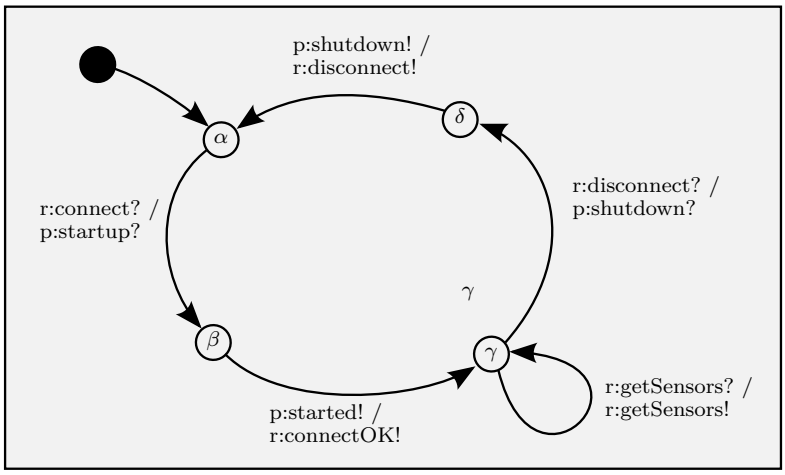


Abbildung 5.18.: Rein dynamisches Adaptermodell zur Vermittlung zwischen einer benötigten Schnittstelle ParkA2 und einer angebotenen ParkA1.

5.8. Zusammenfassung

Das Konzept der geschachtelten Adaptierung mit statischen und dynamischen Adaptern baut auf dem hierarchischen Schnittstellenmodell auf und ermöglicht die Aufteilung des Adapters in zwei Modelle. Das statische Adaptermodell definiert global gültige Adaptierungen in einer einfachen Art und Weise. Das dynamische Adaptermodell stellt eine etwas flexiblere zustandsbasierte Adaptierung dar. Die wesentlichen Eigenschaften dieser Art der Modellierung sind die Vereinfachung von häufigen Adaptierungen, die Robustheit der Ausführung und die Reduzierung der Größe des Zustandsautomaten. Die Modelle und Architekturen wurden in diesem Kapitel unabhängig vom konkreten Zielsystem definiert. Im nächsten Kapitel wird eine konkrete Umsetzug der Modelle für GENIVI Linux vorgestellt.

Kapitel 6

Evaluierung am Beispiel GENIVI Linux

Die beiden vorhergehenden Kapitel stellen die Konzepte der hierarchischen Schnittstellenmodellierung und der geschachtelten Schnittstellenadaptierung mit statischen und dynamischen Adaptern vor. Dieses Kapitel zeigt nun die praktische Anwendung der Konzepte für GENIVI Linux als aktuelle Infotainment-Plattform und bewertet die Lösung in Bezug auf die Aufgabenstellung.

Abschnitt 6.1 definiert den Workflow für die Adapter-Generierung und zeigt die dazugehörigen Werkzeuge. Anschließend beschreibt Abschnitt 6.2 die Umsetzung der hierarchischen Schnittstellenmodellierung für GENIVI Linux mit zwei textuellen und einer grafischen Beschreibungssprache. Aufbauend auf diesen Modellen wird in Abschnitt 6.3 die Umsetzung des geschachtelten Adaptermodells vorgestellt. Abschnitt 6.4 definiert die notwendige Infrastruktur zur Generierung und Ausführung der Adapter. Die abschließende Performancemessung in Abschnitt 6.5 zeigt, welche Geschwindigkeitsvorteile die geschachtelte Adaptierung gegenüber einer rein zustandsbasierten Adaptierung bringt.

Inhaltsverzeichnis

6.1. Workflow für die Adapter-Generierung

Bei dem Konzept der geschachtelten Schnittstellenadaptierung handelt es sich um einen teilautomatisierten Prozess, der im Rahmen dieser Arbeit vollständig mit entsprechender Werkzeugunterstützung hinterlegt wurde. Abbildung 6.1 zeigt den Prozess und gliedert die Adaptererstellung in einzelne Schritte von der Modellierung der Schnittstellen bis hin zur Generierung des lauffähigen Adapters.

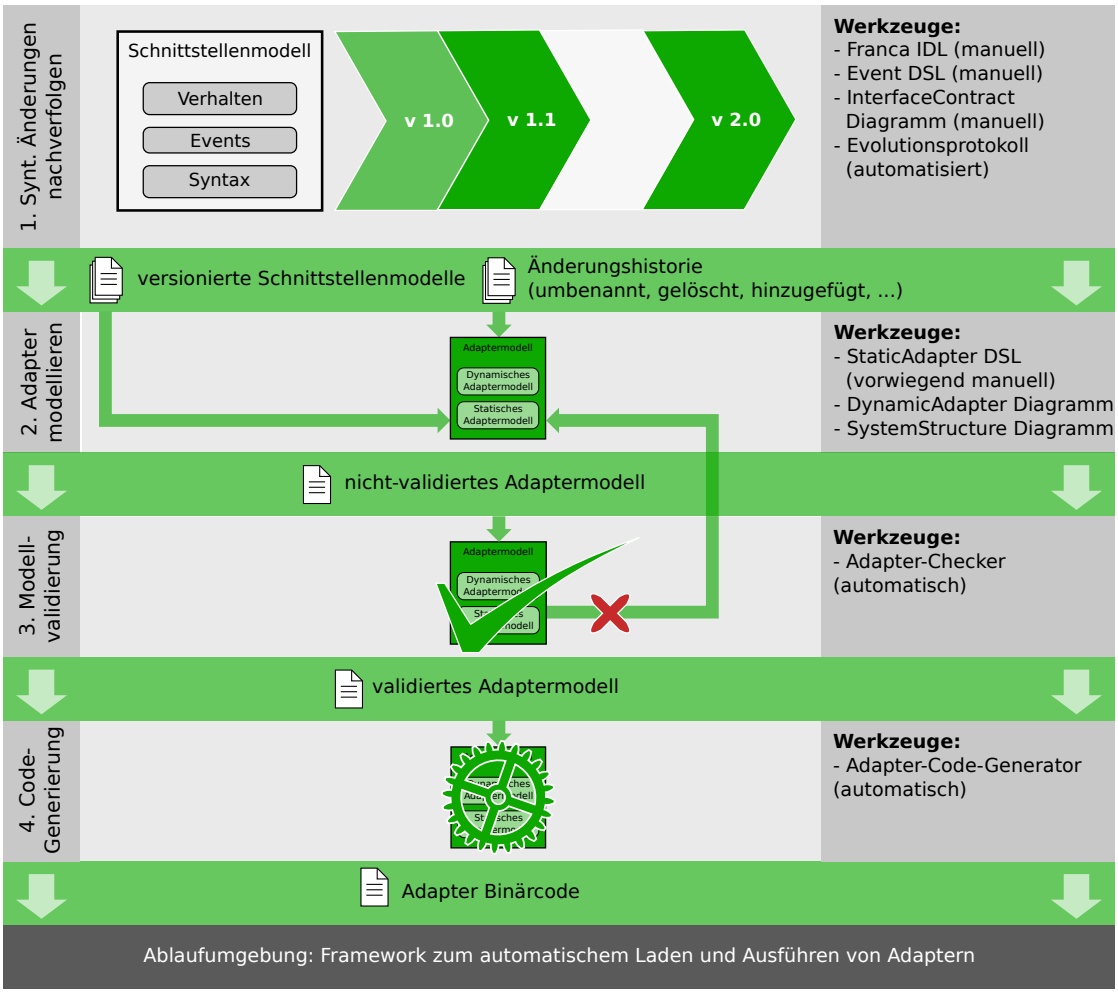


Abbildung 6.1.: Workflow und Werkzeuge für die Adaptergenerierung.

Nachverfolgung von Schnittstellenänderungen: Der erste Schritt setzt die Modellierung der Schnittstellen mit dem hierarchischen Schnittstellenmodell vor-

aus. Diese wird mit den Werkzeugen *Franca IDL*, *Event DSL* (domain-specific language) und *InterfaceContract Diagramm* erstellt. Somit ist jede Schnittstelle in verschiedenen Versionen mit den entsprechenden Modellen beschrieben. Für Schnittstellenänderungen wird ein Evolutionsprotokoll geführt, welches für die nachfolgende Generierung des Adaptermodells genutzt wird. Andrikopoulos [ABP12] beschreibt, wie diese Schnittstellenänderungen nachverfolgt werden können. Für die Migration der Modelle kann die von Hermannsdörfer [Her11] vorgeschlagene Lösung der gekoppelten Metamodell-Evolution angewandt werden. Dabei wird zur Bearbeitung des Modells ein Editor verwendet, der Informationen zur Historie der Schnittstelle (Evolutionsprotokoll) generiert. Jede elementare Modelländerung ist dabei mit einer Vorschrift verknüpft, die später für die automatisierte Migration von Modellen verwendet wird. Die im Rahmen dieser Arbeit entstandene Lösung beschränkt sich auf das Löschen und Umbenennen von Elementen. Diese Historie wird verwendet, um für den nächsten Schritt einen statischen Adapter als Vorschlag zu generieren.

Modellierung des Adapters: Im zweiten Schritt wählt der Modellierer zwei Versionen der Schnittstelle, für die ein Adapter erstellt werden soll. Auf Basis des Evolutionsprotokolls kann bereits ein Vorschlag für ein statisches Adaptermodell generiert werden. Zum Beispiel wird für eine umbenannte Methode bereits ein Deklarations-Mapping eingefügt. Je nachdem, welcher Automatisierungsgrad bei der Erstellung des Adaptermodells erreicht werden soll, kann die automatisierte Generierung eines sinnvollen Adaptermodells sehr komplex werden, vor allem bei der Betrachtung von Verhaltensänderungen. Da für diese Arbeit nicht die automatische Generierung von Adaptermodellen, sondern von lauffähigen Adaptern im Vordergrund steht, wird an dieser Stelle auf weiterführende Literatur [BP06; CPS08; SR02; GMW08] verwiesen. Nach diesem Schritt wird der statische Adapter manuell erweitert und ein dynamischer Adapter hinzugefügt. Die dabei eingesetzten Werkzeuge sind das *StaticAdapter DSL* und das *DynamicAdapter Diagramm*.

Validierung des Adaptermodells: Im nächsten Schritt erfolgt die vollautomatische Validierung des Adapters. Der *Adapter-Checker* prüft das Modell in Bezug auf den in den Schnittstellenmodellen definierten Kontrollfluss mit dem in Abschnitt 5.5 definierten Algorithmus. Im Falle eines Validierungsfehlers muss das Adaptermodell angepasst und erneut validiert werden. Zusätzlich zu den Kontrollfluss-Eigenschaften werden auch weitere Modellierungsrichtlinien überprüft. Zum Beispiel darf es für einen Zustand keine zwei ausgehenden Transitionen mit demselben Event als Trigger geben.

Code-Generierung: Im letzten Schritt wird aus dem validierten Adaptermodell mit dem *Adapter-Code-Generator* Quelltext generiert und in lauffähigen Binärcode übersetzt. Für die minimal-invasive Integration des Adapters in das Zielsystem GENIVI Linux wird Common-API um die Möglichkeit erweitert, während der Laufzeit Adapter zu laden.

6.2. Werkzeuge zur Schnittstellenmodellierung

Die in Kapitel 4.2 definierten Konzepte zur hierarchischen Modellierung von Schnittstellen wurden für eine möglichst breite Anwendbarkeit sehr abstrakt definiert. Dieser Abschnitt beschreibt nun, wie eine konkrete Ausprägung für GENIVI Linux (siehe Abschnitt 3.5) und die dort verwendete Schnittstellenbeschreibungssprache (Franca-IDL) aussehen kann.

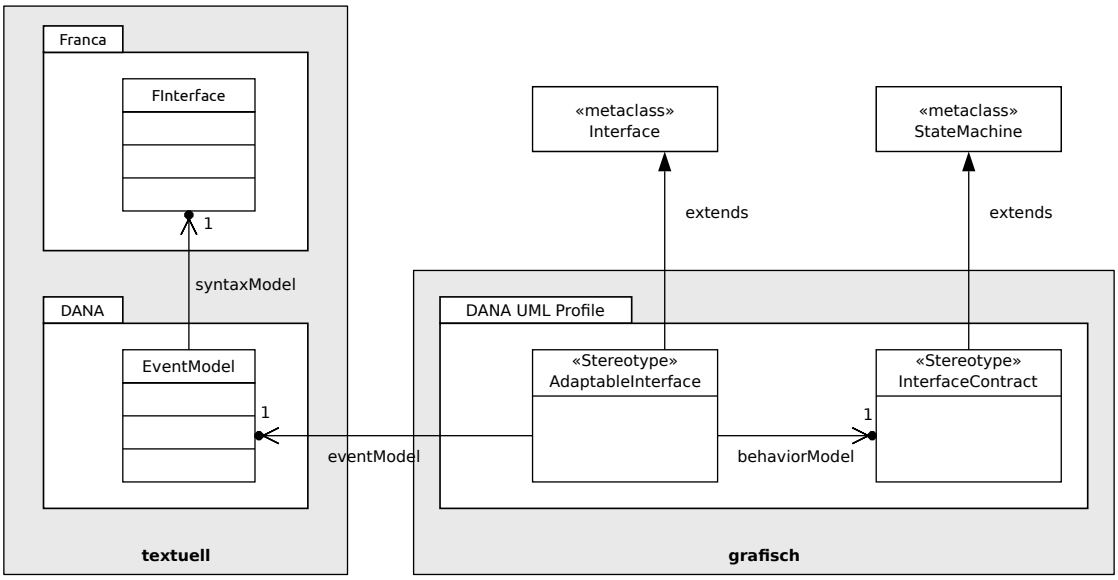


Abbildung 6.2.: Metamodell des hierarchischen Schnittstellenmodells am Beispiel GENIVI.

Ein Ziel bei der Umsetzung der Werkzeuge war es, die UML als Basis zu verwenden. Der Vorteil dabei ist die Möglichkeit, bereits existierende UML Werkzeuge verwenden und erweitern zu können. Praktisch umsetzen lässt sich eine derartige Erweiterung durch ein UML Profil (siehe Abschnitt 3.2.2). Das DANA UML Profil definiert die für die hierarchische Schnittstellenmodellierung notwendigen Erweiterungen. Abbildung 6.2 zeigt einen Ausschnitt der Implementierung als Klas-

sendiagramm. Zentrales Element ist die Klasse *AdaptableInterface*, welche UML *Interface* erweitert. Die Klasse *InterfaceContract* realisiert den Schnittstellenautomaten und erweitert die Klasse *StateMachine* aus der UML. Das UML Profil wird in Kombination mit dem Open-Source UML Editor Papyrus [Ecl14b] zur grafischen Erstellung und Bearbeitung von Modellen verwendet. Während das UML Profil hauptsächlich für grafische Modelle zum Einsatz kommt, wird für die Implementierung der textuellen Modelle das Xtext Framework [Ecl14c] verwendet. Dieses erlaubt die einfache Erstellung von textuellen Sprachen auf Basis einer Grammatik-Definition. Die Klasse *FInterface* wird vom Paket Franca [Bir14] zur Verfügung gestellt und realisiert die syntaktische Schnittstellenbeschreibung. Die Klasse *EventModel* im Paket *DANA* repräsentiert das Eventmodell. Alle Modelle basieren auf dem Eclipse Modeling Framework (EMF) [Ecl14a; Ste+09] und lassen sich dadurch sehr einfach in der Implementierung miteinander verknüpfen.

6.2.1. Franca IDL als Syntaxmodell

Das in Abschnitt 4.2 vorgestellte Syntaxmodell $\mathcal{I}^S$ ist unabhängig von der Aufrufsemantik der Deklarationen definiert. Das bedeutet, dass für Franca IDL definiert werden muss, wie die einzelnen Franca-Elemente den Elementen des Syntaxmodells zugeordnet werden. Abbildung 6.3 zeigt diese Entsprechung zwischen Franca IDL und dem Syntaxmodell. In Franca IDL gibt es zwei Arten von Deklarationen:

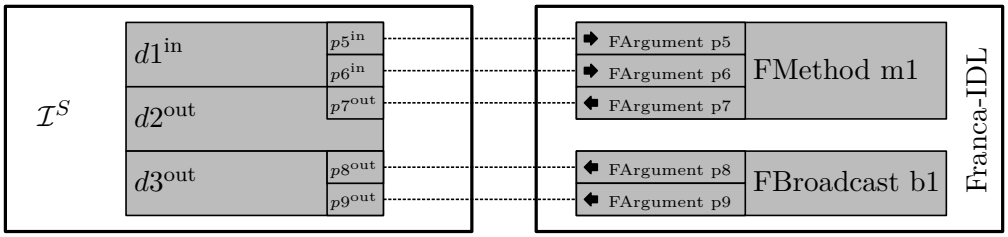


Abbildung 6.3.: Entsprechung zwischen Syntaxmodell und Franca IDL.

Methoden (Klasse *FMethod*) und *Broadcasts* (Klasse *FBroadcast*). Methoden besitzen Ein- und Ausgabeparameter und Broadcasts lediglich Ausgabeparameter. Da im Syntaxmodell konkret zwischen Ein- und Ausgabedeklarationen unterschieden wird, existieren deshalb für jede Franca-Methode zwei Deklarationen im Syntaxmodell. Zum Beispiel sind für die Methode *m1* in Abbildung 6.3 die beiden Deklarationen $d1^{\text{in}}$ und $d2^{\text{out}}$ definiert. Im Gegensatz dazu gibt es für einen Broadcast nur eine Ausgabe-Deklaration im Syntaxmodell. Listing 6.1 zeigt die Schnittstelle *ParkA1* in Franca IDL. Sie entspricht der in Listing 4.4.4 abstrakt definierten

Schnittstelle. Die Schnittstellenbeschreibung beginnt mit der Definition des Pakets in dem sich die Schnittstelle befindet (im Beispiel *com.bmw.demo*). Franca IDL erlaubt es, für Schnittstellen eine Version zu definieren (im Beispiel *major 1 minor 2*). Die Versionsnummer ermöglicht es zwischen Versionen einer gleichnamigen Schnittstelle zu unterscheiden und ist deshalb für die Adaptierung von zentraler Bedeutung. Denn sowohl die Client-Implementierung als auch die Server-Implementierung unterstützen die Abfrage der jeweiligen Schnittstellenversion. So kann festgestellt werden ob aufgrund von unterschiedlichen Versionen ein Adapter geladen werden muss. Zusätzlich zu den genannten Elementen können in einem Paket auch komplexe Datentypen definiert sein (im Beispiel die Enumeration *am_error*).

```
1  package com.bmw.demo
2
3  interface Parka {
4  version { major 2 minor 0 }
5
6     method connect
7     {
8        in { Boolean retry
9             Int8 retryCount }
10       out{
11            am_error result
12            Int8 connectionID }
13    }
14    method disconnect
15    {
16       in { Int8 connectionID }
17       out{ am_error result }
18    }
19    method getSensors {
20       out{
21            am_error result
22            UInt16 fl
23            UInt16 fml
24            UInt16 fmr
25            UInt16 fr
26            UInt16 rl
27            UInt16 rml
28            UInt16 rmr
29            UInt16 rr}
30    }
31    enumeration am_error{
32       OK
33       ERROR
34    }
35 }
36
37
```

Listing 6.1: Schnittstelle ParkA1 in Franca IDL

6.2.2. Event DSL

Als Eventmodell $\mathcal{I}^E$ (siehe Abschnitt 4.2.2) wird, angelehnt an Franca IDL, eine textuelle domänenspezifische Sprache (DSL) verwendet. Listing 6.2 zeigt das Schnittstellenmodell und Listing 6.3 das Eventmodell von ParkA2. Als Basis der Event-Definition wird zuerst die FIDL-Datei importiert (Listing 6.3 Zeile 1). Anschließend können für jede Deklaration des importierten Syntaxmodells beliebig viele Events definiert werden.

```
1  package com.bmw.demo
2
3  interface Parka {
4  version { major 2 minor 0 }
5
6     method connect
7     {
8        in { Boolean retry
9             Int8 retryCount }
10       out{
11            am_error result
12            Int8 connectionID }
13    }
14    method disconnect
15    {
16       in { Int8 connectionID }
17       out{ am_error result }
18    }
19    method getSensors {
20       out{
21            am_error result
22            UInt16 fl
23            UInt16 fml
24            UInt16 fmr
25            UInt16 fr
26            UInt16 rl
27            UInt16 rml
28            UInt16 rmr
29            UInt16 rr}
30    }
31    enumeration am_error{
32       OK
33       ERROR
34    }
35 }
36
37
```

Listing (6.2) Franca IDL für ParkA2

```
1  import "../franca/ParkA2.fidl"
2
3  Interface Parka {
4     CallEvent connectCall {
5        methodRef connect
6        children{
7           CallEvent connect {
8              constraint {retry == false}
9              emulation
10          }
11          CallEvent reconnect {
12             constraint {retry == true}
13             emulation {
14                retry = true
15                retryCount = 1 | connect.retryCount + 1
16             }
17          }
18       }
19    }
20    ResponseEvent connectResponse{
21       methodRef connect
22       children{
23          ResponseEvent connectOK{
24          constraint {(result == OK)
25             && (connectionID != -1)}
26          }
27          ResponseEvent connectERR{
28          constraint {(result == ERROR)
29             || (connectionID == -1)}
30          }
31       }
32    }
33    CallEvent disconnect { methodRef disconnect }
34    ResponseEvent disconnect { methodRef disconnect }
35    CallEvent getSensors { methodRef getSensors }
36    ResponseEvent getSensors { methodRef getSensors }
37 }
```

Listing (6.3) Eventmodell für ParkA2

Für Methodenaufrufe werden *CallEvents*, für Methoden-Rückgaben *ResponseEvents* und für Broadcasts *BroadcastEvents* definiert. Von diesen ist das CallEvent

das einzige Eingabeevent. Bei ResponseEvents und BroadcastEvents hingegen handelt es sich um Ausgabeevents. In der Event-Definition wird mit *methodRef* die Deklaration referenziert, für welche das Event definiert wird. Im Falle vom Event *connectCall* ist dies die Methode *connect* (siehe Listing 6.3 Zeile 5). Innerhalb eines Events kann optional die in Abschnitt 4.2.2 beschriebene Constraint definiert werden. Diese ist ein beliebig geklammerter logischer Ausdruck auf Basis der Deklarations-Parameter. Bei einem Eingabeevent sind dies die Eingabeparameter und bei einem Ausgabeevent die Ausgabeparameter der Methode.

Nach der Constraint erfolgt die Spezifikation der Emulation (siehe Abschnitt 4.2.2). Während die Constraint zur Dekodierung eines Events dient, wird die Emulation zur Enkodierung des Events verwendet. Zum Beispiel wird für das Event *reconnect* in Listing 6.3 Zeile 13 eine Emulation definiert. Innerhalb der Emulation wird für jeden Deklarations-Parameter spezifiziert, wie dieser beim Aufruf der Deklaration gesetzt werden muss. Im Beispiel wird *retry* auf *true* gesetzt. Für *retryCount* hingegen ist ein arithmetischer Ausdruck definiert, der den Parameter immer um eins erhöht. Falls zum Zeitpunkt des Emulationsaufrufs der arithmetische Ausdruck nicht ausgewertet werden kann, wird der Standardwert (im Beispiel *retryCount = 1*) gesetzt. Dies geschieht beim Emulationsaufruf von *reconnect*, falls vorher noch nie die Methode *connect* aufgerufen wurde und deshalb *retryCount* noch nicht gesetzt wurde. Sollten die im Eventmodell zur Verfügung gestellten Konstrukte zur Spezifikation einer Emulation nicht ausreichen, kann eine Emulation auch direkt in C++ umgesetzt werden. Bei der Modellierung muss für diesen Fall der Ausdruck in geschweiften Klammern einfach weggelassen werden (Listing 6.3 Zeile 9). Für diese leere Emulation wird bei der Code-Generierung ein Funktionsrumpf generiert, der frei implementiert werden kann. Somit ist es möglich auch komplexe Abläufe umzusetzen.

Die in Abschnitt 4.2.2 beschriebene Event-Hierarchie wird durch das Schlüsselwort *children* (z.B. Listing 6.3 Zeile 6) definiert. Somit sind die Events *connect* und *reconnect* Kinder des Events *connectCall*.

Der Editor zur Modellierung der DSL wurde mit dem Xtext Framework [Ecl14c] implementiert und unterstützt die Validierung des Modells auf Basis der definierten Sprache. Zum Beispiel wird bereits bei der Eingabe des Modells überprüft, ob die referenzierten Parameter oder Methoden im importierten Franca IDL Modell existieren. Die der Event DSL zugrunde liegende Grammatik kann Anhang A entnommen werden.

6.2.3. InterfaceContract Diagramm

Der in Abschnitt 4.2.3 definierte Schnittstellenautomat orientiert sich sehr stark an Mealy-Automaten [Mea55]. Somit eignen sich UML Zustandsautomaten für eine Implementierung sehr gut, da sie die wesentlichen Mealy-Eigenschaften, wie Trigger und Aktionen beinhalten. Eine Umsetzung mit UML Zustandsautomaten erfordert lediglich die Definition der erlaubten Trigger-Objekte und das Hinzufügen der für die Modellierung von Zeitverhalten notwendigen Elemente.

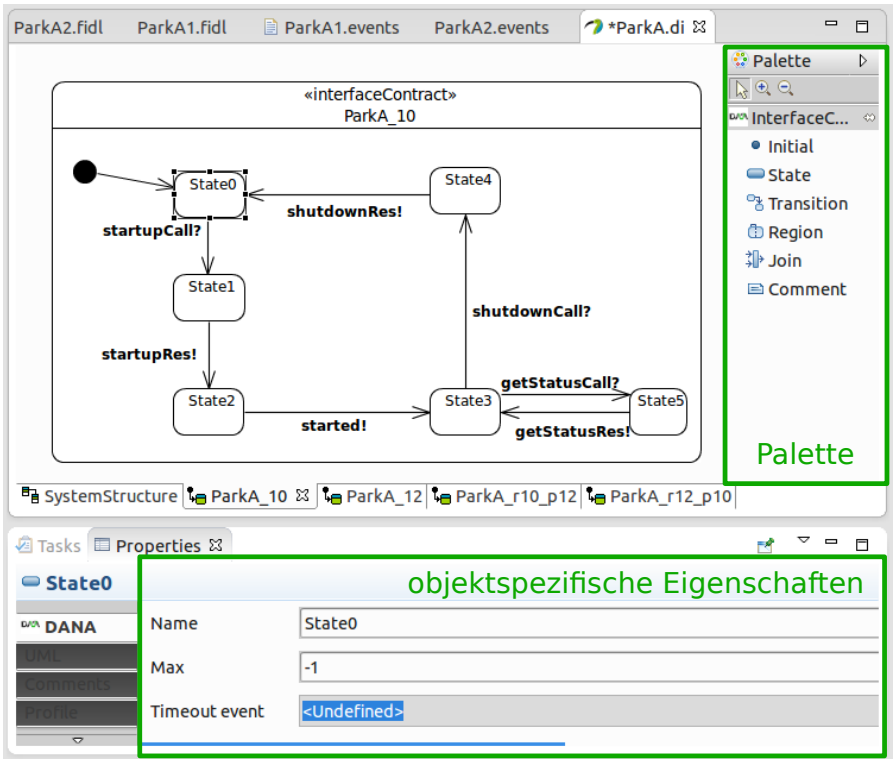


Abbildung 6.5.: Schnittstellenautomat für ParkA1 im DANA Modellierungswerkzeug.

Abbildung 6.5 zeigt das InterfaceContract Diagramm mit dem Schnittstellenautomaten von *ParkA1* als Beispiel. Die UML-Palette mit den Modellierungselementen rechts in der Grafik wird soweit eingeschränkt, dass nur die für die Schnittstellenmodellierung relevanten Werkzeuge sichtbar sind. Die in UML definierten *Regionen* zur Modellierung von Parallelität werden ebenfalls unterstützt. Diese können durch die von Harel [Har87] definierten Methoden wieder in einen flachen Zustandsautomaten ohne Regionen transformiert werden. Zum synchronisierten

Verlassen eines Zustands mit Regionen wird das in UML definierte *Join*-Element verwendet. Im unteren Bereich von Abbildung 6.5 werden die im Profil definierten Eigenschaften der Modellelemente angezeigt. Zum Beispiel sind das für Zustände das Max-Attribut (siehe Abschnitt 4.1.2) und das zugeordnete Timeout-Event. Verschiedene Hilfsfunktionen erleichtern dem Modellierer die Erstellung eines konsistenten Modells. Zum Beispiel erstellt das Modellierungswerkzeug beim Setzen des max-Attributes automatisch ein Timeout-Event für diesen Zustand.

6.3. Werkzeuge zur Adaptermodellierung

Die Grundeinstellungen eines Adapters werden im Diagramm *SystemStructure* modelliert. Dort können die in Abschnitt 6.2 beschriebenen *AdaptpableInterface*-Objekte erstellt werden, die das hierarchische Schnittstellenmodell für eine Schnittstelle repräsentieren. Sie besitzen Verweise auf das Syntaxmodell, das Eventmodell und den Schnittstellenautomaten. Der eigentliche Adapter kann in Form einer UML Assoziation zwischen zwei AdaptableInterface-Diagrammelementen erstellt werden. Bei diesem Schritt wird die Historieninformation aus den Schnittstellenmodellen verwendet, um einen Vorschlag für den statischen Adapter zu generieren (siehe Abschnitt 6.1). Zusätzlich wird ein leerer dynamischer Adapter generiert.

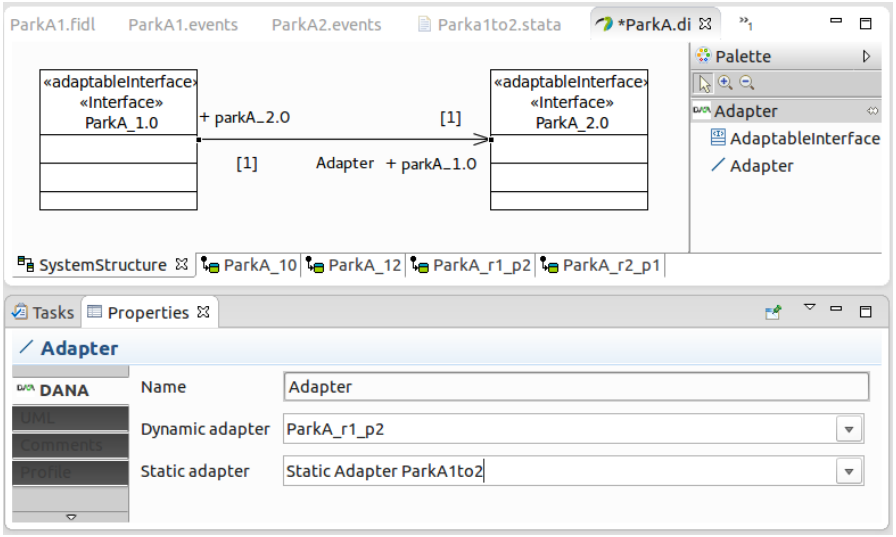


Abbildung 6.6.: Ansicht für Adapter-Grundeinstellungen zur Adaptierung der Schnittstellen ParkA1 und ParkA2.

Abbildung 6.6 zeigt das Werkzeug am Beispiel ParkA. In diesem Fall wird ein

Adapter für einen Client mit Schnittstelle ParkA1 und einen Server mit Schnittstelle ParkA2 erstellt. Dabei wird der Adapter durch eine gerichtete Assoziation modelliert die von Client zu Server verläuft. Die Eigenschaften im unteren Bereich der Grafik zeigen die Zuweisung des statischen und dynamischen Adaptermodells für den Adapter.

6.3.1. StaticAdapter DSL

Die in Franca IDL textuell beschriebenen Schnittstellen bilden die Basis für das statische Adaptermodell $\mathcal{A}^S$. Da es sich bei $\mathcal{A}^S$ um eine einfache Zuordnung von Deklarationen und Parametern handelt, bietet sich auch für das statische Adaptermodell eine textuelle Beschreibung an. Abbildung 6.7 zeigt einen statischen Adapter am Beispiel ParkA. Das Modell beginnt mit dem Import der beiden zu adaptierenden syntaktischen Schnittstellen. Anschließend werden die benötigte (required) und angebotene (provided) Schnittstelle und der Namen des statischen Adapters definiert. Anschließend kann für jede Deklaration ein Mapping definiert werden. Ist dies der Fall, ist es notwendig innerhalb des Deklarations-Mappings alle Parameter mit einem Parameter-Mapping zu spezifizieren. Dieses enthält zwingend die Angabe einer Konstanten (z.B. *Parka.1.0.sensorStatus.front_left = 0*) und optional

Parka1to2.statadapt

```
import "../franca/ParkA1.fidl"
import "../franca/ParkA2.fidl"

required interface Parka.1.0 {
    static adapter ParkA1to2{
        provided interface Parka.2.0

        declaration mapping{
            Parka.1.0.startup = Parka.2.0.connect{
                 Parka.2.0.connect.retry = 0;
                 Parka.2.0.connect.retryCount = 0;
            }
            Parka.1.0.sensorStatus = Parka.2.0.getSensors{
                Parka.1.0.sensorStatus.front_left = 0 |
                    (Parka.2.0.getSensors.fl + Parka.2.0.getSensors.fml)/2;
                Parka.1.0.sensorStatus.front_mid = 0 |
                    (Parka.2.0.getSensors.fml + Parka.2.0.getSensors.fmr)/2;
                Parka.1.0.sensorStatus.front_right = 0 |
                    (Parka.2.0.getSensors.fmr + Parka.2.0.getSensors.fr)/2;
                Parka.1.0.sensorStatus.rear_left = 0 |
                    (Parka.2.0.getSensors.rl + Parka.2.0.getSensors.rml) / 2;
                Parka.1.0.sensorStatus.rear_mid = 0 |
                    (Parka.2.0.getSensors.rml + Parka.2.0.getSensors.rmr) / 2;
                Parka.1.0.sensorStatus.rear_right = 0 |
                    (Parka.2.0.getSensors.rmr + Parka.2.0.getSensors.rr) / 2;
                Parka.1.0.sensorStatus.quality = Parka.1.0.QualityType.CLEAN;
            }
            Parka.1.0.shutdown = Parka.2.0.disconnect{
                Parka.2.0.disconnect.connectionID = 0 | Parka.2.0.connect.connectionID;
            }
            Parka.1.0.started = none
        }
    }
}
```

Abbildung 6.7.: Umsetzung des statischen Adaptermodells am Beispiel ParkA.

einen arithmetischen Ausdruck zur Berechnung des Parameters auf Basis anderer Schnittstellenparameter (z.B. *(Parka.2.0.getSensors.fl + Parka.2.0.getSensors.fml) / 2)*. Bei der Angabe von *none* als Deklarations-Mapping, wird bei der späteren Generierung von Quelltext ein Funktionsrumpf generiert, der frei implementiert werden kann. Somit ist gewährleistet, dass auch mit der vorgestellten Sprache nicht beschreibbare Abläufe implementiert werden können. Die der StaticAdapter DSL zugrunde liegende Grammatik kann Anhang A entnommen werden.

6.3.2. DynamicAdapter Diagramm

Das dynamische Adaptermodell wurde ebenfalls im DANA UML-Profil auf Basis eines UML Zustandsautomaten implementiert. Abbildung 6.8 zeigt den dynamischen Adapter aus Abschnitt 5.4. Das DynamicAdapter Diagramm besitzt eine eigene Palette mit Werkzeugen und die für das dynamische Adaptermodell notwendigen objektspezifischen Einstellungen. Zum Beispiel kann für einen Zustand das *active*-Flag gesetzt werden.

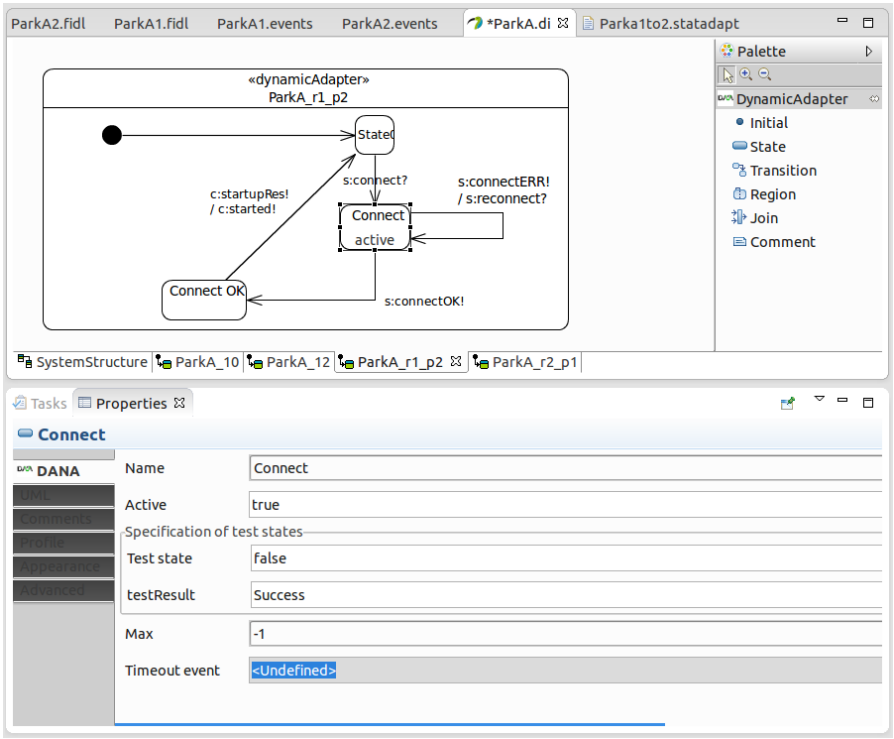


Abbildung 6.8.: Umsetzung des dynamischen Adaptermodells am Beispiel ParkA.

6.3.3. Modellierungsrichtlinien und Modellvalidierung

Für die Validierung des Kontrollflusses wurde der in Abschnitt 5.5 vorgestellte Algorithmus verwendet. Das Eclipse Modelig Framework besitzt eine generische Schnittstelle zur Implementierung eigener Validierungsrichtlinien. Das Ergebnis der Validierung wird standardmäßig als Listenansicht gezeigt und zusätzlich als Symbol an die fehlerhaften grafischen Elemente im Diagramm geheftet. Abbildung 6.9 zeigt das Ergebnis der Validierung im Falle eines Fehlers im Modell. In diesem Fall wurde der Trigger *startupRes!* mit dem Trigger *startupCall* ersetzt. Die Validierung meldet einen Deadlock im Zustand *Connect OK*. Für eine bessere Analyse der Validierungsfehler wird der in Abschnitt 5.5 vorgestellte Validierungsgraph im *DANA validation graph view* visualisiert.

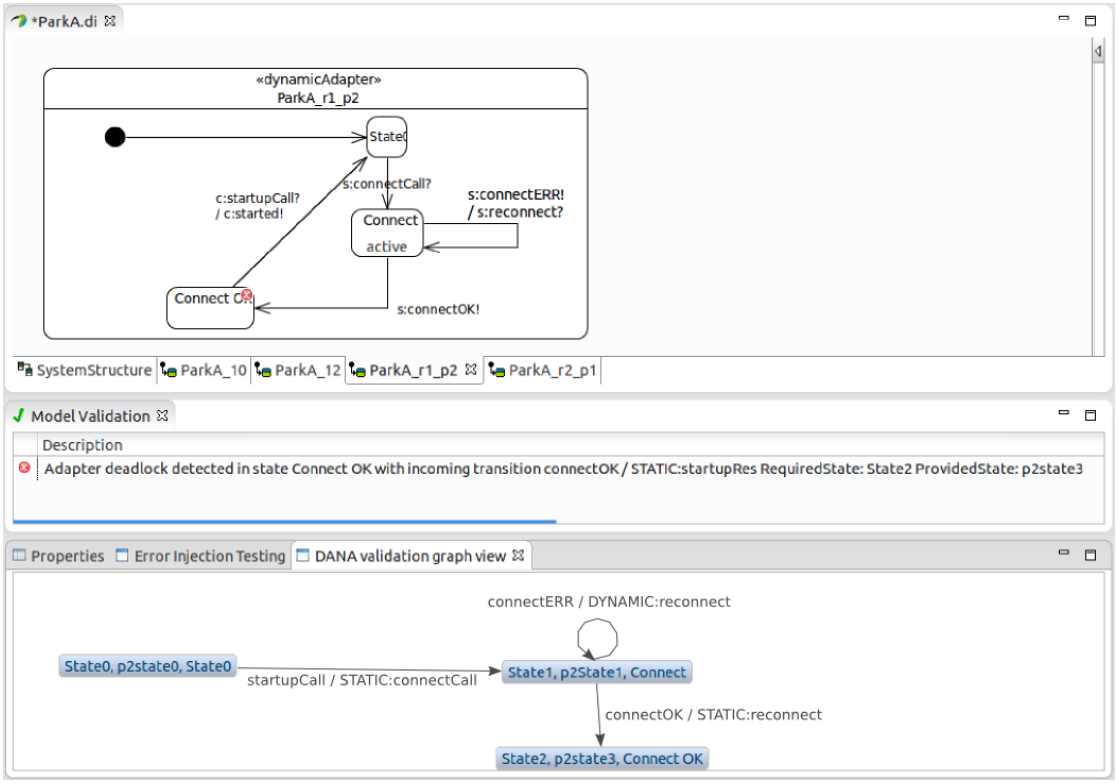


Abbildung 6.9.: Validierung des Adapters im DANA Modellierungswerkzeug.

6.4. Laufzeitumgebung und Code-Generierung

In den letzten Abschnitten wurden die Adapter-Modelle vorgestellt, aus denen lauffähiger Binärcode für das Zielsystem generiert wird. Der nächste Schritt im Workflow ist die Generierung und Ausführung des Adapters aus den Modellen. Deshalb wird nun die in Abschnitt 5.6 vorgestellte Architektur zur Code-Generierung verfeinert und eine Erweiterung des IPC-Mechanismus CommonAPI C++ [Gen14a] vorgestellt, die das Laden von Adaptern zur Laufzeit ermöglicht. Eine Einführung in die GENIVI Kommunikationsarchitektur mit CommonAPI C++ findet der Leser in Abschnitt 3.5. In dieser Arbeit beschreiben wir die Adaptierung der Client-Seite. Die Konzepte können ebenfalls für die Serverseite umgesetzt werden.

6.4.1. Minimal-Invasive Middleware-Integration

Wie in Abschnitt 2.4 beschrieben, bieten heutige Middleware-Architekturen nicht die Möglichkeit, einen Adapter dynamisch zu laden. Das bedeutet konkret, dass eine Adaptierung nur durch die Neu-Übersetzung der Softwarekomponenten ermöglicht wird. Diese Vorgehensweise widerspricht der Anforderung, dass Black-Box Softwarekomponenten gleichermaßen wie White-Box Softwarekomponenten adaptiert werden müssen.

Auch Common-API bietet in der aktuellen Version keine Möglichkeit, eine Komponentenschnittstelle nach der Code-Generierung zu verändern. Deshalb wird hier eine Erweiterung von Common-API vorgestellt, die dies ermöglicht. Die vorgestellte Lösung beruht auf dem Architekturmuster *adaptation by delegation* [Gam+95].

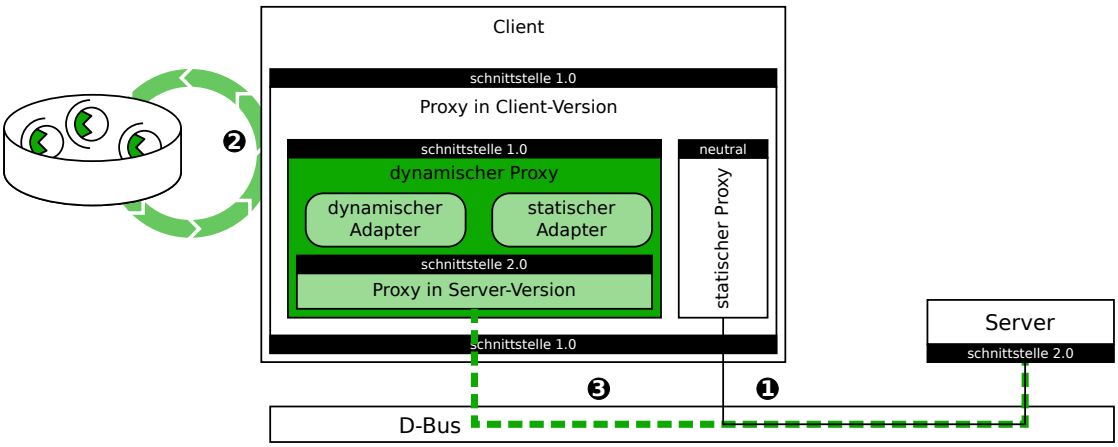


Abbildung 6.10.: Architektur für eine minimal-invasive Middleware-Integration.

Abbildung 6.10 zeigt die Architektur für eine minimal-invasive Integration des Adapters. CommonAPI wurde im Rahmen dieser Arbeit angepasst, sodass jeder Proxy einen statischen Anteil und einen dynamischen besitzt. Diese haben nichts mit statischen oder dynamischen Adaptern zu tun. Der *statische Proxy* ist versionsneutral und enthält lediglich die Funktionen, die von jedem Server angeboten werden (zum Beispiel die Abfrage der Schnittstellenversion). Der *dynamische Proxy* enthält die Anteile, die die eigentliche Funktionalität der Schnittstelle ausmachen. Die Adaptierung erfolgt nun in den in Abbildung 6.10 visualisierten Schritten.

1. Anfrage des statischen Proxys, welche Schnittstellenversion vom Server implementiert wird.
2. Vergleich mit der eigenen Version. Sollte die Version nicht übereinstimmen, wird versucht, den entsprechenden Adapter mit dem Fabriksmuster zu erstellen. Dieser wird als dynamischer Proxy installiert und fortan für die Kommunikation verwendet.
3. Die Interaktion zwischen Client und Server findet nun über den dynamisch geladenen Proxy (Adapter) statt.

6.4.2. Code-Generierung

Der gesamte, für die Adaptierung notwendige, Code wird auf Basis der Schnittstellen- und Adaptermodelle generiert. Abbildung 6.11 zeigt ein vereinfachtes Klassendiagramm der beteiligten Klassen am Beispiel ParkA. CommonAPI wurde um die Klasse *StaticProxy* erweitert, die alle Methoden beinhaltet, die zur Abfrage der Version und Ermittlung des korrekten Adapters benötigt werden. Die Klassen in den Namensräumen *v1_0* und *v2_0* werden durch den Common-API Code-Generator erstellt. Sie bieten die Standard-Implementierung, falls kein Adapter benötigt wird. Für den Adapter wird ein weiterer Namensraum unterhalb des Namensraums der benötigten Schnittstelle (im Beispiel *v1_0*) generiert. Die beteiligten Klassen werden nun genauer beschrieben. Bei der Schnittstelle v1_0 bezeichnet <MAYOR> die Hauptversion (1) und <MINOR> die Nebenversion (0). <REQUIRED> bezeichnet die Version der benötigten und <PROVIDED> die Version der angebotenen Schnittstelle.

- *CommonAPI::Proxy:* Von CommonAPI zur Verfügung gestellte Schnittstelle zur Implementierung von Proxy-Objekten.
- *CommonAPI::DBusProxy:* Von CommonAPI zur Verfügung gestellte Schnittstelle zur Implementierung von DBUS-Proxy-Objekten.

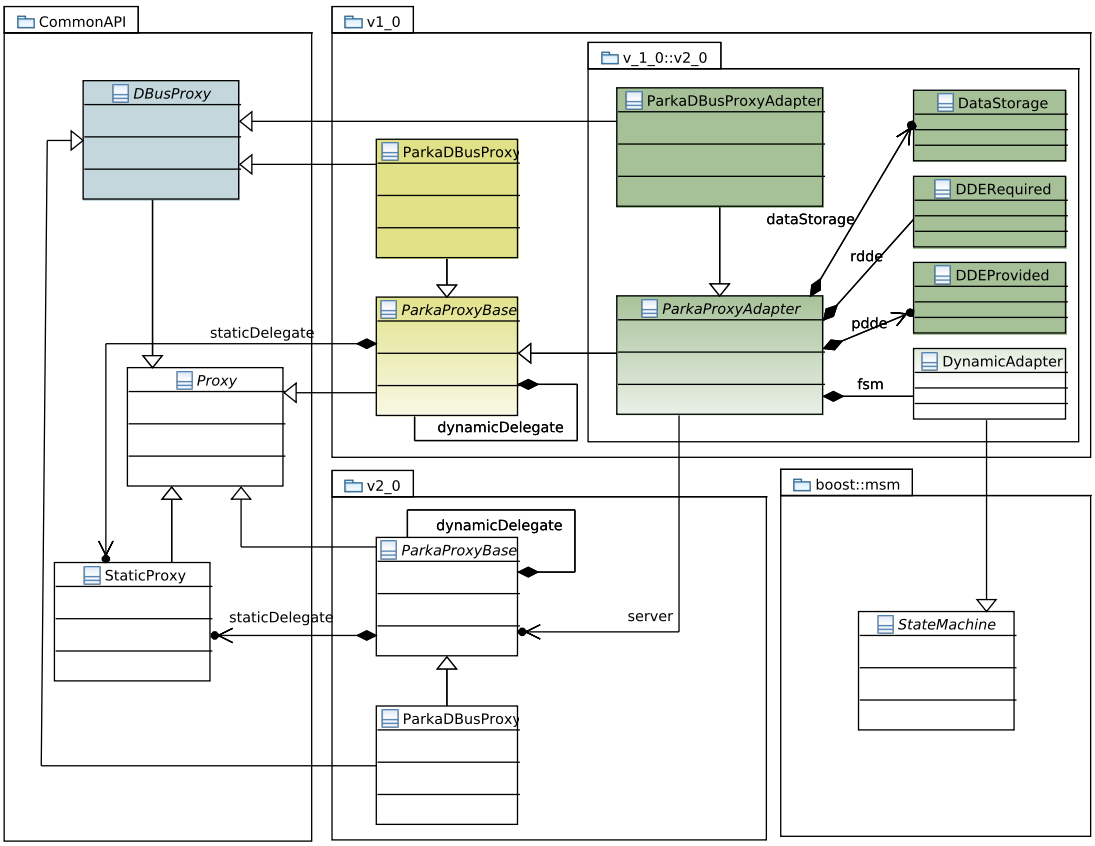


Abbildung 6.11.: Klassendiagramm für das Beispiel ParkA.

- *CommonAPI::StaticProxy:* Eine Erweiterung von CommonAPI zur Unterstützung des minimal-invasiven Ladens von Adaptern. Der *StaticProxy* übernimmt die Ermittlung der Server-Version und das Laden des korrekten dynamischen Proxy-Objektes.

- *v<MAYOR>_<MINOR>::ParkaProxyBase:* Vom Common-API Code-Generator generierte Klasse, die zur Implementierung des Clients verwendet wird. Sie enthält einen *staticDelegate* zur Ermittlung der Server-Version und einen *dynamicDelegate*, der im Falle der Adaptierung geladen wird.

- *v<MAYOR>_<MINOR>::ParkaDBUSProxy:* Vom Common-API Code-Generator generierte Klasse, die im Falle einer DBus Middleware durch die von CommonAPI zur Verfügung gestellte Proxy-Fabrik instantiiert wird.

- *v<REQUIRED>::v<PROVIDED>::ParkaProxyAdapter:* Vom Adapter-Code-Generator generierte Klasse, welche den statischen Adapter implementiert und alle zur Adaptierung notwendigen Elemente enthält. Diese Klasse setzt auf CommonAPI (*ParkaProxyBase*) auf und kann deshalb middlewareunabhängig verwendet werden.

- *v<REQUIRED>::v<PROVIDED>::ParkaDBusProxyAdapter:* Vom Adapter-Code-Generator generierte Klasse, welche die DBus-Variante des Adapters implementiert.

- *v<REQUIRED>::v<PROVIDED>::DynamicAdapter:* Vom Adapter-Code-Generator generierte Klasse, die den Zustandsautomaten des dynamischen Adaptermodells implementiert. Die Implementierung verwendet das boost Meta State Machine framework [boo14], welches auf C++ Templates basiert und eine performante Implementierung von Zustandsautomaten bietet.

- *v<REQUIRED>::v<PROVIDED>::DDERequired/Provided:* Vom Adapter-Code-Generator generierte Klassen, welche die Dekodierung und Enkodierung der Events auf Basis des Eventmodells implementieren.

- *v<REQUIRED>::v<PROVIDED>::DataStorage:* Vom Adapter-Code-Generator generierte Klasse, welche Daten für den Adapter zwischenspeichert.

Für konkrete Beispiele des generierten Quelltextes wird auf [Pra+14] verwiesen.

6.5. Performancemessung

Ein Vorteil der Verwendung von statischen und dynamischen Adaptern ist die Verkleinerung des Zustandsautomaten. Unter der Annahme, dass aus dem Adaptermodell ohne strukturelle Transformation Code generiert wird, wirkt sich diese Verkleinerung direkt auf die Ausführungszeit des Adapters aus.

Zur Veranschaulichung des zu erwartenden Performancegewinns werden hier die Ergebnisse einer Laufzeit-Messung eines Methodenaufrufes für vier verschiedene Szenarien gezeigt. Für die Messung wurden die Methoden *startup* und *connect* aus dem ParkA-Beispiel verwendet (siehe Abschnitt 4.4). Gemessen wurde die Dauer vom synchronen Aufruf einer Methode bis zum Erhalt der Antwort. Für die Ausführung des Zustandsautomaten kommt das *boost Meta State Machine Framework (boost::msm)* [boo14] zum Einsatz, welches mit dem Fokus auf Performanz entwickelt wurde.

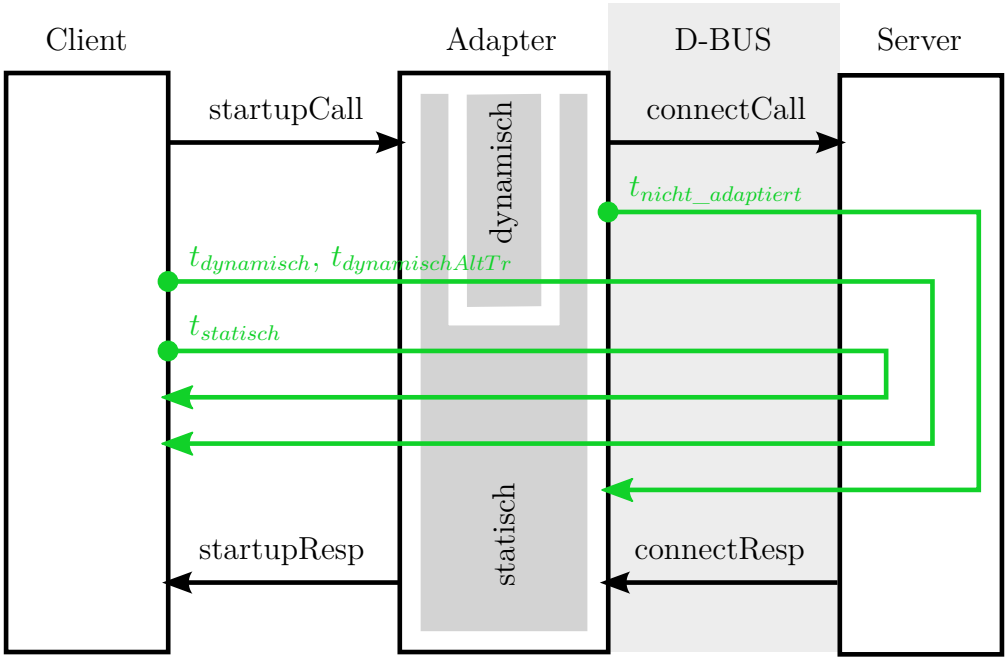


Abbildung 6.12.: Szenarien zur Messung der Laufzeit von statischem und dynamischem Adapter.

Abbildung 6.12 veranschaulicht den Kontrollfluss, der durch die einzelnen Szenarien erfasst wird. $t_{nichtadaptiert}$ stellt den nicht adaptierten Fall mit einer Übertragung über D-BUS dar. $t_{statisch}$ stellt die rein statische Adaptierung des Methodenaufrufs dar. In diesem Fall kommt kein Zustandsautomat zum Einsatz. $t_{dynamisch}$ und $t_{dynamischAltTr}$ sind die Szenarien, in denen der dynamische Adapter involviert ist. Die einzelnen Szenarien werden nun genauer beschrieben:

Nicht adaptierter Aufruf ($t_{nichtadaptiert}$): In diesem Fall wurde dieselbe Client- und Serverversion verwendet. Diese Messung beschreibt die Dauer, die für die Serialisierung, den Versand, den Empfang und die Deserialisierung der Daten für den Aufruf der Methode *connect* benötigt wird.

Rein statische Adaptierung ($t_{statisch}$): Mapping von *startup* auf *connect* mit einem statischen Deklarations-Mapping. Dieses Szenario umfasst die Zeit, die für das Weiterleiten des Methodenaufrufes und die Umrechnung der Parameter benötigt wird.

Rein dynamische Adaptierung ($t_{dynamisch}$): Mapping von *startup* auf *connect* mit einem dynamischen Adapter und leerem statischen Adapter. Bei der dynamischen Adaptierung eines Methodenaufrufes wird eine Transition für den Aufruf, und eine für die Antwort benötigt. Abbildung 6.13 zeigt den hierfür notwendigen dynamischen Adapter.

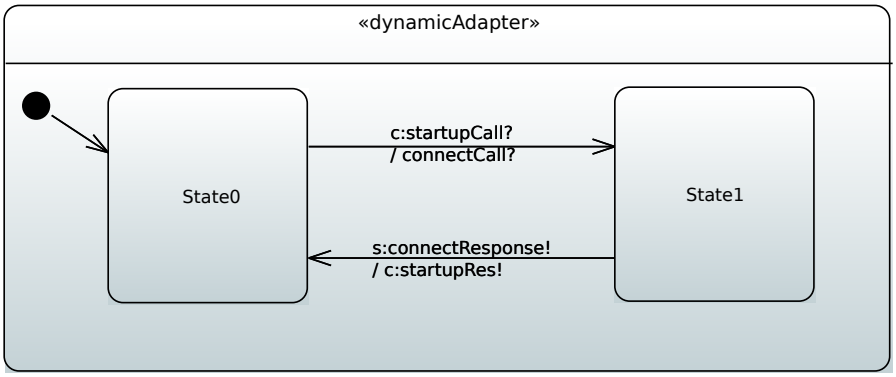


Abbildung 6.13.: Dynamischer Adapter für das Szenario $t_{dynamisch}$.

Rein dynamisch mit alternativer Transition ($t_{dynamischAltTr}$): Rein dynamisches Mapping mit einer zusätzlichen Transition (`c:started!`) an jedem Zustand. Diese Transitionen enthalten Trigger, die weder von der Methode `connect` noch von `startup` gefeuert werden. Abbildung 6.14 zeigt den dynamischen Adapter für dieses Szenario. Dieser Anwendungsfall ist von Bedeutung, falls global gültige

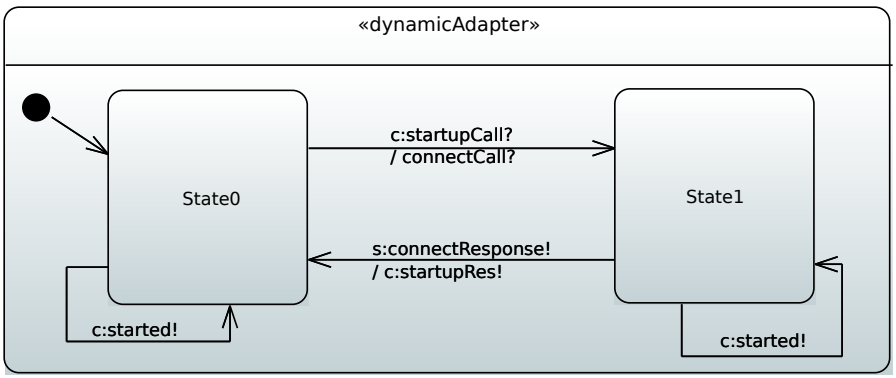


Abbildung 6.14.: Dynamischer Adapter für das Szenario $t_{dynamischAltTr}$.

Adaptierungen (z.B. das Umbenennen einer Methode) mit einem Zustandsautomaten umgesetzt werden sollen. Denn in diesem Fall muss an jedem Zustand eine

Transition, welche die Umbenennung durchführt, existieren. Da dies eine nicht unerhebliche Anzahl von Transitionen sein kann, wird mit diesem Szenario gemessen, welche Auswirkung eine derartige Modellierung auf die Performance des Adapters hat.

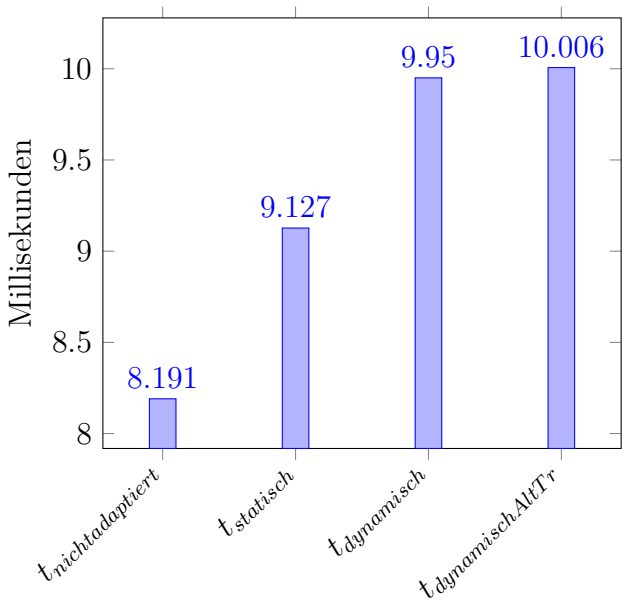


Abbildung 6.15.: Ergebnisse der Laufzeitmessung für die vier Szenarien.

Abbildung 6.15 zeigt die Messergebnisse für die einzelnen Szenarien. Für jedes Szenario wurden 100 Messungen durchgeführt, von denen der Durchschnitt dargestellt ist. Die Ergebnisse zeigen, dass für die Adaptierung in jedem Fall zusätzliche Zeit benötigt wird. Die Variante mit der besten Performance ist die rein statische Adaptierung ($t_{statisch}$), denn bei dieser Variante werden gar keine Events an den Zustandsautomaten weitergereicht.

Beim Einsatz des dynamischen Adapters muss gegenüber der rein statischen Adaptierung mit einer Verzögerung von 0,832ms pro Transition gerechnet werden ($t_{dynamisch}$). Ein großer Teil dieser Verzögerung ist dem Kontextwechsel zwischen den Threads geschuldet, denn in der verwendeten Implementierung läuft der dynamische Adapter in einem separaten Thread. Zusätzliche Zeit nimmt auch die Umrechnung der Events und das Durchführen des Zustandswechsels in Anspruch.

Das letzte Szenario $t_{dynamischAltTr}$ zeigt, dass jede zusätzliche Transition im Automaten, auch wenn sie nicht getriggert wird, eine verlangsamte Ausführung des Adapters zur Folge hat. In diesem Beispiel führt die zusätzliche Transition

`c:started!` an den Zuständen zu einer Verzögerung von 0,056ms. Somit hat jede globale Adaptierung, die nicht mit dem statischen Adapter modelliert wird, eine konstante Verzögerung in jedem Zustand des Adapters zur Folge.

Die Messung zeigt, dass die Trennung von statischem und dynamischem Adapter in Bezug auf die Performance des Adapters sinnvoll ist. Denn durch den geschachtelten Adapter können die zeitintensiven dynamischen Adaptierungen ($t_{dynamisch}$) und die unnötigen alternativen Transitionen ($t_{dynamischAltTr}$) minimiert werden. Je mehr Inkompatibilitäten mit dem statischen Adapter gelöst werden können, desto schneller ist die Ausführung des gesamten Adapters.

6.6. Zusammenfassung

Der Fokus dieses Kapitels lag darin, die Anwendbarkeit der geschachtelten Adaptierung mit statischen und dynamischen Adaptern an einem realen System zu zeigen.

Der vorgestellte Workflow für die Adapter-Generierung definiert die einzelnen Arbeitsschritte und die im Rahmen dieser Arbeit entstandenen Werkzeuge. Sie ermöglichen die teilautomatisierte Erstellung eines lauffähigen Adapters. Die Generierung des Quelltextes und des Binärcodes erfolgt dabei vollautomatisch aus den Modellen. Die in diesem Kapitel vorgestellte Laufzeitumgebung ermöglicht zudem das minimal-invasive Laden der Adapter zur Laufzeit für die in GENIVI verwendete Kommunikations-Middleware. Mit der abschließenden Performancemessung wurde gezeigt, dass das Konzept der geschachtelten Adaptierung einen Performance-Vorteil gegenüber einer rein zustandsbasierten Adaptierung bringt. Voraussetzung dafür ist, dass möglichst viele Adaptierungen statisch modelliert werden können und somit die Größe des Zustandsautomaten verkleinert wird.

Kapitel 7

Zusammenfassung

Zu Beginn dieser Arbeit wurde das Problem der Wiederverwendung von Softwarekomponenten im Fahrzeugumfeld detailliert erläutert und darauf aufbauend die Notwendigkeit einer Lösung zur minimal-invasiven Adaptierung von Softwarekomponenten im Fahrzeug aufgezeigt. Anschließend wurde für diese Aufgabenstellung eine durchgängige Lösung erarbeitet, die den Entwicklungszyklus von der Spezifikationsphase bis hin zur Integration der Software auf einem Steuergerät adressiert.

Dieses abschließende Kapitel fasst die wesentlichen Konzepte zusammen und zeigt einen Ausblick auf weiterführende Fragestellungen und Forschungsfelder.

7.1. Beitrag dieser Arbeit

Abschnitt 2.4 hat gezeigt, dass die Wiederverwendung von Softwarekomponenten in Form eines Baukastens mit bestehenden Konzepten erheblichen Aufwand verursacht. Der wesentliche Grund dafür ist, dass die Anpassung der Schnittstellen von inkompatiblen Softwarekomponenten manuell, auf Basis einer semiformalen Spezifikation erfolgt. Sowohl in der Spezifikationsphase, als auch in der Implementierungs- und Integrationsphase fehlen bisher Konzepte, Schnittstellen automatisiert zu adaptieren.

In dieser Arbeit wurden deshalb für jede Phase passende Konzepte vorgestellt, die eine möglichst automatisierte Erstellung von Software-Adaptern ermöglichen. Die wesentlichen Neuerungen im Vergleich zum Stand der Technik sind folgende:

Hierarchische Schnittstellenmodellierung: Das hierarchische Schnittstellenmodell beschreibt eine Schnittstelle auf drei Ebenen. Die syntaktische Ebene spe-

zifiziert die mit der Schnittstelle ausgetauschten Daten und der Schnittstellenautomat das gültige Protokoll. Die dritte Ebene der Events dient als Übersetzungsebene zwischen Schnittstellenautomat und Syntaxmodell. Diese Art der Modellierung adressiert die in der Spezifikationsphase identifizierten Defizite. Um dem Defizit der rein syntaktischen Schnittstellenmodellierung zu begegnen, ist das hierarchische Schnittstellenmodell so gestaltet, dass die untere Ebene der syntaktischen Beschreibung ausgetauscht werden kann. So können bereits existierende Schnittstellenbeschreibungssprachen als Basis verwendet und mit einem Verhaltensmodell erweitert werden. Das zweite Defizit aktueller Methoden ist die Heterogenität der Modelle, denn sie enthalten oft Quelltext zur Ausführung des Modells, was meist das Modell ungeeignet für Model-Checking macht. Dieses Defizit wird dadurch adressiert, dass das hierarchische Schnittstellenmodell alle Bestandteile enthält, damit lauffähiger Binärcode für ein Zielsystem generiert werden kann. Dies gilt sowohl für die syntaktischen Schnittstellen, als auch für das Verhalten von Softwarekomponenten. Die Modelle wurden so gestaltet, dass wichtige Kompatibilitätseigenschaften, wie beispielsweise die Kontrollfluss-Kompatibilität, mit Model-Checking nachgewiesen werden können.

Geschachtelte Adapter-Modellierung: Bisherige Ansätze zur Adaptierung von Softwareschnittstellen lassen sich grob in die zwei Kategorien der musterbasierten und zustandsbasierten Ansätze unterteilen (siehe Abschnitt 3.3). Der hier vorgestellte Ansatz der geschachtelten Adaptermodellierung vereint die beiden Ansätze und damit auch deren Vorteile. Die musterbasierte Adaptierung ist durch das statische Adaptermodell umgesetzt. Mit diesem können die häufigsten Anwendungsfälle mithilfe einfacher Mappings beschrieben werden. Die zustandsbasierte Adaptierung wird durch das dynamische Adaptermodell realisiert, welches die Auflösung komplexer Inkompatibilitäten ermöglicht. Das geschachtelte Adaptermodell ist so gestaltet, dass es ohne das Hinzufügen von Quelltext die Generierung von lauffähigen Adaptern ermöglicht.

Validierung von geschachtelten Adaptermodellen: Formale Modelle bringen den Vorteil mit sich, dass bereits auf Modellebene bestimmte Aussagen algorithmisch nachgewiesen werden können. Für die Modellierung von Adaptern ist die zentrale Frage die, ob das erstellte Adaptermodell einen korrekten Vermittler für zwei Schnittstellen beschreibt. Das geschachtelte Adaptermodell wird auf Basis der hierarchischen Schnittstellenmodelle zweier Schnittstellen erstellt. Jedes der involvierten Modelle enthält einen Automaten, der den Kontrollfluss einer Softwarekomponente beschreibt. Deshalb ermöglicht der Vergleich der Modelle die Validierung

des Adaptermodells in Bezug auf die Korrektheit des modellierten Kontrollflusses. Diese Validierung wurde für das geschachtelte Adaptermodell mithilfe eines Model-Checking Algorithmus umgesetzt. Somit kann nach der manuellen Modellierung eines geschachtelten Adapters automatisiert festgestellt werden, ob dieser ein korrekter Vermittler für zwei inkompatible Schnittstellen in Bezug auf deren Kontrollfluss ist.

Generierung und Ausführung von geschachtelten Adaptern: Die Durchgängigkeit des vorgestellten Ansatzes ergibt sich daraus, dass aus dem geschachtelten Adaptermodell automatisiert Binärcode für ein Zielsystem generiert werden kann. Dabei wird auf syntaktischer Ebene auf bereits existierende Code-Generatoren zurückgegriffen. Da die Modellierungssprache im wesentlichen auf UML Zustandsautomaten aufbaut, kann auch für die Adaptierung des Verhaltens auf existierende Frameworks zur Ausführung von Zustandsautomaten zurückgegriffen werden.

Minimal-Invasive Middleware-Integration: Eine wesentliche Anforderung bei der Adaptierung von Fahrzeug-Softwarekomponenten war, dass die Adaptierung minimal-invasiv erfolgen soll. Diese Anforderung ergibt sich aus dem in der Automobilindustrie vorherrschenden Businessmodell von OEM und Zulieferer, bei dem Softwarekomponenten meist in kompilierter Form als Black-Boxes geliefert werden. Bei einem minimal-invasiven Ansatz kann die Schnittstelle der Softwarekomponente angepasst werden, ohne die Black-Box zu verändern. Für diesen Zweck wurde ein Konzept definiert, welches das Laden von Adaptern zur Laufzeit ermöglicht. Dabei kann die Kommunikationsinfrastruktur automatisiert eine Inkompatibilität erkennen und den entsprechenden Adapter laden. Für die Erstellung und das Deployment des Adapters muss weder die Implementierung der Black-Box bekannt sein, noch die Black-Box verändert werden.

Umsetzung der Konzepte für GENIVI Linux: Alle in dieser Arbeit vorgestellten Konzepte wurden in einem auf Eclipse basierenden Werkzeug umgesetzt. Dieses ermöglicht die Erstellung von hierarchischen Schnittstellenmodellen, geschachtelten Adaptermodellen und die Generierung von Binärcode für das Zielsystem GENIVI Linux. Zusätzlich unterstützt das Werkzeug die Validierung von Adaptern mit dem vorgestellten Model-Checking Algorithmus.

7.2. Ausblick

Das Ergebnis dieser Arbeit ist ein durchgängiges Konzept zur teilautomatisierten Generierung von geschachtelten Software-Adaptern. Der Ansatz löst viele der technischen Probleme bei der Wiederverwendung von Black-Box Softwarekomponenten im Fahrzeug. Weiterführenden Fragestellungen ergeben sich in Bezug auf den praktischen Einsatz oder die weitere Automatisierung des Modellierungsprozesses.

Praktischer Einsatz im GENIVI Projekt: Im Rahmen dieser Arbeit wurde die Abstraktionsschicht der CommonAPI C++ so erweitert, dass das Laden von Adaptern unterstützt wird, unabhängig davon, ob diese geschachtelter Natur sind oder nicht. Die Implementierung wurde als Entwicklungszweig dem GENIVI Projekt zur Verfügung gestellt. Zum heutigen Zeitpunkt ist noch offen, wann das erste Infotainmentsystem mit GENIVI und der angepassten CommonAPI C++-Variante in Fahrzeugen erhältlich sein wird.

Weitere Unterstützung des Modellierers: Die Erstellung des Adaptermodells erfolgt im vorgestellten Ansatz teilautomatisiert. Zum Beispiel muss das dynamische Adaptermodell manuell modelliert werden. Mit Sicherheit kann der Modellierungsprozess weiter automatisiert werden oder die Modellierung des dynamischen Adaptermodells durch ein Vorschlagssystem auf Basis der Schnittstellenautomaten vereinfacht werden. Als Grundlage dafür kann der in dieser Arbeit vorgestellte Model-Checking Algorithmus verwendet werden. Mithilfe dieses Algorithmus kann genau festgestellt werden, in welchem Zustand des dynamischen Adapters welche Events auf Client- oder Serverseite auftreten können. Zusätzlich kann der Model-Checking Algorithmus mit einer Timing-Analyse erweitert werden. Somit könnten bereits auf Modellebene Timing-Probleme erkannt werden.

Anhang A

Grammatiken

Dieser Anhang stellt die Grammatiken für die in Abschnitt 6 vorgestellten domänenspezifischen Sprachen zur Verfügung. Listing A.1 beschreibt die Event DSL und Listing A.2 die StaticAdapter DSL.

```
1  grammar ida.dana.mapping.dsl.SignalEvents with org.eclipse.xtext.common.Terminals
2
3  generate signalEvents "http://www.dana.ida/mapping/dsl/SignalEvents"
4
5  import "http://core.franca.org" as franca
6  import "http://www.eclipse.org/emf/2002/Ecore" as ecore
7
8  // Wurzelelement
9  Model:
10    imports+=Import*
11    interfaces+=Interface*;
12
13 // Import für Franca-IDL Modelle
14 Import:
15    'import' importURI=STRING;
16
17 // Verweis auf die Schnittstelle, für die Events modelliert werden sollen.
18 // Enthält eine beliebige Anzahl von SignalEvents
19 Interface:
20    'Interface' fInterface=[franca::FInterface] '{'
21       events+=SignalEvent*
22    '}';
23
24
25 // Event-Definition als zentrales Element dieser Sprache.
26 SignalEvent:
27    (OutputEvent | InputEvent)
28    name = ID '{'
29        // Verweis auf Franca Methoden oder Broadcasts
30        (hasMethodRef ?='methodRef' methodRef=[franca::FModelElement])?
```

```
31
32        // Constraint, welche dieses Event charakterisiert
33        (constraint=Constraint)?
34
35        // Definition, wie dieses Event emuliert werden kann.
36        (emulation=Emulation)?
37
38        // Evetuelle Kind-Events
39        (hasChildren ?='children' '{'
40          children+=SignalEvent+
41        '}')?
42     '}'
43  ;
44
45  // Eingabe-Events
46  InputEvent:
47     {CallEvent} 'CallEvent'
48  ;
49
50  // Ausgabe-Events
51  OutputEvent:
52     {BroadcastEvent} 'BroadcastEvent' | {ResponseEvent} 'ResponseEvent'
53  ;
54
55  // Constraint, welche durch einen logischen Ausdruck definiert wird.
56  // Die Prüfung, ob es sich dabei um einen booleschen Ausdruck handelt,
57  // findet nicht in der Grammatik statt.
58  Constraint:
59     head='constraint' '{'expression= Expression '}'
60  ;
61
62  // Emulation, in der für alle Event-Parameter eine Zuweisung definiert werden muss.
63  Emulation:
64     head='emulation' '{' argumentAssignments+=ArgumentAssignment+ '}'
65  ;
66
67  // Zuweisung von Argumenten durch
68  ArgumentAssignment:
69     arg = [franca::FArgument] '=' defaultValue=ArgumentAssignmetConstant (hasExpression?='|'
70        expression=Expression
71     )?
72  ;
73
74  // Bei einer Argument-Zuweisung kann es sich um eine Konstante,
75  // oder um eine Franca Enumeration handeln.
76  ArgumentAssignmetConstant:
77     {AssignmentConstant} constant=Constant |{AssignmentFrancaConstant} valEnum =
78        [franca::FEnumerator]
79  ;
80
81
82  // Die Grammatik-Definitionen ab hier betreffen die logischen und arithmetischen Ausdrücke.
83  Expression:
84     Comparison
85  ;
86
87  Comparison returns Expression:
88     LessGreater (({Equal.left = current} '==' | {NotEqual.left = current} '!=' )
89        right=LessGreater
```

```
90     )*
91  ;
92
93  LessGreater returns Expression:
94     LessGreaterEqual (({Less.left = current} '<' | {Greater.left = current} '>' )
95        right=LessGreaterEqual
96     )*
97  ;
98
99  LessGreaterEqual returns Expression:
100    Conjunction (({LessEqual.left = current} '<=' | {GreaterEqual.left = current} '>=' )
101       right=Conjunction
102    )*
103 ;
104
105 Conjunction returns Expression:
106    Addition (({AND.left = current} '&&' | {OR.left = current} '||' ) right=Addition )*
107 ;
108
109 Addition returns Expression:
110    Multiplication (({Plus.left = current} '+' | {Minus.left = current} '-' )
111       right=Multiplication
112    )*
113 ;
114
115 Multiplication returns Expression:
116    Shift (({Mul.left = current} '*' | {Div.left = current} '/' ) right=Shift )*
117 ;
118
119 Shift returns Expression:
120    PrimaryExpression (({LShift.left = current} '<<' | {RShift.left = current} '>>' )
121       right=PrimaryExpression
122    )*
123 ;
124
125 PrimaryExpression returns Expression:
126    ({NestedExpression} '(' expression=Expression ')' |
127    {ExpressionConstant} constant = Constant |
128    {ExpressionArgument} arg = ModelElementReference (hasIteration ?='<' iteration=INT '>')?);
129
130 ModelElementReference hidden() :
131    element=[franca::FModelElement](Array ?='[' arrayIndex=INT ']')?  (hasNestedElement ?='.'
132       nestedElement=ModelElementReference
133    )?;
134
135 Constant:
136    INT | NEGINT | STRING | BOOLEAN
137 ;
138
139 enum FBasicTypeId returns franca::FBasicTypeId:
140 undefined = 'undefined' |
141 Int8 = 'Int8' |
142 UInt8 = 'UInt8' |
143 Int16 = 'Int16' |
144 UInt16 = 'UInt16' |
145 Int32 = 'Int32' |
146 UInt32 = 'UInt32' |
147 Int64 = 'Int64' |
148 UInt64 = 'UInt64' |
```

```
Boolean = 'Boolean' |
String = 'String' |
Float = 'Float' |
Double = 'Double' |
ByteBuffer = 'ByteBuffer' ;

BOOLEAN: ('true'|'false');

terminal ID   :   '^'?('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;

terminal INT returns ecore::EInt: ('0'..'9')+;
terminal NEGINT:
   ('-') INT;
terminal STRING :
        '"' ( '\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') | !('\\'|'"') )* '"' |
        "'" ( '\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') | !('\\'|"'") )* "'"
     ;
terminal ML_COMMENT : '/*' -> '*/';
terminal SL_COMMENT  : '//' !('\n'|'\r')* ('\r'? '\n')?;

terminal WS   :   (' '|'\t'|'\r'|'\n')+;
```

Listing A.1: Grammatik der Event DSL in Xtext

```
grammar ida.dana.adapter.staticadapterdsl.StaticAdapter with
   org.eclipse.xtext.common.Terminals hidden(WS, ML_COMMENT, SL_COMMENT)
import "http://core.franca.org" as franca
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

generate staticadapter "http://www.dana.ida/adapter/staticadapterdsl/StaticAdapter"


// Wurzelelement der Sprache
Model returns StaticAdapterModel :
   imports+=Import*

   // Verweis auf die Benötigte Schnittstelle, mit allen dazugehörenden statischen Adaptern.
   'required interface' requiredInterface=[franca::FInterface|FQNVERSION] ' {'
      staticAdapters+=StaticAdapter*
   '}'
;

// Statischer Adapter für eine angebotene (provided) und eine benötigte (required) Schnittstelle
StaticAdapter:
   'static adapter' name=ID '{'

      // Verweis auf die angebotene Schnittstelle
      'provided interface' providedInterface=[franca::FInterface|FQNVERSION]

      // Deklarationsmapping
      (hasDeclarationMapping ?='declaration mapping' '{'
         declarationMappings+=DeclarationMapping*
      '}')?

      // Typ-Mapping
```

```
32        (hasTypeMapping ?= 'type mapping' '{'
33           typeMappings+=TypeMapping*
34        '}')?
35     '}'
36  ;
37
38  // Typ-Mapping wird für die Code-Generierung benötigt
39  TypeMapping:
40     sourceType=STRING ':' targetType=STRING
41  ;
42
43  // Import der Franca-IDL Modelle
44  Import:
45     'import' importURI=STRING;
46
47  // Deklarationsmapping in dem eine Eingabedeklaration auf mehrere Ausgabedeklarationen verweisen kann.
48  DeclarationMapping:
49     inDeclaration = [franca::FModelElement|FQNVERSION] '=' outDeclaration=OutDeclaration
50        // Die Parameter-Mappings, denn für jedes Deklarationsmapping müssen alle betroffenen
51        // Parameter gemappt werden.
52        (hasParameterMapping?= '{' parameterMappings+=ParameterMapping* '}')?
53  ;
54
55  // Eine Ausgabedeklaration kann auch 'none' definieren, dann wird ein Funktionsrumpf generiert,
56  // der im Nachgang der Adapter-Generierung mit einer Implementierung befüllt werden kann.
57  OutDeclaration:
58     {NullOutMapping} 'none' | {FrancaOutMapping} declaration = [franca::FModelElement|FQNVERSION]
59  ;
60
61  // Parameter-Mapping in dem einem Parameter eine obligatorische Konstante und ein optionaler
62  // Arithmetischer Ausdruck auf Basis anderer Parameter zugewisen wird.
63  ParameterMapping:
64     outParameter =[franca::FArgument|FQNVERSION] '=' defaultValue = Const
65        (hasEquation?='|' equation = Expression)? ';'
66  ;
67
68  // Die restlichen Grammatik-Definitionen beziehen sich auf arithmetische Ausdrücke und Konstanten.
69  Const:
70     ({ConstVal} valInt = INT) | ({ConstEnum} valEnum = [franca::FEnumerator|FQNVERSION])
71  ;
72
73  Expression returns StaticExpression:
74     Addition
75  ;
76
77  Addition returns StaticExpression:
78     Multiplication (({StaticPlus.left = current} '+' | {StaticMinus.left = current} '-' )
79        right=Multiplication )*
80  ;
81
82  Multiplication returns StaticExpression:
83     Shift (({StaticMul.left = current} '*' | {StaticDiv.left = current} '/' ) right=Shift )*
84  ;
85
86  Shift returns StaticExpression:
87     PrimaryExpression (({StaticLShift.left = current} '<<' | {StaticRShift.left = current} '>>' )
88         right=PrimaryExpression )*
89  ;
90
```

```
91  PrimaryExpression returns StaticExpression:
92     ({StaticNestedExpression} '(' expression=Expression ')' |
93     {StaticLiteral} value=INT |
94     {StaticVariable} param=[franca::FArgument|FQNVERSION](hasIteration ?= '<'iteration=INT'>')?)
95  ;
96
97  FQNVERSION:
98     ID('.'INT'.'INT)?('.'ID)*
99  ;
```

Listing A.2: Grammatik der StaticAdapter DSL in Xtext

Anhang B

Liste eigener Veröffentlichungen

[Dra+13] Christian Drabek, Thomas Pramsohler, Marc Zeller und Gereon Weiss. „Interface Verification Using Executable Reference Models: An Application in the Automotive Infotainment“. In: *Proceedings of the 6th International Workshop on Model Based Architecting and Construction of Embedded Systems*. co-located with ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2013). Nov. 2013.

[PB13] Thomas Pramsohler und Uwe Baumgarten. „Adaptation of Automotive Infotainment Interfaces using Static and Dynamic Adapters“. In: *Proceedings INFORMATIK 2013*. Bd. GI Edition - Lecture Notes in Informatics (LNI). 11. Workshop Automotive Software Engineering. 2013.

[Pra+12] Thomas Pramsohler, Dirk Kaule, Annette Paulić, Marc Zeller und Gereon Weiß. „Modellbasierte Erkennung von Fehlverhalten: Automatisierte Generierung von Verifikationsmodellen zur Absicherung von Funktionsschnittstellen im Bereich Infotainment“. In: *Elektronik automotive* 8-9/2012 (2012), S. 12–17. ISSN: 1614-0125.

[Pra+13] Thomas Pramsohler, Mahmut Kafkas, Annette Paulic, Marc Zeller und Uwe Baumgarten. „Control Flow Analysis of Automotive Software Components Using Model-Based Specifications of Dynamic Behavior“. en. In: *SAE International Journal of Passenger Cars - Electronic and Electrical Systems* 6.2 (Aug. 2013), S. 425–436. ISSN: 1946-4614, 1946-4622. DOI: 10.4271/2013-01-0435.

[Pra+14] Thomas Pramsohler, Simon Schenk, Andreas Barthels und Uwe Baumgarten. „A layered interface-adaptation architecture for distributed component-based systems“. en. In: *Future Generation Computer Systems* (Okt. 2014). ISSN: 0167739X. DOI: 10.1016/j.future.2014.09.011.

[PSB13] Thomas Pramsohler, Simon Schenk und Uwe Baumgarten. „Towards an Optimized Software Architecture for Component Adaptation at Middleware Level“. In: *Software Architecture*. Hrsg. von Khalil Drira. Lecture Notes in Computer Science 7957. Springer Berlin Heidelberg, Jan. 2013, S. 266–281. ISBN: 978-3-642-39030-2 978-3-642-39031-9.

Literatur

[ABP12] V. Andrikopoulos, S. Benbernou und M.P. Papazoglou. „On the Evolution of Services“. In: *IEEE Transactions on Software Engineering* 38.3 (Mai 2012), S. 609–628. ISSN: 0098-5589. DOI: 10.1109/TSE.2011.22.

[AD94] Rajeev Alur und David L. Dill. „A theory of timed automata“. In: *Theoretical Computer Science* 126.2 (Apr. 1994), S. 183–235. ISSN: 0304-3975. DOI: 10.1016/0304-3975(94)90010-8.

[AG97] Robert Allen und David Garlan. „A Formal Basis for Architectural Connection“. In: *ACM Trans. Softw. Eng. Methodol.* 6.3 (Juli 1997), S. 213–249. ISSN: 1049-331X. DOI: 10.1145/258077.258078.

[Aut+08] Marco Autili, Leonardo Mostarda, Alfredo Navarra und Massimo Tivoli. „Synthesis of decentralized and concurrent adaptors for correctly assembling distributed component-based systems“. In: *Journal of Systems and Software* 81.12 (Dez. 2008), S. 2210–2236. ISSN: 0164-1212. DOI: 10.1016/j.jss.2008.04.006.

[BBC05] Andrea Bracciali, Antonio Brogi und Carlos Canal. „A formal approach to component adaptation“. In: *Journal of Systems and Software* 74.1 (Jan. 2005), S. 45–54. ISSN: 0164-1212. DOI: 10.1016/j.jss.2003.05.007.

[BCT04] Boualem Benatallah, Fabio Casati und Farouk Toumani. „Analysis and Management of Web Service Protocols“. In: *Conceptual Modeling – ER 2004*. Hrsg. von Paolo Atzeni, Wesley Chu, Hongjun Lu, Shuigeng Zhou und Tok-Wang Ling. Lecture Notes in Computer Science 3288. Springer Berlin Heidelberg, Jan. 2004, S. 524–541. ISBN: 978-3-540-23723-5 978-3-540-30464-7.

[BD98] Eric Badouel und Philippe Darondeau. „Theory of regions“. In: *Lectures on Petri Nets I: Basic Models*. Hrsg. von Wolfgang Reisig und Grzegorz Rozenberg. Lecture Notes in Computer Science 1491. Springer Berlin Heidelberg, Jan. 1998, S. 529–586. ISBN: 978-3-540-65306-6 978-3-540-49442-3.

[Bec+06] Steffen Becker, Antonio Brogi, Ian Gorton, Sven Overhage, Alexander Romanovsky und Massimo Tivoli. „Towards an Engineering Approach to Component Adaptation“. In: *Architecting Systems with Trustworthy Components*. Hrsg. von Ralf H. Reussner, Judith A. Stafford und Clemens A. Szyperski. Lecture Notes in Computer Science 3938. Springer Berlin Heidelberg, Jan. 2006, S. 193–215. ISBN: 978-3-540-35800-8 978-3-540-35833-6.

[Bec08] Markus Bechter. „Kompatibilitätsabsicherung verteilter eingebetteter Systeme in der Mechatronik“. German. Diss. Göttingen: Sierke, 2008.

[Ben+05] Boualem Benatallah, Fabio Casati, Daniela Grigori, Hamid R. Motahari Nezhad und Farouk Toumani. „Developing Adapters for Web Services Integration“. In: *Advanced Information Systems Engineering*. Hrsg. von Oscar Pastor und João Falcão e Cunha. Lecture Notes in Computer Science 3520. Springer Berlin Heidelberg, Jan. 2005, S. 415–429. ISBN: 978-3-540-26095-0 978-3-540-32127-9.

[Beu+99] A. Beugnard, J. Jezequel, N. Plouzeau und D. Watkins. „Making components contract aware“. In: *Computer* 32.7 (1999), S. 38–45. ISSN: 0018-9162. DOI: 10.1109/2.774917.

[Bir14] Klaus Birken. *Franca IDL*. Feb. 2014. URL: https://code.google.com/a/eclipselabs.org/p/franca/ (besucht am 17. 02. 2014).

[boo14] boost. *boost Meta State Machine*. Aug. 2014. URL: http://www.boost.org/doc/libs/1_55_0/libs/msm/doc/HTML/index.html (besucht am 06. 07. 2015).

[BP06] Antonio Brogi und Razvan Popescu. „Automated Generation of BPEL Adapters“. In: *Service-Oriented Computing – ICSOC 2006*. Hrsg. von Asit Dan und Winfried Lamersdorf. Lecture Notes in Computer Science 4294. Springer Berlin Heidelberg, Jan. 2006, S. 27–39. ISBN: 978-3-540-68147-2 978-3-540-68148-9.

[Bro+04] Antonio Brogi, Carlos Canal, Ernesto Pimentel und Antonio Vallecillo. „Formalizing Web Service Choreographies“. In: *Electronic Notes in Theoretical Computer Science* 105 (Dez. 2004), S. 73–94. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2004.05.007.

[Bro+08] Manfred Broy, Martin Feilkas, Johannes Grünbauer, Alexander Gruler, Alexander Harhurin, Judith Hartmann, Birgit Penzenstadler, Bernhard Schätz und Doris Wild. „Umfassendes Architekturmodell für das Engineering eingebetteter software-intensiver Systeme“. In: *Technische Universität München* (2008).

[BZ83] Daniel Brand und Pitro Zafiropulo. „On Communicating Finite-State Machines“. In: *J. ACM* 30.2 (Apr. 1983), S. 323–342. ISSN: 0004-5411. DOI: 10.1145/322374.322380.

[CMP06] Carlos Canal, Juan Manuel Murillo und Pascal Poizat. „Software Adaptation.“ In: *L'objet* 12.1 (2006), S. 9–31.

[Coo10] MOST Coorperation. *MOST Specification Rev. 3.0 Errata 2*. Aug. 2010. URL: http://www.mostcooperation.com/ (besucht am 12. 07. 2015).

[Cor+98] J. Cortadella, M. Kishinevsky, L. Lavagno und A. Yakovlev. „Deriving Petri nets from finite transition systems“. In: *IEEE Transactions on Computers* 47.8 (Aug. 1998), S. 859–882. ISSN: 0018-9340. DOI: 10.1109/12.707587.

[CPS08] C. Canal, P. Poizat und G. Salaun. „Model-Based Adaptation of Behavioral Mismatching Components“. In: *IEEE Transactions on Software Engineering* 34.4 (2008), S. 546–563. ISSN: 0098-5589. DOI: 10.1109/TSE.2008.31.

[Cue+07] P. Cuenot, DeJiu Chen, S. Gerard, Henrik Lonn, M.-O. Reiser, David Servat, C.-J. Sjostedt, R.T. Kolagari, M. Torngren und M. Weber. „Managing Complexity of Automotive Electronics Using the EAST-ADL“. In: *12th IEEE International Conference on Engineering Complex Computer Systems, 2007*. 2007, S. 353–358. DOI: 10.1109/ICECCS.2007.28.

[DH01a] Luca De Alfaro und Thomas A. Henzinger. „Interface Automata“. In: *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*. ESEC/FSE-9. New York, NY, USA: ACM, 2001, S. 109–120. ISBN: 1-58113-390-1. DOI: 10.1145/503209.503226.

[DH01b] Luca De Alfaro und Thomas A. Henzinger. „Interface Theories for Component-Based Design“. en. In: *Embedded Software*. Hrsg. von Thomas A. Henzinger und Christoph M. Kirsch. Lecture Notes in Computer Science 2211. Springer Berlin Heidelberg, Jan. 2001, S. 148–165. ISBN: 978-3-540-42673-8 978-3-540-45449-6.

[DHS02] Luca De Alfaro, Thomas A. Henzinger und Mariëlle Stoelinga. „Timed Interfaces“. en. In: *Embedded Software*. Hrsg. von Alberto Sangiovanni-Vincentelli und Joseph Sifakis. Lecture Notes in Computer Science 2491. Springer Berlin Heidelberg, Jan. 2002, S. 108–122. ISBN: 978-3-540-44307-0 978-3-540-45828-9.

[DIN98] EN DIN. *45020 (1998): DIN EN 45020: 1998, Normung und damit zusammenhängende Tätigkeiten–Allgemeine Begriffe*. Beuth Verlag, Berlin, 1998.

[DJ05] D. Dig und R. Johnson. „The role of refactorings in API evolution“. In: *Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005. ICSM'05*. 2005, S. 389–398. DOI: 10.1109/ICSM.2005.90.

[Dra+13] Christian Drabek, Thomas Pramsohler, Marc Zeller und Gereon Weiss. „Interface Verification Using Executable Reference Models: An Application in the Automotive Infotainment“. In: *Proceedings of the 6th International Workshop on Model Based Architecting and Construction of Embedded Systems*. co-located with ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2013). Nov. 2013.

[DSW06] Marlon Dumas, Murray Spork und Kenneth Wang. „Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation“. In: *Business Process Management*. Hrsg. von Schahram Dustdar, José Luiz Fiadeiro und Amit P. Sheth. Lecture Notes in Computer Science 4102. Springer Berlin Heidelberg, Jan. 2006, S. 65–80. ISBN: 978-3-540-38901-9 978-3-540-38903-3.

[Ecl14a] Eclipse. *Eclipse Modeling Framework Project (EMF)*. Juli 2014. URL: http://www.eclipse.org/modeling/emf/ (besucht am 20. 07. 2014).

[Ecl14b] Eclipse. *Papyrus UML Editor*. Juli 2014. URL: http://www.eclipse.org/papyrus/ (besucht am 20. 07. 2014).

[Ecl14c] Eclipse. *Xtext*. Juli 2014. URL: http://www.eclipse.org/Xtext/index.html (besucht am 20. 07. 2014).

[Fen+06] Helmut Fennel, Stefan Bunzel, Harald Heinecke, Jürgen Bielefeld, Simon Fürst, Klaus-Peter Schnelle, Walter Grote, Nico Maldener, Thomas Weber, Florian Wohlgemuth u. a. „Achievements and exploitation of the AUTOSAR development partnership“. In: *Convergence* 2006 (2006), S. 10.

[Gam+95] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[GAO09] D. Garlan, R. Allen und J. Ockerbloom. „Architectural Mismatch: Why Reuse Is Still So Hard“. In: *Software, IEEE* 26.4 (Aug. 2009). Cited by 0032, S. 66–69. ISSN: 0740-7459. DOI: 10.1109/MS.2009.86.

[GAO95] D. Garlan, R. Allen und J. Ockerbloom. „Architectural mismatch: why reuse is so hard“. In: *Software, IEEE* 12.6 (Nov. 1995). Cited by 0611, S. 17–26. ISSN: 0740-7459. DOI: 10.1109/52.469757.

[GbR14a] AUTOSAR GbR. *Layered Software Architecture, v3.4.0 R4.1*. 2014. URL: http://www.autosar.org/fileadmin/files/releases/4-1/software-architecture/general/auxiliary/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf (besucht am 06. 07. 2015).

[GbR14b] AUTOSAR GbR. *Methodology, v3.1.0 R4.1*. 2014. URL: http://www.autosar.org/specifications/release-41/methodology-and-templates/methodology/ (besucht am 06. 07. 2015).

[Gen14a] Genivi-Alliance. *CommonAPI C++*. Dez. 2014. URL: http://projects.genivi.org/commonapi/home (besucht am 12. 07. 2015).

[Gen14b] Genivi-Alliance. *GENIVI Alliance*. https://www.genivi.org/. Feb. 2014. URL: https://www.genivi.org/ (besucht am 12. 07. 2015).

[GMW08] Christian Gierds, Arjan J. Mooij und Karsten Wolf. *Specifying and generating behavioral service adapter based on transformation rules*. Techn. Ber. 2008.

[GMW12] C. Gierds, A.J. Mooij und K. Wolf. „Reducing Adapter Synthesis to Controller Synthesis“. In: *IEEE Transactions on Services Computing* 5.1 (März 2012), S. 72–85. ISSN: 1939-1374. DOI: 10.1109/TSC.2010.57.

[GMY84] M. G. Gouda, E. G. Manning und Y. T. Yu. „On the progress of communication between two finite state machines“. In: *Information and Control* 63.3 (Dez. 1984), S. 200–216. ISSN: 0019-9958. DOI: 10.1016/S0019-9958(84)80014-5.

[Gui+11] Ning Gui, Vincenzo De Florio, Hong Sun und Chris Blondia. „Toward architecture-based context-aware deployment and adaptation“. In: *Journal of Systems and Software* 84.2 (Feb. 2011), S. 185–197. ISSN: 0164-1212. DOI: 10.1016/j.jss.2010.09.017.

[Hab+10] Wolfgang Haberl, Markus Herrmannsdoerfer, Stefan Kugele, Michael Tautschnig und Martin Wechs. „Seamless Model-Driven Development Put into Practice“. In: *Leveraging Applications of Formal Methods, Verification, and Validation*. Hrsg. von Tiziana Margaria und Bernhard Steffen. Lecture Notes in Computer Science 6415. Springer Berlin Heidelberg, Jan. 2010, S. 18–32. ISBN: 978-3-642-16557-3 978-3-642-16558-0.

[Hab11] Wolfgang Haberl. „Code Generation and System Integration of Distributed Automotive Applications“. Dissertation. München: Technische Universität München, 2011.

[Har87] David Harel. „Statecharts: a visual formalism for complex systems“. In: *Science of Computer Programming* 8.3 (Juni 1987), S. 231–274. ISSN: 0167-6423. DOI: 10.1016/0167-6423(87)90035-9.

[Hei+08] H. Heinecke, W. Damm, B. Josko, A. Metzner, H. Kopetz, A. Sangiovanni-Vincentelli und M. Di Natale. „Software Components for Reliable Automotive Systems“. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE '08. New York, NY, USA: ACM, 2008, S. 549–554. ISBN: 978-3-9810801-3-1. DOI: 10.1145/1403375.1403508.

[Hen+94] T. A. Henzinger, X. Nicollin, J. Sifakis und S. Yovine. „Symbolic Model Checking for Real-Time Systems“. In: *Information and Computation* 111.2 (Juni 1994), S. 193–244. ISSN: 0890-5401. DOI: 10.1006/inco.1994.1045.

[Her11] Markus Herrmannsdörfer. „Evolutionary Metamodeling“. Dissertation. München: Technische Universität München, 2011.

[IBM14] IBM. *IBM Rational Rhapsody*. 2014. URL: http://www.ibm.com/software/awdtools/rhapsody (besucht am 21. 04. 2014).

[IEE90] IEEE. „IEEE Standard Glossary of Software Engineering Terminology“. In: *IEEE Std 610.12-1990* (1990), S. 1–84. DOI: 10.1109/IEEESTD.1990.101064.

[ITU11] ITU-T. *Recommendation Z.120: Message Sequence Chart (MSC)*. Feb. 2011. URL: http://www.itu.int/rec/T-REC-Z.120/en (besucht am 05. 01. 2014).

[Jia+11] Jian-min Jiang, Shi Zhang, Ping Gong und Zhong Hong. „Message Dependency-Based Adaptation of Services“. In: *Services Computing Conference (APSCC), 2011 IEEE Asia-Pacific*. Dez. 2011, S. 442–449. DOI: 10.1109/APSCC.2011.81.

[Kan+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak und A. S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. en. Techn. Ber. Nov. 1990.

[KD14] Dr Galib Krdzalic und Alexander Driss. „Software-Architektur ohne Autosar Verallgemeinerte Schablone für Embedded-Systeme“. de. In: *ATZelektronik* 9.1 (Feb. 2014), S. 34–37. ISSN: 1862-1791, 2192-8878. DOI: 10.1365/s35658-014-0385-9.

[Kin97] Ekkart Kindler. „A compositional partial order semantics for Petri net components“. In: *Application and Theory of Petri Nets 1997*. Hrsg. von Pierre Azéma und Gianfranco Balbo. Lecture Notes in Computer Science 1248. Springer Berlin Heidelberg, Jan. 1997, S. 235–252. ISBN: 978-3-540-63139-2 978-3-540-69187-7.

[LT89] Nancy A. Lynch und Mark R. Tuttle. „An introduction to input/output automata“. In: *CWI Quarterly* 2 (1989), S. 219–246.

[Mat14] Berner & Mattner. *MODENA - Spezifikations- und Testwerkzeug für Infotainment- und Bodyanwendungen*. 2014. URL: http://www.berner-mattner.com/de/berner-mattner-home/produkte/modena/index.html (besucht am 21. 04. 2014).

[Mea55] George Mealy. „A Method for Synthesizing Sequential Circuits“. In: *Bell System Technical Journal* 34.5 (1955), S. 1045–1079.

[Meg63] Leon C Megginson. „LESSONS FROM EUROPE FOR AMERICAN-BUSINESS“. In: *Southwestern Social Science Quarterly* 44.1 (1963), S. 3–13.

[MPS12] R. Mateescu, P. Poizat und G. Salaün. „Adaptation of Service Protocols Using Process Algebra and On-the-Fly Reduction Techniques“. In: *IEEE Transactions on Software Engineering* 38.4 (Aug. 2012), S. 755–777. ISSN: 0098-5589. DOI: 10.1109/TSE.2011.62.

[MTV09] G. Macario, Marco Torchiano und M. Violante. „An in-vehicle infotainment software architecture based on google android“. In: *IEEE International Symposium on Industrial Embedded Systems, 2009. SIES '09.* Juli 2009, S. 257–260. DOI: 10.1109/SIES.2009.5196223.

[NM95] Oscar Nierstrasz und Theo Dirk Meijler. „Research Directions in Software Composition“. In: *ACM Comput. Surv.* 27.2 (Juni 1995), S. 262–264. ISSN: 0360-0300. DOI: 10.1145/210376.210389.

[OMG10] OMG. *UML Superstructure Specification v2.4.* Techn. Ber. Nov. 2010.

[OMG12] OMG. *Common Object Request Broker Architecture (CORBA) v3.3.* Techn. Ber. Dez. 2012.

[PB13] Thomas Pramsohler und Uwe Baumgarten. „Adaptation of Automotive Infotainment Interfaces using Static and Dynamic Adapters“. In: *Proceedings INFORMATIK 2013.* Bd. GI Edition - Lecture Notes in Informatics (LNI). 11. Workshop Automotive Software Engineering. 2013.

[Pen+14] Havoc Pennington, Anders Carlsson, Alexander Larsson, Sven Herzberg, Simon McVittie und David Zeuthen. *D-Bus Specification Revision 0.23.* Juni 2014. URL: http://dbus.freedesktop.org/doc/dbus-specification.html (besucht am 21. 04. 2014).

[Pop+12] Razvan Popescu, Athanasios Staikopoulos, Antonio Brogi, Peng Liu und Siobhán Clarke. „A formalized, taxonomy-driven approach to cross-layer application adaptation“. In: *ACM Trans. Auton. Adapt. Syst.* 7.1 (Mai 2012), 7:1–7:30. ISSN: 1556-4665. DOI: 10.1145/2168260.2168267.

[Pra+12] Thomas Pramsohler, Dirk Kaule, Annette Paulić, Marc Zeller und Gereon Weiß. „Modellbasierte Erkennung von Fehlverhalten: Automatisierte Generierung von Verifikationsmodellen zur Absicherung von Funktionsschnittstellen im Bereich Infotainment“. In: *Elektronik automotive* 8-9/2012 (2012), S. 12–17. ISSN: 1614-0125.

[Pra+13] Thomas Pramsohler, Mahmut Kafkas, Annette Paulic, Marc Zeller und Uwe Baumgarten. „Control Flow Analysis of Automotive Software Components Using Model-Based Specifications of Dynamic Behavior“. en. In: *SAE International Journal of Passenger Cars - Electronic and Electrical Systems* 6.2 (Aug. 2013), S. 425–436. ISSN: 1946-4614, 1946-4622. DOI: 10.4271/2013-01-0435.

[Pra+14] Thomas Pramsohler, Simon Schenk, Andreas Barthels und Uwe Baumgarten. „A layered interface-adaptation architecture for distributed component-based systems“. en. In: *Future Generation Computer Systems* (Okt. 2014). ISSN: 0167739X. DOI: 10.1016/j.future.2014.09.011.

[Pre+07] Alexander Pretschner, Manfred Broy, Ingolf H. Kruger und Thomas Stauner. „Software Engineering for Automotive Systems: A Roadmap“. In: *2007 Future of Software Engineering*. FOSE '07. Washington, DC, USA: IEEE Computer Society, 2007, S. 55–71. ISBN: 0-7695-2829-5. DOI: 10.1109/FOSE.2007.22.

[PSB13] Thomas Pramsohler, Simon Schenk und Uwe Baumgarten. „Towards an Optimized Software Architecture for Component Adaptation at Middleware Level“. In: *Software Architecture*. Hrsg. von Khalil Drira. Lecture Notes in Computer Science 7957. Springer Berlin Heidelberg, Jan. 2013, S. 266–281. ISBN: 978-3-642-39030-2 978-3-642-39031-9.

[RKK13] Dominik Reinhardt, Dirk Kaule und Markus Kucera. *Achieving a Scalable E/E-Architecture Using AUTOSAR and Virtualization*. SAE Technical Paper 2013-01-1399. Warrendale, PA: SAE International, Apr. 2013.

[RQZ07] Chris Rupp, Stefan Queins und Barbara Zengler. *UML 2 glasklar: Praxiswissen für die UML-Modellierung*. German. München; Wien: Hanser, 2007. ISBN: 978-3-446-41118-0 3-446-41118-6.

[Scr07] Kenn Scribner. *Microsoft® Windows® Workflow Foundation: Step by Step*. First. Redmond, WA, USA: Microsoft Press, 2007. ISBN: 978-0-7356-2335-4.

[SK13] Mahesh Shirole und Rajeev Kumar. „UML Behavioral Model Based Test Case Generation: A Survey“. In: *SIGSOFT Softw. Eng. Notes* 38.4 (Juli 2013), S. 1–13. ISSN: 0163-5948. DOI: 10.1145/2492248.2492274.

[SR02] Heinz W. Schmidt und Ralf H. Reussner. „Generating Adapters for Concurrent Component Protocol Synchronisation“. en. In: *Formal Methods for Open Object-Based Distributed Systems V*. Hrsg. von Bart Jacobs und Arend Rensink. IFIP — The International Federation for Information Processing 81. Springer US, Jan. 2002, S. 213–229. ISBN: 978-1-4757-5268-7 978-0-387-35496-5.

[Ste+09] David Steinberg, Frank Budinsky, Marcelo Paternostro und Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. 2nd. Addison-Wesley Professional, 2009. ISBN: 0-321-33188-5.

[Wie92] G. Wiederhold. „Mediators in the architecture of future information systems“. In: *Computer* 25.3 (März 1992), S. 38–49. ISSN: 0018-9162. DOI: 10.1109/2.121508.

[Wie95] Gio Wiederhold. „Mediation in information systems“. In: *ACM Comput. Surv.* 27.2 (Juni 1995), S. 265–267. ISSN: 0360-0300. DOI: 10.1145/210376.210390.

[Wil+06] Doris Wild, Andreas Fleischmann, Judith Hartmann, Christian Pfaller, Martin Rappl und Sabine Rittmann. „An Architecture-Centric Approach towards the Construction of Dependable Automotive Software“. In: *Proceedings of the SAE 2006 World Congress*. 2006.

[YS97] Daniel M. Yellin und Robert E. Strom. „Protocol specifications and component adaptors“. In: *ACM Trans. Program. Lang. Syst.* 19.2 (1997), S. 292–333. ISSN: 0164-0925. DOI: 10.1145/244795.244801.

[Zee12] Alexander Zeeb. „Plug-and-Play-Lösung für Autosar-Software-Komponenten“. German. In: *ATZelektronik* 7.1 (2012), S. 28–33. ISSN: 1862-1791. DOI: 10.1365/s35658-012-0116-z.