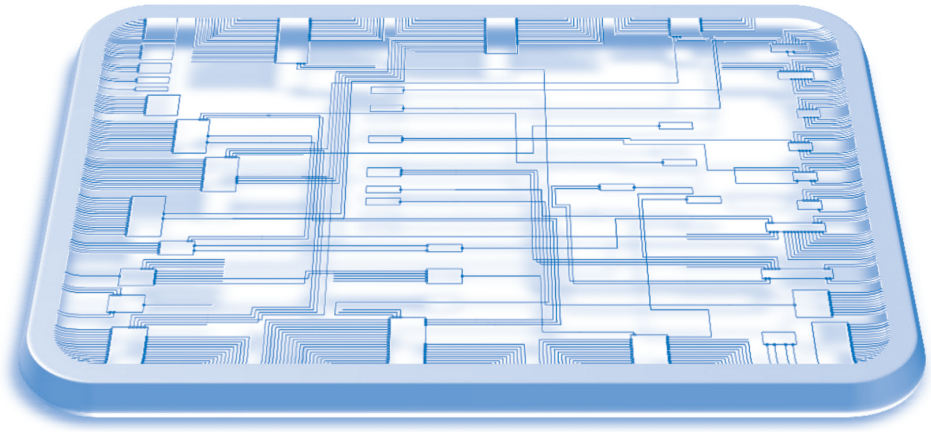
RWTH AACHEN

SE SOFTWARE ENGINEERING

Cem Mengi

Automotive Software

Prozesse, Modelle und Variabilität



Aachener Informatik-Berichte,
Software Engineering

Band 13

Hrsg: Prof. Dr. rer. nat. Bernhard Rumpe
Prof. Dr.-Ing. Dr. h.c. Manfred Nagl

Automotive Software

Prozesse, Modelle und Variabilität

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der
Rheinisch-Westfälischen Technischen Hochschule Aachen
zur Erlangung des akademischen Grades eines Doktors der
Ingenieurwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker
Cem Mengi

aus Aachen

Berichter: Universitätsprofessor Dr.-Ing. Dr. h.c. Manfred Nagl
Universitätsprofessor Dr.-Ing. Stefan Kowalewski

Tag der mündlichen Prüfung: 26. Juni 2012

[Men12] C. Mengi
Automotive Software - Prozesse, Modelle und Variabilität
Shaker Verlag, ISBN 978-3-8440-1262-1.
Aachener Informatik-Berichte, Software Engineering Band 13. 2012.
www.se-rwth.de/publications

Aachener Informatik-Berichte, Software Engineering

herausgegeben von
Prof. Dr. rer. nat. Bernhard Rumpe
Software Engineering
RWTH Aachen University

Band 13

Cem Mengi

Automotive Software

Prozesse, Modelle und Variabilität

Shaker Verlag
Aachen 2012

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Zugl.: D 82 (Diss. RWTH Aachen University, 2012)

Printed in Germany.

ISBN 978-3-8440-1262-0
ISSN 1869-9170

Shaker Verlag GmbH • Postfach 101818 • 52018 Aachen
Telefon: 02407 / 95 96 - 0 • Telefax: 02407 / 95 96 - 9
Internet: www.shaker.de • E-Mail: info@shaker.de

Kurzfassung

Software hat in der Automobilentwicklung eine bedeutende Rolle eingenommen. Sie eröffnet neue Potenziale und ist primärer Innovationstreiber. Gleichzeitig ist Software aber auch ein erheblicher Komplexitäts- und Kostenfaktor. Ein wesentlicher Grund hierfür ist die unzureichende Anwendung geeigneter Methoden und Konzepte zur systematischen Erfassung und Beherrschung von Softwarevarianten im Sinne der Wiederverwendung.

Das Bestreben, Software so zu gestalten, dass es für verschiedene Varianten adaptierbar ist, erfordert geeignete Maßnahmen in allen Phasen des Softwareentwicklungsprozesses: (1) Variabilität muss explizit erfasst werden können, (2) Abhängigkeiten zwischen variablen Entitäten müssen formuliert werden können und (3) Varianten müssen gebunden werden können. Die vorliegende Arbeit schlägt in diesem Zusammenhang verschiedene Lösungskonzepte vor und wendet diese auf drei Entwicklungsebenen an.

Die Basis stellt hierbei ein *Variabilitätsmodell* dar, welches jegliche Form der *Variabilität explizit modelliert* und *strukturiert. Abhängigkeiten* werden durch eine *Restriktionssprache* formuliert. Schließlich werden Varianten durch einen *Konfigurierungsvorgang* und einer anschließenden *Generierung* gebunden. Dieses Variabilitätsmodell wird dann auf allen Entwicklungsebenen eingesetzt.

Für den *konzeptionellen Entwurf mit Funktionsnetzen* wird ein *Top-Down-Modellierungsprozess* vorgeschlagen. Wiederverwendbare Bestandteile werden hierbei zunächst in einer *klassifizierten Domänenbibliothek* modelliert. Diese können dann aus der Bibliothek instanziiert und zur Modellierung von Funktionsnetzen verwendet werden. Zur expliziten und formalen Erfassung von Varianten wird ein *Variabilitätsmechanismus* eingeführt. Dieser wird mit dem Variabilitätsmodell gekoppelt, sodass Funktionsvarianten vollständig beherrscht werden können.

Im *Architekturentwurf mit Simulink-Modellen* wird aufgrund der hierbei etablierten *inkrementellen Variantenentwicklung durch Copy-Paste* ein Bottom-Up-Ansatz verfolgt, um gemeinsame und variable Modellanteile explizit zu identifizieren. Durch die *Anwendung von geeigneten Variabilitätsmechanismen* werden die Modellvarianten in ein Familienmodell überführt. Eine Anbindung an das Variabilitätsmodell komplettiert den Ansatz dieser Phase.

Bei der *Implementierung (mit der Programmiersprache C)* wird ein Ansatz verfolgt, bei dem die *überwachte Variantenimplementierung* mit Anbindung an das Variabilitätsmodell das zentrale Konzept dieser Ebene darstellt. Modifikationen am Quellcode werden an *variantenspezifischen Sichten* durchgeführt und anhand von *Variabilitätsmechanismen* in den ursprünglichen Quellcode überführt.

Die Ansätze der verschiedenen Ebenen sind dadurch charakterisiert, dass sie sowohl auf allen anderen Entwicklungsebenen als auch in Kombination angewendet werden können. Aus den beschriebenen Lösungskonzepten sind eine Reihe *prototypischer Werkzeuge* entstanden. Sie zeigen den Nachweis der Machbarkeit der in dieser Arbeit beschriebenen Ansätze.

Danksagung

An dieser Stelle möchte ich mich bei den lieben Menschen bedanken, die mich in der Zeit meiner Promotion unterstützt haben.

Mein erster und ganz besonderer Dank gilt meinem Doktorvater Herrn Prof. Manfred Nagl. Wie es der Zufall so wollte, habe ich zwar spät aber dennoch rechtzeitig den Weg zum Lehrstuhl für Informatik 3 unter seiner Leitung gefunden. Seine prospektiven Sichtweisen und die intensiven Diskussionen haben zum Gelingen dieser Arbeit wesentlich beigetragen.

Herrn Prof. Stefan Kowalewski danke ich für die Übernahme des Zweitgutachtens. Herrn Prof. Matthias Jarke und Herrn Prof. Berthold Vöcking danke ich für die Bereitschaft als weitere Prüfer. Darüber hinaus möchte ich Herrn Prof. Bernhard Rumpe für seine Unterstützung danken.

Des Weiteren möchte ich mich bei allen bedanken, die ihm Rahmen von studentischen Arbeiten das Automotive-Thema dieser Arbeit vorangetrieben haben. Hierzu gehören Ruben Zimmermann, Antonio Navarro Pérez, Önder Babur, Maxim Pogrebinski, Onur Armaç, Özgür Akcasoy, Youssef Arbach und Jan Pojer. Außerhalb der Uni hatte ich auch Gelegenheiten, andere Talente dieser Gruppe kennenzulernen. Önder Babur ist besonders musikalisch begabt. Seine regelmäßigen Auftritte in Cafés habe ich versucht, nie zu verpassen. Mit Onur Armaç verbindet uns etwas ganz besonderes. Wir sind beide Fans von Galatasaray Istanbul. Gemeinsam haben wir bei vielen Spielen mitgefiebert. Von unserem Rotschopf Youssef Arbach konnte ich viel über die Herkunft von Begriffen aus dem arabischen Raum lernen. Schließlich hat Jan Pojer mir gezeigt, warum die Tschechen die besten Biertrinker der Welt sind.

Weiterhin möchte ich mich bei meinen Kollegen am Lehrstuhl bedanken. Ganz besonders bei Ibrahim Armaç. Er hat mich nicht nur zur Promotion motiviert, sondern stand zu jeder Zeit mit Rat und Tat an meiner Seite. Es war schön, ihn stets als Vorbild am Lehrstuhl zu haben. Thomas Heer war ein liebenswürdiger Büronachbar, mit dem man sich auch außerhalb der Arbeitszeiten sehr schön unterhalten konnte. Ein besonderes Talent, Menschen zum Lachen zu bringen, hatte René Wörzberger. Dank ihm waren insbesondere unsere Weihnachtsfeiern sehr unterhaltsam. Erhard Weinell danke ich für seine Hilfsbereitschaft. Die Ansichten und Ratschläge von Daniel Retkowitz waren für mich immer sehr hilfreich. Theresa Körtgen hat durch die regelmäßigen Sportaktivitäten dafür gesorgt, dass wir nicht am Lehrstuhl einrosten. Gegen die Endphase meiner Promotion hatte ich das Glück, mit Rim Jnidi ein Büro zu teilen. Unsere Unterhaltungen waren immer sehr amüsant. Vielen Dank für die schöne Zeit Rim. Urlaubserfahrungen und -wünsche konnte ich immer am besten mit Marita Breuer teilen. Dir auch einen herzlichen Dank für die Inspirationen. Simon Becker, Thomas Haase, Markus Heller, Bodo Kraft, Ulrike Ranger, Christian Fuß, Christof Mosler, Galina Volkova, Arne Haber, Christoph Herrmann, Thomas Kurpick, Markus Look, Antonio Navarro Pérez, Claas Pinkernell, Holger Rendel, Jan Oliver Ringert und Ingo Weisemöller danke ich für ihre fachliche und organisatorische Unterstützung. Schließlich gilt mein Dank unseren Sekretärinnen Angelika Fleck, Silke Cormann und Sylvia Gunder. Sie waren immer eine großartige Unterstützung

bei sämtlichen organisatorischen Aspekten.

In meiner Freizeit haben meine Freunde dafür gesorgt, den nötigen Abstand von der Arbeit zu bekommen. Ein ganz besonderer Dank gilt Nur und Murad Abu-Tair. Unsere „Spaziergang"- und „Frische Luft"-Expertin war immer Nur. Ihr Ehemann Murad hingegen war auf Autos, Smartphones, Comedy und Burger spezialisiert. Gemeinsam waren sie ein unverzichtbares Duo für viele entspannende Momente. Ebru Armaç hat mit ihren spannenden Geschichten stets für unterhaltsame Abende gesorgt. Gemeinsam mit ihrem Ehemann Ibrahim Armaç waren sie ganz besondere Freunde, die mir immer unterstützend zur Seite standen. Auch die gemeinsamen Wochenendtrips haben immer sehr viel Spaß gemacht und werden hoffentlich in Zukunft fortgeführt. Mit Ismet Aktaş sind wir wie Pech und Schwefel. Seit nun zwölf Jahren schlage ich mit ihm denselben Weg ein. Unsere Gespräche während des Mittagessens oder bei einer Kaffeepause habe ich immer sehr genossen. Aber auch unsere abendlichen Aktivitäten waren immer spaßig. Des Weiteren möchte ich mich bei Gabriella und Giorgio Guarrasi, Vildan und Halil Gülez, Nina und Farzad Afschari, Yasemin und Murat Başaran, Sinem Kuz (vielen Dank für die vielen Korrekturvorschläge), Canan Biçer (meine Kindergarten-, Schul- und Unifreundin) und Canan Kasaci für die tolle Zeit bedanken.

Schließlich gilt mein unendlicher Dank meiner Familie. Meine Eltern, Nezaket und Sefer Mengi, haben mir in jeder Lebenssituation den Rücken gestärkt, mich aufgemuntert und unterstützt. Ohne sie wäre ich wohl heute nicht da, wo ich jetzt bin. Ich hoffe, dass ich ihnen mit dieser Arbeit etwas zurückgeben konnte. *Sevgili anneciğim, sevgili babacığım. Sevginiz ve desteğiniz için sonsuz teşekkürler. Iyiki varsiniz. Sizi çok seviyorum*. Meiner Schwester Aynur und ihrem Ehemann Mustafa Savaşan danke ich für die besonderen gemeinsamen Grillabende. *Ablacığım sen bitanesin (Sende tabiki enişte).* Auch meinen beiden Neffen Cenk und Devin Savaşan danke ich, dass sie mich stets zum Lachen bringen. Meinem Bruder Zafer und seiner Ehefrau Sevgi Mengi danke ich für ihre unerschöpfliche Unterstützung. Ihren beiden Söhnen Enis und Mirkan Mengi danke ich dafür, dass sie mich durch ihre vielen Hausaufgabenfragen zurück in die Schulzeit katapultiert haben. Meinem Bruder Muzaffer und seiner Ehefrau Gülden Mengi danke ich für die lustigen Unterhaltungen, die teilweise in Lachkrämpfen endeten. Meinen beiden Nichten Eyşan und Minel Mengi danke ich, dass sie mir zeigen, wie schön es ist, klein zu sein. Mein ganz besonderer Dank gilt meinem Bruder Alper Mengi, der immer für mich da ist und, wenn nötig, auch Berge für mich versetzen würde. Ein besonderer Dank gebührt auch meinen Schwiegereltern Birgül und Cahit Pişkin, die mich stets unterstützt haben. Ich bin froh euch alle zu haben. Zum Schluss möchte ich mich bei einem ganz wundervollen Menschen an meiner Seite bedanken: meiner Ehefrau Azime. Die vielen gemeinsamen Jahre haben uns zu einem eingespielten Team zusammengeschweißt. Sie ist meine Stütze, auf die ich immer zählen kann. Sie hat mir gezeigt, dass es nichts gibt, das man nicht überwinden kann. Dafür danke ich Dir vom ganzen Herzen mein Engel.

Aachen, Juli 2012
Cem Mengi

Inhaltsverzeichnis

Teil I.

Einleitung

Kapitel 1.

Motivation

Das Automobil hat seit seiner Erfindung im Jahr 1886 durch Karl Benz eine beachtliche Entwicklung zurückgelegt. Nicht zu vergessen sind hierbei auch die Pionierleistungen von Gottlieb Daimler, Wilhelm Maybach oder Henry Ford. All diese und weitere Persönlichkeiten haben den Grundstein für das heutige Fahrzeug gelegt.

Während die ersten Fahrzeuge rein mechanisch betrieben wurden, sind seit Mitte des 20. Jahrhunderts auch Elektrik/Elektronik (E/E)-Komponenten wesentliche Bestandteile eines Automobils. Mittlerweile sind elektronische Steuergeräte, Sensoren, Aktuatoren und Bussysteme ausschlaggebende Komplexitätstreiber in der Entwicklung eines Autos. Der Verbau von nahezu 80 Steuergeräten, die über fünf verschiedene Bussysteme vernetzt sind, Daten zahlreicher Sensoren auswerten und Aktuatoren steuern, ist keine Seltenheit [WH06].

Durch E/E hat auch Software den Weg ins Fahrzeug gefunden. Den historischen Verlauf zeigt Abbildung 1.1. In den 70er Jahren wurden Softwarefunktionen zur Steuerung der Zündung und Einspritzung des Motors entwickelt. Auch erste Komfortfunktionen, wie etwa die Zentralverriegelung, wurden zu dieser Zeit realisiert. Charakteristisch war, dass diese Funktionen isoliert voneinander ausgeführt wurden. Es gab also keinerlei Interaktionen zwischen Funktionen. Erst durch die Einführung von Bussystemen wurden zunehmend kommunizierende Funktionen in die Fahrzeuge integriert. So konnten Sensordaten nicht nur für das direkt angebundene Steuergerät zugänglich gemacht werden, sondern über Bussysteme an weitere über Steuergeräte hinweg verteilte Funktionen gesendet werden. Auf gleiche Weise konnten auch Aktuatoren durch verschiedene Funktionen gesteuert werden. In den 80er und 90er Jahren sowie Anfang des 21. Jahrhunderts sind auf diese Weise zahlreiche softwarebasierte Funktionen entstanden. Sie finden ihre Anwendung in den Bereichen Fahrerassistenz-, Infotainment-, Komfort- und Sicherheitssysteme. Seither sind etwa 270 vom Fahrer wahrnehmbare Funktionen entwickelt, die aus annähernd 2000 Subfunktionen komponiert werden. Die Implementierungssumme dieser softwarebasierten Funktionen beträgt schätzungsweise knapp 1 GB Binärcode [PBKS07, BKPS07]. Die Historie zeigt, dass der Softwareanteil im Fahrzeug in diesem Zusammenhang exponentiell zugenommen hat. Eine Sättigung dieses Trends ist derzeit nicht in Sicht. Dies wird auch dadurch belegt, dass heute nahezu 80% aller Innovationen softwarebasiert sind [Bro06].

Software hat somit im Verlauf der vergangenen Jahrzehnte eine Komplexität erreicht, die nur schwierig zu beherrschen ist. Faktoren, die hierzu beigetragen

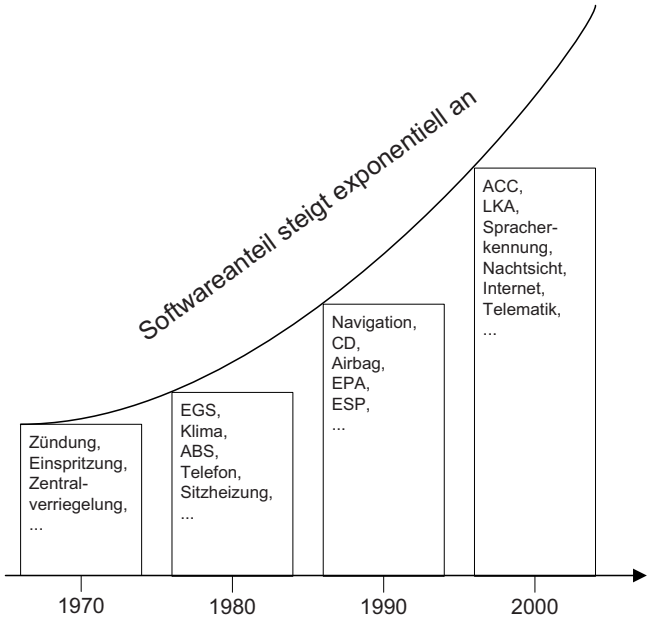


Abbildung 1.1.: Der Softwareanteil in einem Fahrzeug steigt seit Jahrzenten exponentiell an

haben, sind die Folgenden [SZ06]:

- **Hardwaregetriebene Softwareentwicklung:** Bei der Entwicklung eines Fahrzeugs steht traditionell bedingt die Hardwarearchitektur im Vordergrund. Software wird lediglich als vollintegrierter Bestandteil dieser Architektur betrachtet. Sie wird dadurch auch abhängig zu der Hardwarearchitektur realisiert. Prinzipien, wie etwa Abstraktion oder Modularisierung, werden gar nicht oder nur rudimentär angewendet. Die Folgen sind, dass Software schwieriger zu verstehen, zu verändern, zu erweitern und zu warten ist.

- **Proprietäre Lösungen:** Sowohl Automobilhersteller als auch ihre Zulieferer setzen oftmals proprietäre Lösungen für Betriebssysteme, Kommunikationsprotokolle oder Diagnosesoftware ein. Derartige Software ist dadurch schwierig in das Gesamtfahrzeug zu integrieren. Zudem bringt sie den Herstellern keinerlei Wettbewerbsvorteil. Außerdem sind Änderungen nicht ohne großen Aufwand möglich. Schließlich ist es sehr mühselig, Softwarefunktionen zu portieren.

- **Softwareabhängigkeiten:** Im Fahrzeug verteilte Softwarekomponenten stehen oft in Wechselwirkung zueinander. So benötigt eine Softwarekomponente bestimmte Informationen, die von einer anderen Softwarekomponente bereitgestellt wird. Die Auswirkungen bei Änderungen oder Erweiterungen für derartige Fälle sind oftmals nicht bekannt oder nur implizit erfasst, sodass sie nur schwierig nachvollziehbar sind.

- **Geschäftsmodelle:** Software wird mittlerweile nicht nur beim Hersteller entwickelt, sondern auch an Unterauftragnehmer delegiert. Diese haben wiederum ihre eigenen Zulieferer. Eine derartige Struktur kann sich über Stufen fortsetzen. Die Kommunikationskette wird auf diese Weise sehr lang und führt selbst bei kleineren Problemen zu langwierigen Vorgängen.
- **Produktlebenszyklen:** Produktlebenszyklen werden durch lange Lebenszeiten von etwa 20-25 Jahren bestimmt. Software sollte daher robust gegenüber Hardwareveränderungen sein. Zudem sollten Softwareupdates einfach durchgeführt werden können. Aufgrund der bereits erwähnten hardwaregetriebenen Entwicklung ist dies oftmals nicht möglich, sodass Hardwareveränderungen meist auch Softwareveränderungen hervorrufen.
- **Produktvielfalt:** Damit Kundenbedürfnisse erfüllt, größere Marktanteile erlangt und gesetzliche Vorgaben eingehalten werden können, werden Fahrzeuge in verschiedenen Varianten produziert. Für die Entwicklung bedeutet dies, dass Softwarevarianten im gesamten Produktlebenszyklus geeignet erfasst und verwaltet werden müssen.

Unter anderem verantworten die genannten Faktoren, dass Software heute nahezu 40% der Gesamtfahrzeugkosten ausmachen [BKPS07]. Um diese erheblichen Kosten zu reduzieren, bemüht sich die Automobilindustrie Standards in verschiedenen Handlungsfeldern festzulegen. Ziel dabei ist es, überall dort, wo die Konkurrenzfähigkeit eines Herstellers nicht beeinträchtigt wird, eine gemeinsame Basis zu schaffen. So werden beispielsweise durch OSEK VDX Standards unter anderem das Betriebssystem [Por05], die Kommunikation [Por04a] und das Netzwerkmanagement [Por04b] für vernetzte Steuergeräte bestimmt. Mit AUTOSAR werden Softwareschnittstellen und Austauschformate festgelegt (vgl. [AUT10a, AUT10c, AUT10d]). Schließlich wird der Softwareentwicklungsprozess nach dem V-Modell XT organisiert und durchgeführt [VM09].

Diese Maßnahmen sind wertvolle Schritte, um Komplexität zu beherrschen, Software schneller und besser zu entwickeln und Kosten zu senken. Sie umfassen allerdings nicht die gesamte Bandbreite an Problemfeldern. Es gibt immer noch viel Handlungsbedarf in verschiedenen Bereichen. Die vorliegende Arbeit leistet in diesem Zusammenhang einen Beitrag. Sie befasst sich intensiv mit Problemstellungen aus dem Kontext der Produktvielfalt.

Wie bereits weiter oben erwähnt und begründet, streben Automobilhersteller eine hohe Produktvielvalt an. Die Softwareentwicklung wird aber hierdurch deutlich komplexer. In nahezu allen Phasen des Entwicklungsprozesses kann dies beobachtet werden. Die Anfoderungsdefinition kann nicht mehr für ein Produkt formuliert, sondern muss für vielfältige Varianten beschrieben werden. Gleiches gilt für den Architekturentwurf, die Implementierung, das Testen und die Integration. Darüber hinaus sind Varianten auch in der Produktions- sowie in der Betriebs- und Wartungsphase zu berücksichtigen. Variabilität erstreckt sich also im gesamten Prozess. Während für die Anforderungsphase bereits einige Ansätze existieren [KCH$^+$90, AvW00, vdML04, BLP05, BBD$^+$06], die gute Konzepte zur Beherrschung

der Variabilität vorschlagen, sind für die weiteren Phasen noch geeignete Maßnahmen zu treffen. Diese Arbeit setzt an dieser Stelle an und behandelt die frühen Softwareentwicklungsphasen eines Automobils. Diese umfassen den konzeptionellen Entwurf, den Architekturentwurf und die Implementierung. Die Aktivitäten und die erforderlichen sowie resultierenden Softwaredokumente werden hierbei analysiert. Es werden geeignete Lösungskonzepte vorgeschlagen, welche die Softwareentwicklung im Hinblick auf das Variantenproblem unterstützen und somit einen Beitrag leisten, den thematisierten Komplexitätsfaktor zu beheben.

Die folgenden Abschnitte verfolgen zwei Ziele: (1) sie geben einen Überblick hinsichtlich der in dieser Arbeit behandelten Themen und (2) definieren die Probleme detaillierter (vgl. Abschnitt 1.1), skizzieren die Lösungsvorschläge (vgl. Abschnitt 1.2) und beschreiben die hieraus resultierenden wissenschaftlichen Beiträge dieser Arbeit (vgl. Abschnitt 1.3).

1.1. Problemstellung

Im Rahmen dieser Arbeit werden verschiedene Phasen im Softwareentwicklungsprozess betrachtet. In jeder dieser Phasen werden Aktivitäten ausgeführt, die bestimmte Softwaredokumente (auch Artefakte genannt) für die Durchführung benötigen und hieraus Ergebnisdokumente produzieren. In dieser Arbeit werden drei Softwaredokumentklassen genauer betrachtet: (1) Funktionsnetze, (2) Simulink-Modelle und (3) Quellcode. Ihre Eigenschaften und ihr Zusammenhang im Rahmen des Entwicklungsprozesses werden in Kapitel 3 eingehend beschrieben. Im Folgenden werden die in dieser Arbeit betrachteten Problemfelder beschrieben. Dabei wird das Thema Produktvielfalt zunächst durch Beschreibung der allgemeinen Erfordernisse zur Berücksichtigung der Variabilität eingeleitet. Im Anschluss folgen dann die drei zu untersuchenden Softwaredokumente und die Erläuterung ihrer spezifischen Anforderungen.

Variabilität
Das Streben nach einer hohen Produktvielfalt ist ein wesentlicher Komplexitätstreiber. Viele Softwaredokumente im Entwicklungsprozess werden hierdurch beeinflusst, so auch die drei untersuchten Softwaredokumentklassen. Variabilität muss für diese Dokumente erfasst werden, damit wiederverwendbare Bestandteile systematisch realisiert werden können. Wie diese Variabilität identifiziert, modelliert, strukturiert und repräsentiert werden kann, sind Aspekte, die in dieser Arbeit genauer untersucht werden müssen. Hierzu gehört auch die Beantwortung der Fragen, wie variable Eigenschaften definiert, Abhängigkeiten ausgedrückt und schließlich Varianten gebunden werden können.

Funktionsnetze
Ein Funktionsnetz beschreibt Fahrzeugfunktionen und ihre Kommunikation untereinander. Durch diese Beschreibungsform wird die Anforderungsdefinition konkretisiert und als Bindeglied für die E/E-Architektur eingesetzt. Genaueres hierzu wird in

Kapitel 5 erläutert. In der Automobilindustrie werden Funktionsnetze aber sehr unterschiedlich aufgefasst, wodurch auch verschiedene Formalismen und Notationen entstehen. Es muss also aus der Heterogenität ein homogenes Basiskonzept ermittelt werden, damit ein gemeinsames Verständnis von Funktionsnetzen erlangt werden kann, das auf einem einheitlichen Formalismus basiert und entsprechend notiert wird. Wiederverwendung ist ein weiterer Aspekt, der bei der Modellierung von Funktionsnetzen unsystematisch durchgeführt wird. Anstatt wiederverwendbare Bestandteile eines Funktionsnetzes zu identifizieren und geeignet zu verwalten, wird in der Regel die Wiederverwendung durch Kopiervorgänge vorgenommen. Bei Änderungen gibt es daher auch keine Konsistenzerhaltung zwischen den wiederverwendeten Bestandteilen. Dieses Defizit muss durch Einführung geeigneter Wiederverwendungsstrategien, sowohl konzeptionell als auch methodisch, behoben werden. In diesem Zusammenhang spielt auch die Realisierung von Varianten eine wichtige Rolle. Es existiert kein Konzept, mit dem Varianten explizit und formal erfasst werden. Zudem ist kein Mechanismus gegeben, der Varianten methodisch strukturiert und bei Bedarf in das Funktionsnetz einbindet. Es ist also ein Konzept erforderlich, mit dem Varianten in einem Funktionsnetz gekapselt werden können. Zudem muss dieses Konzept auch die Modellierung der Varianten unterstützen, damit sie zum einen explizit dokumentiert und zum anderen durch Konfigurierung gebunden werden können.

Simulink-Modelle

Die zweite Softwaredokumentklasse, welche genauer betrachtet wird, ist Simulink. Simulink ist eine modellbasierte Sprache für den Entwurf des dynamischen Funktionsverhaltens. Insbesondere können Simulink-Modelle vor der Codegenerierung simuliert werden, um den Entwurf validieren zu können. In Kapitel 6 werden Simulink-Modelle genauer behandelt. Varianten entstehen in diesem Kontext inkrementell und evolutionär. Demnach wird ein Simulink-Modell durch Wiederverwendung und entsprechende Modifikationen in ein weiteres Simulink-Modell überführt, welches eine weitere Variante darstellt. In der Regel wird dies aber durch Copy-Paste durchgeführt. Eine derartige Form der Wiederverwendung ist unsystematisch und führt zu einem Abstraktionsverlust. Infolgedessen sind gemeinsame Anteile der Simulink-Modellvarianten nur noch implizit erfasst. Die Unterschiede zwischen den Varianten können daher nur schwer ausfindig gemacht werden. Besonders wichtig ist es daher, das Wissen in Bezug auf Gemeinsamkeiten und Variabilität zwischen Simulink-Modellvarianten wiederherzustellen. Ist dies erreicht, so haben alle Varianten ein gemeinsames Basismodell. Damit dieses Basismodell verwendet werden kann, müssen Varianten als Erweiterungen hiervon modelliert werden. Um dies zu erzielen, muss also die identifizierte Variabilität in den Simulink-Modellvarianten in das Basismodell überführt werden. Hierfür sind adäquate Restrukturierungsmaßnahmen erforderlich. Eine weitere Schwierigkeit hierbei ist, geeignete Mechanismen zu finden, mit denen Varianten modelliert und verwaltet werden können.

Quellcode
Als letzte Softwaredokumentklasse wird der Quellcode betrachtet. Dieser ist entweder aus den Simulink-Modellen generiert oder manuell implementiert. In der Regel wird hierfür die Programmiersprache C verwendet. Weitere Details werden in Kapitel 7 erläutert. Varianten werden im Quellcode durch geeignete Sprachmechanismen, wie etwa Präprozessordirektiven oder Auswahlanweisungen, implementiert. Der Programmierer ist dabei mit allen Varianten konfrontiert. Eine Konzentration auf eine bestimmte Variante ist daher sehr schwierig. Der Programmierer muss also geeignet unterstützt werden, um die Variantenkomplexität zu beherrschen. Ein weiterer Komplexitätsfaktor ist, dass der Quellcode einer Variante verstreut ist. Der Programmierer muss die entsprechenden Stellen manuell ausfindig machen. Es muss also auch hierfür ein geeignetes Konzept entwickelt werden, das diese Komplexität insofern reduziert, dass der Programmierer nur den Code der gewünschten Variante sieht. Eine weitere Schwierigkeit für einen Programmierer ist, Abhängigkeiten zwischen Varianten zu identifizieren, die nicht direkt aus dem Quellcode heraus zu erkennen sind. Es muss also eine Möglichkeit bestehen, diese Abhängigkeiten zu beschreiben und bei Bedarf für den Quellcode anzuwenden, sodass nur die entsprechenden Codefragmente den Programmierer angezeigt werden. In diesem Zusammenhang stellt die fehlende Kenntnis über gültige Konfigurationen eine weitere Problematik dar. Implikationen oder Exklusionen zwischen Varianten sind in der Regel nicht im Quellcode dokumentiert. Der Programmierer muss daher bei der Erstellung gültiger Konfigurationen geeignet unterstützt werden.

1.2. Lösungsansatz

Für die im vorherigen Abschnitt beschriebenen Probleme werden in dieser Arbeit entsprechende Lösungen noch detailliert vorgestellt. Im Folgenden werden diese Lösungen kurz skizziert und somit ein Überblick über bevorstehende Themenfelder gegeben. Die Lösungsskizze ist entsprechend der Problemstellung unterteilt.

Variabilität
Variabilität wird durch Einführung eines Variabilitätsmodells erfasst. Im Rahmen dieser Arbeit wird ein Ansatz verfolgt, der die Vorteile der zwei am häufigsten angewendeten Modellierungsarten (1. hierarchisch strukturierte Variabilitätsmodelle und 2. Auswahlmodelle) kombiniert und um Konzepte zur Modellierung von Variabilitätsmechanismen, Bindungsmechanismen und Bindezeiten erweitert. Zur Strukturierung von Varianten wird ein Gruppierungskonzept eingeführt. Zur Repräsentation der Variabilität wird eine Baumlistenansicht vorgeschlagen. Zur Definition variabler Eigenschaften werden Gruppen- und Variantenkardinalitäten in das Variabilitätsmodell integriert. Weiterhin besteht die Möglichkeit, Abhängigkeiten durch Restriktionen zu formulieren. Zu diesem Zweck wurde eine Restriktionssprache entwickelt, die eine erweiterte Form der Aussagenlogik darstellt. Zudem wurde eine aus der Literatur etablierte Restriktionssprache, Weight Constraint Rule Language (WCRL) [SNS02], als weitere Möglichkeit zur Formulierung von Abhängigkeiten

integriert. Schließlich wurde zur Bindung von Varianten ein Konfigurationsmodell entwickelt, das den Konfigurierungsvorgang proaktiv validiert und somit ausschließlich gültige Konfigurationen zulässt.

Funktionsnetze
Die Existenz verschiedener Formalismen und Notationen bei der Modellierung von Funktionsnetzen wird durch Einführung eines einheitlichen Metamodells zur Beschreibung von Funktionsnetzen homogenisiert. Hier können existierende Ansätze integriert werden. Auch neue Ansätze werden in diesem Rahmen vorgestellt und in das Lösungskonzept integriert, sodass ein umfassendes Modell zur Beschreibung von Funktionsnetzen entsteht. Insbesondere spielen Abstraktionsmerkmale und -regeln eine wichtige Rolle, um die verschiedenen Ansätze in Beziehung zu setzen. Zur Verbesserung der Wiederverwendung von Funktionsnetzen wird ein grundlegend neuer Entwicklungsprozess vorgeschlagen, der aus zwei Teilprozessen besteht. Im ersten Teilprozess werden zunächst wiederverwendbare Bestandteile der Domäne erfasst. Sie dient als eine Art Bibliothek von Funktionsnetzen. Durch Anwendung von objektorientierten Paradigmen wird die Domänenbibliothek klassifiziert, sodass sie in Form einer objektorientierten Baumhierarchie modelliert wird (auch Domänenmodell genannt). Im zweiten Teilprozess ist es vorgesehen, mittels des Domänenmodells sämtliche Abstraktionsstufen des Funktionsnetzes zu modellieren. Auf diese Weise werden wiederverwendbare Teile aus dem Domänenmodell instanziiert und zur Modellierung von Funktionsnetzen herangezogen. Zur Realisierung von Varianten wird das Konzept der Funktionsvarianten eingeführt. Dies ist ein Variabilitätsmechanismus, mit dem Varianten systematisch gekapselt und modelliert werden können. Schließlich werden durch Einführung und Integration des Variabilitätsmodells sämtliche Varianten strukturiert, variable Merkmale definiert, in Beziehung zueinander gesetzt und bei Bedarf für die Konfigurierung verwendet.

Simulink-Modelle
Damit Gemeinsamkeiten und Unterschiede von Simulink-Modellvarianten detektiert werden können, wird ein Differenzierungsprozess eingeführt. Der Prozess besteht aus der Eingabe von zwei Modellvarianten und der Ausgabe von drei Modellen: ein Kommunalitätsmodell und zwei Differenzmodelle. Das Kommunalitätsmodell beinhaltet den gemeinsamen Anteil der beiden Simulink-Modellvarianten und zusätzlich auch sämtliche Markierungen von Variationspunkten. Diese kennzeichnen die Stellen, an denen die Modelle variieren. Da Simulink selbst keine Konzepte zur Markierung von Variationspunkten besitzt, werden zu Visualisierungszwecken farbige Markierungen verwendet. In den beiden Differenzmodellen sind jeweils die spezifischen Teile der Simulink-Modellvarianten enthalten. Sie definieren also die im Kommunalitätsmodell markierten Variationspunkte mit spezifischen Eigenschaften. Damit das Konzept der Differenzierung auch für andere Modelle eingesetzt werden kann, wurde es nicht direkt innerhalb Simulink entwickelt. Entsprechende Metamodelle für Simulink-, Kommunalitäts- und Differenzmodelle sowie Import- und Exportfunktionen sind daher weitere Bestandteile der Lösung. Die Ergebnisse aus der

Differenzierung werden im Weiteren dazu verwendet, die Simulink-Modellvarianten so umzustrukturieren, dass Gemeinsamkeiten und Variabilität in ein Familienmodell überführt werden. Zu diesem Zweck werden Regeln eingeführt, die bei dieser Restrukturierung unterstützend wirken. Ähnlich wie bei Funktionsnetzen wird zur Modellierung und Bindung von Varianten das Variabilitätsmodell eingesetzt.

Quellcode
Verstreute Codefragmente einer Variante können durch Einführung eines Variabilitätsmodells beherrscht werden, indem eine im Variabilitätsmodell modellierte Variante mit allen verstreuten Codefragmenten assoziiert wird. So können auch gleichzeitig alle Varianten erfasst und zentralisiert werden. Abhängigkeiten zwischen Varianten können ebenfalls im Variabilitätsmodell formuliert werden. Da aus dem Variabilitätsmodell ein Konfigurationsmodell abgeleitet werden kann, sind gültige Konfigurationen leicht erstellbar. Das Variabilitätsmodell wird schließlich mit dem Quellcode assoziiert, sodass aus einer Konfiguration eine variantenspezifische Sicht erzeugt wird. Dies führt zu einer signifikanten Reduktion der Komplexität des Quellcodes. Die Implementierung innerhalb einer Sicht wird überwacht und anschließend durch Transformationsregeln in den ursprünglichen Quellcode überführt.

1.3. Wissenschaftliche Beiträge

In dieser Arbeit wurden die beschriebenen Problemstellungen behandelt und entsprechende Lösungskonzepte vorgeschlagen. Verwandte Ansätze wurden in diesem Zusammenhang ebenfalls analysiert und bewertet. Sie werden in den jeweiligen Kapiteln beschrieben. Insbesondere wird dabei erläutert, inwiefern die Lösungsansätze dieser Arbeit sich von den verwandten Ansätzen abgrenzen. Darüber hinaus beinhalten die Ansätze dieser Arbeit neue wissenschaftliche Beiträge. Sie werden im Folgenden erläutert.

Variabilität

Proaktive Unterstützung bei der Variabilitätsmodellierung Die Variabilitätsmodellierung wird in dieser Arbeit als integraler Bestandteil für den Entwurf verschiedener Softwaredokumente betrachtet. Ein wesentlicher Beitrag in diesem Zusammenhang ist die proaktive Unterstützung bei der Variabilitätsmodellierung. Werden Variationspunkte in den Softwaredokumenten durch Variabilitätsmechanismen realisiert, kann dieses Wissen unmittelbar in das Variabilitätsmodell propagiert werden. Dies wird durch ein Assoziationskonzept realisiert, das Entitäten der Softwaredokumente mit dem Variabilitätsmodell verknüpft. Auf diese Weise wird der Aufwand zur Variabilitätsmodellierung reduziert.

Explizite Modellierung von Variabilitätsmechanismen Variabilitätsmechanismen realisieren Variationspunkte in den Softwaredokumenten. In der Regel werden hierfür die Mechanismen der zugrunde liegenden Sprache eingesetzt. Eine Unterscheidung der Variabilitätsmechanismen von anderen Sprachteilen im Softwaredokument ist daher nicht ohne Weiteres möglich. Der Beitrag in dieser Arbeit ist diesbezüglich eine explizite Modellierung von Variabilitätsmechanismen im Variabilitätsmodell. Da das Variabilitätsmodell mit den Softwaredokumenten verknüpft ist, können Variabilitätsmechanismen stets identifiziert werden.

Funktionsnetze

Zweiphasige Funktionsnetzmodellierung Der Mangel an Wiederverwendungsstrategien in Funktionsnetzen wird in dieser Arbeit sowohl methodisch als auch konzeptionell behoben. So ist ein zweigeteilter Modellierungsprozess eine wesentliche Verbesserung beim Entwurf von Funktionsnetzen. Hierbei werden in einer ersten Phase wiederverwendbare Elemente und Strukturen modelliert, die zudem durch objektorientierte Paradigmen in Beziehung gesetzt werden können. Auf diese Weise entsteht eine Domänenbibliothek, die in einer zweiten Phase zur Instanziierung für die eigentliche Funktionsnetzmodellierung verwendet werden kann.

Abstraktionsebenen und -regeln Funktionsnetze dienen als Bindeglied zur Erschließung der Lücke zwischen der Anforderungsspezifikation und der E/E-Architektur. Bisher konnte kein Ansatz diese Lücke vollständig schließen, da sie entweder zu nah an der technischen Realisierung oder zu abstrakt sind. Durch den Beitrag in dieser Arbeit wird die Erschließung nun vollständig erreicht. Zu diesem Zweck wurde ein Ansatz realisiert, bei dem die Funktionsnetzmodellierung auf mehreren Abstraktionsebenen durchgeführt wird. So können je nach Bedarf Details hinzugefügt bzw. vernachlässigt werden. Der Übergang zwischen den verschiedenen Ebenen wird durch Abstraktionsregeln unterstützt. Auf diese Weise können Übergänge teilautomatisiert durchgeführt werden.

Variabilitätsmechanismus Der bisherige Ansatz zur Modellierung von Varianten in Funktionsnetzen besteht aus dem Entwurf des maximalen Funktionsnetzes, welches alle Varianten beinhaltet. Spezifische Ausprägungen werden hierbei durch Wegstreichen nicht erforderlicher Anteile erzeugt. Variationspunkte werden dabei kaum formal erfasst. Varianten können nicht systematisch strukturiert werden. Auch Abhängigkeiten zwischen Varianten können nicht ausgedrückt werden. Im Rahmen dieser Arbeit wurde daher ein Beitrag geleistet, bei dem Varianten durch einen Variabilitätsmechanismus gekapselt und strukturiert werden können. Dies ermöglicht auch gleichzeitig eine nahtlose Anbindung an Variabilitätsmodelle, sodass auch Abhängigkeiten definiert werden können.

Simulink-Modelle

Differenzierung mit Strukturerhaltung Bei der Differenzierung zwischen Simulink-Modellvarianten wird oftmals die Hierarchie in den Modellen entfernt, sodass die Modelle aus genau einer Ebene bestehen. Die Ergebnisse einer Differenzierung können dann aber nur sehr schwer nachvollzogen werden. Der Beitrag in dieser Arbeit besteht aus einem Differenzierungsansatz, bei dem die Struktur der Modelle nicht verändert wird, sodass die Ergebnisse auch direkt nachvollziehbar sind.

Evaluation von Variabilitätsmechanismen Simulink bietet eine Reihe von kommerziellen Variabilitätsmechanismen an. Ihre Anwendung kann aber unterschiedliche Ergebnisse sowohl in der Codegenerierung als auch in der Ausführung verursachen. Die Eignung bzw. die Auswirkungen eines bestimmten Variabilitätsmechanismus in einer bestimmten Situation ist nicht dokumentiert und den Entwicklern oftmals nicht bewusst. Diese Arbeit leistet diesbezüglich einen wesentlichen Beitrag durch Bewertung aller existierender Variabilitätsmechanismen anhand verschiedener Kriterien. Sie stellt somit eine Entscheidungsgrundlage für die Entwickler dar.

Restrukturierung Simulink-Modellvarianten, die durch Copy-Paste entstanden sind, leiden an einer unsystematischen Wiederverwendung, bei dem das Wissen über Gemeinsamkeiten und Variabilität nur noch implizit vorhanden ist. Durch einen Restrukturierungsprozess kann dieses Defizit wieder behoben werden. Der Prozess wird direkt im Anschluss an die Differenzierung durchgeführt. Dabei unterstützen Regeln die Restrukturierung in ein Simulink-Familienmodell. Die Regeln bestehen insbesondere aus der Anwendung von geeigneten Variabilitätsmechanismen.

Quellcode

Variabilitätsmodellierung mit Codeanbindung durch Variabilitätsmechanismen Varianten im Quellcode werden entweder durch Präprozessordirektiven oder Auswahlanweisungen realisiert. Die Modellierung der Variabilität wird dabei unabhängig durchgeführt, sodass keine direkte Assoziation zwischen dem Quellcode und dem Variabilitätsmodell besteht. Der Beitrag in dieser Arbeit liegt diesbezüglich bei der Herstellung einer engen Kopplung zwischen beiden Dokumenten. Die Variabilitätsmodellierung stellt somit einen integralen Bestandteil bei der Programmierung dar. Die Variabilitätsmechanismen der Programmiersprache werden dabei als Anknüpfungspunkte vom Variabilitätsmodell zum Code verwendet. Auf diese Weise besteht stets eine direkte Beziehung zwischen beiden Dokumenten.

Variantengetriebene Implementierung Variabilitätsmodellierung und die Implementierung wurden bisher als zwei separate Aktivitäten betrachtet. Dies führt oft zu der Situation, dass Varianten im Variabilitätsmodell und Varianten im Quellcode

redundant verwaltet werden müssen, sodass Mehraufwand und Inkonsistenzen entstehen. In diesem Zusammenhang ist der Beitrag dieser Arbeit die Einführung eines neuen Prozesses, der die variantengetriebene Implementierung unterstützt. Sie besteht aus der Variabilitätsmodellierung, der Konfigurierung einer Variante, die Sichterzeugung entsprechend der Konfiguration und einer überwachten Implementierung. Bei der überwachten Implementierung werden jegliche Modifikationen mit der entsprechenden Konfiguration verknüpft.

1.4. Struktur der Arbeit

Diese Arbeit ist wie folgt strukturiert:

Kapitel 2 stellt das dieser Arbeit zugrunde liegende Beispielszenario vor. Insbesondere werden die Eigenschaften des Beispielfahrzeugs erläutert, das elektronische System dargestellt, die für diese Arbeit relevanten Funktionen beschrieben und in einem Szenario in Beziehung gesetzt. Im weiteren Verlauf dieser Arbeit werden Beispiele herangezogen, die auf diesem Kapitel basieren.

Kapitel 3 beschreibt den dieser Arbeit zugrunde liegenden Referenzprozess für die Softwareentwicklung. Hier werden die wesentlichen Aktivitäten beleuchtet und zugehörige Softwaredokumente erläutert. Ihr Zusammenhang wird ebenfalls erklärt. Dabei werden sich drei Ebenen im Prozess herauskristallisieren: (1) die Funktionsebene, (2) die Architekturebene und (3) die Codeebene. Diese Ebenen stellen auch gleichzeitig die Hauptgliederung dieser Arbeit dar.

Kapitel 4 motiviert das Thema Variabilität. Der Bedarf für ein Variabilitäts-, Restriktions-, Konfigurations und Generierungsmodell wird hierbei begründet. Im weiteren Verlauf der Arbeit werden diese Modelle auf den verschiedenen Ebenen des Referenzprozesses angewendet.

Kapitel 5 beschreibt die Funktionsebene. Funktionsnetze, ihre Eigenschaften und Besonderheiten werden in diesem Kapitel dargelegt. Die Behandlung der Mängel von Funktionsnetzen stellen die weiteren inhaltlichen Aspekte des Kapitels dar. So werden Konzepte zur Funktionsnetzmodellierung vorgestellt, Abstraktionsebenen und -regeln bei der Modellierung beschrieben und die Variabilitätsmodellierung eingeführt.

Kapitel 6 beschreibt die Architekturebene. Simulink-Modelle werden hierbei genauer betrachtet. Insbesondere wird die Differenzierung von Simulink-Modellvarianten erläutert. Auch die Restrukturierung und die Variabilitätsmodellierung sind wesentliche Inhalte dieses Kapitels.

Kapitel 7 beschreibt die Codeebene. Hier wird der Prozess zur variantengetriebenen Implementierung vorgestellt. Im Weiteren Verlauf des Kapitels wird auf die

einzelnen Aktivitäten dieses Prozesses näher eingegangen.

Kapitel 8 fasst die Arbeit schließlich zusammen. Zusätzlich werden in diesem Zusammenhang auch Anreize für weitere Themenfelder gegeben.

Kapitel 2.

Ein Beispielszenario

Dieses Kapitel beschreibt ein Beispiel, das im Rahmen dieser Arbeit verwendet wird, um die untersuchten Problemstellungen zu erläutern und die Lösungskonzepte darzustellen. In diesem Beispiel werden drei wichtige Bereiche verknüpft: (1) der *Fahrer*, (2) das *Fahrzeug* und (3) die *Umwelt* [SZ06, WH06]. Abbildung 2.1 illustriert dies.

Der Bereich Fahrer umfasst hierbei auch weitere Benutzer, wie etwa Beifahrer. Diese sind zur Vereinfachung nicht in der Abbildung dargestellt. Der Fahrer kann Funktionen des Fahrzeugs direkt oder indirekt beeinflussen und nimmt ihre Auswirkungen bewusst oder unbewusst wahr.

Das Fahrzeug ist aus vielfältigen Elementen zusammengesetzt. So sind beispielsweise Motoren, Getriebe, Reifen, Türen, Spiegel und Lichter grundlegende Bauteile von Fahrzeugen. Das elektronische System stellt ebenfalls einen wichtigen Bestandteil eines Fahrzeugs dar und spielt in den folgenden Betrachtungen eine wesentliche Rolle. Typischerweise besteht ein elektronisches System aus Sollwertgebern, Sensoren, Aktuatoren, elektronischen Steuergeräten und Bussystemen. Durch sie werden viele Funktionen des Fahrzeugs realisiert, zum Beispiel die Getriebe- bzw. Motorsteuerung. Aber auch innovative Fahrerassistenzsysteme, wie etwa Parkassistenten, adaptive Geschwindigkeitsregelung und Spurhalteassistenten, werden durch elektronische Systeme realisiert.

Die Umwelt umfasst alle Komponenten, die durch das Zusammenspiel von Fahrer und Fahrzeug beeinflusst werden. Diese sind weitere Fahrzeuge, Verkehrsschilder, Kommunikationsinfrastrukturen aber auch weitere elektronische Systeme, wie beispielsweise Diagnosecomputer in einer Werkstatt.

Die Schnittstelle zwischen Fahrer und Fahrzeug wird primär durch den Sollwertgeber beeinflusst. Ein Beispiel hierfür ist das Gaspedal. Mit ihr wird die gewünschte Fahrgeschwindigkeit eingestellt. Aber auch Benutzer- bzw. Bedienschnittstellen aus dem Multimediabereich erlauben dem Fahrer, Informationen mit dem Fahrzeug auszutauschen.

Darüber hinaus ist eine Schnittstelle zwischen Fahrzeug und Umwelt unabhängig vom Fahrer vorhanden. Dabei ist das Fahrzeug in eine übergeordnete Kommunikationsinfrastruktur eingebettet, um mit seiner Umwelt zu kommunizieren. Zum Beispiel ist das Fahrzeug mit dem Global Positioning System (GPS) zur Positionsbestimmung oder mit sogenannten Car-2-Infrastructure-Kommunikationssystemen zur Interaktion mit weiteren Fahrzeugen, Gebäuden und Verkehrsschildern ausgestattet.

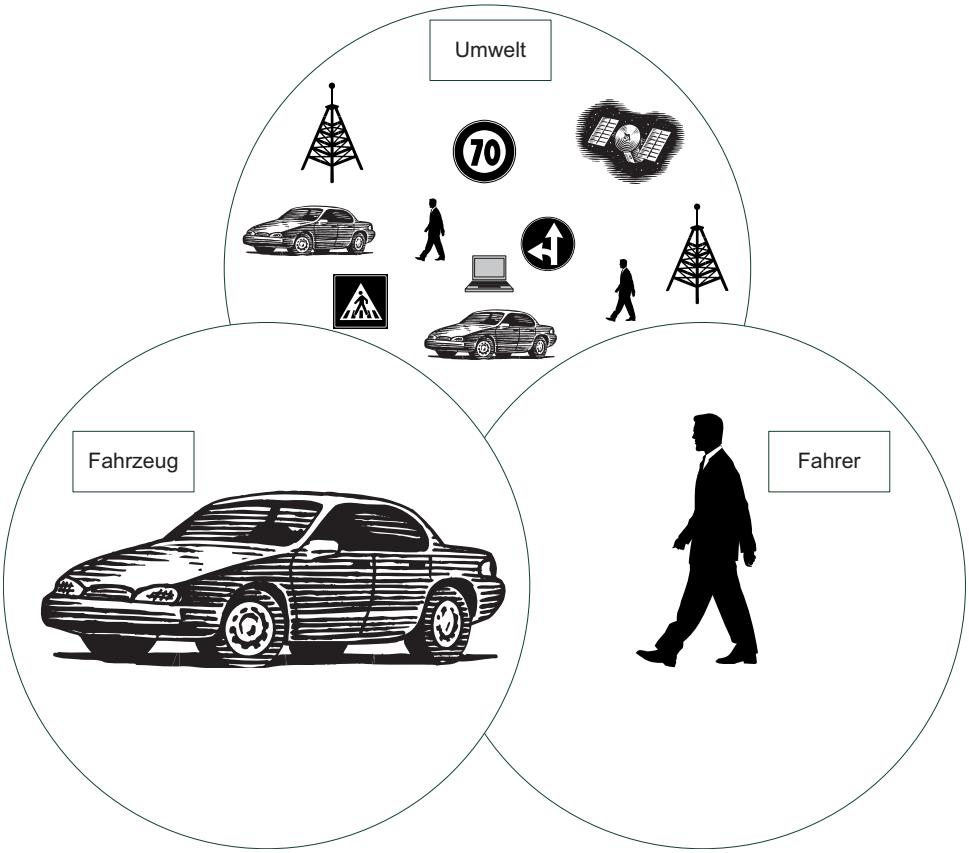


Abbildung 2.1.: Das Zusammenspiel zwischen Fahrer, Fahrzeug und Umwelt

Das Zusammenspiel zwischen Fahrer, Fahrzeug und Umwelt kann insbesondere zur Realisierung fahrerunterstützender Funktionen (den Fahrerassistenzsystemen) herangezogen werden. Das in diesem Kapitel betrachtete Beispiel bezieht die Entitäten Fahrer, Fahrzeug und Umwelt, ihre Schnittstellen sowie ihr Zusammenspiel ein.

Im Folgenden wird das Fahrzeug genauer vorgestellt (vgl. Abschnitt 2.1, Abschnitt 2.2 und Abschnitt 2.3). Der Fahrer und die Umwelt werden später genauer beschrieben, wenn das Zusammenspiel der drei Entitäten in einem Szenario (vgl. Abschnitt 2.4) erläutert wird.

2.1. Das Fahrzeug: BMW X5 xDrive50i

Das in dieser Arbeit verwendete Beispiel basiert auf einem Fahrzeug des Herstellers Bayerische Motoren Werke (BMW). Abbildung 2.2 stellt eine Baumstruktur dar, in der die Personenkraftwagen (PKW)-Domäne von BMW illustriert ist. Sie beinhaltet verschiedene Fahrzeugmodelle, Motorisierungen, Editionen und Pakete, Innenraumausstattungen sowie Grund- und Sonderausstattungen. Aus diesen Features kann ein Kunde seine individuellen Wünsche für das Fahrzeug festlegen. Die grau hinterlegten Knoten stellen eine mögliche Auswahl für ein Fahrzeug dar. Es handelt sich dabei um einen BMW X5 xDrive50i. Dieses Fahrzeugmodell beinhaltet als Grundausstattung unter anderem einen Start-Stop-Knopf, elektrische Fensterheber und eine Zentralverrieglung mit Funkfernbedienung. Die Sonderausstattungen sind optional und können vom Kunden als zusätzliche Features ausgewählt werden. In dem Beispiel aus Abbildung 2.2 sind die Sonderausstattungen Komfortzugang, Adaptive Drive und Auto-Start-Stop ausgewählt.

2.2. Das elektronische System

Das *elektronische System* besteht aus den Bestandteilen *Sollwertgeber*, *Sensoren*, *Aktuatoren*, *Steuergeräte*, *Bussysteme* und schließlich *(Software-)Funktionen*, die über alle Steuergeräte hinweg verteilt sind. Typischerweise wird ein elektronisches System in Subsysteme unterteilt. Diese sind der Antriebsstrang, das Fahrwerk, die Karosserie und das Multimedia.

Der Antriebsstrang umfasst alle Komponenten, die für die Steuerung von Antrieb und Getriebe erforderlich sind. Im Fahrwerk befinden sich u.a. Komponenten zur Regelung der Bremsen, Lenkung und Federung. Im Karosseriesubsystem existieren größtenteils Komponenten für Komfort, wie beispielsweise Zentralverriegelung und Fensterheber, und Sicherheit, wie zum Beispiel Airbag und Gurtbelegung. Schließlich beinhaltet das Multimediasubsystem Komponenten wie das Navigationssystem, Radio und Telefon.

Durch den Einsatz von Bussystemen werden Steuergeräte innerhalb von Subsystemen miteinander verknüpft. Durch die Verwendung eines zentralen Steuergeräts, das als Gateway fungiert, werden schließlich Subsysteme miteinander verbunden. Somit ist es möglich, insbesondere Sensordaten subsystemübergreifend mehreren Steuergeräten zur Verfügung zu stellen. Dadurch wird der Verbau von direkten Leitungskabeln zwischen Sensoren und Steuergeräten reduziert. Zum Beispiel ist die Fahrzeuggeschwindigkeit, die über entsprechende Sensoren an den Rädern ermittelt wird, sowohl für die Motor- und Getriebesteuerung als auch für die Zentralverriegelung relevant.

Funktionen werden in der Regel in Subfunktionen zerlegt. Jede dieser Subfunktionen erfüllt eine bestimmte Aufgabe. Diese müssen nicht notwendigerweise gemeinsam in einem Steuergerät integriert sein. Stattdessen können sie in mehreren Steuergeräten ausgeführt werden. Die Ausführung einer Funktion wirkt sich letztlich

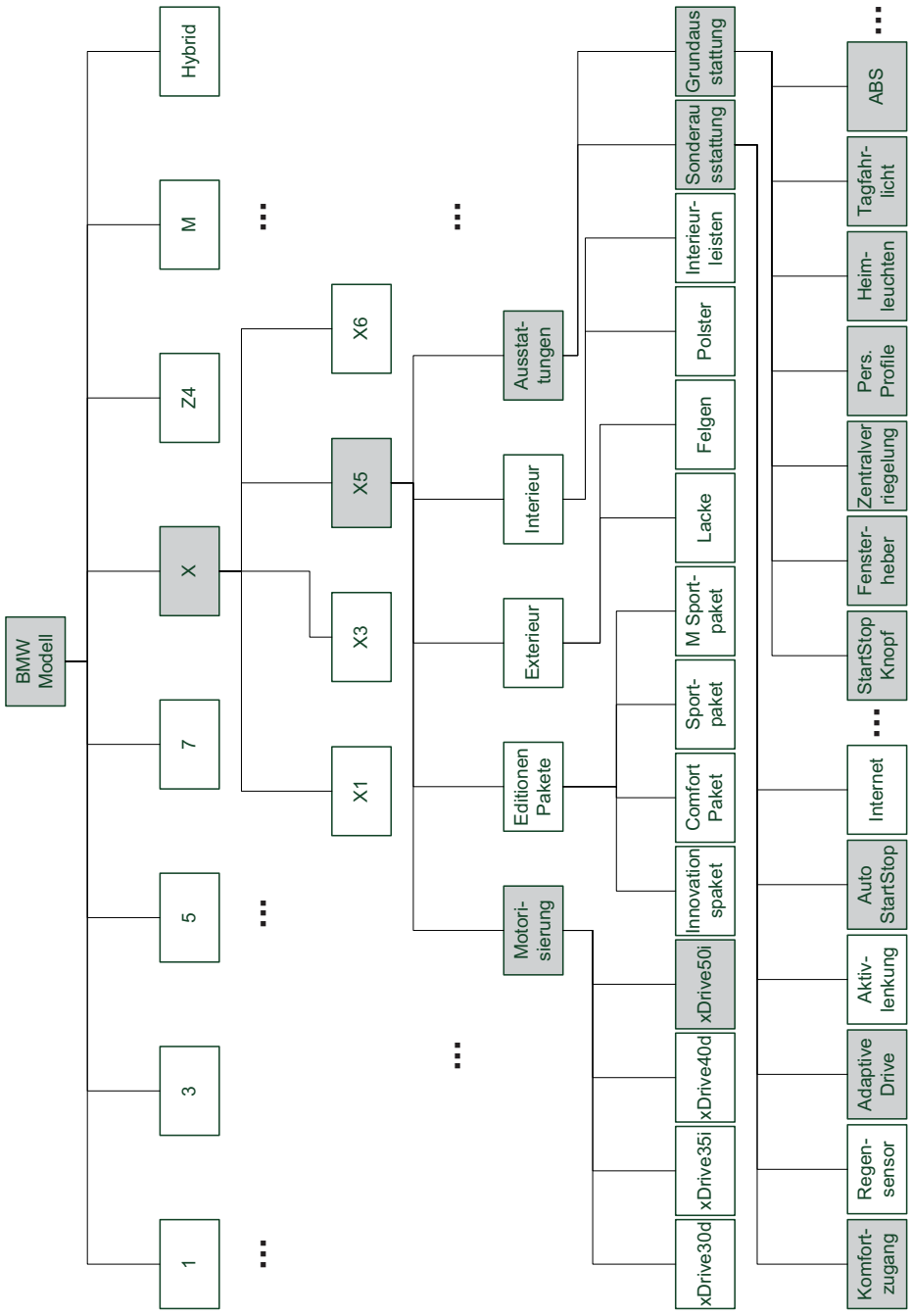


Abbildung 2.2.: Eine Klassifikation der PKW-Domäne des Herstellers BMW

oftmals auf ein oder mehrere Aktuatoren aus. Beispielsweise regelt die Motorsteuerung die Aktuatoren Zündung und Einspritzung.

Abbildung 2.3 stellt das Bussystem mit den angebundenen Steuergeräten für das Fahrzeugmodell BMW X5 xDrive50i dar. In der Regel spiegelt das Bussystem die verschiedenen Fahrzeugsubsysteme wieder. Das Karosseriesubsystem ist im linken Teil der Abbildung zu sehen, das Multimediasubsystem im mittleren Teil und der Antriebsstrang und das Fahrwerk schließlich im rechten Teil der Abbildung. Insgesamt können in diesem Fahrzeugmodell bis zu 90 Steuergeräte verbaut sein (einige Steuergeräte werden mehrfach verbaut). Die Erläuterungen zu den jeweiligen Steuergeräten sind in Tabelle 2.1 und Tabelle 2.2 aufgeführt.

Jedes Steuergerät initiiert oder führt eine oder mehrere Funktionen aus. Beispielsweise wird die Zentralverriegelung vom Steuergerät Car Access System (CAS) initiiert. Die eigentliche Ausführung findet in Junction Box (JB) statt. Die Kommunikation zwischen Steuergeräten findet über 10 verschiedene Bussysteme statt. Das Controller Area Network (CAN)-Bussystem wird für die Diagnose, das Fahrwerk, die Karosserie, und den Antriebsstrang eingesetzt. Auch wenn das zugrunde liegende Kommunikationsprotokoll dasselbe ist, unterscheiden sie sich in Bitübertragungsraten. Weiterhin kommen Bussysteme wie FlexRay, Local Interconnect Network (LIN), Local CAN (LoCAN) und Media Oriented Systems Transport (MOST) zum Einsatz. Darüber hinaus werden konventionelle Kommunikationsprotokolle wie Bit-Serielle Datenschnittstelle (BSD) und Karosserie-Bus (K-Bus) eingeführt.

Im Folgenden werden die relevanten Funktionen für das Beispiel beschrieben. Ihre Funktionsweisen werden, wenn erforderlich, auf Basis des zugrunde liegenden elektronischen Systems detailliert erläutert. Insbesondere werden hier das Deployment der Funktionen auf Steuergeräte sowie die Kommunikation unter den Steuergeräten zur Realisierung der Funktionen beschrieben.

2.3. Die Funktionen

Der BMW X5 xDrive50i verfügt über eine Vielzahl von Ausstattungen wie beispielsweise für Optik und Komfort sowie für Fahrerassistenzsysteme und Sicherheit. Als Anwendungsbeispiel wird das *Fahrzeugzugangssystem* herangezogen (in Anlehnung auf [SAV06a, SAV06b, SAV06c, SAV06f, SAV06d]). Im Folgenden werden die wesentlichen Bestandteile vorgestellt, die für das Verständnis des Beispiels erforderlich sind. Es umfasst alle Subfunktionen, die beim Verriegeln bzw. Entriegeln des Fahrzeugs in Betracht kommen. Hierbei werden insbesondere alle Subfunktionen im Kontext des elektronischen Systems erklärt. Das Zusammenspiel zwischen diesen Subfunktionen und dem elektronischen System wird im Verlauf dieser Arbeit an vielen Stellen zur konkreten Problembeschreibung erneut aufgegriffen.

Abbildung 2.4 stellt den Zusammenhang aller Subfunktionen im Kontext des Fahrzeugzugangssystem grafisch dar. Die Rechtecke sind dabei die Subfunktionen. Die Verbindungen zwischen den Subfunktionen deuten darauf, dass ein Datenaustausch zwischen diesen stattfindet. Der Zweck dieser Abbildung ist, dem Leser eine grobe Übersicht über die im Folgenden noch zu erläuternden Funktionen zu geben.

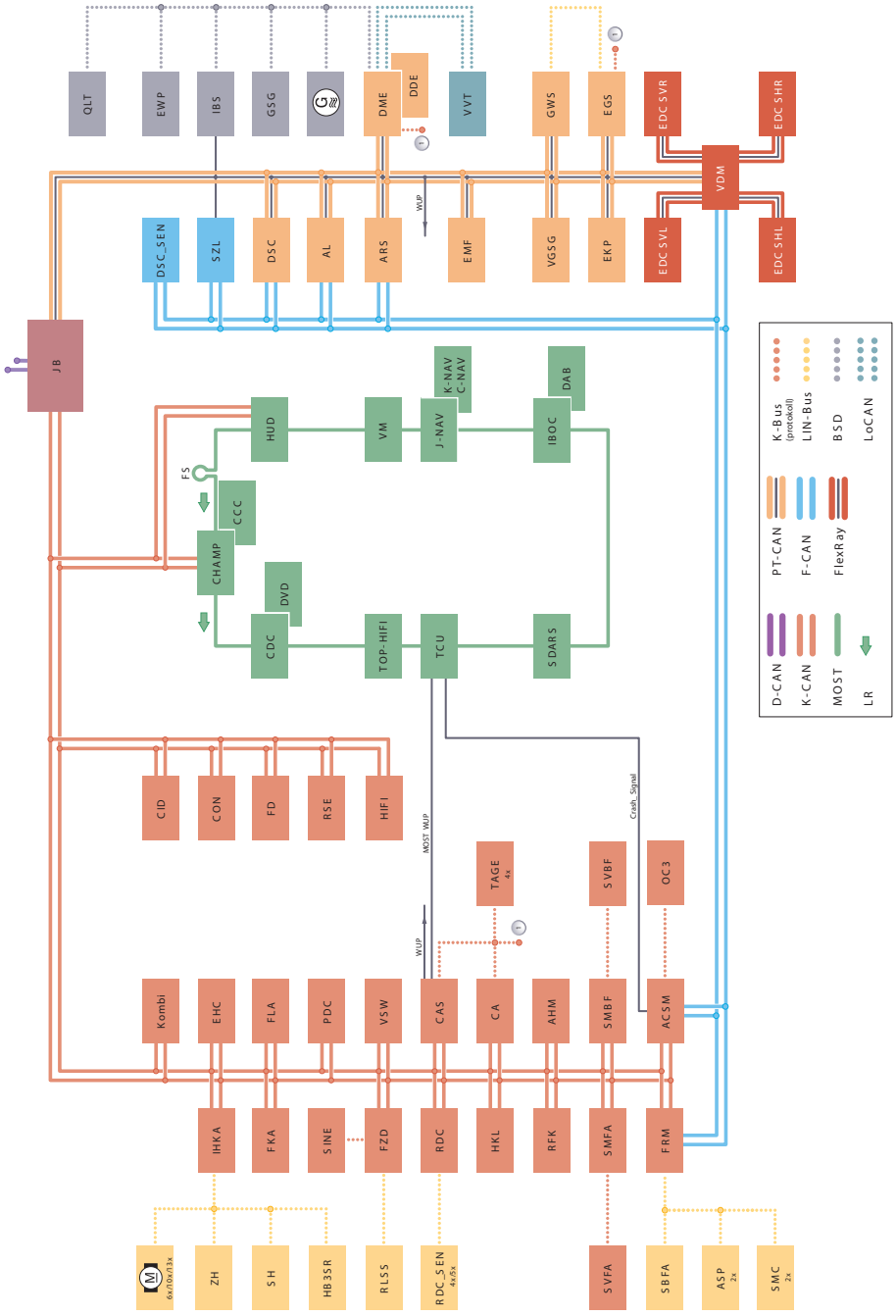


Abbildung 2.3.: Das Bussystem im BMW X5 xDrive50i und die Vernetzung der Steuergeräte (Quelle: [SAV06e])

Abkürzung	Erläuterung
ACSM	Advanced Crash Safety Management
AL	Aktive Lenkung
ARS	Active Roll Stabilization
ASP	Außenspiegel
CA	Comfort Access
CAS	Car Access System
CCC	Car Communication Computer
CDC	Compact Disk Changer
M-ASK	Multi-Audio System Kontroller
CHAMP	Central Head unit and Multimedia Platform
CID	Central Information Display
CON	Controller
DME	Digitale Motorelectronik
DSC	Dynamic Stability Control
DSC-SEN	DSC Sensor
DVD	Digital Video Disc Changer
EDC SHL	Electronic Damper Control, Satellit hinten links
EDC SHR	Electronic Damper Control, Satellit hinten rechts
EDC SVL	Electronic Damper Control, Satellit vorne links
EDC SVR	Electronic Damper Control, Satellit vorne rechts
EGS	Elektronische Getriebesteuerung
EHC	Electronic Height Control
EKP	Elektronische Kraftstoffpumpe
EMF	Elektro-mechanische Feststellbremse
FD	Fond Display
FKA	Fond Klimaautomatik
FLA	Fernlichtassistent
FRM	Fußraummodul
FZD	Funktionszentrum Dach
GWS	Gangwahlschaltung
HB3SR	Heizung/Belüftung 3. Sitzreihe
HiFi	HiFi Verstärker
HKL	Heckklappenlift
HUD	Head-Up Display
IBOC	In-Band On-Channel (HD Radio)
IBS	Intelligenter Batteriesensor
IHKA	Intergrierte Heiz- und Klimaautomatik
JB	Junction Box
Kombi	Kombinationsinstrument
OC3	Seat Occupancy Sensor
PDC	Park Distance Control
QLT	Quality, Level, Temperature Sensor für Öl-Status
RDC	Reifendruck Control
RDC-SEN	RDC-Sensor
RFK	Rückfahrkamera
RLSS	Regen-/Fahrlicht Solar Sensor
RSE	Rücksitz-Entertainment
SBFA	Schalterblock Fahrer
SDARS	Satellite Digital Audio Radio Services
SINE	Sirene und Neigungssensor

Tabelle 2.1.: Erläuterung der Abkürzungen für die Bussysteme und Steuergeräte des BMW X5 xDrive50i (Teil 1)

Abkürzung	Erläuterung
SMBF	Sitzmodul Beifahrer
SMC	Schrittmotor Controller
SMFA	Sitzmodul Fahrer
SVBF	Sitzverstellung Beifahrer
SVFA	Sitzverstellung Fahrer
SZL	Schaltzentrum Lenksäule
TAGE	Türaußengriffelektronik
TCU	Telematics Control Unit
TONS	Thermalöl Niveau Sensor
TOP-HIFI	Top HiFi Verstärker
VDM	Vertikaldynamik Management
VGSG	Verteilergetriebe Steuergerät
VVT	Variabler Ventiltrieb
BSD	Bit-Serielle Datenschnittstelle
Crash-Sig	Crash Signal
D-CAN	Diagnose CAN (Controller Area Network)
F-CAN	Fahrwerk CAN
FlexRay	FlexRay
K-Bus	Karosserie Bus
K-CAN	Karosserie CAN
LIN-Bus	Local Interconnect Network Bus
LoCAN	Local CAN
MOST	Media Oriented Systems Transport
MOST WUP	MOST Wake-Up
PT-CAN	Powertrain CAN
WUP	Wake-Up
1	CAS Bus Verbindung

Tabelle 2.2.: Erläuterung der Abkürzungen für die Bussysteme und Steuergeräte des BMW X5 xDrive50i (Teil 2)

Die Zentralverriegelung ist hierbei die Kernfunktionalität. Sie initiiert die Verriegelung/Entriegelung. Als optionale Sonderausstattung gibt es die Komfortzugangsfunktion. Sie realisiert das passive Verriegeln/Entriegeln, das heißt ohne aktive Verwendung eines mechanischen Schlüssels oder einer Funkfernbedienung. Darüber hinaus gibt es weitere Komfort- und Sicherheitsfunktionen, wie zum Beispiel das automatische Verriegeln und die Unfallerkennung. Beide Funktionen interagieren mit der Zentralverriegelung, damit sie ihre Aufgaben erfüllen können. Zusätzlich spielen sowohl die Innen- als auch die Außenbeleuchtung eine wichtige Rolle, wenn das Fahrzeug verriegelt/entriegelt wird. Schließlich wird die Elektronische Wegfahrsperre (EWS) beim Verriegeln aktiviert und beim Entriegeln deaktiviert, sodass auch hier eine Interaktion erforderlich ist.

Die folgenden Erläuterungen können unter Zuhilfenahme von Abbildung 2.3 gelesen werden. Insbesondere werden auf diese Weise das Deployment der Funktionen, die Verbindungen von Sensoren/Aktuatoren zu Steuergeräten und schließlich der Datenfluss ersichtlich. Für umfassendere Informationenen sei auf die Literatur aus

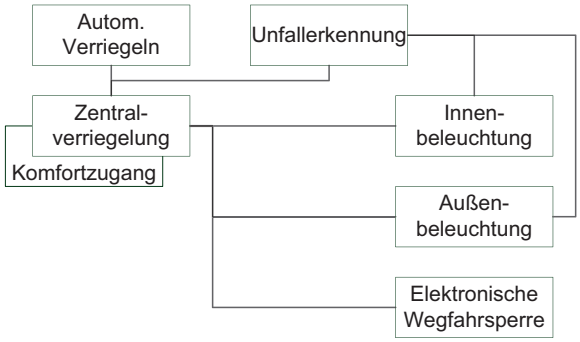


Abbildung 2.4.: Überblick der betrachteten Funktionen und ihre Beziehungen untereinander für das Fahrzeugzugangssystem

[SAV06a, SAV06b, SAV06c, SAV06f, SAV06d] verwiesen.

2.3.1. Die Zentralverriegelung

Die *Zentralverriegelung* ist für die Verriegelung bzw. Entriegelung aller Türen, der Heckklappe sowie der Tankklappe zuständig (für ausführliche Informationen vgl. [SAV06b]). Betrieben wird die Zentralverriegelung durch einen mechanischen Schlüssel über den Türschlosszylinder, eine Funkfernbedienung, die optional mit einem Funkempfänger für den Komfortzugang (siehe Abschnitt 2.3.2) ausgestattet sein kann, und einer Zentralverriegelungstaste im Innenraum des Fahrzeugs. Hierbei geht die Initiierung der Sollwertgeber stets vom Fahrer aus. Darüber hinaus wird die Zentralverriegelung auch vom Fahrzeug ausgelöst. Beispielsweise wird das Fahrzeug automatisch verriegelt, wenn es eine bestimmte Geschwindigkeit überschreitet. In diesem Fall ist der Fahrer nur indirekt beteiligt.

Für die Zentralverriegelung spielen viele (Teil-)Funktionen bei der Ausführung der Verriegelung/Entriegelung eine wichtige Rolle. Grundlegend kann allerdings gesagt werden, dass die Funktion im Steuergerät CAS deployed ist. Die Ausführung der Befehle zum Verriegeln/Entriegeln findet aber im Steuergerät JB statt. Da sich das Zusammenspiel zwischen der Funktion Zentralverriegelung und dem elektronischen System je nach verwendetem Sollwertgeber unterscheidet, werden die folgenden Erläuterungen entsprechend unterteilt.

In den meisten Fällen wird ein mechanischer Schlüssel zusätzlich zu einer Funkfernbedienung ausgeliefert. Dabei ist der mechanische Schlüssel in die Funkfernbedienung integriert, sodass er bei Bedarf aus dieser herausgenommen werden kann. Der mechanische Schlüssel ist ein Sollwertgeber, mit der die Zentralverriegelungsfunktion initiiert wird. Dazu muss der Schlüssel in den Türschlosszylinder gesteckt werden und in die gewünschte Richtung (rechts zum Verriegeln, links zum Entriegeln) gedreht werden. Das Schloss ist mit einem Hallsensor ausgestattet, der die Bewegung des Schlüssels in eine Richtung ermittelt und den Zustand an das Steuergerät Fußraummodul (FRM) weiterleitet. Zusätzlich erfasst ein zweiter Hall-

sensor den Status des Türkontakts, also ob die Fahrertüre offen oder geschlossen ist. Diese Information ist zum einen für das Verriegeln erforderlich. Denn nur bei geschlossener Fahrertüre kann die Verriegelung ausgeführt werden. Zum anderen wird diese Information benötigt, um die Innenbeleuchtung ein-/auszuschalten (vgl. Abschnitt 2.3.3). Beide Statusinformationen werden über das Karosserie-CAN (K-CAN)-Bussystem an das CAS gesendet. CAS empfängt die Nachricht und initiiert den Verriegelungs- bzw. Entriegelungswunsch. Dieser wird wiederum über K-CAN an das Steuergerät JB übermittelt. JB enthält schließlich die Relaisschalter, um die Verriegelung/Entriegelung durchzuführen. Diese sind

- Fahrertüre,
- Beifahrertüre,
- Türe hinten links,
- Türe hinten rechts,
- Heckklappe und
- Tankklappe.

Mit diesen werden die entsprechenden Aktuatoren (4 x Türen, Heckklappe, Tankklappe) gesteuert.

Weiterhin kann das Fahrzeug über die Zentralverriegelungstaste verriegelt/entriegelt werden. Sie ist ein Sollwertgeber, der im Innenraum des Fahrzeugs angebracht ist. Durch Betätigen der Zentralverriegelungstaste, wird der Fahrerwunsch direkt zur JB weitergeleitet. Die Zentralverriegelungstaste ist mit dem Steuergerät JB über eine direkte Verkabelung verbunden. Es wird also kein Bussystem verwendet, um die Verriegelung/Entriegelung auszuführen. An dieser Leitung liegt eine hohe elektrische Spannung von 12 V an, wenn die Zentralverriegelungstaste nicht gedrückt wurde. Sobald die Zentralverriegelungstaste gedrückt wird, ändert sich die hohe Spannung zu einer niedrigen von etwa 0 V. JB überprüft diesen Spannungswechsel und schaltet die Relais' für alle Fahrzeugtüren und der Heckklappe. Die Tankklappe wird hierbei nicht berücksichtigt. Durch die Schaltung der Relais' werden die entsprechenden Aktuatoren aktiviert, die das Verriegeln/Entriegeln durchführen. Damit die Funktion der Zentralverriegelung den aktuellen Schließstatus kennt, wird über K-CAN eine Nachricht an CAS gesendet.

Die Funkfernbedienung ist ein weiterer Sollwertgeber bzw. eine Bedienschnittstelle für den Fahrer zum Verriegeln/Entriegeln der Fahrzeugtüren, der Tank- und Heckklappe. Weiterhin kann mit der Funkfernbedienung die Heckklappe separat geöffnet werden. Für diese Aktionen sind drei Knöpfe auf der Funkfernbedienung enthalten (entriegeln, verriegeln, Heckklappe öffnen). Außerdem besitzt die Funkfernbedienung einen Datenspeicher von 512 MB. In diesem Speicher können folgende Informationen gehalten werden:

- Die aktuellen Kilometer des Fahrzeugs

- Die Identifikationsnummer des Fahrzeugs
- Die Identifikationsnummer der Funkfernbedienung
- Einträge bzgl. aufgetretener Fehler
- Die Version der DVD für die Navigation
- Informationen zum aktuellen Ölstand
- Batteriestatus der Funkfernbedienung

Wird der Knopf zur Entriegelung gedrückt, erreicht das Signal zunächst die Antenne im hinteren Fahrzeugbereich. Das Signal wird verstärkt und an CAS weitergesendet. Die Antenne ist dabei direkt über eine Leitung an das Steuergerät angebunden. Die Zentralverriegelungsfunktion verifiziert anhand der Identifikationsnummer der Funkfernbedienung die Gültigkeit des Signals. Wird sie als gültig bewertet, sendet CAS das Entriegelungssignal an JB weiter. JB verarbeitet das Signal entsprechend, d.h., die Relais' werden geschaltet und die entsprechenden Aktuatoren werden dadurch aktiviert.

Wird der Knopf zur Verriegelung gedrückt, überprüft CAS (sobald das Signal angekommen ist) zusätzlich noch den Status des Türkontakts der Fahrertüre. Diese wird von FRM über K-CAN bekannt gegeben. Ist die Türe geschlossen und die Identifikationsnummer der Funkfernbedienung gültig, wird das Verriegelungssignal an JB weitergeleitet. Falls die Türe offen ist, wird die Verriegelung nicht durchgeführt. Der weitere Ablauf ist analog zur Entriegelung.

Die Zentralverriegelung verfügt zusätzlich zu der konventionellen Verrieglung/Entriegelung Erweiterungen wie etwa das automatische Verriegeln/Entriegeln und das automatische Entriegeln bei Unfallerkennung. Diese werden im Folgenden genauer erläutert.

2.3.1.1. Automatisches Verriegeln und Entriegeln

Sobald das Fahrzeug die Geschwindigkeit 16 km/h überschreitet, aktiviert sich die Zentralverriegelung und verriegelt automatisch das Fahrzeug. Die Drehratensensoren an den Reifen werden von Dynamic Stability Control (DSC) empfangen, um die Fahrzeuggeschwindigkeit zu ermitteln. Diese wird über Fahrwerk-CAN (F-CAN) an Advanced Crash Safety Management (ACSM) gesendet, das wiederum das Signal über K-CAN für CAS bereitstellt. CAS initiiert schließlich die Prozedur zur Verriegelung. Die Entriegelung wird aktiviert, wenn die Funkfernbedienung aus der Nut entnommen wird, also wenn der Motor ausgestellt wird. Beim Komfortzugang muss nicht notwendigerweise die Funkfernbedienung in der Nut stecken. In diesem Fall reicht es aus, den Start-Stop-Knopf zu drücken, damit die Entriegelung ausgelöst wird.

2.3.1.2. Automatisches Entriegeln bei Unfallerkennung

Über einen sogenannten Crash-Sensor wird ein Unfall erkannt und entsprechende Maßnahmen werden durchgeführt. Der Crash-Sensor ist an das Steuergerät ACSM angebunden. Das Signal wird über K-CAN an CAS und JB gesendet. CAS blockiert die Signale der Funkfernbedienung und JB sperrt die Zentralverriegelungstaste. Gleichzeitig entriegelt JB das Fahrzeug, die Innen- und Außenbeleuchtung (Warnblinkanlage) werden eingeschaltet (vgl. Abschnitt 2.3.3 und Abschnitt 2.3.4).

2.3.2. Der Komfortzugang

Der *Komfortzugang* ist eine Sonderausstattung, die das „passive“ Verriegeln bzw. Entriegeln des Fahrzeugs ermöglicht (für ausführliche Informationen vgl. [SAV06c]). Die Funkfernbedienung wird dabei durch einen Funkempfänger erweitert. Zusätzlich werden Türaußengriffe angebracht, die über Sensoren verfügen, um eine Berührung am Griff zu erkennen. Außerdem werden die Türaußengriffe mit Sendeantennen ausgestattet. Zusätzlich werden im Innenraum des Fahrzeugs Antennen angebracht, um diesen Bereich ausreichend abzudecken.

Befindet sich die Funkfernbedienung innerhalb eines Radius von zwei Metern vom Fahrzeug und wird der Türgriff umfasst, so wird eine Authentifizierung zwischen Funkfernbedienung und Fahrzeug initiiert. Ist die Authentifizierung erfolgreich, wird das Fahrzeug entriegelt. Die Antennen im Innenraum des Fahrzeugs erkennen, wenn sich die Funkfernbedienung im Innenraum des Fahrzeugs befindet. In diesem Fall führt das Betätigen des Start-Stop-Knopfs bei erfolgreicher Authentifizierung zum Starten des Motors. Das heißt, dass die Funkfernbedienung nicht notwendigerweise im Zündschloss stecken muss. Beim Verlassen des Fahrzeugs müssen alle Türen geschlossen sein, der Fahrer muss eine der Türaußengriffe berühren und die Authentifikation über die äußeren Antennen muss erfolgreich abgeschlossen sein.

Für die Entriegelung muss der Fahrer die Funkfernbedienung bei sich haben und die Griffmulde umfassen. Diese ist mit einem kapazitiven Drucksensor (kapazitiver Sensor 1) ausgestattet, der bei Berührung einen Impuls auslöst, sodass Türaußengriffelektronik (TAGE) und Comfort Access (CA) aktiviert werden. TAGE steuert die Sendeantennen und bewirkt, dass diese ein Signal im Frequenzbereich 125 kHz an die Funkfernbedienung aussenden. Da die Funkfernbedienung mit einem Empfänger ausgestattet ist, kann sie dieses Signal entsprechend empfangen. Das Signal dient als Authentifikationsanfrage. Die Funkfernbedienung sendet daraufhin die Identifikationsnummer in einem Frequenzbereich von 868 MHz, die über die äußere Empfangsantenne des Fahrzeugs empfangen wird. Diese leitet das Signal weiter an CAS, in der die Überprüfung stattfindet und die Entriegelung initiiert wird. Der restliche Ablauf ist analog zu den Erläuterungen aus Abschnitt 2.3.1.

Wenn das Fahrzeug verriegelt wird, müssen zunächst alle Türen geschlossen sein und eine sensitive Fläche am Türaußengriff berührt werden. Diese Fläche ist mit einem weiteren kapazitiven Drucksensor (kapazitiver Sensor 2) ausgestattet, sodass bei Berührung ein Impuls ausgelöst wird, der von TAGE empfangen wird. TAGE sendet den Verriegelungswunsch über CAS-Bus an CA weiter. Genau wie bei der

Entriegelung senden die Sendeantennen von TAGE eine Authentifikationsanfrage im Frequenzbereich von 125 kHz. Die Funkfernbedienung sendet nach Empfang der Anfrage ein Signal mit der Identifikationsnummer im Frequenzbereich von 868 MHz, die von der äußeren Antenne empfangen wird und weiter verarbeitet wird. Das Signal wird weiter an CAS gesendet, das bei gültiger Authentifizierung die Verriegelung initiiert.

2.3.3. Die Innenbeleuchtung

Die Funktion zur *Innenbeleuchtung* ist für das Ein- und Ausschalten der Beleuchtung im Fahrzeuginneren zuständig (für ausführliche Informationen vgl. [SAV06f]). Sie umfasst den Dachbereich, den Kofferraum, den Fußraum und die Türbereiche. Die Beleuchtung kann entweder manuell ein-/ausgeschaltet werden oder sie wird automatisch abhängig von bestimmten Ereignissen ein-/ausgeschaltet. Diese treten ein, wenn zum Beispiel die Zentralverriegelung ausgeführt wird oder ein Unfall stattgefunden hat. Wenn das Fahrzeug durch einen mechanischen Schlüssel oder durch eine Funkfernbedienung entriegelt wurde und eine Tür geöffnet wird, dann schaltet sich die Innenbeleuchtung im Dachbereich sowie Tür- und Fußraum der geöffneten Tür ein. Wird die Tür geschlossen, dann schaltet sich die Beleuchtung wieder aus. Bei offener Tür schaltet sich die Beleuchtung spätestens nach 20 sec aus. Außerdem schaltet sich die Innenbeleuchtung im Falle eines Unfalls permanent ein. Das Innenbeleuchtungssystem ist in zwei Varianten verfügbar, (1) die Standardausstattung oder (2) die optionale Premiumausstattung. In der Premiumausstattung sind u.a. die Exit-Beleuchtung, Make-Up-Beleuchtung, Handschuhfachbeleuchtung etc. vorhanden.

Die Innenbeleuchtung für den Türbereich, Fußraum und Kofferraum wird von FRM gesteuert. Der Dachbereich hingegen wird von Funktionszentrum Dach (FZD) geregelt. Wird das Fahrzeug durch einen mechanischen Schlüssel entriegelt, wird FRM als Erstes informiert, da an diesem die Hallsensoren für den Türschlosszylinder und Türkontakt angebunden sind. Wenn das Schloss entriegelt und die Türe geöffnet wurde, aktiviert die Innenbeleuchtungsfunktion den Fußraum und den Türbereich der geöffneten Türe und sendet zusätzlich über K-CAN ein Signal an FZD den Dachbereich zu beleuchten. Sobald FZD dieses Signal empfängt, führt es die Aktion aus. Wird entweder die Türe geschlossen oder es vergeht eine Zeit von 20 sec, schaltet die Innenbeleuchtungsfunktion in FRM und FZD die Beleuchtung aus.

Wird das Fahrzeug mit der Funkfernbedienung entriegelt, erreicht das Signal als Erstes das Steuergerät CAS. CAS sendet das Signal über K-CAN an JB. Da FRM am gleichen Bussystem befestigt ist, empfängt es ebenfalls das Signal. Wenn die Türe geöffnet wird, führt FRM die Innenbeleuchtungsfunktion aus.

Bei einem Unfall hingegen sendet ACSM das Crash-Signal über K-CAN. FRM und FZD empfangen das Signal und schalten die Innenbeleuchtung permanent ein.

2.3.4. Die Außenbeleuchtung

Im Zusammenhang mit der Zentralverriegelung steuert die *Außenbeleuchtungsfunktion* lediglich die Blinkerleuchten (für ausführliche Informationen vgl. [SAV06d]). Diese dienen dem Fahrer als visuelles Feedback, dass der Verriegelungs- / Entriegelungswunsch ausgeführt wurde. Ein weiteres Feature ist das Einschalten der Warnblinkanlage bei einem Unfall. Die Außenbeleuchtung verfügt darüber hinaus über weitere Funktionalitäten, wie beispielsweise das Tagfahrlicht, Willkommensleuchten und Heimleuchten. Diese spielen allerdings im Rahmen der Erläuterungen zur Zentralverriegelung keine wesentliche Rolle.

Die Außenbeleuchtung wird vollständig vom Steuergerät FRM geregelt. Bei Verwendung des mechanischen Schlüssels zur Verriegelung/Entriegelung kann FRM die Außenbeleuchtung aktivieren, sobald der Status des Hallsensors am Türschlosszylinder eine Änderung aufweist. Hierbei werden die Blinkerleuchten einmal aufgeleuchtet.

Die Signale zur Verriegelung/Entriegelung durch die Funkfernbedienung werden von FRM über K-CAN empfangen, wenn CAS diese aussendet. Bei Erhalt wird die Außenbeleuchtung aktiviert.

Bei einem Unfall wird das Crash-Signal über K-CAN empfangen. In diesem Fall wird von der Außenbeleuchtungsfunktion in FRM die Warnblinkanlage eingeschaltet.

2.3.5. Die elektronische Wegfahrsperre

Die EWS wird in Verbindung mit der Zentralverriegelung ausgeführt (für ausführliche Informationen vgl. [SAV06a]). Sie verhindert das unautorisierte Starten des Motors. Wenn die Entriegelung aktiviert wurde, zum Beispiel über die Taste auf der Funkfernbedienung oder über den Komfortzugang, wird eine Authentifizierung mit der Motor- und Getriebesteuerung durchgeführt. Ist die Authentifizierung erfolgreich, wird EWS für Motor und Getriebe abgeschaltet.

EWS ist in das Steuergerät CAS deployed. Zur Authentifikation wird ein Verschlüsselungsprotokoll verwendet. Dazu wird ein 128-bit geheimer Schlüssel in CAS und Digitale Motorelektronik (DME) und ein weiterer 128-bit geheimer Schlüssel in CAS und Elektronische Getriebesteuerung (EGS) gespeichert. Damit verwaltet CAS zwei geheime Schlüssel und DME und EGS verwalten jeweils einen. Wird eines dieser Steuergeräte funktionsunfähig, müssen somit alle drei Steuergeräte ausgetauscht werden. Die Kommunikation zwischen den Steuergeräten erfolgt zum einen über K-CAN über JB und Powertrain-CAN (PT-CAN), zum anderen über CAS-Bus als redundanter Kommunikationspfad. Es gibt zwei Phasen der Authentifikation: (1) Authentifizieren beim Entriegeln und (2) Authentifizieren beim Starten des Motors. In der ersten Authentifikation werden die Getriebefunktionen freigeschaltet und in der zweiten Authetifikation die Wegfahrsperre für den Motor.

Sobald der Entriegelungswunsch CAS erreicht, beginnt der Datenaustausch mit EGS, um das Getriebe freizuschalten. Dazu wird ein sogenanntes Challenge-Response Verfahren eingesetzt. Dieser verläuft folgendermaßen ab:

- EGS sendet über CAS-Bus und PT-CAN eine Zufallszahl an CAS. Diese Zufallszahl wird durch einen Zufallszahlengenerator erzeugt.
- CAS empfängt diese Zufallszahl und berechnet aus dieser und dem gespeicherten geheimen Schlüssel eine Antwort für EGS. Die Antwort wird über CAS-Bus und K-CAN an EGS gesendet.
- Gleichzeitig berechnet EGS die erwartete Antwort mit seinem geheimen Schlüssel.
- CAS und EGS verwenden somit die gleiche Zufallszahl und den gleichen Algorithmus zur Berechnung der Antwort.
- Wenn die Antwort von CAS mit der erwarteten Antwort übereinstimmt, schaltet EGS die Wegfahrsperre ab.

Die Wegfahrsperre für den Motor wird abgeschaltet, wenn das Fahrzeug gestartet wird. Dabei muss zunächst die Funkfernbedienung anhand der Identifikationsnummer identifiziert werden - entweder über das Zylinderschloss oder über die inneren Antennen im Fahrzeug bei einem Komfortzugang. CAS verifiziert die Identifikationsnummer und startet das Challenge-Response-Verfahren mit DME. Wenn das Protokoll erfolgreich beendet wird, schaltet DME die Zündung und Einspritzung frei, sodass der Motor gestartet werden kann.

2.4. Das Szenario

In den vergangenen Abschnitten wurde das betrachte Fahrzeug vorgestellt, das zugrunde liegende elektronische System erläutert und zwei Funktionen, das Fahrzeugzugangssystem und die Anhaltewegverkürzung, mit ihren Subfunktionen genauer beschrieben. In diesem Abschnitt wird in einem Szenario, unter Berücksichtigung der drei Komponenten Fahrer, Fahrzeug und Umwelt, die Anwendung der Funktionen vorgestellt. Das betrachtete Szenario umfasst die folgenden Aktivitäten:

1. Der Fahrer entriegelt das Fahrzeug.
2. Der Fahrer startet den Motor und fährt los.
3. Der Fahrer stoppt den Motor und steigt aus.
4. Der Fahrer verriegelt das Fahrzeug.

Das Entriegeln des Fahrzeugs scheint im ersten Moment eine recht einfache Aktivität zu sein. Bei genauerem Betrachten wird erkennbar, dass die Komplexität größer ist als erwartet. Der Fahrer hat mehrere Möglichkeiten, das Fahrzeug von außen zu entriegeln. Dazu kann er den mechanischen Schlüssel oder die Funkfernbedienung verwenden. Darüber hinaus kann er das Fahrzeug über den optionalen Komfortzugang entriegeln. Es wurde anhand des elektronischen Systems gezeigt, dass die

Kommunikationspfade dieser drei Möglichkeiten unterschiedlich sind, bis sie die Zentralverriegelungsfunktion erreichen. Dies hat zur Folge, dass Subfunktionen in allen Steuergeräten deployed werden müssen, die den Entriegelungswunsch erkennen und an die Zentralverriegelung weiterleiten. Typischerweise wird die Erkennung durch die Verwendung verschiedener Sensoren realisiert. Zudem muss bei der Entwicklung berücksichtigt werden, dass in Zukunft durchaus weitere Möglichkeiten zur Entriegelung eingeführt werden können. Zum Beispiel wäre es vorstellbar, das Fahrzeug mit einem Smartphone über Bluetooth oder UMTS zu entriegeln. Zusätzlich werden weitere Funktionen aktiviert. So werden die Innen- und Außenbeleuchtungen eingeschaltet und die Getriebefunktionalität freigeschaltet. Der Entriegelungsvorgang ist also deutlich komplexer, als es den Anschein hat. Es muss die Vielfalt eingesetzter Features und Interaktionen bekannt sein.

Zum Starten des Motors gibt es wiederum mehrere Möglichkeiten. Zum einen kann die Funkfernbedienung in die Nut gesteckt werden oder im Fall des Komfortzugangs über die inneren Antennen des Fahrzeugs identifiziert werden. In beiden Fällen wird die Identifikationsnummer der Funkfernbedienung verifiziert, um die Wegfahrsperre für den Motor zu deaktivieren. Die Kommunikation verläuft in beiden Fällen über verschiedene Sensoren und Steuergeräte (vgl. Abschnitt 2.3). Hier ist erneut die Handhabung vielfältiger Features und Interaktionen zu berücksichtigen.

Beim Fahren bewirkt die Funktion zur automatischen Verriegelung, dass das Fahrzeug bei Überschreiten einer bestimmten Geschwindigkeit verriegelt wird. Die Fahrzeuggeschwindigkeit wird durch entsprechende Sensoren ermittelt. Somit empfängt die Zentralverriegelungsfunktion über einen weiteren Kommunikationspfad den Befehl zum Verriegeln/Entriegeln. Wenn ein Unfall nicht mehr vermieden werden kann, reagiert die Unfallerkennung, indem es das Fahrzeug entriegelt, die Warnblickanlage einschaltet und die Zentralverrieglungstaste sowie die Funkfernbedienung in ihren Funktionen blockiert.

Erreicht der Fahrer sein Ziel ohne Unfall, beendet er die Fahrt, indem der Motor abgeschaltet wird. Entweder entnimmt der Fahrer die Funkfernbedienung aus dem Zündschloss oder sie ist bereits in der Tasche des Fahrers, da er den Komfortzugang genutzt hat. In beiden Fällen muss das Fahrzeug in der Lage sein die Wegfahrsperre für den Motor erneut einzuschalten.

Nachdem der Fahrer ausgestiegen ist und die Türe geschlossen hat, verriegelt er das Fahrzeug. Dazu kann er erneut entweder den mechanischen Schlüssel, die Funkfernbedienung oder den Komfortzugang nutzen. Ähnlich zum Entriegeln erfolgt auch hier die Kommunikation im elektronischen System über verschiedene Wege. Zusätzlich wird die Außenbeleuchtung als visuelles Feedback eingeschaltet und die Getriebefunktionalität deaktiviert.

2.5. Zusammenfassung

In diesem Kapitel wurde ein Beispiel vorgestellt, das im Verlauf dieser Arbeit zur Beschreibung der Problemstellungen und Lösungskonzepte ihren Zweck erfüllen wird. Insbesondere wurde das betrachtete Fahrzeug, der BMW X5 xDrive50i, samt

seiner Ausstattungen präsentiert und das zugrundeliegende elektronische System beschrieben. Bereits hier wird erkennbar mit welcher Komplexität die Softwareentwicklung verbunden ist. Anschließend wurde das Fahrzeugzugangssystem im Detail beschrieben. Bei den Erläuterungen für das Fahrzeugzugangssystem wurden insbesondere auch die Datenflüsse basierend auf dem elektronischen System ausführlich erklärt. Abschließend wurde in einem Szenario die Anwendung der Funktionen anhand der drei Entitäten Fahrer, Fahrzeug und Umwwelt beschrieben.

Teil II.

Prozesse, Variabilität und Variabilitätsmodell

Kapitel 3.

Der Referenzprozess

Produktlebenszyklen von Fahrzeugen sind durch lange Laufzeiten charakterisiert. So erstreckt sich die Entwicklung eines Fahrzeugs auf knapp drei Jahre, die Produktion auf sieben Jahre nach der Entwicklung und schließlich die Betriebs- und Wartungsphase auf weitere 10 - 15 Jahre [SZ06], die parallel zur Produktion verläuft. Insgesamt ist der Produktlebenszyklus eines Automobils also nahezu 20 - 25 Jahre. Für die Softwareentwicklung bedeutet dies, dass die entstehenden Softwaredokumente möglichst stabil für Hardwareveränderungen sein sollten, sodass Softwareportierungen ohne großen Aufwand durchgeführt werden können. Außerdem müssen Softwareupdates ohne umfangreiche Austauschvorgänge ermöglicht werden [Bal01]. Automobilhersteller haben hierfür Softwareentwicklungsprozesse in den Gesamtprozess integriert, die mehr oder weniger den genannten Herausforderungen gewachsen sind und sich über die Jahre hinweg etabliert haben. Dabei unterscheiden sich Softwareentwicklungsprozesse verschiedener Automobilhersteller an den ein oder anderen Stellen in geringfügigem Maße. In dieser Arbeit wird daher basierend auf den Erfahrungen mit verschiedenen Automobilherstellern ein *Referenzprozess* vorgestellt, der die wesentlichen Prozessschritte beinhaltet, die in allen Softwareentwicklungsprozessen der verschiedenen Hersteller enthalten sind. Insbesondere wird hierbei die frühe Phase der Entwicklung betrachtet. So sind die Test- und Integrationsphase sowie die Betriebs- und Wartungsphase nicht Bestandteil des betrachteten Referenzprozesses.

Abbildung 3.1 illustriert den Referenzprozess als diskretes Phasenmodell. Die Rechtecke stellen dabei *Aktivitäten* dar, die Lochstreifen sind die *Ergebnisse* bzw. die aus der Aktivität entstehenden Softwaredokumente. Die Semantik der Pfeile sind die folgenden: (1) Aktivität -> Ergebniss: „ist Ergebnis von“ und (2) Ergebnis -> Aktivität: „wird benötigt für“ [Nag90].

Im linken Teil der Abbildung ist außerdem eine Unterteilung der Aktivitäten mit ihren Ergebnissen in verschiedene Abstraktionsebenen vorgenommen. Die Abstraktionsebenen umfassen die *Featureebene*, die *Funktionsebene*, die *Architekturebene* und die *Codeebene*. Diese werden nachfolgend im Einzelnen beschrieben.

3.1. Featureebene

Die *Featurebene* umfasst die Aktivität *Problemanalyse* mit dem Ergebnis der *Anforderungsdefinition*. Neben einer Istanalyse wird das Sollkonzept mit der Beschreibung

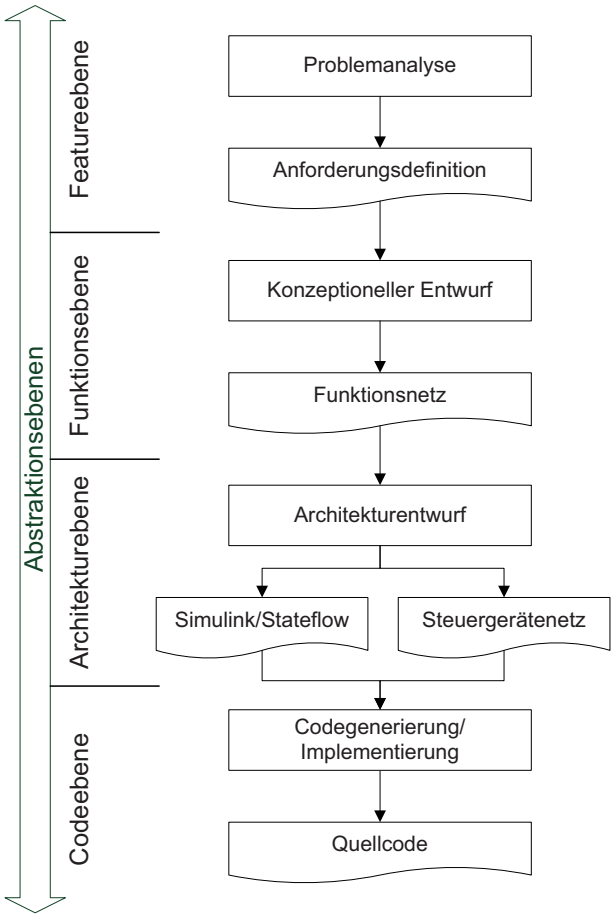


Abbildung 3.1.: Der Referenzprozess als diskretes Phasenmodell mit der Einteilung in verschiedene Abstraktionsebenen

seiner Funktionen sowie eine Durchführbarkeitsstudie mit Risikoabschätzung aufgestellt [Nag90]. Diese werden in Form einer Anforderungsdefinition notiert. In der Automobilindustrie hat sich als Werkzeug für das Anforderungsmanagement IBM Rational DOORS etabliert [Webb]. Typischerweise wird DOORS verwendet, um die Anforderungsdefinition in textueller Form zu spezifizieren. In dieser Arbeit wird die Featureebene nicht im Detail betrachtet. Insbesondere werden die im Verlauf dieser Arbeit vorgestellten Konzepte nicht auf diese Ebene angewendet. Die Beschreibung dieser Ebene dient primär zur besseren Verständlichkeit der darauffolgenden Ebenen.

3.2. Funktionsebene

Die *Funktionsebene* beinhaltet die Aktivität *Konzeptioneller Entwurf* und liefert das Ergebnis in Form eines *Funktionsnetzes*. In dieser Phase werden die Funktionen auf einer konzeptionellen Ebene entworfen. Bei der Realisierung werden die funktionalen Anforderungen konkretisiert, aber es wird dennoch weitesgehend von Detailinformationen abstrahiert [WH06]. Das Ergebnis ist eine erste virtuelle und logische Realisierung der fahrzeugweiten Funktionen in Form eines grafischen Funktionsnetzes.

Die Vorteile eines derartigen Funktionsnetzes ist die intuitive und leicht verständliche Darstellung, die als Kommunikationsgrundlage für Entwickler zur Diskussion, Dokumentation, Simulation und Validierung dient. Auf diese Weise können Spezifikationsfehler in den Anforderungen frühzeitig erkannt werden.

In der Industrie wird allerdings der konzeptionelle Entwurf unterschiedlich gehandhabt. Während einige Automobilhersteller den Entwurf eines Funktionsnetzes auf einer hohen Abstraktionsebene durchführen, also nah an den wahrnehmbaren Features, realisieren andere Hersteller das Funktionsnetz auf einem sehr niedrigen Abstraktionsniveau, also nah an der Hardware. Beide Methoden haben ihre Vor- und Nachteile. In dieser Arbeit wird in Kapitel 5 ein Ansatz vorgestellt, der mehrere Abstraktionsniveaus abdeckt und somit das Verständnis von Funktionsnetzen für unterschiedliche Automobilhersteller erfasst.

3.3. Architekturebene

Die *Architekturebene* enthält die Aktivität *Architekturentwurf* mit den Ergebnissen *Simulink/Stateflow* und *Steuergerätenetz*. In dieser Phase wird das Funktionsnetz als Basis verwendet, dieses in einem Partitionierungsschritt in Software- und Hardwareanteile aufgeteilt und hierfür jeweils die Software- und Hardwarearchitektur gebildet. Die Hardwarearchitektur beschreibt das elektronische System mit der Spezifikation der Steuergeräte, Sensoren, Aktuatoren sowie Bustechnologien mit konkreten Leitungssätzen und Stromlaufplänen. Die Hardwarearchitektur wird in dieser Arbeit nicht weiter im Detail betrachtet. Der Fokus dieser Arbeit liegt vielmehr softwareseitig. Die Softwarearchitektur wird oftmals durch die visuelle Spezifikationssprache Simulink realisiert [Webc]. Simulink ist eine datenflussorientierte Sprache, mit der sowohl kontinuierliche als auch diskrete Systeme modelliert werden können. Die wesentlichen Konzepte in Simulink sind Blöcke und Verbindungen zwischen Blöcken. Jeder Block hat eine bestimmte Semantik mit einem zugrunde liegenden Berechnungsmodell, das eine Eingabe in eine Ausgabe transformiert. Ausnahme hierbei sind Quellblöcke, die keine Eingabe haben, und Zielblöcke, die keine Ausgabe haben (aber dennoch ein Berechnungsmodell). Ein besonderer Vorteil von Simulink ist die Möglichkeit, die modellierten Funktionen zu simulieren, um frühzeitig das Verhalten zu testen. Hierbei wird typischerweise von der Verfügbarkeit unendlicher Ressourcen ausgegangen. Zudem abstrahiert die Simulation von der Hardwarearchitektur, sodass zum Beispiel Speicher eines Steuergeräts oder

Bustopologien nicht berücksichtigt werden. Die Trennung zwischen Software- und Hardwarearchitektur muss in einem weiteren Deploymentschritt wieder zusammengeführt werden. Hier wird also entschieden welche Softwarekomponenten auf welchen Steuereinheiten integriert werden sollen. Ist dieser Schritt durchgeführt, kann im Anschluss der Quellcode für den spezifischen Hardwarebaustein generiert oder implementiert werden.

3.4. Codeebene

Die *Codeebene* umfasst die Aktivität *Codegenerierung/Implementierung* mit dem Ergebnis *Quellcode*. Diese Phase beinhaltet entweder die Generierung von Quellcode anhand der realisierten Simulink-Modelle oder die manuelle Implementierung der spezifizierten Softwarekomponenten. Typischerweise wird aus den Simulink-Modellen der Quellcode für das dedizierte Steuergerät generiert. Insbesondere wird dabei das Modell (oder auch die manuelle Implementierung) durch Echtzeiteigenschaftem angereichert, Datenstrukturen und Datentypen optimiert und an die Schnittstellen der Steuergeräte angepasst. Der übersetzte Quellcode ist schließlich auf dem Steuergerät ausführbar.

3.5. Zusammenfassung

In diesem Kapitel wurde der dieser Arbeit zugrunde liegende Referenzprozess vorgestellt. Der Prozess umfasst vier Abstraktionsebenen: (1) Featureebene, (2) Funktionsebene, (3) Architekturebene und (4) Codeebene. Insbesondere werden die drei letztgenannten Ebenen in dieser Arbeit im Detail behandelt. Zusätzlich werden Ansätze vorgestellt, die Variabilität im Prozess berücksichtigen.

Kapitel 4.

Variabilität: Modellierung und Bindung

4.1. Einleitung und Motivation

Softwareentwicklungsprozesse im Automobilbau sind durch einen hohen Komplexitätsgrad gekennzeichnet, für die es vielerlei Ursachen gibt. Schon das Geschäftsmodell führt zu einem enormen Verwaltungsaufwand. So müssen Automobilhersteller mit verschiedenen Zulieferern kooperieren. Die Zulieferer selbst haben wiederum eigene Zulieferer, etc. Die Kommunikationskette wird auf diese Weise sehr lang und führt auch bei kleinen Fragestellungen oder Problemen zu langwierigen Vorgängen. Ein weiterer Grund für die erhebliche Komplexität ist der Bedarf nach Maßnahmen zur Absicherung sicherheitskritischer Systeme. Diese müssen durch Simulations-, Verifikations-, Validierungs- und Testmethoden über mehrere Bewertungsrunden überprüft werden, damit sie zur Integration freigegeben werden können. In der Integrationsphase kommen schließlich weitere Überprüfungen hinzu. Insbesondere erschweren proprietäre Lösungen der Zulieferer die Integration verschiedener Bausteine in das elektronische System und stellen somit eine weitere Komplexitätsquelle dar. Ein für diese Arbeit wesentlicher Auslöser der Komplexitätssteigerung ist die *Entwicklung variantenreicher Softwaresysteme*. So muss die Vielfalt an Kundenbedürfnissen, die sich in Software niederschlagen, im gesamten Entwicklungsprozess geeignet berücksichtigt werden. Dieses Kapitel beschäftigt sich genau mit diesem Anspruch.

In [Sch05] wird beschrieben, dass in den Jahren 2004 und 2005 insgesamt 1,1 Millionen Mercedes-Benz A-Klasse Fahrzeuge produziert wurden, von denen nur zwei identisch waren. Auch wenn dieses Ergebnis nicht allein aufgrund vielfältiger Software entstanden ist, sondern primär durch Interieur- und Exterieurausstattungen, wie Farb-, Sitz- oder Lenkradvarianten, so trägt sie dennoch zu dieser mannigfaltigen Angebotspalette bei. Zudem wurde in [Sch05] festgehalten, dass diese Diversität eher ein typisch deutsches Problem ist (französische Automobilhersteller haben 70% und japanische Automobilhersteller haben sogar 90% weniger Varianten als deutsche Automobilhersteller). Der wesentliche Faktor ist aber, dass der hohe Grad an Variationen knapp *20% der gesamten Entwicklungs- und Produktionskosten* ausmachen. Softwarevielfalt wird vermutlich zu einem nicht zu vernachlässigenden Anteil zu diesen Kosten beitragen.

Es stellt sich nun die Frage, wieso insbesondere deutsche Automobilhersteller ein derartig vielschichtiges Ausstattungsspektrum anbieten, obwohl Konkurrenten deutlich sparsamer sind. Die Antwort auf diese Frage wird in einer Studie aus [AG09] gegeben. Hier wurde ermittelt, dass Produktvielfalt nicht immer für Gewinne, aber für *Kundenzufriedenheit* und *größere Marktanteile* sorgt. Neben diesen seitens der Automobilhersteller erwünschten positiven Aspekten, treiben wiederum andere Faktoren, wie zum Beispiel *gesetzliche Vorgaben* und *Produktdifferenzierung*, zwangsläufig die Vielfalt in die Höhe [GBRW11].

In Kapitel 2 wurde ein Beispiel mit der Beschreibung des Fahrzeugs BMW X5 xDrive50i eingeführt. Die Sonderausstattungen dieses Fahrzeugmodells werden im Folgenden genauer untersucht, um einen Eindruck der angebotenen Vielfalt zu bekommen. Tabelle 4.1 listet die Sonderausstattungen für den BMW X5 xDrive50i auf. Diese können von einem Kunden zusätzlich zur Grundausstattung des Fahrzeugs hinzugefügt werden. Beispielsweise sind der *Fernlichtassistent*, der *Komfortzugang* und die *Soft-Close-Automatik* Features, die dem Bereich Komfort/Nutzen zugeordnet werden können. Die *Spurverlassenswarnung* und das *adaptive Kurvenlicht* sind weitere Features, die zur Fahrsicherheit beitragen. Wäre jedes einzelne Feature unabhängig voneinander selektierbar, so würden sich für die Liste der Sonderausstattungen aus Tabelle 4.1 insgesamt

$$2^{38} = 274.877.906.944$$

mögliche Fahrzeugkombinationen ergeben. Diese enorme Zahl resultiert allein aus den *für Kunden wahrnehmbaren Features*.

Im Entwicklungsprozess vervielfacht sich diese Menge aufgrund weiterer Variationsquellen. Diese betreffen primär Realisierungsentscheidungen einzelner Features. Abbildung 4.1 illustriert diesen Sachverhalt. Hier ist der Referenzprozess aus Kapitel 3 dargestellt. Im Hintergrund dieses Prozesses ist angelehnt an [PBvdL05] eine Pyramide dargestellt, die zwischen (1) *externer Vielfalt* und (2) *interner Vielfalt* unterscheidet. Die externe Vielfalt entspricht den bereits oben beschriebenen wahrnehmbaren Features, wie zum Beispiel der Komfortzugang oder das adaptive Kurvenlicht. Im Referenzprozess schlägt sich der Großteil der externen Vielfalt auf der Featureebene, d.h. in der Problemanalysephase, nieder. Im Verlauf des Entwicklungsprozesses nimmt sie ab und wird zunehmend von der internen Vielfalt abgelöst. Der Unterschied der internen Vielfalt zur externen ist, dass sie nicht von einem Kunden wahrnehmbar ist, sondern spezifische Aspekte der Realisierung beinhaltet. Die Aktivitäten und Ergebnisse der Funktions-, Architektur- und Codeebene sind daher primär durch interne Vielfalt gekennzeichnet. Zu der extrem hohen Vielfalt, die auf Featureebene entsteht, kommt also noch *Realisierungsvielfalt* hinzu.

Ein Beispiel soll dies verdeutlichen. Der Komfortzugang ist eine Sonderausstattung, die als zusätzliche Option zur Zentralverriegelung hinzugefügt werden kann. Angenommen der Komfortzugang wird durch zehn Funktionen auf Funktionsebene realisiert. Zum Deployment dieser Funktionen stehen drei Steuergeräte zur Verfügung. Wenn nun die Funktionen beliebig auf die Steuergeräte verteilt werden können, so ergeben sich

$$(2^{10})^3 = 1.073.741.824$$

Komfort/Nutzen
Fernlichtsassistent
Komfortzugang
Regensensor und automatische Fahrlichtsteuerung
Rückfahrkamera mit Top View
Soft-Close-Automatik für Türen
Panorama Glasdach
Adaptive Drive
Alarmanlage
Klimakomfort-Frontscheibe
Aktivlenkung
Aktive Geschwindigkeitsregelung mit Stop&Go Funktion
Geschwindigkeitsregelung mit Bremsfunktion
Polsterung, Sitze
Aktive Sitzbelüftung vorne
Sitzheizung für Fahrer und Beifahrer
Sitzheizung für Fondsitze
Komfortsitze vorne, elektrisch verstellbar
Sitzverstellung, elektrisch mit Memory für Fahrersitz
Radio, Audio, Kommunikation, Info
Apps
BMW Assist
BMW Head-Up Display
BMW Online
Internet
Navigationssystem Professional mit integrierter Handyvorbereitung Bluetooth
Speed Limit Info
Spracheingabesystem
DAB Tuner
BMW Individual High End Audiosystem
HiFi Lautsprechersystem
HiFi System Professional
Navigationssystem Professional
Handyvorbereitung Business mit Bluetooth Schnittstelle
TV-Funktion
Sicherheit
Adaptives Kurvenlicht
Außenspiegel automatisch abblendend
Innenspiegel automatisch abblendend
Side View
Spurverlassenswarnung
Park Distance Control

Tabelle 4.1.: Ein Ausschnitt aus möglichen Sonderausstattungen für den BMW X5 xDrive50i (ermittelt aus [Weba])

Möglichkeiten. Diese enorme Zahl entsteht alleine aus der Tatsache, dass es für die zehn Funktionen verschiedene Deploymentmöglichkeiten auf drei Steuergeräte existieren. Darüber hinaus gibt es noch weitere vielfältige Realisierungsentscheidungen, die beispielsweise Simulink-Modelle (zum Beispiel Signalflüsse) oder die Implementierung (zum Beispiel Datentypen) betreffen können.

Automobilhersteller müssen daher bereits jetzt verschiedene Maßnahmen treffen, diese Diversität auf Realisierungsebene zu beherrschen. Insbesondere ist die Erfassung von Abhängigkeiten zwischen verschiedenen Abstraktionsebenen angesichts dieser Vielfaltsexplosion für Automobilhersteller eine herausfordernde Aufgabe. Bei den oben dargestellten Zahlen scheint eine angemessene Lösung nahezu unmöglich zu sein. Dies ist allerdings nicht die ganze Wahrheit.

Die obigen zwei Berechnungen sind nämlich pessimistische Zählungen, da angenommen wurde, dass zum einen alle einzelnen Features unabhängig voneinander selektierbar und zum anderen die realisierenden Funktionen beliebig auf Steuergeräte verteilbar sind. Praktisch werden diese Möglichkeiten so nie auftreten. Für beide Fälle wird dies im Folgenden erläutert.

Typischerweise stehen Features in Beziehung zueinander, sodass bestimmte Featureselektionen die Auswahl weiterer Features bewirken. Derartige Beziehungen werden auch oft als *Restriktionen* oder *Einschränkungen* (engl. *Constraints*) bezeichnet. Auf diese Weise reduziert sich der Raum möglicher Kombinationen drastisch. Auch für die Sonderausstattungen aus Tabelle 4.1 gilt diese Aussage. Beispielsweise bewirkt die Auswahl der *Rückfahrkamera mit Top View* das Hinzufügen der Features *Park Distance Control* und *Außenspiegel automatisch abblendend*. Die *Aktive Geschwindigkeitsregelung mit Stop&Go-Funktion* und *Geschwindigkeitsregelung mit Bremsfunktion* stehen im gegenseitigen Ausschluss zueinander. Das heißt, dass beide Features nie gleichzeitig selektiert werden können. Die Auswahl der *aktiven Sitzbelüftung vorne* impliziert u.a. das Entfernen der *Sitzverstellung, elektrisch mit Memory für Fahrersitz*, sodass auch in diesem Fall diese Features nie gleichzeitig ausgewählt werden können. Die beschriebenen Beispiele stellen nur einen kleinen Teil der Restriktionen dar. Es gibt noch viele weitere, die hier nicht aufgeführt werden (diese können zum Beispiel mit Hilfe des aus dem BMW-Konfigurators ermittelt werden [Weba]). Durch diese Restriktionen werden sich die Möglichkeiten aller Fahrzeugkombinationen merklich reduzieren. So wird das Ergebnis deutlich vielversprechender als die erste Berechnung sein. Dennoch ist es einleuchtend, dass die Problematik im Entwicklungsprozess behandelt werden muss [Beh00, Bör94].

Für die Funktionsverteilung auf Steuergeräte gilt für die Praxis, dass diese enorme Menge an Verteilungsmöglichkeiten natürlich niemals ausgenutzt wird. Stattdessen wird eine Verteilung festgelegt, die nicht ohne Weiteres verändert wird. Auf diese Weise werden Alternativen nicht zugelassen, sodass die obige Zahl praktisch auf genau eine Verteilungsmöglichkeit reduziert wird. Der Trend in der Automobilindustrie zeigt aber auch, dass es Bestrebungen gibt, mehr Flexibilität in Bezug auf Funktionsverteilung zu erreichen. Dies wird in den Spezifikationen des Automotive Open System Architecture (AUTOSAR)-Standards deutlich [AUT10b, AUT10c, AUT10a, AUT10d]. Es ist also wichtig, auf allen Ebenen im Ent-

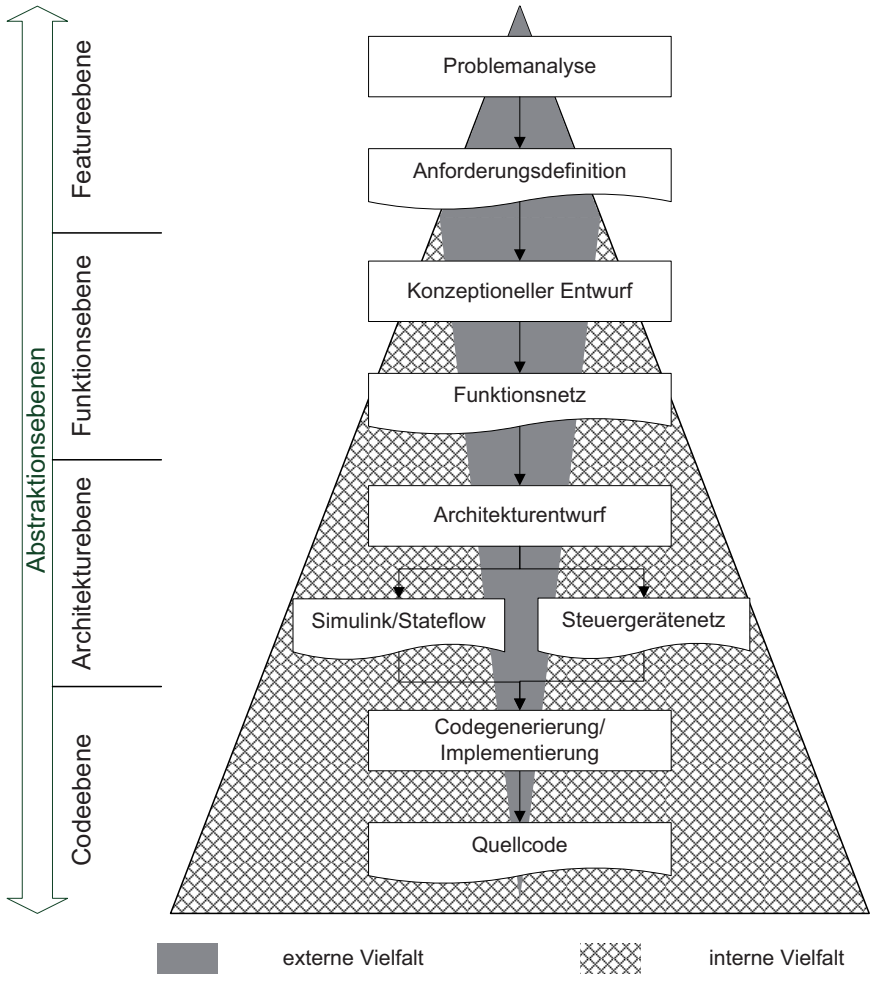


Abbildung 4.1.: Der Referenzprozess und die zu berücksichtigende Variabilität

wicklungsprozess jegliche Art der Vielfalt zu erfassen und geeignet in den Prozess zu integrieren.

Es sei an dieser Stelle verdeutlicht, dass sich die obigen Betrachtungen auf ein Fahrzeugmodell beziehen, dem BMW Modell X5. Das Diversitätsproblem erschwert sich um weitere Dimensionen, wenn auch *modell-*, *baureihen-* und *markenübergreifende Vielfalt* betrachtet wird. Abbildung 4.2 gibt diesbezüglich eine Übersicht. Die Baureihe X besteht beispielsweise aus den weiteren Modellen X1, X3 und X6. Die Sonderausstattungen der jeweiligen Modelle variieren bereits innerhalb einer Baureihe. Beispielsweise ist die *aktive Geschwindigkeitsregelung mit Stop&Go Funktion* im Model X1 nicht als Sonderausstattung verfügbar. Hier wird nur die *Geschwindigkeitsregelung mit Bremsfunktion* bereitgestellt. Variation innerhalb einer Baureihe hat also einen anderen Charakter als in einem Modell der Baureihe. Diese modellübergreifende Variation zu erfassen, stellt Automobilhersteller vor eine große Herausforderung, da in der Regel für jedes Fahrzeugmodell ein eigener Entwicklungsprozess angestoßen wird. Dies wird in der Abbildung durch die Darstellung mehrerer Entwicklungsprozesse für verschiedene Fahrzeugmodelle verdeutlicht. In der PKW-Domäne von BMW gibt es wiederum mehrere Baureihen, die sich in weitere Modelle unterteilen. So gibt es neben der Baureihe X, die weiteren Baureihen 1, 3, 5, 7, Z4, M und Hybrid. Die baureihenübergreifende Variation verschärft die Situation um eine weitere Dimension. Schließlich besteht die Marke BMW aus weiteren Automarken, wie zum Beispiel der Marke MINI. Für diese werden weitestgehend dieselben Features angeboten wie für die BMW Baureihen. Die möglichen Kombinationen variieren allerdings hier ebenfalls.

Vielfalt führt angesichts der Angebote zu Kundenzufriedenheit und größeren Marktanteilen, aber der Softwareentwicklungsprozess wird deutlich komplexer. Es müssen also geeignete Konzepte entworfen, realisiert und in den Entwicklungsprozess integriert werden. Bisher wurde das Problem der Vielfalt auf einem recht abstrakten Niveau beschrieben. Um Lösungen für den betrachteten Entwicklungsprozess erarbeiten zu können, muss zunächst eine saubere Terminologie eingeführt werden. So reicht es nicht alleine aus, von Vielfalt oder Diversität zu sprechen. Es müssen geeignete Begriffe eingeführt werden, die sowohl das Problem als auch die Lösung eindeutig beschreiben. Zu diesem Zweck wird in Abschnitt 4.1.1 zunächst die in dieser Arbeit verwendete Terminologie erläutert. Der betrachtete Referenzprozess schließt hauptsächlich die frühe Phase der Entwicklung ein. Die Aktivitäten, die verwendeten Hilfsmittel in der jeweiligen Aktivität als auch die resultierenden Ergebnisse müssen untersucht werden, um eine angemessene Lösung entwickeln zu können. Daher werden in Abschnitt 4.1.2 die bevorstehenden Herausforderungen und die hieraus resultierenden Anforderungen für den Referenzprozess beschrieben. Sind diese ermittelt, müssen geeignete Konzepte entwickelt werden, die den Anforderungen gerecht werden. Zu diesem Zweck werden in Abschnitt 4.2 und Abschnitt 4.3 Konzepte vorgestellt, die es ermöglichen, Vielfalt im Referenzprozess zu erfassen. Weiterhin werden in Abschnitt 4.4 die wichtigsten Aspekte vorgestellt, die der Realisierung der beschriebenen Konzepte dienen. Da dieses Thema bereits öfters in Bezug auf unterschiedliche Fragestellungen behandelt wurde, werden

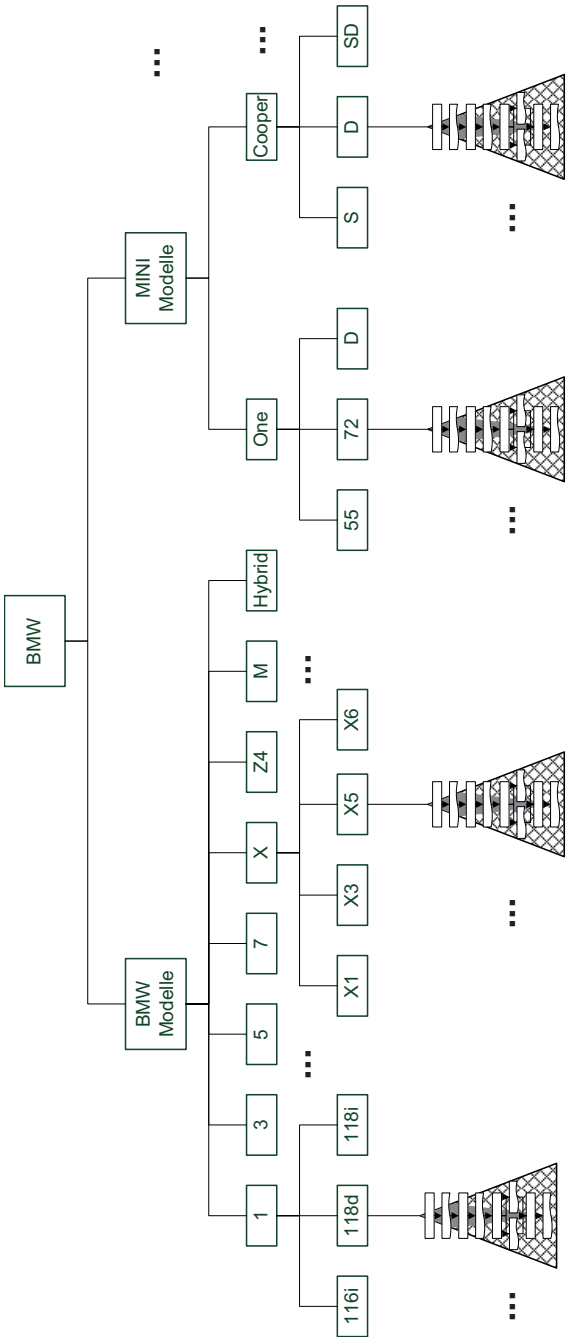


Abbildung 4.2.: Modell-, baureihen- und markenübergreifende Variabilität

in Abschnitt 4.5 verwandte Arbeiten vorgestellt. Insbesondere beinhaltet dieser Abschnitt einen Vergleich mit den Konzepten dieser Arbeit. Schließlich endet das Kapitel mit einer Zusammenfassung.

4.1.1. Terminologie

In diesem Abschnitt werden Begriffe eingeführt, die zur Beschreibung der Konzepte erforderlich sind. Im Wesentlichen werden dabei zwei Oberbegriffe eingeführt: Variabilität und Variationspunkt. Der Variationspunkt wird dabei weiter verfeinert, indem Begriffe wie Variabilitätsmechanismus, Variante, Bindungsmechanismus und Bindezeit näher beschrieben werden.

4.1.1.1. Variabilität

In der Softwareentwicklung bezeichnet *Variabilität* die Eigenschaft der *Veränderlichkeit von Merkmalen* eines Softwaresystems oder eines Softwaredokuments. Merkmale mit dieser Eigenschaft werden auch als *variable Merkmale* bezeichnet [CE00, PBvdL05, CN07, vdLSR07].

Variable Merkmale sind beispielsweise vom Kunden optional selektierbare Features eines Automobils, im gegenseitigen Ausschluss stehende Anforderungen, optionale Funktionen in einem Funktionsnetz oder variable Datentypen (zum Beispiel `int16` oder `int8`) im Quellcode. Die Gründe, die zu Variabilität führen sind sehr vielfältig. So entsteht Variabilität aufgrund verschiedener Kundenbedürfnisse, unterschiedlicher Marktpräsenz, vielfältiger Technologien etc.

4.1.1.2. Variationspunkt

Ein *Variationspunkt* beschreibt ein variables Merkmal in einem Softwaredokument. Die Beschreibung eines Variationspunktes beinhaltet den Variabilitätsmechanismus, die Varianten, den Bindungsmechanismus und die Bindezeit. Diese werden im Folgenden genauer erläutert.

Variabilitätsmechanismus Ein *Variabilitätsmechanismus* realisiert einen Variationspunkt durch Konstrukte der zugrunde liegenden Sprache. Die Sprache kann hierbei u.a. eine natürliche, eine Modellierungs- oder eine Programmiersprache sein.

Für die deutsche Sprache als natürliche Sprache sind beispielsweise die Ausdrücke

- *oder*,
- *optional*,
- *wenn ... dann ... sonst ...*

mögliche Sprachmechanismen, um variable Merkmale zu definieren.

In einer Modellierungssprache wie beispielsweise Simulink kommen wiederum spezifische Mechanismen zum Einsatz (vgl. [Wei08, Sch10]):

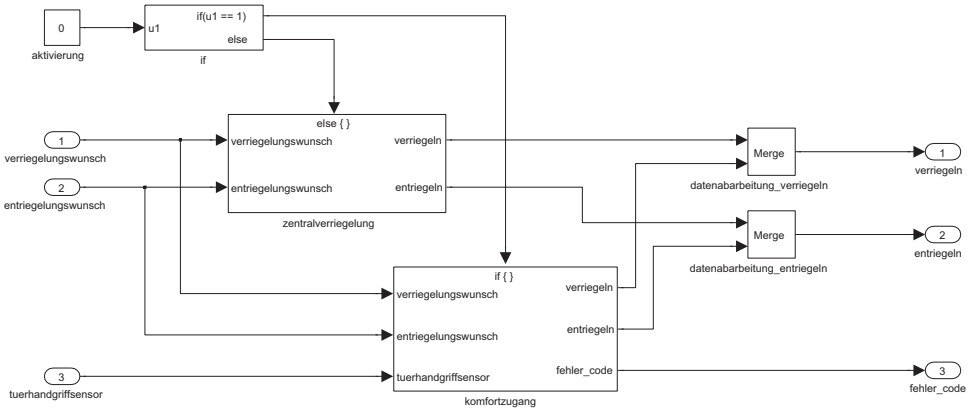


Abbildung 4.3.: Ein Beispiel für einen Variabilitätsmechanismus mit bedingt ausführbaren Subsystemen mittels `If Action Subsystem`

- Bedingt ausführbare Modelle
- Bedingt ausführbare Subsysteme
- Konfigurierbare Subsysteme
- Signalrouting
- Logische Gatter
- Parameter

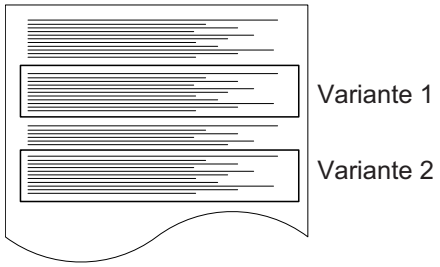
In Kapitel 6 werden Variabilitätsmechanismen in Simulink ausführlich behandelt. An dieser Stelle sei anhand Abbildung 4.3 ein Beispiel für bedingt ausführbare Subsysteme illustriert. Das variable Merkmal in diesem Modell ist das Fahrzeugzugangssystem und wird in den `If Action Subsystem`-Blöcken `zentralverriegelung` und `komfortzugang` gekapselt. Die Ausführung dieser beiden Blöcke wird durch den Block `If` gesteuert. Die Bedingung zur Ausführung von `komfortzugang` ist `u1 == 1`. Der Wert für `u1` wird durch den Block `aktivierung` festgelegt. Wenn die Bedingung positiv ausgewertet wird, so wird auch `komfortzugang` ausgeführt, andernfalls wird `zentralverriegelung` ausgeführt.

In Programmiersprachen können ebenfalls verschiedene Mechanismen eingesetzt werden. So sind Konstrukte wie

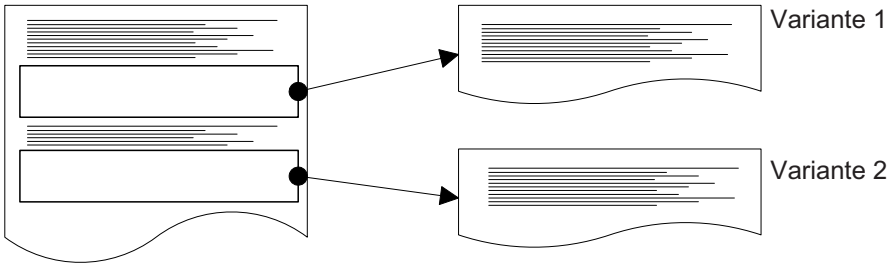
```
if ... then ... else ...
```

oder

```
switch ... case ...
```



(a) Varianten werden in einem einzigen Softwaredokument realisiert



(b) Varianten werden in separaten Softwaredokumenten realisiert

Abbildung 4.4.: Integrierte und separierte Organisation von Varianten in Softwaredokumenten

gängige Kontrollstrukturen. In den Programmiersprachen C/C++ können weiterhin Präprozessordirektiven eingesetzt werden. In objektorientierten Programmiersprachen, wie zum Beispiel Java, kann die Vererbung als weiterer Variabilitätsmechanismus verwendet werden. Weiterhin gibt es in der Programmiersprache Ada das Konzept der Generizität [Nag03, Bar06], um beispielsweise Datentypen als variables Merkmal zu kennzeichnen.

Variante Eine *Variante* ist eine Ausprägung eines variablen Merkmals. Sie ist somit eine Manifestation in Form eines Repräsentanten des variablen Merkmals [CE00, PBvdL05, CN07, vdLSR07]. In Abbildung 4.3 stellt beispielsweise die innere Struktur vom Subsystem `zentralverriegelung` eine Variante dar (in der Abbildung nicht zu sehen).

Varianten werden abhängig vom Variabilitätsmechanismus auf zwei Arten organisiert: (1) integriert oder (2) separiert. *Integrierte Varianten* werden in einem einzigen Softwaredokument erfasst. Abbildung 4.4(a) illustriert diese Konstellation. Ein konkretes Beispiel ist in Abbildung 4.3 gegeben. Hier werden sowohl `zentralverriegelung` als auch `komfortzugang` in einem einzigen Simulink-Modell erfasst. *Separierte Varianten* hingegen werden in jeweils verschiedenen Softwaredokumenten erfasst und verwaltet. Hier muss allerdings der Bezug zum Kerndokument hergestellt werden. Abbildung 4.4(b) veranschaulicht diese Situation.

Bindungsmechanismus Eine *Bindung* löst Variabilität durch eine Variante auf. Nach einer Bindung wird somit ein variables Merkmal zu einem festen Merkmal überführt. In der natürlichen Sprache würde beispielsweise Variabilität in der Aussage

wenn X dann A sonst B

entweder durch Variante *A* oder durch Variante *B* aufgelöst werden. In Simulink würde beispielsweise für das Modell in Abbildung 4.3 der Datenfluss entweder zum Subsystem `zentralverriegelung` oder zu `komfortzugang` weitergeleitet. Analog würden Kontrollstrukturen in Programmiersprachen wie etwa

```
if Bedingung then Anweisungsblock1 else Anweisungsblock2
```

durch `Anweisungsblock1` oder `Anweisungsblock2` gebunden und ausgeführt werden.

Eine Bindung wird durch einen *Bindungsmechanismus* realisiert. Der Bindungsmechanismus legt fest, wie eine Variante gebunden werden soll. In der Regel kommen vier Bindungsmechanismen zum Einsatz:

1. Selektion
2. Aktivierung
3. Substitution
4. Generierung

Jeder Bindungsmechanismus wird durch verschiedene Auswertungsmaschinen berechnet. Folgende Auswertungsmaschinen kommen zur Berechnung in Betracht:

- Konfigurationsmaschine
- Updatemaschine
- Solver
- Präprozessor
- Compiler
- Ausführungsmaschine

Durch die *Selektion* wird eine Variante, ohne eine in den Variabilitätsmechanismus integrierte explizite Bedingung, innerhalb eines Konfigurierungsvorgangs ausgewählt. Typischerweise ist hierfür eine Benutzerinteraktion erforderlich. Der BMW Konfigurator [Weba] ist ein Beispiel für einen Bindungsmechanismus durch Selektion mit einer Konfigurationsmaschine zur Validierung sämtlicher Benutzereingaben sowie Berechnung von Entscheidungshilfen.

Die *Aktivierung* unterscheidet sich von der Selektion durch das Vorhandensein einer Bedingung, die je nach Auswertung eine bestimmte Variante automatisch

aktiviert. Daher ist in diesem Fall nicht notwendigerweise eine Benutzerinteraktion erforderlich, da die Bedingung bereits explizit im Variabilitätsmechanismus integriert ist. Abbildung 4.3 illustriert ein Beispiel für einen Bindungsmechanismus durch Aktivierung, der zum Beispiel durch einen Solver in einer Simulation berechnet wird. Hier wird abhängig vom Ergebnis des `If`-Blocks entweder das Subsystem `zentralverriegelung` oder `komfortzugang` aktiviert.

Bei der *Substitution* enthält der Variabilitätsmechanismus einen Platzhalter, der durch die jeweiligen Varianten ersetzt wird. Dabei kann die Ersetzung entweder durch einen Konfigurierungsvorgang manuell bestimmt werden oder automatisch durch Auswertung entsprechender Bedingungen festgesetzt werden. Das Generizitätskonzept in Ada ist ein Beispiel für einen Bindungsmechanismus durch Substitution mit dem entsprechenden Compiler, der die Ersetzung durchführt. Hier werden beispielsweise Variablen im generischen Teil des Programms mit einem Platzhalter für den Datentyp deklariert. Dieser kann dann bei Kompilierung durch verschiedene Datentypen ersetzt werden.

Bei der *Generierung* ist der Variabilitätsmechanismus so ausgelegt, dass Varianten generiert werden. Hier kann die Generierung ebenfalls manuell oder automatisch ausgelöst werden. Die Präprozessordirektiven der Programmiersprachen C/C++ und die zugehörigen Präprozessoren und Compiler sind Beispiele für einen Bindungsmechanismus durch Generierung. Der Präprozessor wertet die Bedingungen der Präprozessordirektiven aus und generiert für den Compiler nur die als gültig ausgewerteten Anweisungsteile, aus denen der Compiler den Objektcode erzeugt. Diese Anweisungsteile könnten beispielsweise jeweils eine Variante darstellen.

Die Bindungsmechanismen Selektion, Aktivierung, Substitution und Generierung können allesamt sowohl für integrierte als auch separierte Varianten angewendet werden.

Bindezeit Eine Bindung wird durch einen Bindungsmechanismus zu einer bestimmten Bindezeit durchgeführt. Die *Bindezeit* beschreibt die Zeit, in der eine Variante gebunden werden kann. In diesem Zusammenhang werden in dieser Arbeit folgende Bindezeiten betrachtet:

- Modellkonstruktionszeit
- Codegenerierungszeit
- Compilezeit
- Laufzeit

Im weiteren Verlauf dieser Arbeit werden die Bindezeiten erneut aufgegriffen und im Zusammenhang mit Variabilitätsmechanismen, Variantenarten und Bindungsmechanismen erläutert.

4.1.2. Herausforderungen und Anforderungen

Nachdem eine allgemeine Einführung und Motivation in den Themenkomplex gegeben und die begriffliche Basis geschaffen wurde, dient dieser Abschnitt zur Beschreibung der bevorstehenden Herausforderungen sowie den hieraus resultierenden Anforderungen. In diesem Zusammenhang haben sich zwei wesentliche Schwerpunkte herauskristalisiert: (1) die Modellierung und (2) die Bindung von Variabilität.

Der erste Aspekt behandelt primär die Fragestellung, wie Variabilität im Entwicklungsprozess identifiziert und dokumentiert werden kann. Der zweite Punkt adressiert die Frage nach der Bindung der erfassten Variabilität, um eine nahtlose Integration in den Entwicklungsprozess zu gewährleisten. Beide Faktoren werden in den folgenden Abschnitten genauer analysiert, sodass der Handlungsbedarf für die im weiteren Verlauf dieses Kapitels vermittelten Konzepte identifiziert werden kann.

4.1.2.1. Modellierung

Die Modellierung umfasst zwei Bereiche. Es werden nachfolgend Problemstellungen in Bezug auf Variabilitätsmodellierung und Restriktionsmodellierung erörtert. Die Restriktionsmodellierung ist ein Teil der Variabilitätsmodellierung. Sie wird aber hier gesondert behandelt, da sie einen enorm wichtigen Punkt darstellt.

Variabilitätsmodellierung

Art der Modellierung Variabilität ist ein Aspekt, der im Entwicklungsprozess berücksichtigt werden muss, um wiederverwendbare Software auf systematische und effiziente Weise zu realisieren. In Abbildung 4.2 wurde bereits das Ausmaß von Variabilität im Referenzprozess dieser Arbeit dargestellt. Sie entsteht bereits auf Featureebene und erstreckt sich in den gesamten Prozess. Die wesentliche Frage an dieser Stelle ist, wie Variabilität geeignet im Entwicklungsprozess erfasst werden kann. Dazu gehören die Identifikation der Variabilität und dessen Repräsentation.

Die Domänenanalyse ist zu dieser Fragestellung eine wichtige und erforderliche Aktivität. Hierbei werden gemeinsame Aspekte einer Menge ähnlicher Softwaresysteme systematisch identifiziert. Dies kann durch Domänenexperten manuell oder durch Einsatz von Softwarewerkzeugen weitestgehend automatisiert durchgeführt werden.

Zur Dokumentation und Repräsentation der identifizierten Variabilitätsinformationen werden sogenannte Variabilitätsmodelle verwendet. In der Literatur haben sich im Wesentlichen zwei Typen von Variabilitätsmodellen etabliert [SD07]: (1) *hierarchisch strukturierte Variabilitätsmodelle* und (2) *Auswahlmodelle* (engl. *Choice Models*). Erstere strukturiert Variabilität in zusammenhängende Merkmale, die in der Regel auf mehreren Hierarchieebenen modelliert werden. Typischerweise stehen weitere Ausdrucksmittel zur Verfügung, um variable Merkmale, also Variationspunkte, zu kennzeichnen. Letzteres hingegen basiert hauptsächlich auf die zur Auswahl stehenden Varianten der Variationspunkte. Im einfachsten Fall würde

bereits eine Liste von Varianten ausreichen. In der Literatur werden allerdings oftmals zwei Ebenen eingeführt: Eine Ebene repräsentiert den im Softwaredokument identifizierten Variationspunkt und die zugehörige Subebene beinhaltet die Varianten.

Zur Illustration beider Modellierungstypen wird als Beispiel das Fahrzeugzugangssystem aus Abschnitt 2.3 herangezogen. Hier wurde zum einen die Zentralverriegelung als Grundausstattung und zum anderen der Komfortzugang als Sonderausstattung vorgestellt. Wenn der Komfortzugang als Sonderausstattung selektiert wird, dann muss das elektronische System des Fahrzeugs mit einem CA-Steuergerät und vier weiteren TAGE-Steuergeräten ausgestattet werden. Außerdem müssen die Türaußengriffe jeweils mit Sendeantennen (also vier Stück) sowie der Innenraum des Fahrzeugs mit fünf weiteren Sende- und Empfangsantennen ausgestattet werden. Beide Features realisieren die Funktionen zum Verriegeln und Entriegeln der Fahrzeugtüren. Initiiert werden die Funktionen entweder durch einen Sollwertgeber, gesteuert von einem Fahrer, oder automatisch vom Fahrzeug abhängig (von bestimmten Zuständen). Sollwertgeber sind der mechanische Schlüssel, die Funkfernbedienung, die Funkfernbedienung mit Funkempfänger und die Zentralverriegelungstaste. Die Funkfernbedienung mit Funkempfänger wird nur in Kombination mit dem Komfortzugang ausgeliefert. Das automatische Verriegeln erfolgt bei Überschreitung einer bestimmten Geschwindigkeit. Wird das Fahrzeug gestoppt, so entriegeln die Türen, sobald der Schlüssel aus dem Zündschloss entnommen wird bzw. im Falle des Komfortzugangs durch Drücken des Start-Stop-Knopfs.

Die Variabilität in der Beschreibung obiger Informationen soll nun als hierarchisch strukturiertes Variabilitätsmodell einerseits und als Auswahlmodell andererseits dargestellt werden. Die am weitverbreitetsten hierarchisch strukturierten Variabilitätsmodelle sind sogenannte Featuremodelle [KCH$^+$90] sowie auf diesen basierende Erweiterungen [GFd98, CK05]. Featuremodelle wurden von *Kang et al.* im Jahre 1990 vorgestellt [KCH$^+$90]. Ein *Feature* ist dabei ein benutzersichtbarer Aspekt oder eine Eigenschaft der Domäne. Die Features der Domäne werden in Form einer Baumstruktur mit entsprechenden Notationen zur Beschreibung variabler Features repräsentiert.

Abbildung 4.5 illustriert das Resultat der obigen Beschreibung in Form eines Featuremodells. Jedes Feature stellt einen Knoten im Baum dar. So sind die Funktionen `Zentralverriegelung` und `Komfortzugang` zwei Features im Featuremodell. Das Modell ist dabei durch die Einführung von Subfeatures organisiert. Zum Beispiel ist `Steuergeräte` ein Subfeature von `Elektronisches System`. Auf diese Weise lässt sich die Domäne hierarchisch strukturieren. Variabilität wird im Wesentlichen durch zwei Konstrukte realisiert. Features, die optional sind, werden über eine Kante mit einem weißen Kreis oberhalb des optionalen Features dargestellt. Das Feature `Komfortzugang` ist zum Beispiel ein optionales Feature. Alternative Features werden durch einen Bogen, der die Kanten der entsprechenden Features verbindet, dargestellt. Zum Beispiel sind `Funkfernbedienung` und `Funkfernbedienung mit Funkempfänger` alternativ zueinander. Schließlich können Restriktionen zwischen Features über Teilbaumgrenzen hinweg ausgedrückt werden. Die Relation `requires`

ist eine derartige Restriktion. Sie drückt aus, dass ein Feature ein anderes Feature erfordert. Zum Beispiel benötigt das Feature Komfortzugang die Features CA, TAGE F, TAGE BF usw. (umgekehrt benötigt CA, TAGE F, TAGE BF usw. auch Komfortzugang; daher sind die Pfeile in beide Richtungen abgebildet). Eine weitere Relation, nicht in der Abbildung dargestellt, ist excludes, mit welcher der Ausschluss eines Features ausgedrückt wird.

Die Modellierung der Variabilität in einem Auswahlmodell ist in Abbildung 4.6 dargestellt. Sie basiert auf der Notation orthogonaler Variabilitätsmodelle nach *Pohl et al.* [PBvdL05]. Die zwei wesentlichen Konzepte sind der Variationspunkt und die Variante. Durch den Variationspunkt wird ein variables Merkmal identifiziert. Die Varianten stellen die möglichen Ausprägungen des Variationspunktes dar. Ein Variationspunkt wird in der grafischen Notation in Form eines Dreiecks dargestellt. In der Abbildung sind Funktionen, Steuergeräte, Sollwertgeber und Antennen jeweils Variationspunkte. Varianten hingegen werden als Rechtecke notiert. Zum Beispiel sind Zentralverriegelung, Komfortzugang und Funkfernbedienung jeweils Varianten. Die Relationen zwischen einem Variationspunkt und Varianten drücken die vorherrschenden variablen Eigenschaften aus. Eine Variante kann dabei verbindlich sein. Dies wird durch eine durchgezogene schwarze Kante zwischen dem Variationspunkt und der Variante festgelegt. Beispielsweise ist die Variante Zentralverriegelung eine verbindliche Variante des Variationspunktes Funktionen. Weiterhin kann eine Variante optional sein. Optionale Varianten werden anhand einer gestrichelten schwarzen Kante zwischen dem Variationspunkt und der Variante visualisiert. Zum Beispiel ist der Komfortzugang eine optionale Variante des Variationspunktes Funktionen. Außerdem können Gruppen von Varianten als alternativ gekennzeichnet werden. Dazu werden alle zur Gruppe gehörigen Varianten durch einen schwarzen Bogen an den Kanten, die den Variationspunkt und die Varianten miteinander verbinden, umschlossen. Die Funkfernbedienung und Funkfernbedienung mit Funkempfänger stellen beispielsweise alternative Varianten des Variationspunktes Sollwertgeber dar. Schließlich können wie bei Featuremodellen ebenfalls Restriktionen definiert werden. Es gibt dabei requires- und excludes-Restriktionen, die zwischen (1) Varianten, (2) Varianten und Variationspunkten und (3) Variationspunkten auftreten können. Dargestellt werden sie über gepunktete Kanten mit einer entsprechenden Beschriftung. In Abbildung 4.6 sind ausschließlich requires-Restriktionen zwischen Varianten dargestellt. Zum Beispiel erfordert der Komfortzugang die Funkfernbedienung mit Funkempfänger.

Beide Modellierungstypen haben ihre Vor- und Nachteile. Mit Featuremodellen nach *Kang et al.* kann die analysierte Domäne in zusammenhängende und hierarchisch organisierte Features strukturiert werden. Dies führt zu einem Modell, das jegliche Variabilitätsinformationen zentral erfasst. Um dies zu erreichen, werden allerdings sehr viele Features eingeführt, die nur zur Umsetzung der Strukturierung dienen. Dadurch wird das Modell wesentlich größer und komplexer. Die Übersichtlichkeit und Verständlichkeit geht somit verloren.

Orthogonale Variabilitätsmodelle nach *Pohl et al.* hingegen eignen sich durch die zwei Kernkonzepte Variationspunkt und Variante besonders gut zur Erfassung von

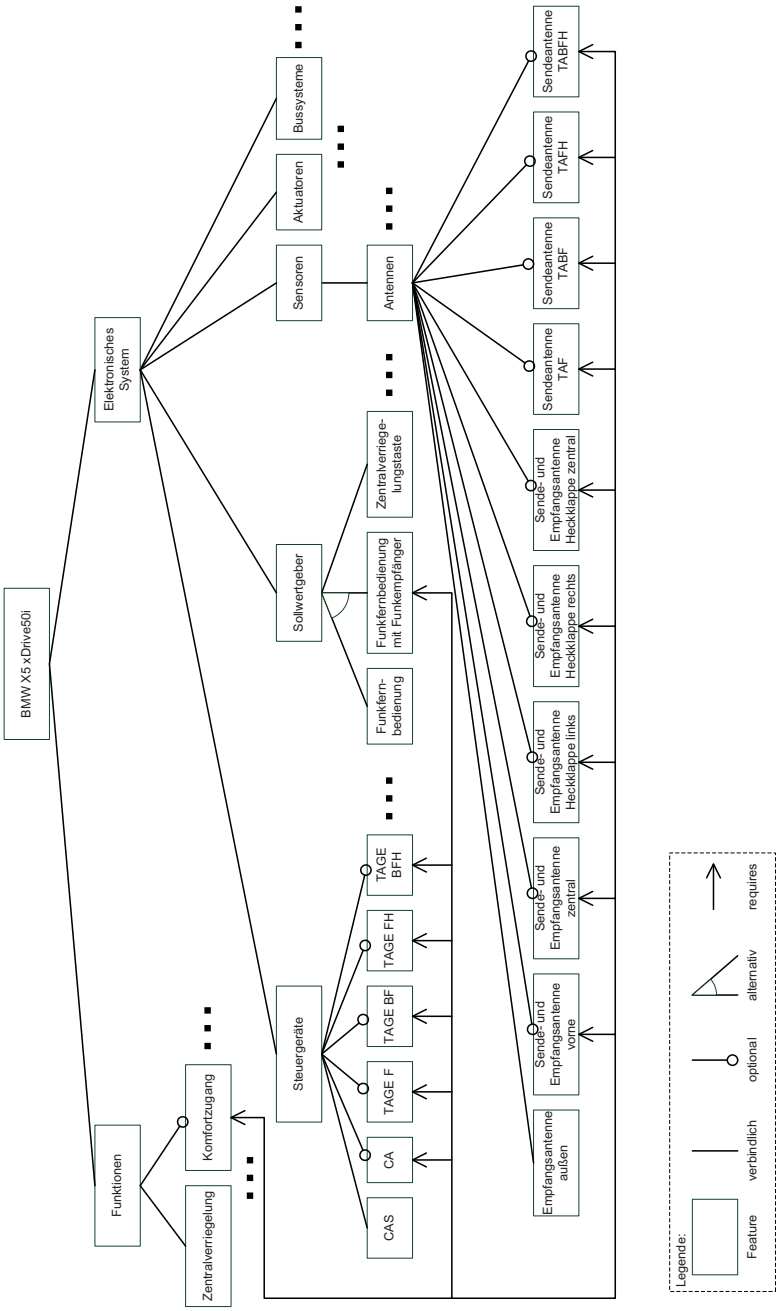


Abbildung 4.5.: Das hierarchisch strukturierte Variabilitätsmodell für das Fahrzeugzugangssystem in Form eines Featuremodells nach FODA

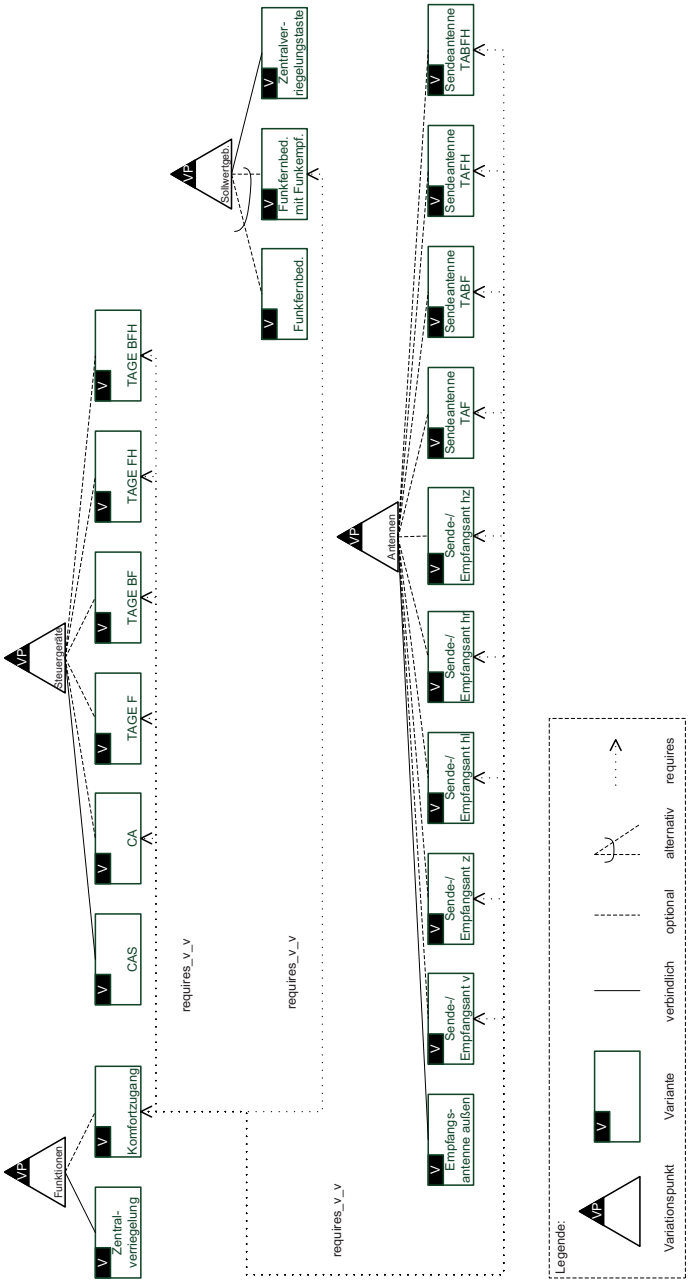


Abbildung 4.6.: Das Auswahlmodell für das Fahrzeugzugangssystem in Form eines orthogonalen Variabilitätsmodells (nach [PBvdL05])

variablen Merkmalen in Softwaredokumenten. Allerdings ist dies das einzige Strukturierungskonzept. Auf diese Weise entstehen unzusammenhängende Teilbäume, die schwer zu verwalten sind.

Weiterhin gilt für beide Modellierungstypen, dass keine Konzepte zur Modellierung von Variabilitätsmechanismen, Bindungsmechanismen und Bindezeiten existieren. Zudem fehlt die Berücksichtigung des Entwicklungsprozesses, welche die eigentliche Variabilität hervorruft.

Angesichts der erwähnten Vor- und Nachteile wird im Rahmen dieser Arbeit ein Ansatz verfolgt, der die Vorteile beider Modellierungstypen kombiniert und um die fehlenden Aspekte erweitert. Die Grundlage in diesem Ansatz stellt das Auswahlmodell dar. Das heißt, dass die Konzepte Variationspunkt und Variante wesentlicher Bestandteil der Modellierung sind. Auf diese Weise können Variationspunkte in Softwaredokumenten intuitiver modelliert werden. Um aber isolierte Teilbäume zu vermeiden, Bedarf es geeigneter Strukturierungskonzepte. Fehlende Konzepte müssen geeignet in die Modellierung einbezogen werden. Zur Vermeidung unübersichtlicher Variabilitätsmodelle, die aufgrund der Spannbaumdarstellung entstehen, wird in dieser Arbeit eine Baumlistenrepräsentation präferiert.

Durch die Integration eines derartigen Variabilitätsmodells in den Entwicklungsprozess werden die nur implizit verwalteten Variationspunkte explizit erfasst. Es ermöglicht die Modellierung und bildet die Basis zur Bindung von Varianten auf eine einfach verständliche Art. Zudem können die Informationen aus dem Modell zur weiteren Verarbeitung im gesamten Entwicklungsprozess verwendet werden.

Strukturierungsmaßnahmen Bisher wurde das Konzept des Variationspunktes als Vaterknoten und Varianten als Kindknoten in einem Auswahlmodell vorgestellt. Dieses Konzept alleine reicht allerdings nicht aus, eine feinere Strukturierung im Variabilitätsmodell vorzunehmen. Somit mangelt es an ausreichender Übersicht, wodurch das Modell schwieriger zu verstehen ist. Daher ist die primäre Fragestellung, wie das Konzept Variationspunkt-Variante geeignet verfeinert werden kann, um eine detailliertere Strukturierung zu erhalten.

Zur Lösung dieses Problems ist ein Gruppierungskonzept erforderlich, das feingranularere Strukturierungen, auch über mehrere Hierarchieebenen hinweg, erlaubt. Somit können logisch zusammenhängende Varianten zweckmäßiger gruppiert und bei Bedarf in mehrere Hierarchiestufen angegliedert werden. Dadurch wird eine höhere Übersicht erzielt, die auch zu einem besseren Verständnis des Variabilitätsmodells führt.

Als Anwendungsfall sei das folgende Beispiel betrachtet: Das Fahrzeugzugangssystem besteht aus den Funktionen (1) Zentralverriegelung und (2) Komfortzugang. Darüber hinaus sind die Antennen zur Erkennung des Entriegelungs- / Verriegelungswunsches wesentliche Bestandteile dieses Systems. Durch eine Strukturierung der Funktionen und Antennen in jeweils verschiedene Gruppen wird die Übersicht enorm erhöht. Das Ergebnis einer derartigen Strukturierung könnte wie folgt zusammengestellt sein:

- Variationspunkt: Fahrzeugzugangssystem
  - Gruppe: Funktionen
    - Variante: Zentralverriegelung
    - Variante: Komfortzugang
  - Gruppe: Sensoren
    - Variante: Empfangsantenne außen
    - Variante: Sendeantenne außen
    - Variante: Sende- und Empfangsantenne innen

Berücksichtigung von Abstraktionsebenen Der dieser Arbeit zugrunde liegende Referenzprozess besteht aus mehreren Abstraktionsebenen. In jeder Ebene müssen Variationspunkte behandelt werden. Dabei gibt es verschiedene Arten.

Zum einen existieren Variationspunkte, die Variabilität aus der höheren Abstraktionsebene detaillieren. In diesem Fall gibt es also eine Korrespondenz zwischen Variationspunkten verschiedener Abstraktionsebenen. Ein Beispiel hierfür ist der Übergang aus der Featureebene zur Funktionsebene. Während in der Featureebene das Fahrzeugzugangssystem ein Variationspunkt mit den Varianten Zentralverriegelung und Komfortzugang ist, der eher in textueller Form beschrieben wird, detailliert die Funktionsebene diese Beschreibung durch Modellierung feingranularer Funktionen, wie etwa mit Sensor- und Aktuatorfunktionen sowie Datenaufbereitungs- und Datenabarbeitungsfunktionen.

Zum anderen entstehen in einer bestimmten Abstraktionsebene neue Variationspunkte, die auf charakteristische Aktivitäten der Abstraktionsebene zurückzuführen sind. So entsteht beispielsweise auf der Funktionsebene Variabilität in der Schnittstellenbeschreibung der Funktionen, die zusätzlich identifiziert und modelliert werden muss.

In Anbetracht dieser Situation stellt sich die elementare Frage, ob ein einziges zentrales Variabilitätsmodell, das alle Ebenen abdeckt, ausreichen wird. Diese Frage kann verneint werden, wenn die steigende Anzahl und Komplexität der Variabilität im Verlauf des Entwicklungsprozesses betrachtet wird. Den gesamten Entwicklungsprozess durch ein zentrales Variabilitätsmodell zu erfassen, scheint daher nicht praktikabel, da dieser dadurch sehr unübersichtlich und schwierig zu verstehen ist. Ein weiterer wichtiger Aspekt hierbei ist, dass eine Aufteilung der Variabilitätsinformation deutlich aufwendiger wird.

Im Rahmen dieser Arbeit wird daher ein Ansatz gewählt, der es erlaubt, das Variabilitätsmodell auf alle Abstraktionsebenen anzuwenden. Dabei müssen insbesondere mehrere Variabilitätsmodelle für verschiedene Abstraktionsebenen eingesetzt werden können, um der steigenden Komplexität entgegenzuwirken. Entsprechend bedarf es geeigneter Konstrukte, welche die Zuordnung eines Variabilitätsmodells zu einer Abstraktionsebene festlegen und darüber hinaus auch identifizieren können. Außerdem müssen Korrespondenzen zwischen Variationspunkten der verschiedenen Ebenen hergestellt werden können.

Unterstützung bei der Variabilitätsmodellierung Die Integration der Variabilitätsmodellierung als Aktivität in den Entwicklungsprozess bedeutet natürlich zunächst einen höheren Aufwand. Variationspunkte müssen identifiziert und darüber hinaus im Variabilitätsmodell dokumentiert werden. Für einen Modellierer gibt es sehr viele Möglichkeiten, die Variabilitätsinformationen zu strukturieren. Beispielsweise können Variationspunkte und Varianten beliebig bezeichnet werden, also ohne Betrachtung des Softwaredokuments, das die Variabilität enthält. Dies führt zu Inkonsistenzen, sodass das Variabilitätsmodell schwieriger zu verstehen und nachzuvollziehen ist. Somit ergibt sich die Frage, wie Variabilitätsmodelle integraler Bestandteil des Entwicklungsprozesses werden können, um den Modellierungsaufwand so gering wie möglich zu halten.

Um dies zu erreichen, müssen die aus den Aktivitäten hervorgehenden Softwaredokumente Ausgangspunkt für die Variabilitätsmodellierung sein. Bei der Modellierung wird also stets eine Assoziation zu einem entsprechenden Softwaredokument erforderlich. Dadurch wird es ermöglicht, erlaubte und unerlaubte Aktionen zu erkennen. So können proaktiv ungültige Aktionen verhindert und gültige vorgeschlagen werden. Auf diese Weise werden die Möglichkeiten, Variabilität zu modellieren, geeignet eingeschränkt und Inkonsistenzen vermieden. Die Modellierung kann wesentlich schneller durchgeführt werden und resultiert in leichter verständlicheren Modellen.

Als Anwendungsfall sei das folgende Beispiel betrachtet: Es besteht eine Assoziation zwischen einem Funktionsnetz und einem Variabilitätsmodell. Im Funktionsnetz wird die Funktion Fahrzeugzugangssystem modelliert. Als Subfunktionen werden die Funktionen Zentralverriegelung und Komfortzugang modelliert. Im Variabilitätsmodell wird das Fahrzeugzugangssystem als Variationspunkt markiert. Dies hat zur Folge, dass alle Subfunktionen automatisch Varianten darstellen. Die Namen des Variationspunktes und der Varianten werden aus dem Funktionsnetz übernommen und sind nicht im Variabilitätsmodell überschreibbar.

Modellierung von Variabilitätsmechanismen Zur Realisierung von Variationspunkten in Softwaredokumenten werden Variabilitätsmechanismen eingesetzt. Diese werden vollständig aus Konstrukten der zugrunde liegenden Sprache definiert. Allerdings führt dies zu der Situation, dass der Unterschied zwischen Sprachelementen der funktionalen Logik und des Variabilitätsmechanismus nur schwer erkennbar ist. Zudem wird es auch deutlich schwieriger, die Begründung für den Einsatz eines bestimmten Variabilitätsmechanismus nachzuvollziehen. Hieraus ergibt sich nun die zentrale Frage, wie Variabilitätsmechanismen als Bestandteil in die Variabilitätsmodellierung eingebettet werden können.

Das Variabilitätsmodell muss also ein Strukturierungskonzept beinhalten, das die explizite Modellierung des Variabilitätsmechanismus eines Variationspunktes unterstützt. Hier müssen die Sprachkonstrukte beschrieben werden können, mit denen der Variationspunkt im Softwaredokument realisiert wird. Auf diese Weise wird der Unterschied zwischen Sprachbestandteilen der funktionalen Logik und des Variabilitätsmechanismus klar herausgestellt. Zudem können Entwurfsentscheidungen in Bezug auf den Einsatz bestimmter Variabilitätsmechanismen leichter

nachvollzogen werden. Weiterhin können die modellierten Elemente zur weiteren Verarbeitung einfacher verwendet werden.

Als Anwendungsfall sei das folgende Beispiel betrachtet: Abbildung 4.3 zeigt einen Variabilitätsmechanismus in einem Simulink-Modell, das den Variationspunkt eines Fahrzeugzugangssystems realisiert. Auf einem Blick ist es nur schwer zu erkennen, welche Bestandteile der Funktionslogik und welche dem Variabilitätsmechanismus zuzuordnen sind. Die Blöcke, die zur Realisierung des Variabilitätsmechanismus eingesetzt werden, sind die folgenden:

- `Constant: aktivierung`
- `If: if`
- `IF Action Subsystem: zentralverriegelung`
- `IF Action Subsystem: komfortzugang`
- `Merge: datenabarbeitung_verriegeln`
- `Merge: datenabarbeitung_entriegeln`

Die Erfassung dieser Information in einem Variabilitätsmodell schafft mehr Klarheit für das Verständnis des Modells.

Modellierung von Variantenarten Im Entwicklungsprozess ist jede Abstraktionsebene durch spezifische Eigenschaften in Bezug auf eingesetzte Sprachen, Methoden und Werkzeuge charakterisiert. Demnach werden auch Varianten spezifisch behandelt. So werden Varianten in einigen Fällen als integraler Bestandteil modelliert, in anderen Fällen als separierte Entitäten. Diese Information wird im Variabilitätsmodell bisher nicht erfasst. Das Fehlen dieser Information führt allerdings zu einem Verlust in Bezug auf Flexibilität bei der Aufgabenplanung von Projekten. Separierte Varianten eignen sich besonders gut für Arbeitsteilung. Ohne die Informationen, ob eine Variante integriert oder separiert ist, könnten zeitweise Fehlplanungen gemacht werden. Weiterhin stellt diese Information eine wichtige Entscheidungsgrundlage beim Deployment von Software auf Steuergeräte. So ist das Deployment von separierten Varianten deutlich flexibler gestaltbar als bei integrierten Varianten. Das Fehlen der Information im Variabilitätsmodell könnte daher zu falschen Deploymententscheidungen führen. Aus diesen Aspekten resultiert die Frage, wie die Informationen, ob Varianten integriert oder separiert sind, im Variabilitätsmodell geeignet erfasst werden können.

Um dieses Problem zu lösen, ist also ein Konzept erforderlich, das diese Information im Variabilitätsmodell geeignet erfasst. Die Information ist eine Eigenschaft einer Variante und sollte daher auch im Variabilitätsmodell als weitere Information der Varianten modelliert werden. Bei separierten Varianten ist eine Referenz erforderlich, die die Variante adressiert (typischerweise eine Dateipfadangabe). Bei integrierten Varianten sind zusätzlich zur Adressierung, die Entitäten zu referenzieren, welche die Varianten darstellen, um diese von den restlichen Teilen des Softwaredokuments

unterscheiden zu können. Das Vorhandensein dieses Konzepts erhöht die Flexibilität bei der Aufgabenplanung. Beispielsweise kann das Konzept der separierten Varianten gewählt werden, wenn Ressourcen wie etwa Arbeitsspeicher knapp sind. Zudem wird dadurch die Portierbarkeit erhöht. So kann zum Beispiel durch separierte Varianten der Realisierungsauftrag an weitere Zulieferer übergeben werden.

Als Anwendungsfall sei das folgende Beispiel betrachtet: Abbildung 4.3 zeigt ein Simulink-Modell, das die beiden Varianten für das Fahrzeugzugangssystem beinhaltet: die Zentralverriegelung und der Komfortzugang. Da die Funktionen in einem Simulink-Modell gemeinsam modelliert werden, sind sie somit integrierte Varianten. Im Variabilitätsmodell wird dies durch einen entsprechenden Eintrag festgelegt. Der Eintrag beinhaltet einen Verweis auf das Ursprungsdokument mit der Angabe des Bezeichners der Variante innerhalb des Dokuments. Die folgende beispielhafte Darstellung illustriert diese Situation:

- Variationspunkt: Fahrzeugzugangssystem
  - Gruppe: Funktionen
    - Variante: Zentralverriegelung
      - integriert: `$(workspace)$/fahrzeugzugangssystem.mdl -> zentralverriegelung`
    - Variante: Komfortzugang
      - integriert: `$(workspace)$/fahrzeugzugangssystem.mdl -> komfortzugang`

Restriktionsmodellierung

Definition variabler Eigenschaften Die bisher eingeführten Konzepte reichen noch nicht aus, um variable Eigenschaften für Gruppen im Variabilitätsmodell auszudrücken. Es ist beispielsweise nicht möglich, Optionalität oder Alternativität auszudrücken. In diesem Zusammenhang ergibt sich also die Frage, wie beliebige variable Eigenschaften einer Gruppe definiert werden können.

Hierfür ist die Einführung von sogenannten Gruppenkardinalitäten erforderlich. Eine Gruppenkardinalität ist dabei ein Intervall $[i..j]$, mit $i \leq j$ und $i, j \in N$ und legt die variablen Eigenschaften einer Gruppe fest. Diese sind unter anderem (1) Optionalität innerhalb von Gruppen (Gruppenkardinalität $[0..1]$), (2) Alternativität innerhalb von Gruppen (Gruppenkardinalität $[1..1]$) und (3) Partitionalität innerhalb von Gruppen (Gruppenkardinalität $[m..n]$). Letzteres erlaubt beliebige Zerlegungen von Gruppen. Variable Eigenschaften einer Gruppe können somit sehr flexibel ausgedrückt werden, da viele Ausdrucksmöglichkeiten zur Verfügung stehen.

Als Anwendungsfall sei das folgende Beispiel betrachtet: Die Zentralverriegelung und der Komfortzugang sind Bestandteil einer Gruppe. Beide Funktionen stehen im gegenseitigen Ausschluss zueinander. Durch die Gruppenkardinalität $[1..1]$ kann dies zum Ausdruck gebracht werden. Folgende beispielhafte Notation illustriert diese Situation:

- Variationspunkt: Fahrzeugzugangssystem
  - Gruppenkardinalität: $[1..1]$
    - Variante: Zentralverriegelung
    - Variante: Komfortzugang

Die Angabe der Gruppenkardinalität oberhalb der Varianten umschließt diese ein und drückt aus, dass beide Varianten im gegenseitigen Ausschluss stehen.

Neben variablen Gruppeneigenschaften können auch variable Varianteneigenschaften existieren. Dies ist im Rahmen der bisherigen Erläuterungen noch nicht möglich. Demnach können keine verbindlichen, optionalen oder multiplen Varianten angegeben werden. Es stellt sich also die Frage, wie beliebige variable Eigenschaften von Varianten definiert werden können.

Ähnlich wie bei Gruppenkardinalitäten werden für Varianten sogenannte Variantenkardinalitäten eingeführt. Eine Variantenkardinalität ist dabei ein Intervall $[i..j]$, mit $i \leq j$ und $i, j \in N$ und definiert die variablen Eigenschaften einer Variante. Diese sind unter anderem (1) Optionalität von Varianten (Variantenkardinalität $[0..1]$), (2) Verbindlichkeit von Varianten (Variantenkardinalität $[1..1]$) und (3) Multiplizität von Varianten (Variantenkardinalität $[m..n]$). Variable Eigenschaften einer Variante sind somit sehr flexibel ausdrückbar. Es stehen beliebig viele Möglichkeiten zur Festlegung des Intervalls vor. Das Konzept erhöht zudem die Verständlichkeit von Varianteneigenschaften.

Als Anwendungsfall sei das folgende Beispiel betrachtet: Die Zentralverriegelung und der Komfortzugang besitzen eine optionale Eigenschaft. Durch die Variantenkardinalität [0..1] kann dies zum Ausdruck gebracht werden. Folgende beispielhafte Notation kann dies zum Ausdruck bringen:

- Variationspunkt: Fahrzeugzugangssystem
  - Gruppenkardinalität: $[1..1]$
    - Variante: Zentralverriegelung (Variantenkardinalität: $[0..1]$)
    - Variante: Komfortzugang (Variantenkardinalität: $[0..1]$)

Es sei an dieser Stelle zu beachten, dass die Variantenkardinalität auch von der Gruppenkardinalität abhängt. So kann nicht gleichzeitig für die Zentralverriegelung und den Komfortzugang die Kardinalität $[1]$ angegeben werden, da die Gruppenkardinalität eine Alternativität voraussetzt.

Restriktionssprachen In den vorangegangenen Abschnitten wurde das Erfordernis der Gruppen- bzw. Variantenkardinalitäten festgestellt, um variable Eigenschaften von Gruppen bzw. Varianten festzulegen. Diese stellen somit eine besondere Form von Restriktionen dar, da durch die Angabe von Kardinalitäten das Modell entsprechend eingeschränkt wird. In vielen Fällen sind allerdings auch Restriktionen über Gruppen- bzw. Variationspunktgrenzen hinweg erforderlich. Allein mit den Konzepten der Gruppen- bzw. Variantenkardinalitäten müsste das Modell umorganisiert

werden. Dies führt allerdings zu unlogischen Strukturierungen, die nicht ohne Weiteres nachzuvollziehen sind. Das Modell wird also dadurch weniger verständlich und unübersichtlich. Demnach stellt sich in diesem Zusammenhang die Frage, wie Restriktionen über Gruppen- bzw. Variationspunktgrenzen hinweg formuliert werden können.

Zur Lösung dieser Situation Bedarf es an einer formalen Restriktionssprache, sodass derartige Restriktionen ausgedrückt werden können. Diese Sprache muss ausdrucksstark und zugleich einfach sein. Sie muss analysierbar sein, um diese für weitere Zwecke einzusetzen. Wichtig ist vor allem, dass Restriktionen im Variabilitätsmodell zentral gesammelt werden. So wird es deutlich einfacher die Restriktionen wiederzufinden und zudem werden unnötige Überlegungen, wo eine Restriktion aufgeführt werden sollte, vermieden. Durch die Restriktionssprache können folglich beliebige und flexible restriktive Aussagen in Bezug auf Variabilität getroffen werden. Das Variabilitätsmodell bleibt dadurch übersichtlicher, da unlogische Strukturierungen vermieden werden.

Als Anwendungsfall sei das folgende Beispiel betrachtet: Bei Selektion des Komfortzugangs sind im Außenbereich des Fahrzeugs Sendeantennen zu installieren. Im Innenbereich sind noch zusätzlich Sende- und Empfangsantennen anzubringen. Da der Komfortzugang und die Antennen in verschiedenen Gruppen strukturiert sind, Bedarf es an Restriktionen, um den Zusammenhang zwischen dem Komfortzugang und den Antennen herzustellen. Das folgende Beispiel illustriert diese Abhängigkeit:

- Variationspunkt: Fahrzeugzugangssystem
  - Gruppe: Funktionen
    - Variante: Zentralverriegelung
    - Variante: Komfortzugang
  - Gruppe: Sensoren
    - Variante: Empfangsantenne außen
    - Variante: Sendeantenne außen
    - Variante: Sende- und Empfangsantenne innen
  - Restriktionen
    - Komfortzugang **erfordert** Sendeantenne außen
    - Komfortzugang **erfordert** Sende- und Empfangsantenne innen

4.1.2.2. Bindung

Als Bindungsmechanismus wurden im Rahmen dieser Arbeit zwei Arten betrachtet: (1) die Selektion mit einer Konfigurationsmaschine und (2) die Generierung mit einer Inferenzmaschine. Die wichtigsten Aspekte zur Realisierung beider Arten werden nachfolgend genauer erläutert.

Konfigurierung

Konfiguration der Variabilität Das Variabilitätsmodell bietet die Möglichkeit, Variationspunkte und Varianten zu erfassen und geeignet zu strukturieren. Im Entwicklungsprozess wird es früher oder später zu der Situation kommen, dass Variabilität aufgelöst werden muss, d.h. Varianten gebunden werden müssen. Beispielsweise ist dies der Fall, wenn das Verhalten eines Simulink-Modells simuliert wird. Hierfür ist es erforderlich, die Simulation für eine Variante durchzuführen. Zu diesem Zweck muss eine Variante also vor der Simulation gebunden werden. Das Variabilitätsmodell bietet allerdings keine Möglichkeit, Varianten zu binden. Daher ist die zentrale Frage, wie Varianten gebunden werden können.

Zur Unterstützung der Bindung von Varianten wird ein Konfigurierungsprozess erforderlich sein, der als eigenständige Aktivität durchgeführt wird. Dadurch wird die Modellierung von der Bindung geeignet getrennt. Demnach ist ein Konfigurationsmodell notwendig, das als separates Modell bereitgestellt wird und nur aus den für die Konfigurierung relevanten Informationen besteht. Diese sind (1) Variationspunkte, (2) Gruppenkardinalitäten, (3) Varianten und (4) Variantenkardinalitäten. Die Informationen müssen aus dem Variabilitätsmodell extrahiert werden. Bei Änderungen im Variabilitätsmodell muss das Konfigurationsmodell entsprechende Anpassungen automatisch durchführen. Auf diese Weise bleiben beide Modelle stets synchron. Als Bindungsmechanismus eignet sich besonders die Selektion, da sie einfach und effizient ist. Voraussetzung hierfür ist, dass die Varianten bereits vollständig realisiert sind, sodass auf diese mittels Selektion zugegriffen werden kann. Zusätzlich zur Selektion muss die Angabe von Variantenkardinalitäten möglich sein.

Als Anwendungsfall sei das folgende Beispiel betrachtet: Zur Bindung der Varianten des Fahrzeugzugangssystems wird ein Konfigurierungsprozess angestoßen. Das Konfigurationsmodell extrahiert die relevanten Daten aus dem Variabilitätsmodell. Aus dem Konfigurationsmodell wird der Komfortzugang mit Kardinalität 1 selektiert. Das folgende Beispiel visualisiert das entsprechende Konfigurationsmodell:

- Konfiguration
  - Variationspunkt: Fahrzeugzugangssystem
    - Gruppenkardinalität: [1..1]
      - Variante: Zentralverriegelung | [0] | ☐ |
      - Variante: Komfortzugang | [1] | ☑ |

Unterstützung bei der Konfigurierung Durch die Möglichkeit der Restriktionsformulierung in Variabilitätsmodellen wird der Konfigurierungsprozess deutlich erschwert. Der Grund hierfür ist, dass gültige bzw. ungültige Aktionen, die durch Restriktionen entstehen, manuell ausgewertet werden müssen. Bei einer Vielzahl von Restriktionen ist dies nahezu unmöglich. Daher stellt sich die Frage, wie diese Auswertung automatisiert durchgeführt werden kann.

Zur Lösung dieses Problems ist eine Validierung jedes einzelnen Konfigurierungsschritts erforderlich. Hierfür müssen sämtliche Restriktionen zur Überprüfung herangezogen werden. Somit kann die Gültigkeit jeder Selektion bzw. Deselektion in jedem Konfigurierungsschritt validiert werden. Zudem ist eine Erkennung von Implikationen notwendig, die automatisch ausgeführt werden muss.

Als Anwendungsfall sei das folgende Beispiel betrachtet: In einem Variabilitätsmodell wurden Restriktionen festgelegt, die bei Selektion des Komfortzugangs die Sendeantennen im äußeren Bereich, sowie Sende- und Empfangsantennen im Inneren des Fahrzeugs erfordern. Das entsprechende Variabilitätsmodell wird wie folgt notiert:

- Variationspunkt: Fahrzeugzugangssystem
  - Gruppenkardinalität: [1..1]
    - Variante: Zentralverriegelung
    - Variante: Komfortzugang
  - Gruppenkardinalität: [1..10]
    - Variante: Empfangsantenne außen
    - Variante: Sendeantenne außen
    - Variante: Sende- und Empfangsantenne innen
  - Restriktionen
    - Komfortzugang **erfordert** Sendeantenne außen
    - Komfortzugang **erfordert** Sende- und Empfangsantenne innen

Bei Selektion des Komfortzugangs in einem Konfigurierungsschritt wird zunächst überprüft, ob die Aktion gültig ist. Dazu werden alle definierten Restriktionen herangezogen. Dabei wird ermittelt, dass sowohl Sendeantennen als auch Sende- und Empfangsantennen erforderlich sind. Diese Implikationen werden ausgeführt, indem entsprechende Varianten im Konfigurationsmodell automatisch selektiert werden. Das Ergebnis der Validierung in einem Konfigurationsmodell sieht dann wie folgt aus:

- Konfiguration
  - Variationspunkt: Fahrzeugzugangssystem
    - Gruppenkardinalität: [1..1]
      - Variante: Zentralverriegelung | [0] | ☐ |
      - Variante: Komfortzugang | [1] | ☑ |
    - Gruppenkardinalität: [1..10]
      - Variante: Empfangsantenne außen | [0] | ☐ |
      - Variante: Sendeantenne außen | [4] | ☑ |
      - Variante: Sende- und Empfangsantenne innen | [5] | ☑ |

Generierung

Inferenz durch Selektion gebundener Softwaredokumente Durch den Konfigurierungsprozess werden Varianten selektiert und an Softwaredokumente gebunden. Dies reicht allerdings noch nicht aus, das gebundene Softwaredokument zu erhalten. Hierfür müssen sämtliche Variationspunkte aufgelöst werden und durch die Varianten ersetzt werden. Ohne dies können keine Simulationen, Verifikationen und Tests durchgeführt werden. Fehler im Softwaredokument sind somit nicht auszuschließen. Demnach folgt die Frage, wie gebundene Softwaredokumente aus der Konfigurierung heraus gebildet werden können.

Zu diesem Zweck ist ein Inferierungsprozess erforderlich, der die entsprechenden Softwaredokumente erzeugt. Für diesen Prozess ist eine Inferenzmaschine einzusetzen, der die Inferierung automatisiert durchführt. Die Generierung als Bindungsmechanismus ist an dieser Stelle geeignet. Durch das Binden von Varianten und der Inferierung wird es zu jedem Zeitpunkt im Entwicklungsprozess möglich, Überprüfungen durchzuführen. Die Inferierung erhöht also die Flexibilität im Entwicklungsprozess. Zudem können Fehler früh erkannt oder Entwurfsentscheidungen angepasst werden.

Als Anwendungsfall sei das folgende Beispiel betrachtet: Für ein Funktionsnetz wird aufgrund von Arbeitsteilungen eine Variante für die Zentralverriegelung und eine weitere Variante für den Komfortzugang durch jeweils eine Konfiguration gebunden. Die Inferierung erzeugt hieraus zwei Funktionsnetze, die je eine Konfiguration repräsentieren. Beide Funktionsnetze können nun unabhängig voneinander verteilt werden. Dies ist insbesondere für einen Automobilhersteller hilfreich, der die Funktionen bei verschiedenen Zulieferern realisieren lässt.

4.2. Modellierung

In dem vorangegangenen Abschnitt wurden in Bezug auf Variabilität wesentliche Fragestellungen eingeführt, deren Bearbeitung das Modellieren und Verwalten von Variationen im Entwicklungsprozess geeignet unterstützt. So wurden die Probleme beschrieben, die Nachteile erläutert, der Bedarf ermittelt und jeweils in einem Anwendungsfall illustriert. In diesem Abschnitt wird die konzeptionelle Basis zur Realisierung der Erfordernisse hinsichtlich der Modellierung von Variabilität aufgestellt. Im nächsten Abschnitt wird dann auf die Bindung von Variabilität eingegangen.

4.2.1. Variabilitätsmodell

Variabilitätsmodellierung ist eine Aktivität, in der Variabilität innerhalb eines Softwaresystems oder eines Softwaredokuments identifiziert sowie in einem Modell dokumentiert und repräsentiert wird. Das Ergebnis dieser Aktivität wird als *Variabilitätsmodell* bezeichnet.

In Abschnitt 4.1.2.1 wurde diesbezüglich das Auswahlmodell als geeignet betrachtet. Es stellt die Basis bei der Modellierung dar. Erweiterte Konzepte, wie

etwa (1) Strukturierungsmaßnahmen durch Gruppierungen, (2) Berücksichtigung von Abstraktionsebenen, (3) Unterstützung bei der Modellierung, (4) Modellierung von Variabilitätsmechanismen und (5) Modellierung von Variantenarten, vervollständigen das Variabilitätsmodell. Im Folgenden werden diese Konzepte detailliert beschrieben.

4.2.1.1. Konkrete Syntax

Abbildung 4.7 illustriert ein Variabilitätsmodell, das im Rahmen dieser Arbeit entwickelt wurde. Es ist ein erweitertes Auswahlmodell, das die Anforderungen aus Abschnitt 4.1.2.1 erfüllt. Anstatt eines aufgespannten Baums wird das Modell in Form einer Baumliste dargestellt. Das Lesen einer deratigen Struktur ist deutlich einfacher, als ein komplex aufgespannter Baum mit beliebig vielen Querbeziehungen.

Die Wurzel beinhaltet die Beschreibung der modellierten Abstraktionsebene sowie die Bezeichnung des konkret erfassten Softwaredokuments. In Kapitel 3 wurden die Abstraktionsebenen Funktionsebene, Architekturebene und Codeebene eingeführt. Alternativ wäre hier auch die Unterteilung in Arbeitsbereiche möglich. In der Funktionsebene wird der konzeptionelle Entwurf in Form eines Funktionsnetzes realisiert. Auf Architekturebene wurden Simulink-Modelle und Steuergerätenetze eingeführt. Schließlich können auf Codeebene verschiedene Programmiersprachen wie etwa C/C++, Java oder Ada spezifiziert werden. Somit ergeben sich für die Wurzel folgende Möglichkeiten:

- `Funktionsebene: Funktionsnetz`
- `Architekturebene: Simulink`
- `Architekturebene: Steuergerätenetz`
- `Codeebene: C/C++`
- `Codeebene: Java`
- `Codeebene: Ada`
- ...

Für Abbildung 4.3 würde das Variabilitätsmodell die Festlegung `Architekturebene: Simulink` beinhalten.

Darauffolgend können die Variationspunkte des spezifizierten Softwaredokuments beschrieben werden. Die Beschreibung eines Variationspunktes beinhaltet zunächst den im Softwaredokument eingesetzten Variabilitätsmechanismus. Dokumentspezifische Variabilitätsmechanismen werden im späteren Verlauf dieser Arbeit genauer beschrieben. An dieser Stelle sei beispielhaft der Variabilitätsmechanismus für das Fahrzeugzugangssystem aus Abbildung 4.3 herangezogen. In dieser Abbildung werden die zwei Varianten `zentralverriegelung` und `komfortzugang` durch das `If Action`-Konstrukt realisiert. So würde als Variabilitätsmechanismus im Variabilitätsmodell genau dieses `If-Action`-Konstrukt angegeben werden. Die Informationen, die bisher erfasst wurden, sind nun Folgende:

- Architekturebene: Simulink
- Variationspunkt: Fahrzeugzugangssystem
  - Variabilitätsmechanismus: If Action

Die genaue Zusammensetzung des If Action-Variabilitätsmechanismus' kann durch Subknoten detailliert beschrieben werden, wie etwa die verwendeten Simulink-Blöcke, Bezeichner, Werte etc. In Abbildung 4.7 ist dies nicht weiter ausgeführt. Für das Beispiel aus Abbildung 4.3 würde sich folgende Komposition ergeben:

- Architekturebene: Simulink
- Variationspunkt: Fahrzeugzugangssystem
  - Variabilitätsmechanismus: If Action
    - Constant: aktivierung
    - If: if
    - If Action Subsystem: zentralverriegelung
    - If Action Subsystem: komfortzugang
    - Merge: datenabarbeitung_verriegeln
    - Merge: datenabarbeitung_entriegeln

Als Nächstes wird im Variabilitätsmodell der Bindungsmechanismus beschrieben. In Abbildung 4.4 wurde der Bindungsmechanismus als eine Realisierung der Variantenbindung definiert. Es wurden dabei vier Bindungsmechanismen - (1) Selektion, (2) Aktivierung, (3) Substitution und (4) Generierung - sowie sechs Auswertungsmaschinen - (1) Konfigurationsmaschine, (2) Updatemaschine, (3) Solver, (4) Präprozessoren, (5) Compiler und (6) Ausführungsmaschinen - identifiziert. Wird ein Bindungsmechanismus definiert, muss weiterhin eine Auswertungsmaschine angegeben werden. Für das Beispiel aus Abbildung 4.3 ergibt sich als Bindungsmechanismus die Aktivierung, da durch eine Bedingung, realisiert im If-Block, ohne Benutzerinteraktion je nach Auswertung der Bedingung die eine oder andere Variante aktiviert wird. Als Auswertungsmaschine kann beispielsweise eine Ausführungsmaschine der Programmiersprache C wie etwa gcc festgelegt werden. Es sei an dieser Stelle angemerkt, dass hier durchaus auch ein Simulink-Solver wie etwa ode3 angegeben werden kann, wenn der Bindungsmechanismus für die Simulation realisiert wurde. Demnach sind nun folgende Informationen für das Beispiel erfasst:

- Architekturebene: Simulink
- Variationspunkt: Fahrzeugzugangssystem
  - Variabilitätsmechanismus: If Action
    - Constant: aktivierung
    - If: if

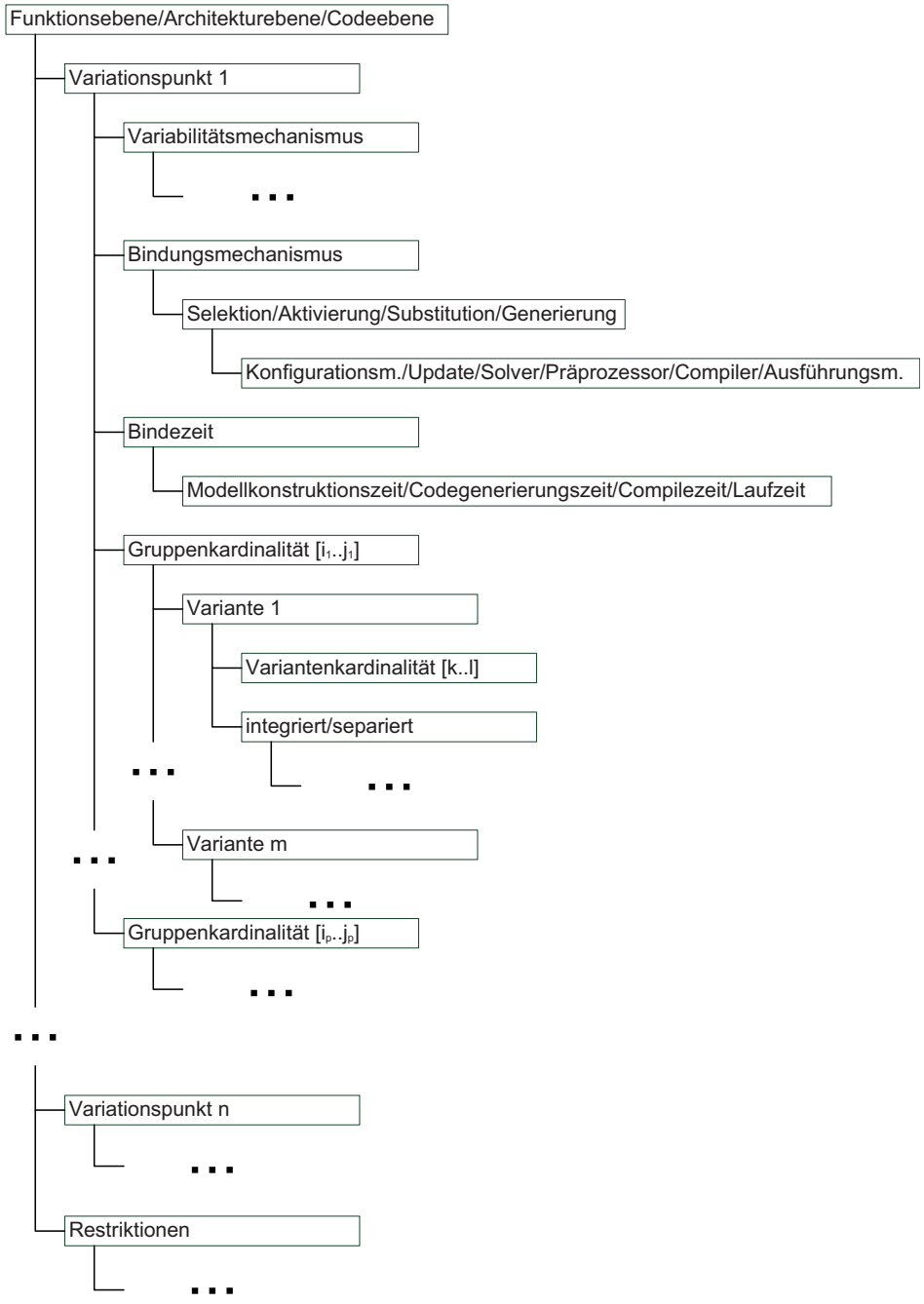


Abbildung 4.7.: Das Variabilitätsmodell mit den zu modellierenden Konzepten

        * If Action Subsystem: zentralverriegelung
        * If Action Subsystem: komfortzugang
        * Merge: datenabarbeitung_verriegeln
        * Merge: datenabarbeitung_entriegeln
    - Bindungsmechanismus: Aktivierung
        * Auswertungsmaschine: Ausführungsmaschine gcc

Weiterhin wird die Bindezeit im Variabilitätsmodell spezifiziert. Sie ist primär abhängig von der Auswertungsmaschine, da diese nur zu bestimmten Bindezeiten eingesetzt werden kann. Ist beispielsweise die Modellkonstruktionszeit als Bindezeit definiert, so wird sich typischerweise eine Updatemaschine eignen. Für das Modell in Abbildung 4.3 wird aufgrund der Ausführungsmaschine die Bindung zur Laufzeit erfolgen. Daraus ergibt sich die folgende Zusammenstellung:

- Architekturebene: Simulink
- Variationspunkt: Fahrzeugzugangssystem
    - Variabilitätsmechanismus: If Action
        * Constant: aktivierung
        * If: if
        * If Action Subsystem: zentralverriegelung
        * If Action Subsystem: komfortzugang
        * Merge: datenabarbeitung_verriegeln
        * Merge: datenabarbeitung_entriegeln
    - Bindungsmechanismus: Aktivierung
        * Auswertungsmaschine: Ausführungsmaschine gcc
    - Bindezeit: Laufzeit

Anschließend werden die Varianten spezifiziert. Hierfür wird ein Gruppierungskonzept angewendet. Dadurch können zusammenhängende Varianten in Gruppen zusammengefasst werden. Jede Gruppe hat eine *Gruppenkardinalität*. Eine Gruppenkardinalität ist dabei ein Intervall $[i..j]$, mit $i \leq j$ und $i, j \in N$. Sie legt die variablen Eigenschaften der Gruppe fest. Im Folgenden sind einige Beispiele mit ihren Bedeutungen aufgelistet:

- Gruppenkardinalität $[0..1]$: Die Varianten in dieser Gruppe sind optional; es kann allerdings stets nur maximal eine Variante im Softwaredokument enthalten sein.
- Gruppenkardinalität $[1..1]$: Die Varianten in dieser Gruppe sind alternativ zueinander; es muss immer genau eine Variante im Softwaredokument enthalten sein.

- Gruppenkardinalität $[1..m]$: Es muss mindestens eine Variante im Softwaredokument enthalten sein. Es können aber auch bis zu m Varianten enthalten sein.
- Gruppenkardinalität $[n..m]$: Es müssen mindestens n Varianten im Softwaredokument enthalten sein. Es können aber auch bis zu m Varianten enthalten sein.

Das Konzept der Gruppenkardinalität stammt aus den Arbeiten von *Czarnecki et al.* [CHE05a, CK05]. Es ist ein geeignetes Konzept, um eine häufig vorkommende Variabilität auf einfache Art auszudrücken. Die zwei Varianten `zentralverriegelung` und `komfortzugang` aus Abbildung 4.3 sind im Simulink-Modell so realisiert, dass sie im gegenseitigen Ausschluss stehen. Dies liegt daran, dass die Realisierung des Komfortzugangs bereits den Anteil der Funktionslogik der standardmäßigen Zentralverriegelung beinhaltet und daher auch bei der Auswahl des Komfortzugangs auch die Zentralverriegelung vorhanden ist. Als Gruppenkardinalität würde sich daher $[1..1]$ ergeben.

Weiterhin wird zusätzlich zur Gruppenkardinalität eine *Variantenkardinalität* eingeführt. Eine Variantenkardinalität ist dabei ein Intervall $[k..l]$, mit $k \leq l$ und $k, l \in N$. Sie gibt die Anzahl möglicher Ausprägungen der Variante wieder. Dieses Konzept ist erforderlich, da das elektronische System oftmals die gleiche Funktion für verschiedene Sensoren bzw. Aktuatoren benötigt. Beispielsweise ist die Funktion zur Datenaufbereitung der im Fahrzeug verteilten Antennen mehrfach erforderlich. Diese Funktion kann daher repliziert und an die entsprechende Antenne angepasst werden. Für die Varianten `zentralverriegelung` und `komfortzugang` ergibt sich jeweils die Variantenkardinalität $[0..1]$. An dieser Stelle wird deutlich, dass es Unterschiede zwischen benutzersichtbaren Features und ihrer Realisierung gibt. Während die Zentralverriegelung eine Grundausstattung ist, wird sie im Simulink-Modell als ein optionaler Baustein realisiert. Der Grund hierfür ist, dass der Komfortzugang bereits die Zentralverriegelung beinhaltet, sodass bei Auswahl des Komfortzugangs auch automatisch die Zentralverriegelung vorhanden ist. Die Entwurfsentscheidung sei an dieser Stelle nicht weiter diskutiert. Es kann allerdings festgehalten werden, dass derartige Strukturen in der Praxis häufig auftreten.

Außerdem wird die Variantenart spezifiziert, also ob es sich um eine integrierte oder separierte Variante handelt. Bei einer integrierten Variante wird der Pfad des Softwaredokuments im Dateisystem angegeben. Zudem wird der Bezeichner angegeben, mit der die Variante identifiziert werden kann. Bei einer separierten Variante reicht der Pfad alleine bereits aus, da die Varianten dadurch bereits eindeutig identifiziert werden können. Sowohl `zentralverriegelung` als auch `komfortzugang` sind jeweils integrierte Varianten. Für diese müssen jeweils der Pfad und der Bezeichner angegeben werden. Insgesamt ergeben sich also aus den obigen Erläuterungen folgende Informationen:

- `Architekturebene: Simulink`
- `Variationspunkt: Fahrzeugzugangssystem`
  - `Variabilitätsmechanismus: If Action`
    - `Constant: aktivierung`

```
    * If: if
    * If Action Subsystem: zentralverriegelung
    * If Action Subsystem: komfortzugang
    * Merge: datenabarbeitung_verriegeln
    * Merge: datenabarbeitung_entriegeln
- Bindungsmechanismus: Aktivierung
    * Auswertungsmaschine: Ausführungsmaschine gcc
- Bindezeit: Laufzeit
- Gruppenkardinalität: [1..1]
    * Variante: Zentralverriegelung
        · Variantenkardinalität: [0..1]
        · integriert: ${workspace}$/fahrzeugzugangssystem.mdl
          -> zentralverriegelung
    * Variante: Komfortzugang
        · Variantenkardinalität: [0..1]
        · integriert: ${workspace}$/fahrzeugzugangssystem.mdl
          -> komfortzugang
```

Schließlich können im Variabilitätsmodell Restriktionen angegeben werden, die nicht durch die oben beschriebenen Konzepte spezifiziert werden können. So benötigt beispielsweise der Komfortzugang weitere Steuergeräte und Antennen. Durch die Beschreibung entsprechender Regeln können derartige Sachverhalte ausgedrückt werden. In Abbildung 4.7 ist dies nicht weiter detailliert. Dies wird im weiteren Verlauf dieser Arbeit in Abschnitt 4.2.2 genauer beschrieben.

Abbildung 4.8 illustriert das bisher entwickelte Ergebnis am Beispiel des Fahrzeugzugangsystems aus Abbildung 4.3. Die besonderen Eigenschaften dieses Modells sind: (1) die Erfassung aller relevanten Variabilitätskonzepte wie etwa Variabilitätsmechanismus, Bindungsmechanismus, Bindezeit etc., (2) die fest vorgeschriebene Reihenfolge der Modellierung sowie (3) die enge Assoziation zu den Softwaredokumenten. Insbesondere kommt dieses Variabilitätsmodell zum Tragen, wenn es in Kombination mit den Softwaredokumenten betrachtet wird.

4.2.1.2. Abstrakte Syntax

Nachdem in Abschnitt 4.2.1.1 die konkrete Syntax des vorgeschlagenen Variabilitätsmodells vorgestellt wurde, wird in diesem Abschnitt die abstrakte Syntax, also das Metamodell, beschrieben. Die abstrakte Syntax definiert die weiter oben erläuterten Konzepte in Form eines formalen Klassendiagramms (Abbildung 4.9) [GA02]. Die konkrete Syntax korrespondiert also zur abstrakten Syntax. Mit anderen Worten ist sie eine Instanz der abstrakten Syntax.

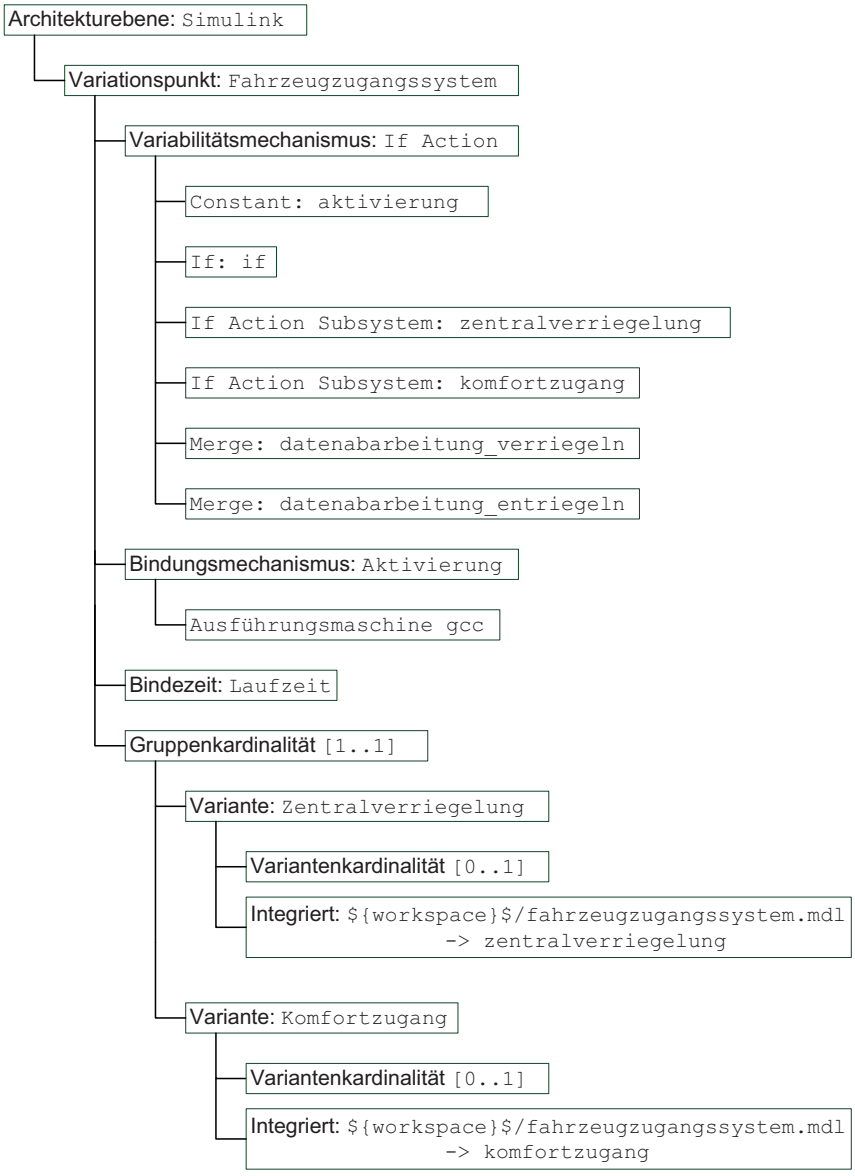


Abbildung 4.8.: Das Variabilitätsmodell für das Fahrzeugzugangsystem auss Abbildung 4.3

Alle beschriebenen Konzepte sind in der abstrakten Syntax erfasst. Das Variabilitätsmodell `VariabilityModel` setzt sich aus einer Liste von Variationspunkten (`VariationPoint`) und aus Containern für Restriktionsregeln (`AuditContainer`) zusammen. Der `AuditContainer` wird in Abschnitt 4.2.2 weiter verfeinert. Außerdem wird im Variabilitätsmodell die Abstraktionsebene spezifiziert (`abstractionLevel : AbstractionLevel`). Der Typ `AbstractionLevel` ist ein Aufzählungstyp mit den Werten `FUNCTIONNET`, `SIMULINK` und `C`.

Der Variationspunkt setzt sich aus dem Variabilitätsmechanismus (`VariabilityMechanism`), dem Bindungsmechanismus (`BindingMechanism`), der Bindezeit (`BindingTime`) und der Gruppenkardinalität (`GroupCardinality`) für Varianten zusammen. `VariabilityMechanism` beschreibt den zur Realisierung der Variabilität verwendeten Mechanismus (`pattern : String`) und verwaltet eine Liste an Elementen (`Element`), die bei der Realisierung verwendet wurden. `BindingMechanism` spezifiziert einen Bindungsmechanismus (durch die Klasse `BindingMechanismType` mit den Werten `SELECTION`, `ACTIVATION`, `SUBSTITUTION` und `GENERATION`) und eine zugehörige Auswertungsmaschine (`EvalEngine`). `BindingTime` beinhaltet die Bindezeit, die durch den Variabilitätsmechanismus ermöglicht wird. Als mögliche Werte können hier über den Aufzählungstypen `BindingTimeType` die Werte `MODELCONTSTRUCTION`, `CODEGENERATION`, `COMPILATION` und `RUN` festgelegt werden. Durch die Klasse `GroupCardinality` werden für die untergeordneten Varianten eine Gruppenkardinalität festgelegt. Da es zudem auch eine Variantenkardinalität gibt, wurde eine allgemeine Klasse `Cardinality` eingeführt, von der die Klassen `GroupCardinality` und `Variant` erben. Die Varianten werden dann jeweils durch die Gruppenkardinalität gruppiert. Für jede Variante wird der Typ durch den Aufzählungstyp `VariantType` mit den Werten `INTEGRATED` und `SEPERATED` spezifiziert, der Pfad der Variante angegeben (`path : String`) und wenn erforderlich (bei integrierten Varianten) der Bezeichner (`identifier : String`) festgelegt.

Schließlich sei noch erwähnt, dass durch die Kompositionsrelationen der Baum für das Variabilitätsmodell aufgespannt wird. Das beschriebene Metamodell stellt die Basis für die im späteren Verlauf der Arbeit noch zu erläuternde Realisierung dar (vgl. Abschnitt 4.4).

4.2.2. Restriktionsmodell

In Abschnitt 4.2.1.1 wurde bereits das Gruppierungskonzept mit der Angabe von Gruppenkardinalitäten vorgestellt. Dadurch ist es möglich, restriktive Aussagen innerhalb einer Gruppe von Varianten zu stellen. Beispielsweise stehen die zwei Varianten `Zentralverriegelung` und `Komfortzugang` aus Abbildung 4.8 im gegenseitigen Ausschluss. Diese Restriktion wird durch die Gruppenkardinalität `[1..1]` ausgedrückt. In vielen Fällen ist allerdings auch die Definition von Restriktionen über Gruppengrenzen hinweg erforderlich. So sollte es möglich sein, Restriktionen zwischen: (1) Variationsgruppen innerhalb eines Variationspunktes, (2) Variationspunkte derselben Abstraktionsebene sowie (3) Variationspunkte auf unterschiedlichen Abstraktionsebenen auszudrücken.

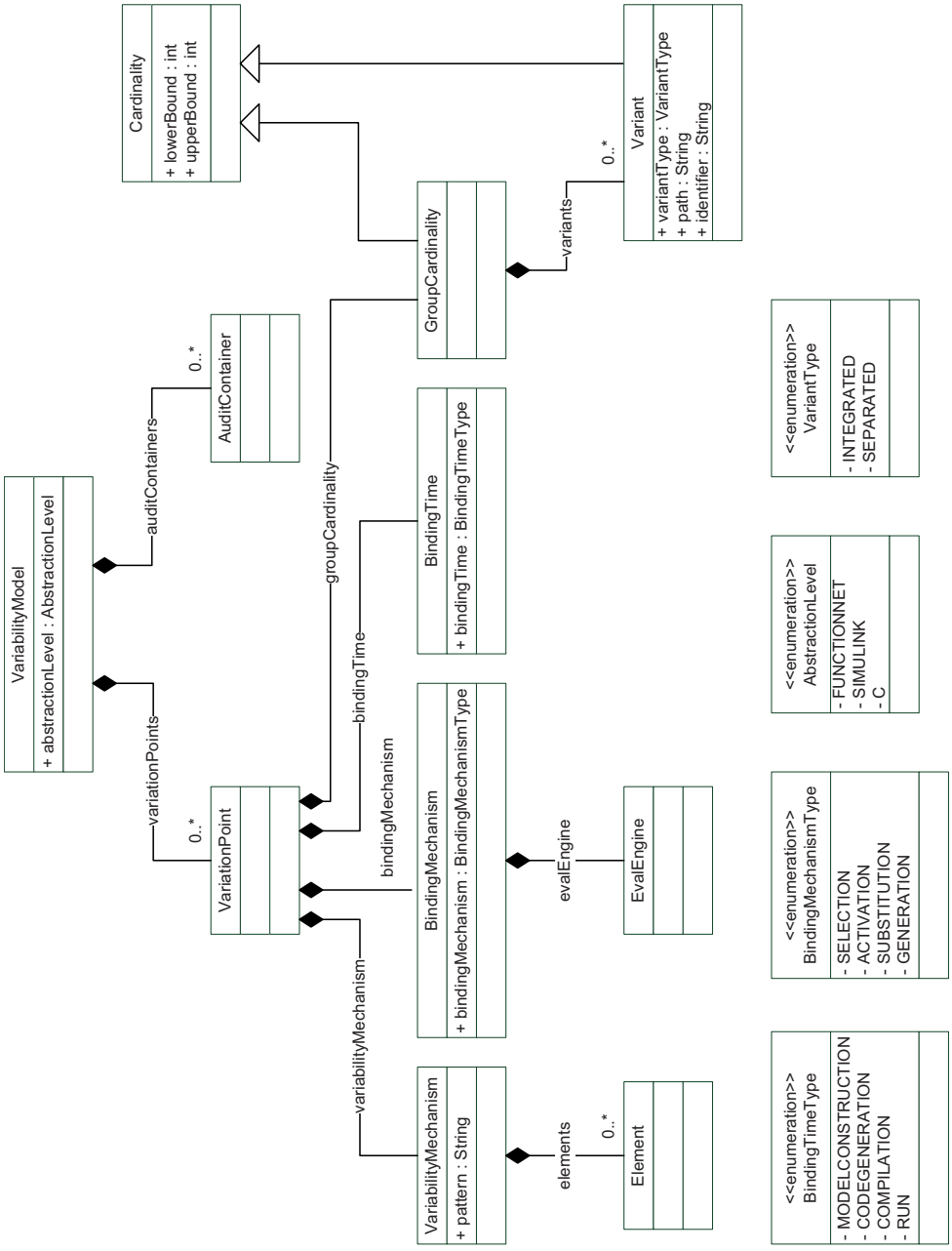


Abbildung 4.9.: Die abstrakte Syntax für das Variabilitätsmodell in Form eines Klassendiagramms

Abbildung 4.10 illustriert ein Beispiel für Restriktionen zwischen Variationspunkten derselben Abstraktionsebene. Zur besseren Lesbarkeit sind nicht relevante Bestandteile des Variabilitätsmodells ausgeblendet. Das `Fahrzeugzugangsystem` ist eine für den Kunden angebotene Funktionalität. Die Antennen sind Bestandteil dieser Funktionalität, da sie die Anfragen empfangen und weiter verarbeiten. Je nachdem, ob eine `Zentralverriegelung` oder ein `Komfortzugang` im Fahrzeug integriert ist, sind verschiedene Antennentypen mit unterschiedlicher Anzahl erforderlich. So benötigt die Zentralverriegelung lediglich eine `Empfangsantenne außen` zum Datenempfang. Der Komfortzugang hingegen benötigt vier weitere `Sendeantennen außen`, die Daten senden sowie fünf weitere `Sende- und Empfangsantennen innen`, die Daten senden und empfangen können. Umgekehrt benötigen die Varianten `Sendeantenne außen` und `Sende- und Empfangsantenne innen` die Komfortzugangsfunktion.

Es ist daher ein Ausdrucksmittel erforderlich, um derartige Restriktionen formulieren zu können. Typische Anwendungsfälle sind die folgenden:

- Eine Variante erfordert die Existenz einer anderen Variante (Implikation).
- Eine Variante schließt die Existenz einer anderen Variante aus (Exklusion).
- Varianten implizieren sich gegenseitig (Äquivalenz).
- Varianten schließen sich gegenseitig aus (Antivalenz).

Darüber hinaus können oben genannte Fälle durch Angaben von Kardinalitäten erweitert werden. So kann beispielsweise eine Variante die Existenz mehrerer Varianten implizieren. Besonders wichtig hierbei ist es, die Komplexität derartiger Restriktionen gering zu halten und die Intuitivität zu steigern. Eine wesentliche Frage diesbezüglich ist, wo im Variabilitätsmodell Restriktionen formuliert werden sollten? Eine Möglichkeit wäre die Restriktion direkt mit einer Variante zu verknüpfen. Dies führt allerdings zu einer Verstreuung aller Restriktionen. Daher wurde in dieser Arbeit ein Ansatz gewählt, in der die Summe aller Restriktionen als eigenständige Entität im Variabilitätsmodell beschrieben wird.

Im Folgenden werden zwei Ansätze zur Restriktionsformulierung vorgestellt, die im Rahmen dieser Arbeit entwickelt bzw. verwendet wurden. Zum einen wurde die Aussagenlogik als einfaches und intuitives Sprachmittel gewählt und für die Ziele dieser Arbeit geeignet erweitert. Dieser Ansatz wird im folgenden Abschnitt 4.2.2.1 beschrieben. Zum anderen wurde eine bereits aus der Literatur bekannte Restriktionssprache ausgewählt, die ausdrucksstärker aber dafür komplexer ist. Diese Sprache wurde ebenfalls in die Konzepte dieser Arbeit integriert. Der Ansatz wird in Abschnitt 4.2.2.2 genauer erläutert.

4.2.2.1. Restriktionen durch erweiterte Aussagenlogik

Die Syntax und Semantik für Restriktionen durch Konzepte der Aussagenlogik wird im Folgenden iterativ erläutert. Dabei wird die Aussagenlogik als Basis verwendet und für die Zwecke dieser Arbeit angepasst. Die folgenden Erläuterungen wurden im Rahmen einer Bachelorarbeit konzipiert und entwickelt [Pog10]. Für Informationen

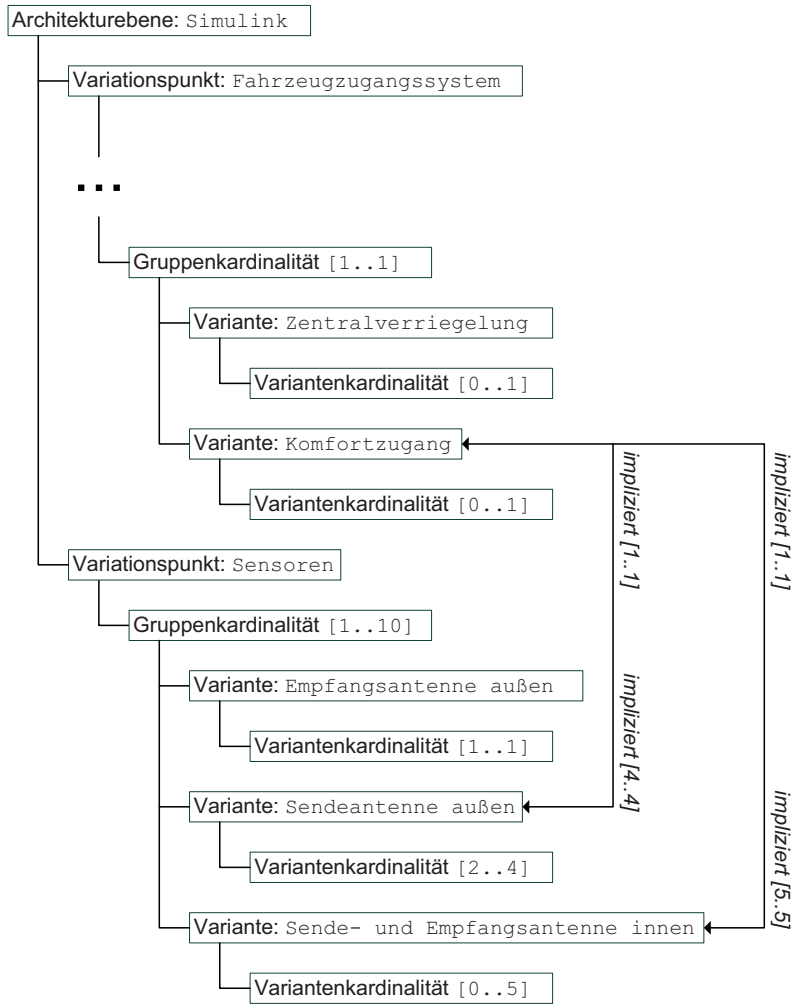


Abbildung 4.10.: Ein Beispiel für Restriktionen zwischen Variationspunkten der selben Abstraktionsebene

zur Aussagenlogik sei auf weiterführende Literatur verwiesen [RN03, Kle02, Bur97, EFT96].

Konkrete Syntax und Semantik Ein aussagenlogischer Ausdruck besteht aus atomaren Aussagen, den sogenannten Aussagenvariablen, die mit aussagenlogischen Junktoren verknüpft werden. Jede Aussagenvariable wird durch die Wahrheitswerte 0 (für falsch/false) und 1 (für wahr/true) interpretiert. Das zugrunde liegende Alphabet Γ setzt sich somit durch folgende Symbole zusammen:

$$\Gamma = \alpha \cup \chi \cup \{0, 1, (,)\},$$

wobei

$$\alpha = \{V_1, V_2, \ldots\}$$

die Menge aller Aussagenvariablen,

$$\chi = \{\neg, \wedge, \vee, \rightarrow\}$$

die Menge der aussagenlogischen Junktoren, 0 und 1 Boolesche Konstanten und (und) die Klammersymbole darstellen (vgl. [Grä11]).

Eine aussagenlogische Formel ψ ist ein Wort über dem Alphabet Γ. Da nicht jedes Wort aus Γ auch eine Formel ist, wird folgende induktive Definition zur Beschreibung von Formeln gegeben (vgl. [Grä11]): Die Menge $A \subset \Gamma$ der aussagenlogischen Formeln ist induktiv definiert durch

1. $0, 1 \in A$,
2. $\alpha \subseteq A$,
3. Wenn $\psi, \phi \in A$, dann sind auch die Wörter $\neg\psi$, $(\psi \wedge \phi)$, $(\psi \vee \phi)$ und $(\psi \rightarrow \phi)$ Formeln aus A.

Somit sind Boolesche Konstanten und Aussagenvariablen aussagenlogische Formeln als auch ihre Komposition mit den definierten Junktoren. Boolesche Konstanten und Aussagenvariablen werden auch als atomare Formeln bezeichnet. Die Syntax einer aussagenlogischen Formel kann anstatt dieser induktiven Definition auch durch folgende Erweiterte Backus-Naur-Form (EBNF) festgelegt werden:

AussagenlogischeFormel	=	AtomareFormel | KomplexeFormel
AtomareFormel	=	BoolscheKonstante | Aussagenvariable
BoolscheKonstante	=	'0' | '1'
Aussagenvariable	=	'V_1' | 'V_2' | ...
KomplexeFormel	=	[¬]AtomareFormel | {'('KomplexeFormel[('∧' | '∨' | '→')KomplexeFormel]')'}

Für jede Formel $\psi \in A$ wird weiterhin $\alpha(\psi) \subseteq \alpha$ als die Menge der in ψ vorkommenden Aussagenvariablen definiert. Eine aussagenlogische Interpretation ist eine Abbildung $I : \sigma \longrightarrow \{0,1\}$, mit $\sigma \subseteq \alpha$. Sie ist passend für eine Formel $\psi \in A$, wenn $\alpha(\psi) \subseteq \sigma$. Jede zu ψ passende Interpretation I definiert einen Wahrheitswert $[\![\psi]\!]^I \in \{0,1\}$, durch die folgenden Festlegungen (vgl. [Grä11]):

1. $[\![0]\!]^I := 0$, $[\![1]\!]^I := 1$
2. $[\![V]\!]^I := I(V)$, mit $V \in \sigma$
3. $[\![\neg\psi]\!]^I := 1 - [\![\psi]\!]^I$
4. $[\![\psi \wedge \phi]\!]^I := min([\![\psi]\!]^I, [\![\phi]\!]^I)$
5. $[\![\psi \vee \phi]\!]^I := max([\![\psi]\!]^I, [\![\phi]\!]^I)$
6. $[\![\psi \rightarrow \phi]\!]^I := [\![\neg\psi \vee \phi]\!]^I$

Eine Interpretation I erfüllt eine Formel $\psi \in A$, wenn $[\![\psi]\!]^I = 1$. I wird dann auch als Modell von ψ bezeichnet.

Damit die oben beschriebene Aussagenlogik als Sprachmittel zur Formulierung von Restriktionen über Variabilität verwendet werden kann, bedarf es noch an kleineren Anpassungen, die im Folgenden erläutert werden:

1. Das zugrunde liegende Alphabet Γ wird durch die Einführung eckiger Klammern und der Menge der natürlichen Zahlen N erweitert. Demnach setzt sich das Alphabet wie folgt zusammen:

$$\Gamma = \alpha \cup \chi \cup N \cup \{0, 1, (,), [,]\}.$$

2. Die Menge der Aussagenvariablen α entsprechen den Varianten im Variabilitätsmodell. Für das Variabilitätsmodell aus Abbildung 4.10 ergibt sich beispielsweise

$$\alpha = \{\texttt{Zentralverriegelung}, \texttt{Komfortzugang}, \texttt{Empfangsantenne außen}, \texttt{Sendeantenne außen}, \texttt{Sende- und Empfangsantenne innen}\}.$$

3. Die Menge der aussagenlogischen Junktoren χ wird um den Antivalenzjunktor $\oplus$ erweitert. Es ergibt sich somit

$$\chi = \{\neg, \wedge, \vee, \rightarrow, \oplus\}.$$

4. Jede Aussagenvariable in einer Formel kann durch die Angabe einer Kardinalität $[i..j]$, mit $i \leq j; i, j \in N$ erweitert werden.

5. Die Menge $R \subset \Gamma$ aller gültigen Restriktionsregeln wird durch folgende EBNF definiert:

Restriktionsregel	=	LinkeRegelseite ’→’ RechteRegelseite
LinkeRegelseite	=	Regel
RechteRegelseite	=	Regel
Regel	=	[¬] [’[’N ’..’ N’]’]AtomareFormel
		| {’(’Regel[(’∧’ | ’∨’ | ’⊕’)Regel]’)’}
N	=	’0’ | ’1’ | ’2’ | ’3’ | . . .
AtomareFormel	=	BoolscheKonstante | Aussagenvariable
BoolscheKonstante	=	’0’ | ’1’
Aussagenvariable	=	’V_1’ | ’V_2’ | . . .

Bei der Kardinalitätsangabe sei die Bedingung aus Punkt 4 zu beachten.

Eine Restriktionsregel ist somit stets eine spezielle Form einer aussagenlogischen Formel mit der Erweiterung von Kardinalitätsangaben. Dabei besteht eine Restriktionsregel aus einer linken und rechten Regelseite. Beide Regelseiten werden durch den Implikationsjunktor $\rightarrow$ miteinander verknüpft. Der Implikationsjunktor darf darüber hinaus innerhalb einer Restriktionsregel an keiner weiteren Stelle verwendet werden. Jede Regelseite wird dann durch eine aussagenlogische Formel ausgedrückt, die zusätzlich aus Kardinalitätsangaben und Antivalenzjunktoren besteht (siehe die Definition des Nichtterminalsymbols Regel aus der obigen EBNF). Bei der Interpretation einer Restriktionsregel werden die Kardinalitätsangaben ignoriert. Diese werden lediglich auf Übereinstimmung zu den Kardinalitätsangaben im Variabilitätsmodell überprüft. Daher kann die obige Definition zu einer aussagenlogischen Interpretation beibehalten werden.

Im Folgenden werden einige Beispiele für Restriktionsregeln bezogen auf die Varianten aus Abbildung 4.10 dargestellt:

- Die Zentralverriegelung ist eine verbindliche Variante:

$$1 \rightarrow \texttt{Zentralverriegelung}$$

- Die Zentralverriegelung und der Komfortzugang schließen sich gegenseitig aus:

$$1 \rightarrow \texttt{Zentralverriegelung} \oplus \texttt{Komfortzugang}$$

Die obigen zwei Restriktionsregeln hätten auch durch die Gruppen- und Variantenkardinalität des Variabilitätsmodells ausgedrückt werden können. Der Mehrwert der Anwendung von Restriktionsregeln schlägt sich insbesondere für gruppenübergreifende Restriktionen nieder. Einige Beispiele für derartige Regeln sind die folgenden:

- Der Komfortzugang impliziert vier Sendeantennen außen:

$$\texttt{Komfortzugang} \rightarrow \texttt{[4..4] Sendeantennen außen}$$

- Der Komfortzugang sowie fünf Sende- und Empfangsantennen implizieren sich gegenseitig:

$$(\texttt{Komfortzugang} \vee \texttt{[5..5] Sende- und Empfangsantennen}) \rightarrow (\texttt{Komfortzugang} \wedge \texttt{[5..5] Sende- und Empfangsantennen})$$

In Abbildung 4.11 sind die in Abbildung 4.10 informell beschriebenen Restriktionen als formaler Bestandteil des Variabilitätsmodells dargestellt. Die Restriktionen hätten durch die Verwendung weiterer Junktoren komponiert werden können. Zur besseren Lesbarkeit wurde in diesem Beispiel darauf verzichtet.

Abstrakte Syntax In Abbildung 4.9 wurde bereits angedeutet, dass ein Variabilitätsmodell (`VariabilityModel`) aus der Definition des Variationspunktes (`VariationPoint`) und aus einer Menge von Restriktionsregeln (`AuditContainer`) besteht. Abbildung 4.12 detailliert die abstrakte Syntax des Variabilitätsmodells um Konzepte der Restriktionsmodellierung. Durch das Containerkonzept werden Restriktionsregeln gemeinsam verwaltet. Eine Restriktion (`Constraint`) besteht aus der textuellen Darstellung der Regel (`ruleString : ParsedConstraint`) und aus der repräsentierenden Datenstruktur (`ConstraintRule`).

Abbildung 4.13 illustriert die abstrakte Syntax einer Restriktionsregel. Eine Regel setzt sich aus zwei Ausdrücken zusammen (`ConstraintExp`), die durch den Implikationsjunktor verknüpft werden (`ImpliesRule`). Die linke (`source`) und rechte Seite (`target`) einer Regel werden durch Aussagenvariablen (`NamedLiteralExp`) und Junktoren (`ConstraintJunctor`) zusammengesetzt. Die Aussagenvariablen speichern den Namen einer Variante (`cExp : ParsedConstraint`) und die Kardinalität (`lowerBound : int` und `upperBound : int`). Die Junktoren unterscheiden sich in einstellige und zweistellige Junktoren. Der einstellige Junktor `NotExp` besteht aus einem Parameter, die zweistelligen Junktoren `AndExp`, `OrExp` und `XorExp` hingegen aus zwei Parametern. Schließlich können durch die Klasse `NestedExp` komplexere Regeln formuliert werden.

4.2.2.2. Restriktionen durch Restriktionssprachen

In Abschnitt 4.2.2.1 wurde eine formale Sprache vorgestellt, welche die Aussagenlogik um weitere Konzepte erweitert und für Zwecke der Restriktionsformulierung eingesetzt wird. Dadurch ist es möglich, Regeln auf intuitive Weise zu formulieren. Allerdings sind Sprachen, die auf Aussagenlogik basieren, nicht immer ausdrucksstark genug. Insbesondere können keine Aussagen über Strukturen bzw. Elemente von Strukturen getroffen werden. Daher wurde im Rahmen dieser Arbeit ein weiteres ausdrucksstärkeres Sprachmittel zur Formulierung von Restriktionen integriert. Die Prädikatenlogik (engl. First-Order Logic) wäre beispielsweise eine mögliche Alternative, um Restriktionen auszudrücken. Es gibt allerdings in der Literatur Restriktionssprachen (engl. Constraint Language), welche die Ausdrucksstärke einer prädikatenlogischen Sprache besitzen und zudem speziell für Restriktionen entwickelt wurden. Diese eignen sich daher besonders für die in dieser Arbeit beschriebenen Problemstellungen bzgl. Restriktionen.

In der Masterarbeit von *Babur* [nB10] wurden existierende Restriktionssprachen hinsichtlich ihrer Eignung zur Formulierung von Restriktionen innerhalb von Varia-

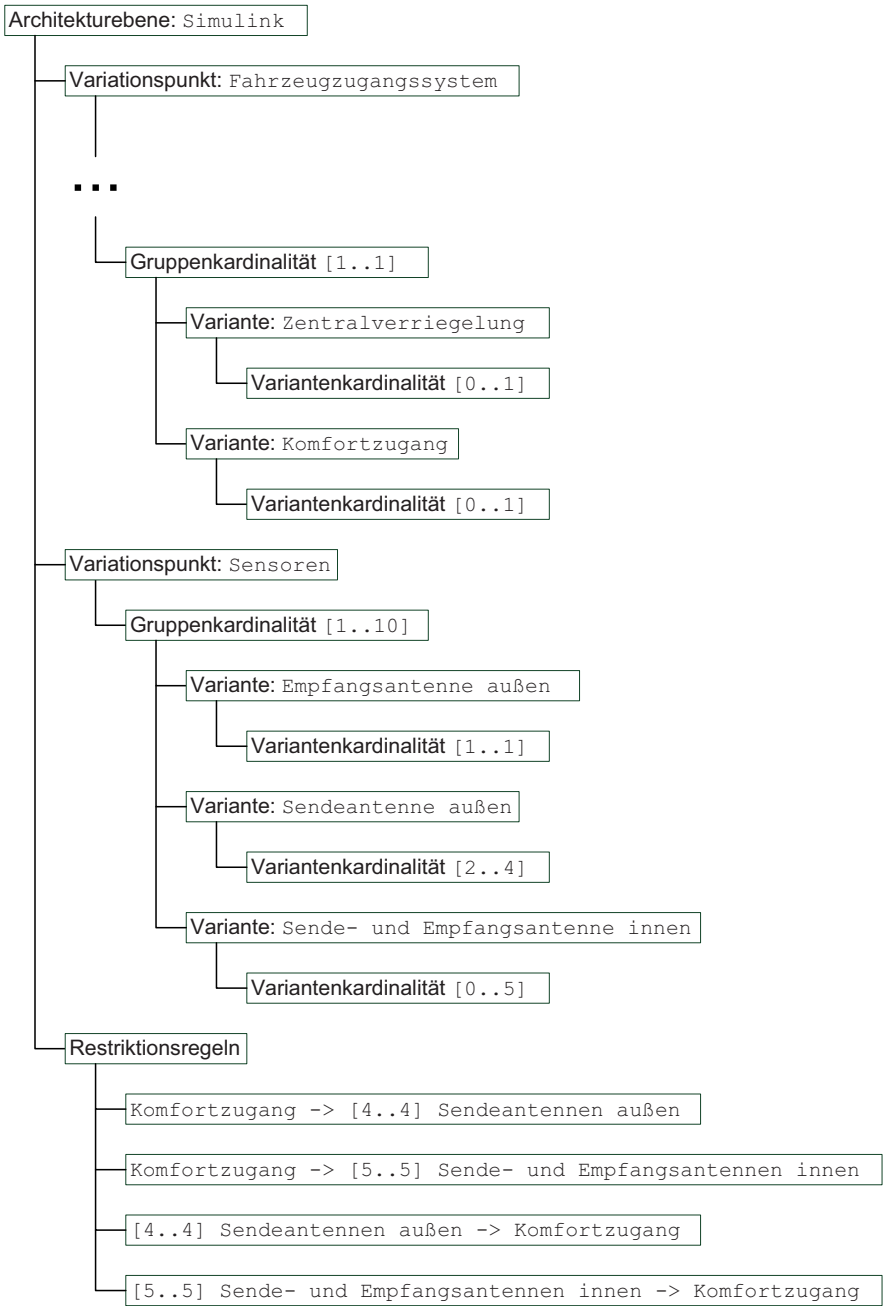


Abbildung 4.11.: Die konkrete Syntax von Restriktionen im Variabilitätsmodell

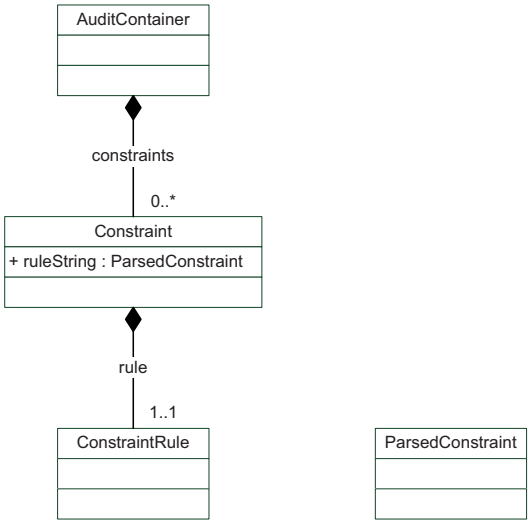


Abbildung 4.12.: Die Erweiterung der abstrakten Syntax des Variabilitätsmodell um Konzepte der Restriktionsmodellierung

bilitätsmodellen untersucht. Als Ergebnis wurde die Restriktionssprache WCRL als adäquate Sprache ermittelt. Sie wurde von *Simons et al.* konzipiert und entwickelt [SNS02] und im Rahmen dieser Arbeit verwendet und integriert. Im Folgenden wird diese Sprache genauer erläutert.

Konkrete Syntax und Semantik WCRL ist eine deklarative Regelsprache, die sich besonders für die Repräsentation von Konfigurationswissen eignet. Das Konfigurationswissen wird durch sogenannte gewichtete Restriktionsregeln (engl. Weight Constraint Rules) erfasst. Diese werden durch eine effiziente Inferenzmaschine, *smodels*, aufgelöst.

Das Alphabet Γ setzt sich aus folgenden Symbolen zusammen:

$$\Gamma = \tau \cup \alpha \cup \chi \cup \delta \cup N \cup \{0, 1, =, \{, \}, (,)\},$$

wobei

$$\tau = \bigcup_{i \in N} FS^i \cup \bigcup_{i \in N} PS^i$$

die Menge aller i-stelligen Funktions- und Relationssymbolen für $i \in N$,

$$\alpha = \{V_0, V_1, \ldots\}$$

die Menge aller Variablen,

$$\chi = \{\neg, \wedge, \leftarrow\}$$

die Menge der Junktoren,

$$\delta = \{C_0, C_1, \ldots\}$$

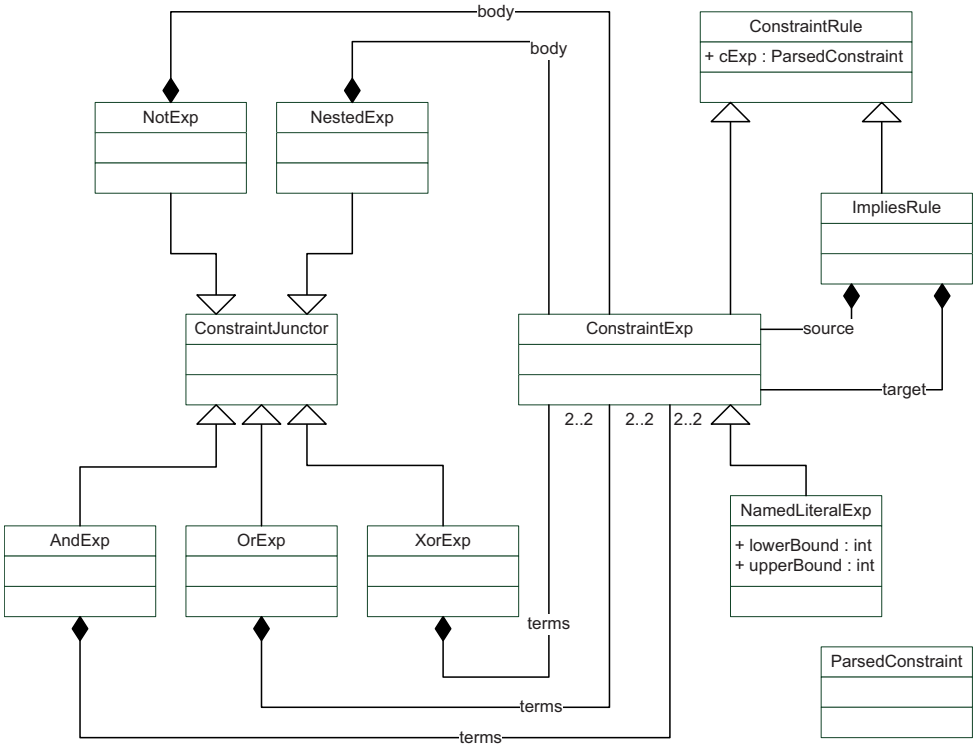


Abbildung 4.13.: Die abstrakte Syntax einer Restriktionsregel

die Menge aller gewichteten Restriktionen (engl. Weight Constraints), N die natürlichen Zahlen, 0 und 1 die Booleschen Konstanten, $=$ das Gleichheitssymbol und schließlich {, },(und) die Klammersymbole darstellen.

Die Menge T aller Terme wird definiert durch

1. $\alpha \subseteq T$, d.h. alle Variablen sind Terme.
2. Alle Konstantensymbole $C \in FS^0$ sind Terme.
3. Sind $T_1, \ldots, T_n$ Terme und $F \in FS^n$, dann ist auch $F(T_1, \ldots, T_n) \in T$.

Ein Grundterm ist ein Term, in dem keine Variablen auftreten. Grundterme werden auch als Grundatome bzw. Atome bezeichnet. Zum Beispiel sei $C \in FS^0$, $F \in FS^1$ und $G \in FS^2$. Dann sind alle folgenden Terme Atome:

- C,
- $F(C)$,

- $G(C, C)$ und
- $G(F(C), C)$.

Seien weiterhin $T_1, \ldots, T_n \in T$ Terme und $P \in PS^n$ ein n-stelliges Relationssymbol, dann ist $P(T_1, \ldots, T_n)$ eine Formel. Sind $T_1, \ldots, T_n$ Grundterme, so wird die Formel $P(T_1, \ldots, T_n)$ ebenfalls als Atom bezeichnet.

Eine gewichtete Restriktion C_i ist definiert durch

$$L \leq \{A_1 = w_{A_1}, \ldots, A_n = w_{A_n}, \neg B_1 = w_{B_1}, \ldots, \neg B_m = w_{B_m}\} \leq U,$$

wobei

- $L, U \in N$ untere und obere Schranken,
- $A_1, \ldots, A_n, B_1, \ldots, B_m$ Atome und
- $w_{A_1}, \ldots, w_{A_n}, w_{B_1}, \ldots, w_{B_m} \in N$ Gewichte sind.

Atome und deren Negation werden auch als Literale bezeichnet. So sind beispielsweise A_i und $\neg B_i$ jeweils Literale. Zum Beispiel ist der folgende Ausdruck eine gewichtete Restriktion:

$$C = 20 \leq \{A = 10, \neg B = 10, D = 20\} \leq 30.$$

Eine Menge S von Atomen erfüllt eine gewichtete Restriktion genau dann, wenn

$$L \leq \sum_{A_i \in S} w_{A_i} + \sum_{B_i \notin S} w_{B_i} \leq U.$$

Als Beispiel sei erneut die obige gewichtete Restriktion C und folgende vier Mengen gegeben:

- $S_1 = \{\}$,
- $S_2 = \{A\}$,
- $S_3 = \{A, B\}$,
- $S_4 = \{A, B, C\}$.

Die leere Menge S_1 erfüllt nicht die Restriktion C, denn

$$20 \not\leq 10.$$

Die Menge S_2 erfüllt die Restriktion C, denn

$$20 \leq 20 \leq 30.$$

Die Menge S_3 erfüllt nicht die Restriktion C, denn

$$20 \not\leq 10.$$

Die Menge S_4 erfüllt die Restriktion C, denn

$$20 \leq 30 \leq 30.$$

Weiterhin werden folgende Konventionen als abkürzende Schreibweise für gewichtete Restriktionen eingeführt:

- Die mathematischen Symbole $\leq$ können ausgelassen werden.
- Wenn L und/oder U nicht angegeben werden, sind diese als $-\infty$ bzw. $+\infty$ zu interpretieren.
- Gewichte mit dem Wert 1 können ausgelassen werden.
- Die Klammerungen können weggelassen werden, wenn die Restriktion aus genau einem Atom besteht.

Eine gewichtete Restriktionsregel ist durch folgende Syntax definiert:

$$C_0 \leftarrow C_1 \wedge \ldots \wedge C_n,$$

wobei $C_i \in \delta, i \in \{0, \ldots, n\}, n \in N$ eine gewichtete Restriktion ist. C_0 wird auch als Kopf der Regel und alle weiteren C_i mit $i \in \{1, \ldots n\}$ als Rumpf der Regel bezeichnet. Das Implikationssymbol ist in WCRL umgedreht. So impliziert die rechte Regelseite die linke Regelseite.

Eine Menge S erfüllt eine gewichtete Restriktionsregel genau dann, wenn S den Kopf der Regel C_0 immer dann erfüllt, wenn S jede Restriktion $C_1, \ldots, C_n$ im Rumpf der Regel erfüllt.

Eine WCRL-Regel mit fehlendem Kopf C_0 hat die Form

$$\leftarrow C_1 \wedge \ldots \wedge C_n.$$

C_0 wird in diesem Fall als nicht erfüllbar interpretiert (Boolesche 0).

Eine WCRL-Regel mit fehlendem Rumpf $C_1, \ldots, C_n$ hat die Form

$$C_0 \leftarrow .$$

Die rechte Regelseite wird in diesem Fall stets als erfüllbar interpretiert (Boolesche 1). Derartige Regeln werden auch als Fakten bezeichnet.

Die Verwendung von Variablen $V_i \in \alpha$ anstelle von Atomen ermöglicht oft eine kompaktere Schreibweise für Restriktionen. Aus Restriktionen mit Variablen werden alle Atome instanziiert, sodass die gleiche Semantik für Restriktionen mit Variablen verwendet werden kann.

Um eine Menge von Literalen auf kompakte Weise zu formulieren, werden sogenannte bedingte Literale der Form $Lit : P$ eingeführt. Lit ist dabei ein Literal und die Bedingung P ein Atom mit einer Relation. Als Beispiel für bedingte Literale seien folgende Restriktionen gegeben:

$$
\begin{aligned}
C_1 &= Antennentyp(Sendeantenne) \\
C_2 &= Antennentyp(Empfangsantenne) \\
C_3 &= Antennentyp(SendeEmpfangsantenne) \\
C_4 &= \{Antenne(A, Sendeantenne), Antenne(A, Empfangsantenne), \\
&\quad Antenne(A, SendeEmpfangsantenne)\}
\end{aligned}
$$

Die gewichtete Restriktion C_4 kann auch in Form eines bedingten Literals wie folgt dargestellt werden:

$$C_4 = \{Antenne(A, X) : Antennentyp(X)\}.$$

Um nun die Restriktionen aus Abbildung 4.10 in Form von WCRL auszudrücken, sei Folgendes gegeben:

- $FZGS \in RS^1$ ist ein einstelliges Relationssymbol für das Fahrzeugzugangssystem.
- $ZV \in FS^0$ ist ein Konstantensymbol für die Zentralverriegelung.
- $Komf \in FS^0$ ist ein Konstantensymbol für den Komfortzugang.
- $Sens \in RS^1$ ist ein einstelliges Relationssymbol für die Menge der Sensoren.
- $EmpAnt \in FS^0$ ist ein Konstantensymbol für Empfangsantennen.
- $SendAnt \in FS^0$ ist ein Konstantensymbol für Sendeantennen.
- $SendEmpAnt \in FS^0$ ist ein Konstantensymbol für Sende- und Empfangsantennen.

Durch folgende Regeln können die Restriktionen nun angegeben werden (beachte, dass Kurzschreibweisen verwendet werden):

- $4\{Sens(SendAnt) = 4\}4 \leftarrow FZGS(Komf)$
- $5\{Sens(SendEmpAnt) = 5\}5 \leftarrow FZGS(Komf)$
- $FZGS(Komf) \leftarrow 4\{Sens(SendAnt) = 4\}4 \wedge 5\{Sens(SendEmpAnt) = 5\}5$

Die primäre Verwendung von WCRL liegt in dieser Arbeit allerdings nicht in der Formulierung von Restriktionen. Vielmehr wird WCRL zusammen mit der Inferenzmaschine smodels für die Herleitung gebundener Softwaredokumente angewendet. Dies wird im Laufe dieser Arbeit noch detaillierter erläutert.

4.3. Bindung

Die Modellierung von Variabilität ermöglicht die explizite Erfassung von Variationspunkten und Varianten im gesamten Referenzprozess. Das Variabilitätsmodell reicht allerdings nicht aus, um Varianten zu binden. In diesem Abschnitt werden daher zwei Bindungsmechanismen vorgestellt, die im Rahmen dieser Arbeit entwickelt wurden: (1) *die Selektion mit einer Konfigurationsmaschine* (vgl. Abschnitt 4.3.1) und (2) *die Generierung mit einer Inferenzmaschine* (vgl. Abschnitt 4.3.2).

4.3.1. Konfigurationsmodell

Die Bindung wird in einem Konfigurierungsprozess durchgeführt. Diese Aktivität wird aber nicht am Variabilitätsmodell verrichtet. Stattdessen wird der Konfigurierungsprozess an einem *Konfigurationsmodell* ausgeführt, das alle relevanten Informationen aus dem Variabilitätsmodell extrahiert und stets synchron zu diesem bleibt. Durch das Konzept der Selektion können die Varianten im Konfigurationsmodell gebunden werden. Durch diese Trennung können die Modellierung und die Bindung als zwei separate Aktivitäten behandelt werden. Die konkrete und abstrakte Syntax des Konfigurationsmodells werden in Abschnitt 4.3.1.1 und Abschnitt 4.3.1.2 beschrieben.

Die Existenz von Restriktionen erschwert allerdings den Konfigurierungsprozess deutlich. Beispielsweise erfordern sich der `Komfortzugang`, die `Sendeantenne außen` und die `Sende- und Empfangsantenne innen` gegenseitig. So müssen bei Selektion des Komfortzugangs auch die beiden Antennentypen selektiert werden, um eine gültige Konfiguration zu erhalten. Derartige Implikationen sind aber nur schwer nachzuvollziehen, insbesondere bei sehr vielen und komplexen Restriktionsregeln. Daher ist es wünschenswert, den Konfigurierungsprozess so zu gestalten, dass in jedem Konfigurierungsschritt Restriktionsregeln ausgewertet und Implikationen automatisch abgeleitet und selektiert werden. Dies wird in einer *Validierung* innerhalb des Konfigurierungsprozess gewährleistet (vgl. Abschnitt 4.3.1.3).

4.3.1.1. Konkrete Syntax

Abbildung 4.14 zeigt die konkrete Syntax eines Konfigurationsmodells. Es besteht im Wesentlichen aus drei Spalten mit folgendem Inhalt: Die erste Spalte beinhaltet eine Variantenbaumliste, die aus dem Variabilitätsmodell abgeleitet wird. Die zweite Spalte enthält die Kardinalitätsangaben für die jeweiligen Varianten. Schließlich umfasst die dritte Spalte die Selektionsentscheidungen.

Die Variantenbaumliste des Konfigurationsmodells bezieht aus dem Variabilitätsmodell alle modellierten Variationspunkte, die Gruppenkardinalitäten und die zugehörigen Varianten. Weitere Informationen wie etwa der Variabilitätsmechanismus oder der Bindungsmechanismus sind für die Konfigurierung nicht erforderlich. Die Kardinalitäten und Selektionen müssen jeweils von einem Benutzer angegeben werden. Somit ist die Konfigurierung ein interaktiver Prozess.

In dem Beispiel aus Abbildung 4.14 wird nun folgendermaßen vorgegangen: Der Benutzer weiß, dass er sich für genau ein Fahrzeugzugangssystem entscheiden muss. In diesem Beispiel wird der `Komfortzugang` mit einer Kardinalität von `[1]` selektiert. Nach diesem Schritt ist allerdings die aktuelle Konfiguration invalide, da Restriktionsregeln die Selektion von vier `Sendeantenne außen` und fünf `Sende- und Empfangsantenne innen` erfordern. An dieser Stelle unterstützt die Validierung den Benutzer, indem es diese Auswahl automatisch tätigt (genaueres hierzu in Abschnitt 4.3.1.3). Dem Benutzer wird die automatische Selektion durch ein Schlosssymbol an der Auswahlbox mitgeteilt. Es herrscht nun erneut eine gültige Konfiguration vor.

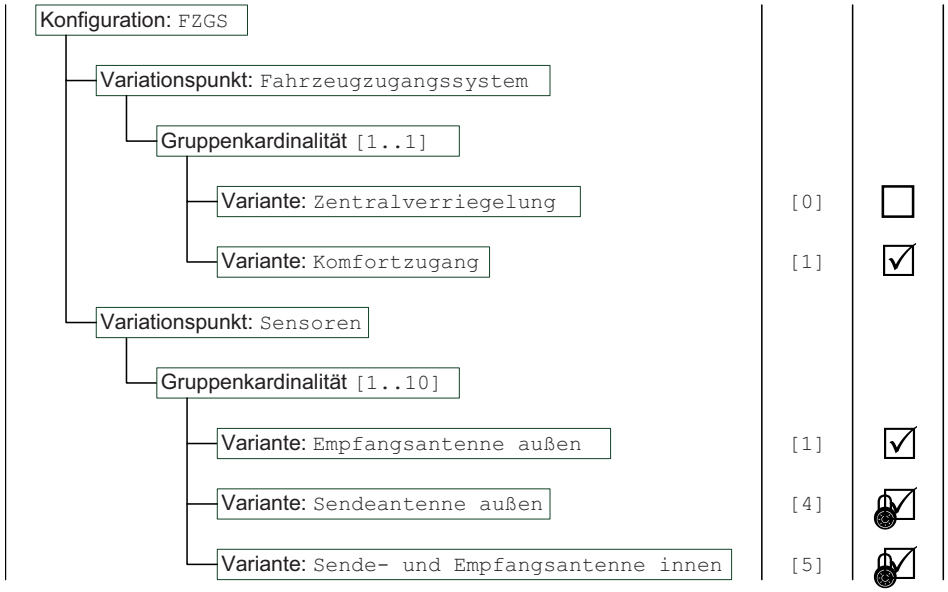


Abbildung 4.14.: Die konkrete Syntax des Konfigurationsmodells

4.3.1.2. Abstrakte Syntax

Die abstrakte Syntax des Konfigurationsmodells ist in Abbildung 4.15 dargestellt. Ein Konfigurationsmodell (`ConfigModel`) verwaltet mehrere Konfigurationen (`Configuration`). Durch die Klasse `ConfigurationItem` wird jede Konfiguration rekursiv aufgebaut. Die Assoziation zum Variabilitätsmodell wird durch die Klasse `Element` hergestellt. Sie ist eine Oberklasse für alle Objekte des Variabilitätsmodells. Zur Validierung der Kardinalitäten in einer Konfigurierung wird die Gruppenkardinalität aus dem Variabilitätsmodell benötigt (`conflicetdVariantGroup : GroupCardinality`). Zusätzlich wird der Status jeder Variante benötigt (`VariantToVariantStatusMap`). Die Statusinformationen werden in Form von Schlüssel-Wert-Paaren gespeichert. Der Schlüssel referenziert eine Variante im Variabilitätsmodell (`key : Variant`) und der Wert beinhaltet den Status der Variante (`value : VariantStatus`). Der Status einer Variante wird insbesondere für die Validierung herangezogen. Folgende Informationen werden hierfür festgehalten:

- Selektionszustand: Eine Variante kann entweder selektiert oder deselektiert werden. Der Selektionszustand einer Variante (`selectionStatus : VariantSelectionStatus`) beinhaltet die aktuelle Information innerhalb des Konfigurierungsprozesses. Der Zustand wird vom Benutzer modifiziert. Der Aufzählungstyp (`VariantSelectionStatus`) stellt hierfür die Werte `SELECTED` und `DESELECTED` zur Verfügung.
- Validierungszustand: Nach jedem Validierungsschritt wird ein entsprechender Va-

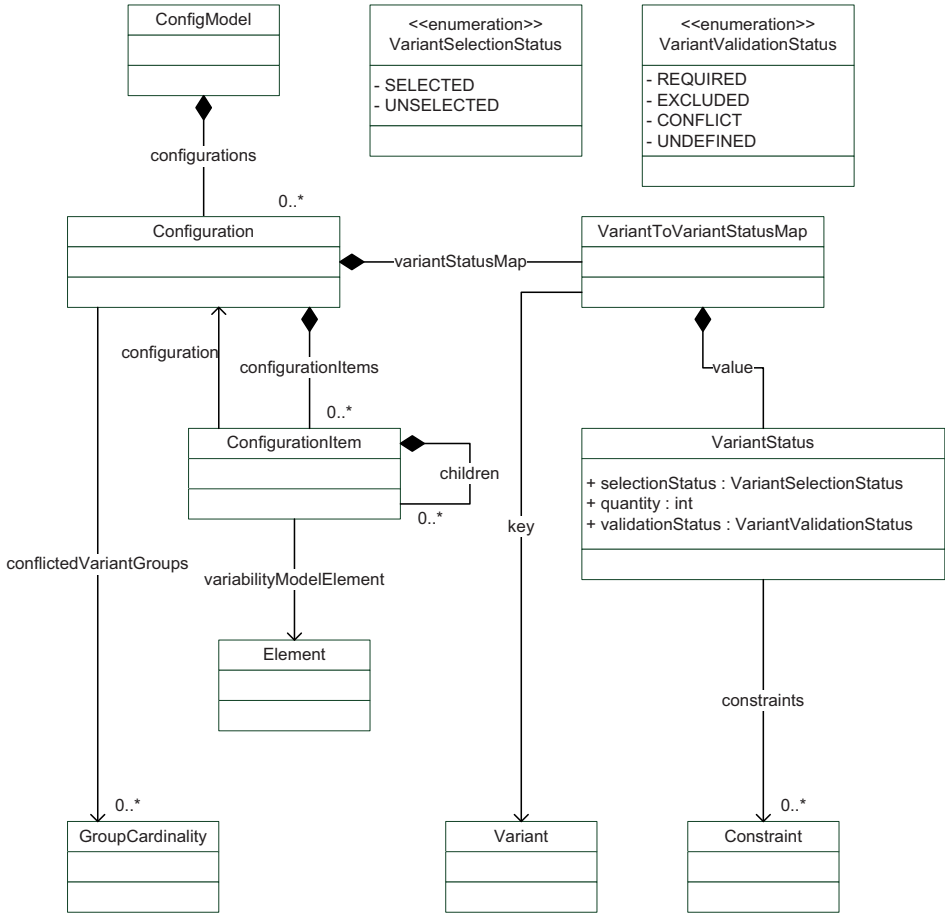


Abbildung 4.15.: Die abstrakte Syntax des Konfigurationsmodells

lidierungszustand einer Variante zugeordnet (`validationStatus : VariantValidationStatus`). Der Aufzählungstyp `VariantValidationStatus` beinhaltet dabei die folgenden Werte:

- `UNDEFINED`: Nach jedem Validierungsschritt wird der Validierungszustand einer Variante mit diesem Wert initiiert.
- `REQUIRED`: Wenn eine Variante aufgrund einer Restriktionsregel erfordert wird, bekommt die Variante diesen Zustand zugewiesen.
- `EXCLUDED`: Wenn eine Variante aufgrund einer Restriktionsregel ausgeschlossen wird, bekommt die Variante diesen Zustand zugewiesen.
- `CONFLICT`: Wenn in einem Validierungsschritt nicht eindeutig festgestellt werden kann, ob eine Variante `REQUIRED` oder `EXCLUDED` ist, wird die Variante mit diesem Validierungszustand belegt. In diesen Fällen ist ein Benutzerein-

greifen erforderlich.

- Kardinalität: Anhand der Kardinalität wird die konkrete Anzahl der Ausprägungen einer Variante festgelegt (`quantity : int`). Diese wird in der Validierung mit den Schranken der Variantenkardinalität verglichen.
- Aktive Restriktionen: Hierbei werden alle Restriktionen referenziert, die den aktuellen Status einer Variante hervorrufen (`constraints : Constraint`). So kann schnell nachvollzogen werden, wie die aktuelle Konfiguration zustande gekommen ist.

4.3.1.3. Validierung

Wie bereits weiter oben erläutert, wird ein Konfigurierungsprozess erstrebt, der den Benutzer bei der Selektion bzw. Deselektion unterstützt, indem alle Restriktionsregeln ausgewertet und aus dieser Auswertung automatisch die nächsten Konfigurierungsschritte hergeleitet und ausgeführt werden. Zu diesem Zweck wird zu jedem Konfigurierungsschritt (sowohl zu manuell vom Benutzer eingegebenen als auch automatisch hergeleiteten) eine Validierung angestoßen. Es wird dabei eine Kombination aus einer proaktiven und reaktiven Validierung eingesetzt. Die proaktive Validierung wird bei jeder Selektion/Deselektion von Varianten eingesetzt und dient zur Vermeidung von invaliden Konfigurationen innerhalb jedes Konfigurierungsschrittes. Dem Benutzer werden auf diese Weise aufgrund von Restriktionen entstehende Konfigurerungsschritte abgenommen und automatisch ausgeführt. Die reaktive Validierung wird zur Überprüfung der Kardinalitäten angewendet. Hier ist das Eingreifen des Benutzers erforderlich, um valide Konfigurationen wiederherstellen zu können. Hieraus ergibt sich der aus Algorithmus 4.1 dargestellte Ablauf.

Algorithmus 4.1 Proaktive und reaktive Validierung in der Konfigurierung

```
n ← #Varianten
m ← #Restriktionen
counter ← 0
repeat
  validiere Restriktionen
  counter ← counter + 1
until Konfiguration wurde nicht verändert or counter > n · m
validiere Kardinalitäten
```

Die Validierung der aktuellen Konfiguration wird solange durchgeführt, bis die Restriktionsregeln keine Änderungen in der Konfiguration hervorrufen. Es kann allerdings passieren, dass aufgrund fälschlicherweise zyklisch modellierter Restriktionen Unendlichschleifen entstehen. Um dies zu vermeiden, werden die Anzahl der Varianten und die Anzahl der Restriktionen herangezogen. Wenn nun n Varianten und m Restriktionen vorhanden sind und jede Restriktion von maximal n Varianten den Status modifizieren kann, so ergibt sich bei m Restriktionen maximal $n \cdot m$

Modifizierungen in der Konfigurierung. Also ist dies eine gesicherte obere Schranke in der Validierung der Konfigurierung.

Die Anweisung `validiere Restriktionen` innerhalb der Schleife stellt die proaktive Validierung dar. Sie stellt sicher, dass stets valide Konfigurationen durch Auswertung der Restriktionsregeln hergeleitet werden. Die Anweisung `validiere Kardinialitäten` hingegen stellt die reaktive Validierung dar. Hier wird überprüft, ob die Kardinalitäten die vordefinierten Schranken nicht unter- bzw. überschreiten.

Die proaktive Validierung der Restriktionen in jedem Konfigurierungsschritt besteht wiederum aus einer sogenannten Vorwärtsvalidierung und einer Rückwärtsvalidierung. In der Vorwärtsvalidierung wird zunächst der linke Teil der Restriktion auf seinen logischen Wert überprüft, also ob der linke Ausdruck wahr oder falsch ist. Wird der Ausdruck als wahr ermittelt, so wird der rechte Teil der Restriktion ausgeführt. Wurde die Vorwärtsvalidierung für alle Restriktionen durchgeführt, folgt im Anschluss die Rückwärtsvalidierung. Sie dient zur Überprüfung des rechten Teils der Restriktion auf seinen aktuell im Konfigurierungsschritt entstandenen logischen Wert. Ist der Wert ermittelt, so können anhand der Restriktionen Schlussfolgerungen über den linken Teil getroffen und entsprechende Maßnahmen durchgeführt werden. Sowohl die Vorwärtsvalidierung als auch die Rückwärtsvalidierung werden im Folgenden genauer erläutert.

Vowärtsvalidierung Aus Abbildung 4.15 ist erkennbar, dass sich der Status einer Variante (`VariantStatus`) aus den Zuständen der Selektion (`selectionStatus : VariantSelectionStatus`) und Validierung (`validationStatus : VariantValidationStatus`) ergibt (die Kardinalität sei an dieser Stelle zunächst vernachlässigt). Aus diesen beiden Zuständen können insgesamt acht Kombinationen abgeleitet werden, von denen zwei aufgrund der proaktiven Validierung nie auftreten:

Selektionszustand	Validierungszustand
SELECTED	UNDEFINED
SELECTED	REQUIRED
~~SELECTED~~	~~EXCLUDED~~
SELECTED	CONFLICT
UNSELECTED	UNDEFINED
~~UNSELECTED~~	~~REQUIRED~~
UNSELECTED	EXCLUDED
UNSELECTED	CONFLICT

Weiterhin wird für jeden atomaren Ausdruck einer Restriktionsregel eine sogenannte Gewichtung zugeordnet. Sie wird aus dem Syntaxbaum der jeweiligen Restriktionsregeln abgeleitet. Dabei wird jedem Element im Syntaxbaum eine Gewichtung zugeordnet. Jedes Element erhält die Gewichtung seines Vorgängers. Dann wird überprüft, ob das Element eine Änderung an der Gewichtung hervorruft. Wenn ja, wird die Gewichtung entsprechend geändert. Die Gewichtung besteht dabei aus einer sogenannten Orientierung und einem Konfliktstatus. Die Orientierung legt fest, ob ein Element im Syntaxbaum `POSITIV` oder `NEGATIV` ist. Die Orientierung eines Elements ist dabei `POSITIV`, wenn das Element

- keine Negationsoperation ($\neg$) ist und die Orientierung `POSITIV` ist, oder
- eine Negationsoperation ($\neg$) ist und die Orientierung `NEGATIV` ist.

Die Orientierung eines Elements ist `NEGATIV`, wenn das Element

- keine Negationsoperation ($\neg$) ist und die Orientierung `NEGATIV` ist, oder
- eine Negationsoperation ($\neg$) ist und die Orientierung `POSITIV` ist.

Der Konfliktstatus legt fest, ob ein Element im Syntaxbaum in Konflikt steht oder nicht (angegeben durch die Wahrheitswerte `TRUE` und `FALSE`). In Konfliktsituationen ist es nicht mehr automatisch entscheidbar, welchen Selektionszustand eine Variante bekommen soll. Der Konfliktstatus eines Elements ist `TRUE`, wenn das Element

- eine Oder-Operation ($\vee$) ist und die Orientierung `POSITIV` ist, oder
- eine Xor-Operation ($\oplus$) ist und die Orientierung `POSITIV` ist, oder
- eine Und-Operation ($\wedge$) ist und die Orientierung `NEGATIV` ist.

In allen anderen Fällen wird der Konfliktstatus vom Vorgänger beibehalten.

Die Gewichtung eines Elements im Syntaxbaum wird also aus der Gewichtung des Vaterknotens und des Elements selber ermittelt. Tabelle 4.2 listet alle möglichen Kombinationen diesbezüglich auf. Dabei steht das Symbol `A` stellvertretend für alle atomaren Aussagenvariablen einer Restriktionsregel.

Abbildung 4.16 illustriert vier Beispiele für Restriktionsregeln mit den jeweils entsprechenden Syntaxbäumen und zugehörigen Gewichtungen. Das erste Beispiel zeigt den gewichteten Syntaxbaum für die Restriktionsregel $A \rightarrow B$. Da jede Restriktionsregel aus einem linken und rechten Teil besteht, die durch die Implikation $\rightarrow$ verbunden werden, ist die Wurzel im Syntaxbaum stets das $\rightarrow$-Element. Initiiert wird die Gewichtung dieses Elements immer mit `(POS, FALSE)`, d.h. die Orientierung ist `POSITIV` und der Konfliktstatus ist `FALSE`. Wird Tabelle 4.2 zur Ermittlung der Gewichtungen herangezogen, ist festzustellen, dass die Elemente `A` und `B` weiterhin die Gewichtung `(POS, FALSE)` besitzen.

Im zweiten Beispiel wird der Syntaxbaum für die Restriktionsregel $A \rightarrow B \vee C$ dargestellt. Während der linke Teilbaum identisch zum vorherigen Beispiel ist, wird im rechten Teilbaum eine Verknüpfung mit einer Oder-Operation abgeleitet. Nach den weiter oben beschriebenen Regeln wird der Konfliktstatus auf `TRUE` gesetzt, wenn die Orientierung des Vaterknotens `POSITIV` und das aktuelle Element eine Oder Operation ($\vee$) ist. Dieser Fall tritt an dieser Stelle auf, sodass die Gewichtung des $\vee$-Elements auf `(POS, TRUE)` gesetzt wird. Dies wäre auch anhand Tabelle 4.2 ablesbar. Da darauffolgend atomare Aussagenvariablen abgeleitet werden, wird die Gewichtung des Vaterknotens übernommen. Somit ist an dieser Stelle nicht entscheidbar, wie der Selektionszustand für die Aussagenvariablen `B` und `C` festgelegt werden soll.

Das dritte Beispiel erweitert das zweite Beispiel durch einen Negationsoperator. Die Restriktionsregel lautet hier $A \rightarrow \neg(B \vee C)$. Durch die Negierung wird die

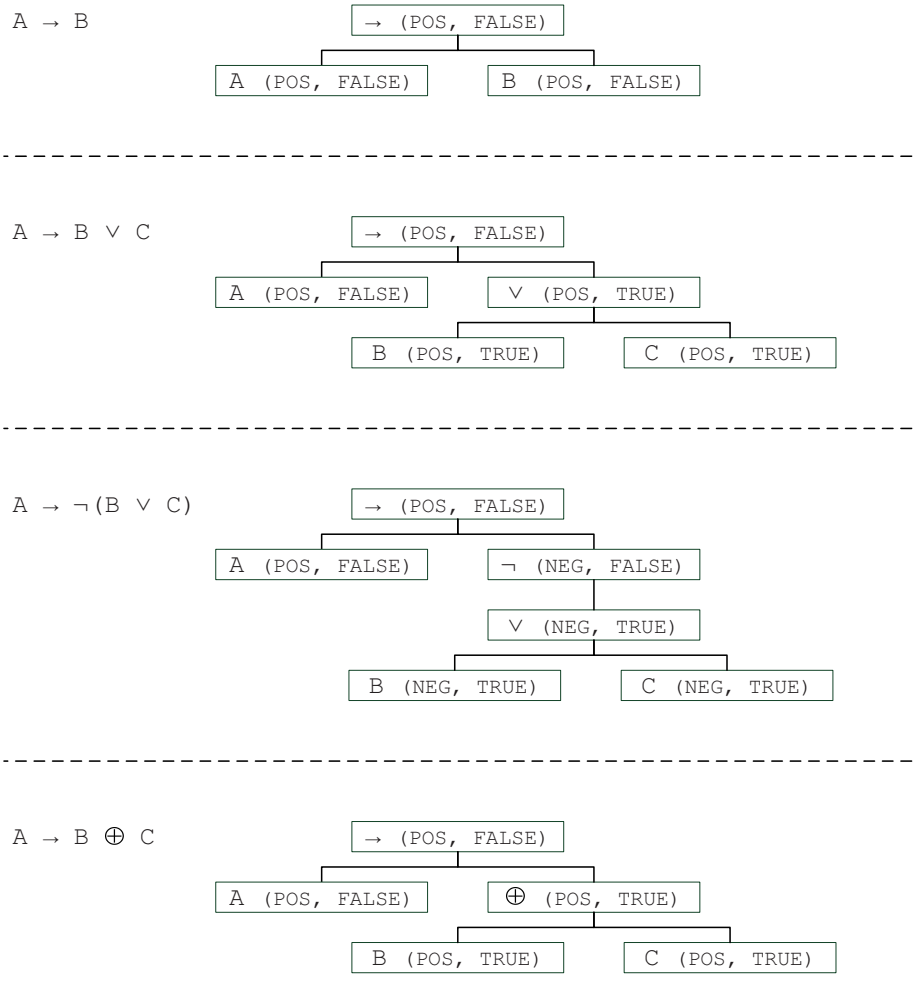


Abbildung 4.16.: Beispiele für Syntaxbäume von Restriktionsregeln mit den zugehörigen Gewichtungen

Gewichtung Vaterknoten		Element im Syntaxbaum	Gewichtung Element	
Orientierung	Konfliktstatus		Orientierung	Konfliktstatus
NEGATIV	FALSE	A	NEGATIV	FALSE
		¬	POSITIV	FALSE
		∧	NEGATIV	TRUE
		∨	NEGATIV	FALSE
		⊕	NEGATIV	FALSE
NEGATIV	TRUE	A	NEGATIV	TRUE
		¬	POSITIV	TRUE
		∧	NEGATIV	TRUE
		∨	NEGATIV	TRUE
		⊕	NEGATIV	TRUE
POSITIV	FALSE	A	POSITIV	FALSE
		¬	NEGATIV	FALSE
		∧	POSITIV	FALSE
		∨	POSITIV	TRUE
		⊕	POSITIV	TRUE
POSITIV	TRUE	A	POSITIV	TRUE
		¬	NEGATIV	TRUE
		∧	POSITIV	TRUE
		∨	POSITIV	TRUE
		⊕	POSITIV	TRUE

Tabelle 4.2.: Ermittlung der Gewichtungen aus der Gewichtung des Vaterknotens und des aktuell betrachteten Elements im Syntaxbaum

Gewichtung im rechten Teilbaum auf `(NEG, FALSE)` gesetzt und durch den Oder-Operator im Anschluss auf `(NEG, TRUE)`. Diese Gewichtung erstreckt sich dann bis zu den Blättern. Das heißt, dass an dieser Stelle, wie im obigen Fall, nicht entscheidbar ist, wie der Selektionszustand für die Aussagenvariablen `B` und `C` zugeordnet werden soll. Das letzte Beispiel sei dem Leser zur Analyse überlassen.

Ausschlaggebend für den Validierungszustand sind die Blätter im Syntaxbaum. Diese stellen die atomaren Aussagenvariablen dar, die mit den Varianten im Variabilitätsmodell korrespondieren. Die Gewichtung dieser Variablen legt den Validierungszustand fest und somit auch den Selektionszustand. Tabelle 4.3 illustriert alle möglichen Variantenstatus nach einem Validierungsdurchlauf. Da in jedem Validierungsdurchlauf der Validierungszustand als `UNDEFINED` initiiert wird, gibt es somit immer zwei Variantenstatus vor einem Validierungsdurchlauf:

- `(SELECTED, UNDEFINED)`
- `(UNSELECTED, UNDEFINED)`

Durch die Validierung werden den atomaren Aussagenvariablen Gewichtungen zugeordnet. In Tabelle 4.3 sind alle möglichen Gewichtungen für die beschriebenen zwei Fälle dargestellt (jeweils vier Gewichtungen). Aus diesen Gewichtungen ergeben sich die Validierungszustände der Aussagenvariablen (d.h. der Varianten). Wurde

Variantenstatus vor Validierung		Gewichtung durch Validierung		Variantenstatus nach Validierung	
Selektionszust.	Validierungszust.	Orientierung	Konfliktstatus	Validierungszust.	Selektionszust.
SELECTED	UNDEFINED	NEGATIV	FALSE	EXCLUDED	UNSELECTED
		NEGATIV	TRUE	CONFLICT	UNSELECTED
		POSITIV	FALSE	UNDEFINED	SELECTED
		POSTIIV	TRUE	CONFLICT	SELECTED
UNSELECTED	UNDEFINED	NEGATIV	FALSE	EXCLUDED	UNSELECTED
		NEGATIV	TRUE	CONFLICT	UNSELECTED
		POSITIV	FALSE	REQUIRED	SELECTED
		POSTIIV	TRUE	CONFLICT	SELECTED

Tabelle 4.3.: Variantenstatus nach Durchlauf einer Validierung

der Validierungszustand festgelegt, kann aus diesen der neue Selektionszustand ermittelt werden. Es ergeben sich dabei folgende Fälle:

- `(NEGATIV, FALSE)` ⇒ `EXCLUDED`: Ist die Orientierung der Aussagenvariable `NEGATIV` und herrscht kein Konflikt vor (`FALSE`), so wird der Validierungszustand der Variablen als `EXCLUDED` festgelegt. Dies hat zur Folge, dass der Selektionszustand als `UNSELECTED` bestimmt wird (unabhängig, wie der Selektionszustand vor der Validierung war).

- `(NEGATIV, TRUE)` ⇒ `CONFLICT`: Ist die Orientierung der Aussagenvariable `NEGATIV` und herrscht ein Konflikt vor (`TRUE`), so wird der Validierungszustand der Variablen als `CONFLICT` festgelegt. Dies hat zur Folge, dass der Selektionszustand als `UNSELECTED` bestimmt wird (unabhängig, wie der Selektionszustand vor der Validierung war).

- `(POSITIV, FALSE)` ⇒ `UNDEFINED`: Ist der Variantenstatus einer Aussagenvariable vor der Validierung `(SELECTED, UNDEFINED)` und ergibt die Validierung eine Gewichtung `(POSITIV, FALSE)`, so wird der Validierungszustand als `UNDEFINED` festgelegt. Hieraus folgt der Selektionszustand `SELECTED`. Dieser Fall tritt immer ein, wenn die Variable auf der linken Regelseite steht.

- `(POSITIV, FALSE)` ⇒ `REQUIRED`: Ist der Variantenstatus einer Aussagenvariable vor der Validierung `(UNSELECTED, UNDEFINED)` und ergibt die Validierung eine Gewichtung `(POSITIV, FALSE)`, so wird der Validierungszustand als `REQUIRED` festgelegt. Hieraus folgt der Selektionszustand `SELECTED`. Dieser Fall tritt immer ein, wenn die Variable auf der rechten Regelseite steht.

- `(POSITIV, TRUE)` ⇒ `CONFLICT`: Ist die Orientierung der Aussagenvariable `POSITIV` und herrscht ein Konflikt vor (`TRUE`), so wird der Validierungszustand der Variablen als `CONFLICT` festgelegt. Dies hat zur Folge, dass der Selektionszustand als `SELECTED` bestimmt wird (unabhängig, wie der Selektionszustand vor der Validierung war).

Varianten	Variantenstatus	
	Selektionszust.	Validierungszust.
A	SELECTED	UNDEFINED
B	UNSELECTED	UNDEFINED
C	UNSELECTED	UNDEFINED
D	UNSELECTED	UNDEFINED
E	UNSELECTED	UNDEFINED
F	UNSELECTED	UNDEFINED

Tabelle 4.4.: Variantenstatus nach Selektion der Variante A

Varianten	Variantenstatus	
	Selektionszust.	Validierungszust.
A	SELECTED	UNDEFINED
B	SELECTED	REQUIRED
C	SELECTED	REQUIRED
D	UNSELECTED	UNDEFINED
E	UNSELECTED	UNDEFINED
F	UNSELECTED	UNDEFINED

Tabelle 4.5.: Variantenstatus nach der Vorwärtsvalidierung

Rückwärtsvalidierung Der Bedarf der Rückwärtsvalidierung wird im Folgenden anhand eines Beispiels genauer erläutert. Gegeben seien die Varianten A, B, C, D, E und F mit folgenden Restriktionen:

1. $A \rightarrow B$
2. $B \rightarrow C$
3. $D \rightarrow \neg C$
4. $E \rightarrow F$

Während einer Konfigurierung wird benutzerseitig Variante A selektiert. Es resultieren die Variantenstatus aus Tabelle 4.4. Da in jedem Konfigurierungsschritt die Validierung angestoßen wird, folgt nach der Selektion zunächst die Vorwärtsvalidierung.

Das Ergebnis der Vorwärtsvalidierung ist in Tabelle 4.5 dargestellt. Die Varianten B und C wurden aufgrund der Restriktionen automatisch selektiert. Wird nun die Restriktion $D \rightarrow \neg C$ erneut betrachtet, ist erkennbar, dass die Selektion der Variante D den Ausschluss der Variante C zur Folge hat. Dies steht allerdings im Widerspruch zur Restriktion $B \rightarrow C$, wodurch der Variantenstatus von C auf den Wert (SELECTED, REQUIRED) gesetzt wurde. Es muss also verhindert werden, dass Variante D selektiert werden kann.

Um dies zu erreichen, wird, wie bereits weiter oben erwähnt, nach der Vorwärtsvalidierung die Rückwärtsvalidierung angestoßen. Sie hat zur Folge, dass der Variantenstatus aller Aussagenvariablen der linken Seite nach Auswertung der

Varianten	Variantenstatus	
	Selektionszust.	Validierungszust.
A	SELECTED	UNDEFINED
B	SELECTED	REQUIRED
C	SELECTED	REQUIRED
D	UNSELECTED	EXCLUDED
E	UNSELECTED	UNDEFINED
F	UNSELECTED	UNDEFINED

Tabelle 4.6.: Variantenstatus nach der Rückwärtsvalidierung

rechten Seite festgelegt wird. Für das obige Beispiel würde somit die Variante `D` nach der Rückwärtsvalidierung den Status `(UNSELECTED, EXCULDES)` erhalten. Dies ist in Tabelle 4.6 dargestellt.

Die Schlussfolgerung für die linke Seite wird durch den logischen Wert der rechten Seite ermittelt. Hierbei wird der Wahrheitswert der rechten Seite der Restriktion überprüft. Ergibt die Überprüfung den Wert `FALSE` und kann ausgeschlossen werden, dass der Ausdruck jemals `TRUE` werden kann, dann wird für die Aussagenvariablen der linken Seite ein geeigneter Variantenstatus festgelegt. Aus den möglichen acht Status können bei der Rückwärtsvalidierung insgesamt folgende sechs Status für die Aussagenvariablen auf der linken Seite auftreten:

Selektionszustand	Validierungszustand
SELECTED	UNDEFINED
SELECTED	REQUIRED
~~SELECTED~~	~~EXCLUDED~~
SELECTED	CONFLICT
UNSELECTED	UNDEFINED
~~UNSELECTED~~	~~REQUIRED~~
UNSELECTED	EXCLUDED
UNSELECTED	CONFLICT

Aus den obigen Erläuterungen zur Vor- und Rückwärtsvalidierung ergibt sich der in Algorithmus 4.2 dargestellte Ablauf. In jedem Konfigurierungsschritt wird die proaktive Validierung angestoßen. Dabei wird zunächst für alle Restriktionen die Vorwärtsvalidierung durchgeführt und im Anschluss erfolgt die Rückwärtsvalidierung.

4.3.2. Generierungsmodell

In Abbildung 4.4 wurde als Bindungsmechanismus neben der Selektion, welche Basis für das Konfigurationsmodell aus Abschnitt 4.3.1 ist, auch die Generierung beschrieben. In diesem Abschnitt wird ein entsprechendes Generierungsmodell vorgestellt. Ziel dabei ist es, aus einem variantenreichen Softwaredokument, dessen Variabilitätsinformationen in einem Variabilitätsmodell erfasst und mit dem

Algorithmus 4.2 Validierungszyklus mit Vor- und Rückwärtsvalidierung

```
for all Restriktionen do
  if linker Teil ist TRUE then
    führe rechten Teil aus
  end if
end for
for all Restriktionen do
  if rechter Teil ist FALSE und kann nicht mehr TRUE werden then
    führe linken Teil aus
  end if
end for
```

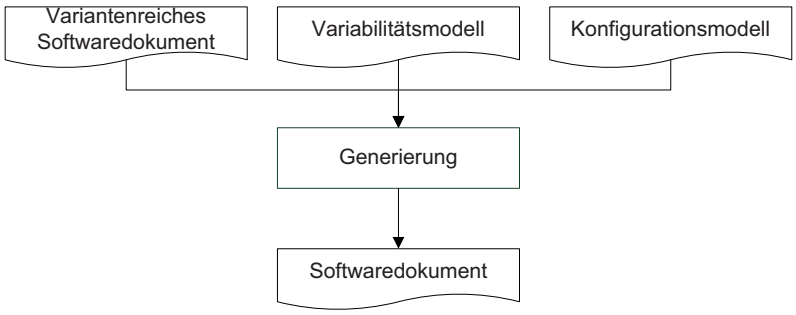


Abbildung 4.17.: Der Prozess zur Generierung von konkreten Softwaredokumenten

Dokument verknüpft sind sowie aus einem Konfigurationsmodell heraus, ein gebundenes Softwaredokument zu generieren. Abbildung 4.17 illustriert den groben Prozessablauf mit der beschriebenen Konstellation.

Bisher wurden allerdings noch keine Softwaredokumente beschrieben. Diese werden noch im Verlauf dieser Arbeit näher erläutert. Daher wird in diesem Abschnitt das Generierungsmodell ohne die Betrachtung von Softwaredokumenten dargestellt. Es bildet somit eine Basis zur Beschreibung der Generierung von gebundenen Softwaredokumenten in den entsprechenden Abschnitten.

In Abschnitt 4.2.2.2 wurde bereits die Restriktionssprache WCRL eingeführt. Es bietet eine Möglichkeit, Restriktionen in einem Variabilitätsmodell in kompakter Form zu formulieren. In dem Abschnitt wurde allerdings auch angedeutet, dass die primäre Anwendung von WCRL in dieser Arbeit nicht in der Formulierung von Restriktionen ist, sondern in der Ableitung bzw. Generierung von Softwaredokumenten liegt. Hierfür wird die effiziente Inferenzmaschine smodels, die auf WCRL-Regeln operiert, verwendet. Da nun die erforderlichen Modelle nicht in WCRL vorliegen, bedarf es einer Überführung bzw. einer Transformation. Sind die Modelle zu WCRL überführt, kann im Anschluss die Inferenzmaschine auf die Regeln angewendet werden. Das Resultat des Inferierens ist ein WCRL-Modell, das alle Informationen für ein gebundenes Softwaredokument besitzt. Es ist allerdings in der Notation von WCRL. Also ist zusätzlich eine Rücktransformation in die formale Sprache des

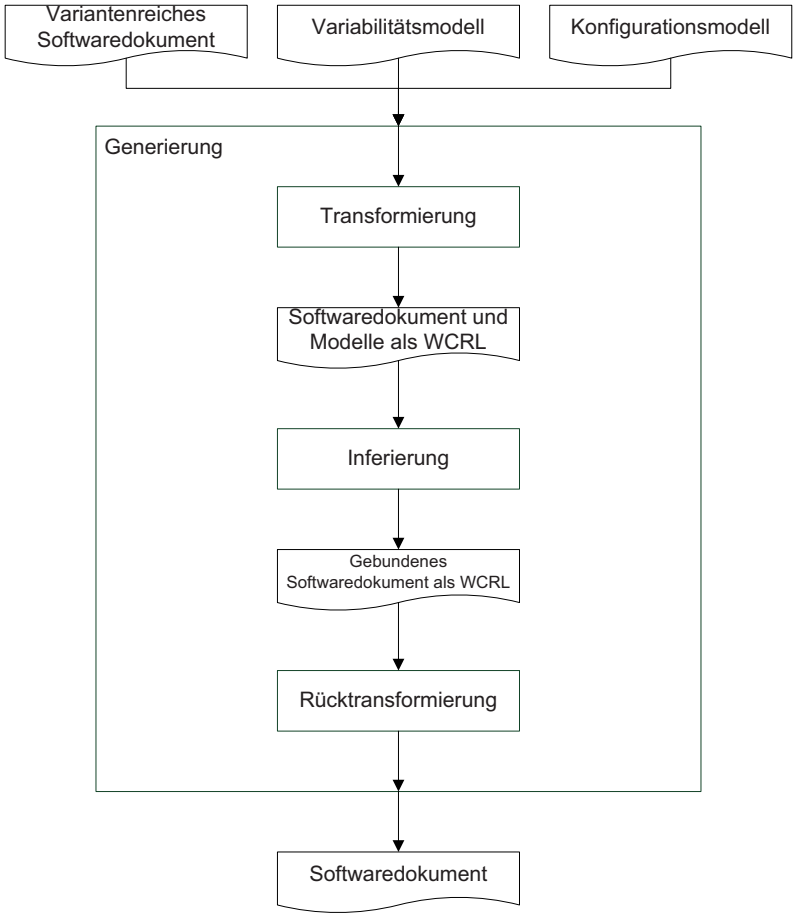


Abbildung 4.18.: Der detaillierte Ablauf des Generierungsprozesses

Softwaredokuments erforderlich.

Abbildung 4.18 zeigt den beschriebenen Ablauf. Die einzelnen Aktivitäten und Ergebnisse sind Bestandteil der Generierung. Im Folgenden werden diese Aktivitäten näher erläutert.

4.3.2.1. Transformierung

Das Ergebnis der Transformierung ist eine Wissensbasis in Form von gewichteten Restriktionsregeln. Die gesamte Aktivität wird strukturiert und das Ergebnis hierarchisch aufgebaut. So wird eine modularisierte Transformation erhalten, die nachvollziehbar ist.

Abbildung 4.19 illustriert diesen Ansatz. Zunächst wird die Eingabe zur Transformierung in drei Bestandteile zerlegt und auf diese Weise gehandhabt. Diese sind

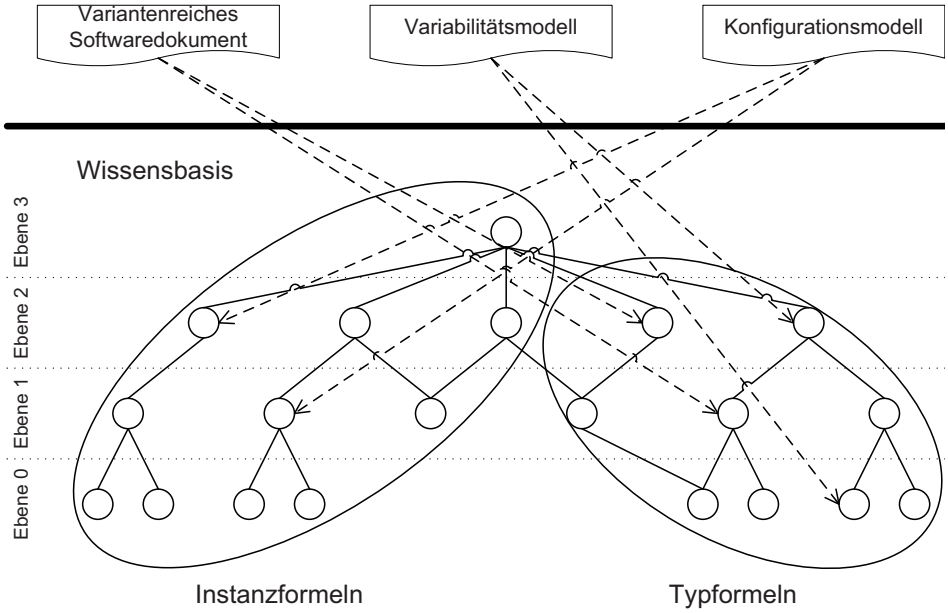


Abbildung 4.19.: Die hierarchische Struktur der Wissensbasis

das variantenreiche Softwaredokument (wie bereits erwähnt, werden betrachtete Softwaredokumente in den entsprechenden Kapiteln behandelt), das Variabilitätsmodell und das Konfigurationsmodell. Die Informationen dieser Ergebnisse werden in die Wissensbasis überführt. Da WCRL keinen Instanziierungsmechanismus und keine Instanzverwaltung bietet, wurde im Rahmen dieser Arbeit ein Konzept zur entsprechenden Instanziierung und Verwaltung realisiert. Auf diese Weise ist es möglich, nach der Inferierung aus den Instanzen das gebundene Softwaredokument abzuleiten. Dies wird im Verlauf dieses Abschnitts detaillierter erläutert. Die Wissensbasis besteht somit aus zwei Teilen:

1. Instanzformeln: Sie beinhalten alle Instanzen, die aus den Informationen des Konfigurationsmodells transformiert werden. Somit ist gewährleistet, dass stets eine ausreichende Menge an Instanzformeln vorhanden ist. Die Instanzen werden mit den entsprechenden Typen in Beziehung gesetzt.

2. Typformeln: Sie beinhalten alle Entitäten, Attribute und Beziehungen untereinander, die aus dem variantenreichen Softwaredokument und dem Variabilitätsmodell zustande kommen.

Die Wissensbasis wird nachfolgend genauer beschrieben. Im Anschluss folgen die Transformationsregeln, die alle Modelle in gewichtete Restriktionsregeln überführen.

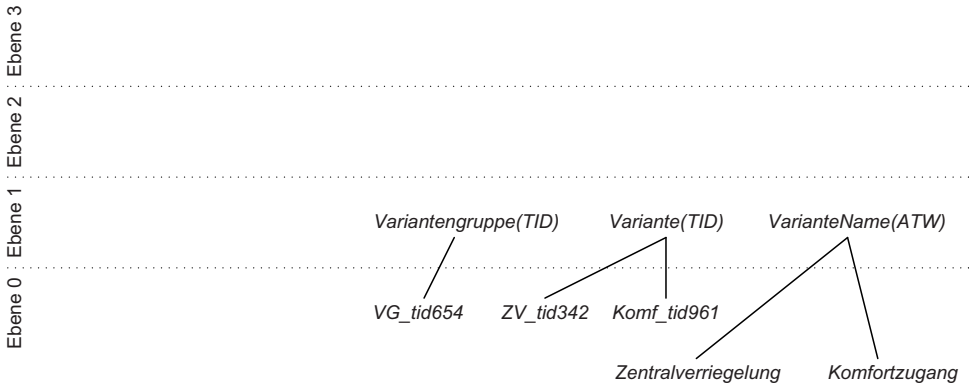


Abbildung 4.20.: Hierarchie von Typen und Werten mit zugehörigen Grundatomen

Wissensbasis Wie bereits erwähnt, ist die Wissensbasis hierarchisch strukturiert sowie in Typ- und Instanzformeln geteilt. Zur Wissensrepräsentation werden im Folgenden Relationen, Atome und Formeln eingeführt, die für die Transformationen erforderlich sind. Zunächst wird die Strukturierung der Typformeln beschrieben und im Anschluss folgen die Instanzformeln.

1. Sei $T \in RS^1$ eine einstellige Relation und $TID \in FS^0$ ein Grundatom. Ein *Typ* ist eine atomare Formel $T(TID)$, wobei T den Typ und TID die Identität des Typs bezeichnet. Die Atome $Variantengruppe(VG\_tid654)$, $Variante(ZV\_tid342)$ und $Variante(Komf\_tid961)$ sind Beispiele für Typen. Grundatome und Typen bilden die unteren Ebenen der Wissensbasis.

2. Sei $W \in RS^1$ und $ATW \in FS^0$ ein Grundatom. Ein *Wert* ist eine atomare Formel $W(ATW)$, wobei W die Bezeichnung eines Attributs und ATW der Wert des Attributs ist. Die Atome $VariantenName(Zentralverriegelung)$ und $VariantenName(Komfortzugang)$ sind Beispiele für Werte. Genau wie bei Typen bilden Werte ebenfalls die unteren Ebenen der Wissensbasis.

Abbildung 4.20 illustriert am Beispiel des Fahrzeugzugangssystems die Struktur der Wissensbasis, die aus den obigen zwei Typformeln entstanden ist. Um nun die obigen Formeln, bestehend aus einstelligen Relationen, miteinander zu verknüpfen, bedarf es an weiteren mehrstelligen Relationen. Diese werden im Folgenden beschrieben:

3. Sei $W\_T \in RS^2$ eine zweistellige Relation und seien $ATW, TID \in FS^0$ Grundatome. Ein *Attribut* ist eine atomare Formel $W\_T(ATW, TID)$, wobei W_T die Bezeichnung des Attributs sowie dessen Typ, ATW den Wert des Attributs und TID die Identität des Typs darstellen. $Name\_Id(Zentralverriegelung, ZV\_tid342)$ und $Name\_Id(Komfortzugang, Komf\_tid961)$ sind Beispiele für Attribute. Sie sind in der Struktur in der nächsthöheren Ebene angesiedelt.

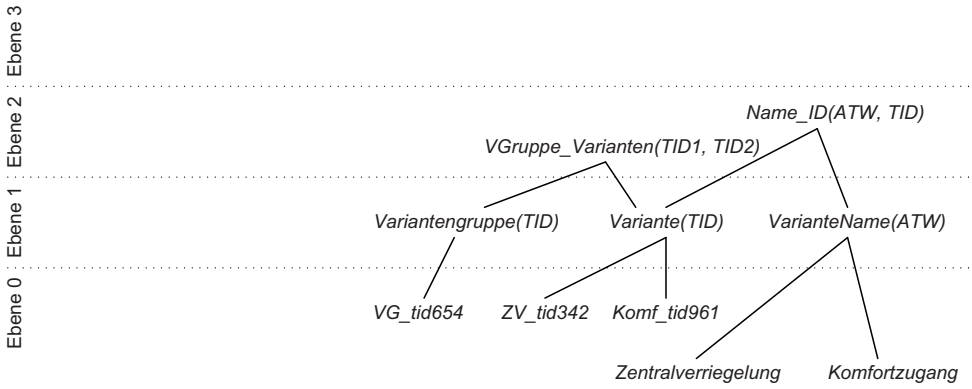


Abbildung 4.21.: Erweiterung der Hierarchie mit mehrstelligen Relationen für Attribute und Typ-Assoziationen

4. Sei $TA \in RS^n$ eine n-stellige Relation und seien $TID_1, \ldots, TID_n \in FS^0$ Grundatome. Eine *Typassoziation* ist eine atomare Formel $TA(TID_1, \ldots, TID_n)$, wobei TA die Bezeichnung der Assoziation und $TID_1, \ldots, TID_n$ die Identitäten der Typen sind. $VGruppe\_Varianten(VG\_tid654, ZV\_tid342)$, $VGruppe\_Varianten(VG\_tid654, Komf\_tid961)$ sind Beispiele für Assoziationen. Sie sind in der Struktur in der nächsthöheren Ebene angesiedelt.

Abbildung 4.21 erweitert das Fahrzeugzugangssystem durch die Einführung von Attributen und Typassoziationen. Auf diese Weise können Typformeln der unteren Ebenen miteinander verknüpft werden. Als Nächstes werden notwendige Formeln für den Instanziierungsmechanismus eingeführt, d.h. Instanzformeln.

5. Seien $I \in RS^1$ eine einstellige Relation und $IID \in FS^0$ ein Grundatom. Eine *Instanz* ist eine atomare Formel $I(IID)$, wobei I die Bezeichnung der Instanz und IID die Identität der Instanz darstellen. $Instanz(Iid001)$, $Instanz(Iid002)$, $Instanz(Iid003)$, ... sind Beispiele für Instanzen. Sie sind den unteren Ebenen der Hierarchie zugehörig.

Abbildung 4.22 erweitert das Beispiel um die Menge der Instanzen. Da ähnlich wie bei Typassoziationen auch Assoziationen für Instanzen existieren können, werden sogenannte Instanzassoziationen eingeführt.

6. Sei $IA \in RS^n$ eine n-stellige Relation und seien $IID_1, \ldots, IID_n \in FS^0$ Grundatome. Eine *Instanzassoziation* ist eine atomare Formel $IA(IID_1, \ldots, IID_n)$, wobei IA die Bezeichnung der Assoziation und $IID_1, \ldots, IID_n$ die Identitäten der Instanzen darstellen. $IVGruppe\_IVarianten(Iid001, Iid002)$ und $IVGruppe\_IVarianten(Iid001, Iid003)$ sind Beispiele für Instanzassoziationen. Sie sind in der Struktur in der nächsthöheren Ebene angesiedelt.

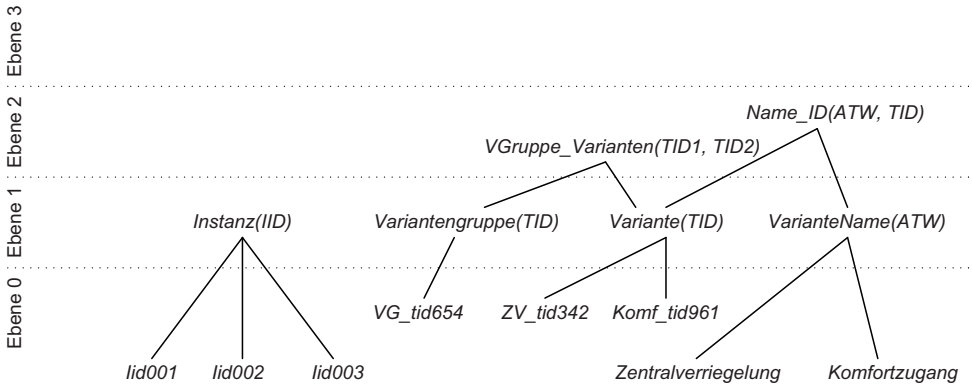


Abbildung 4.22.: Erweiterung der Hierarchie durch Instanzen

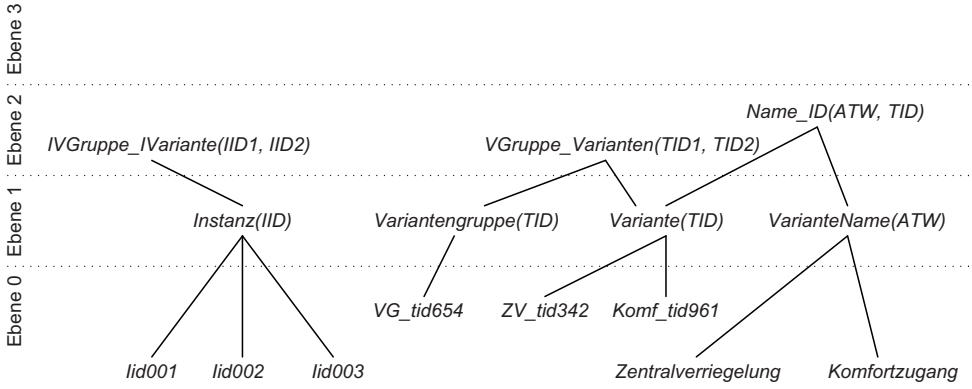


Abbildung 4.23.: Erweiterung der Hierarchie durch Instanzassoziationen

Abbildung 4.23 fügt in die Wissensbasis Instanzassoziationen hinzu. Sie korrespondieren zu den Typassoziationen. Um nun Instanzen und Typen miteinander zu verknüpfen, ist eine weitere Formulierung erforderlich. Diese sind sogenannte Instanz-Typ-Assoziationen und werden im Folgenden vorgestellt.

7. Sei $I\_T \in RS^{2n}$ und seien $IID_1, \ldots, IID_n, TID_1, \ldots, TID_n \in FS^0$ Grundatome. Eine *Instanz-Typ-Assoziation* ist eine Formel $I\_T(IID_1, \ldots, IID_n, TID_1, \ldots, TID_n)$, wobei I_T die Bezeichnung der Instanz-Typ-Assoziation, $IID_1, \ldots, IID_n$ Identitäten von Instanzen und $TID_1, \ldots, TID_n$ Identitäten von Typen sind. $Instanz\_Typ(Iid001, VG\_tid654)$ ist ein Beispiel für eine Instanz-Typ-Assoziation. Sie sind in den höheren Ebenen der Struktur zugeordnet.

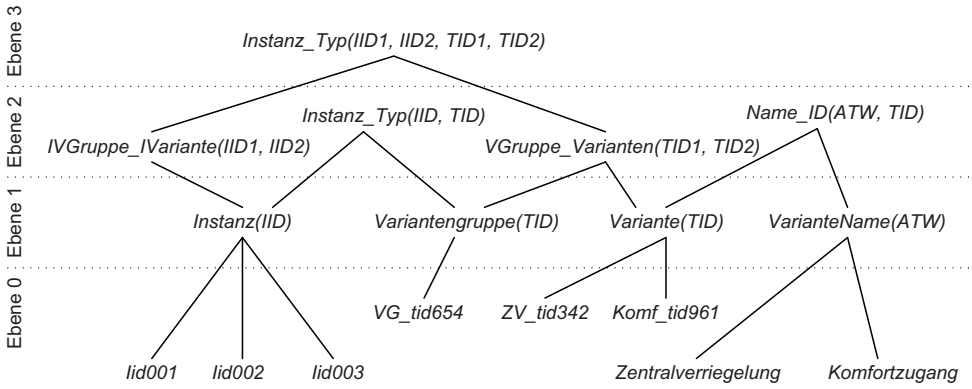


Abbildung 4.24.: Erweiterung der Hierarchie durch Instanz-Typ-Assoziationen

Abbildung 4.24 stellt die letzte Erweiterung der Wissensbasis dar. Mit der Einführung von Instanz-Typ-Assoziationen können beliebige Kardinalitäten abgedeckt und so in kompakter Form abgebildet werden.

Die Wissensbasis wird auf diese Weise zum einen in Typ- und zum anderen in Instanzformeln aufgeteilt. Somit ist sie geeignet strukturiert und hierarchisch aufgestellt. Um die Wissensbasis entsprechend dieser Überlegungen aufbauen zu können, werden im Weiteren die Transformationsregeln bestimmt, die zur Überführung in WCRL erforderlich sind.

4.3.2.2. Transformationsregeln

Nachdem nun die Eingabedokumente sowie die hierarchische Strukturierung der Wissensbasis diskutiert wurden, können in diesem Abschnitt die Transformationsregeln definiert werden. Auf diese Weise wird die Eingabe in gewichtete Restriktionsregeln überführt, sodass diese zur weiteren Verarbeitung durch die Inferenzmaschine verwendet werden können. Abbildung 4.25 gibt einen groben Überblick über den vorgesehenen Ablauf. Die Transformationsregeln werden in zwei Arten unterteilt:

1. Modellspezifische Transformationsregeln

2. Ontologische Transformationsregeln

Modellspezifische Transformationsregeln sind abhängig von den Eingabemodellen. Auch wenn die Transformation die gleiche Struktur erzeugt, unterscheiden sie sich für jedes verschiedene Modell. Die Transformation ist daher spezifisch. Variantenreiche Softwaredokumente, Variabilitätsmodelle sowie Konfigurationsmodelle werden allesamt nach modellspezifischen Regeln transformiert.

Ontologische Transformationsregeln sind im Gegensatz dazu unabhängig von modellspezifischen Eigenschaften. Hier werden Regeln erfasst, die eine konsistente

Inferenz ermöglichen. Ein Beispiel hierfür ist die Sicherstellung, dass jede Instanz genau einen Typ hat.

Im Folgenden werden sechs modellspezifische und drei ontologische Transformationsregeln beschrieben. Es sei an dieser Stelle erwähnt, dass diese Regeln im weiteren Verlauf dieser Arbeit noch ausgeweitet werden. Insbesondere werden zusätzliche Regeln definiert, wenn die spezifischen Eigenschaften der im Referenzprozess eingeführten Softwaredokumente erläutert werden.

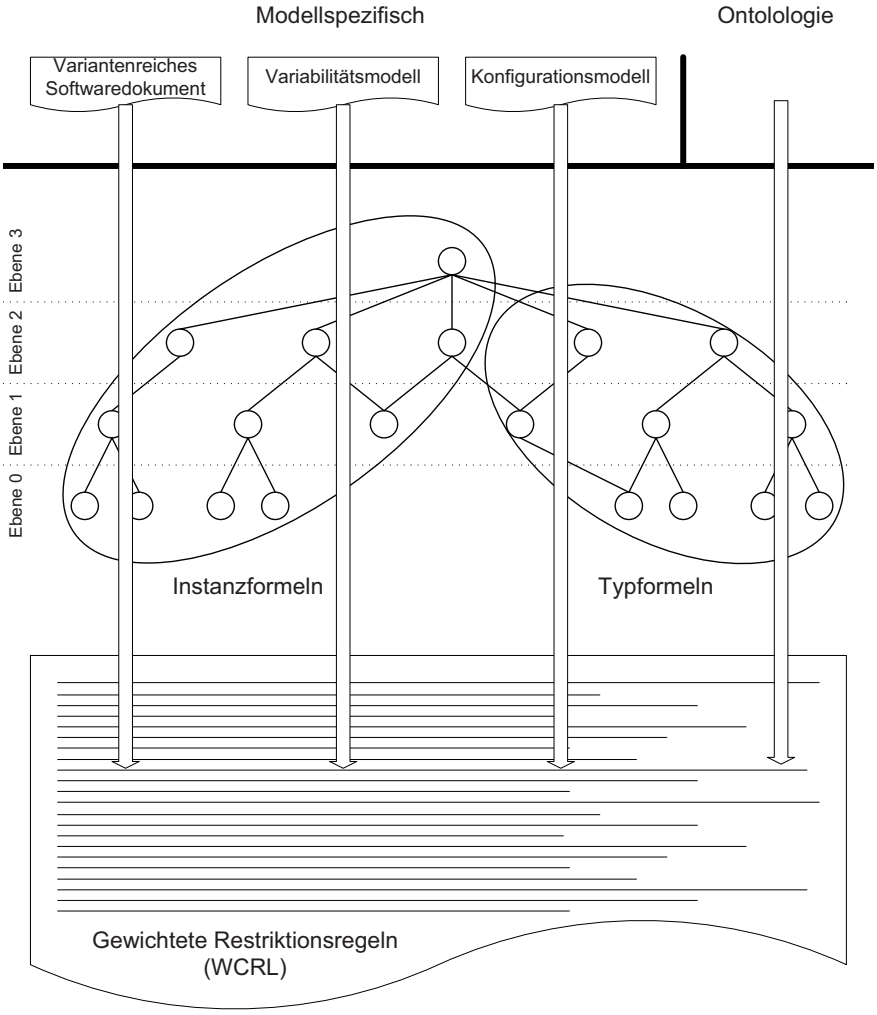


Abbildung 4.25.: Die Unterteilung der Transformationsregeln in modellspezifische einerseits und ontologische andererseits

Modellspezifische Transformationsregel 1 Entitäten in den Modellen werden zu Fakten der Form

$$T(TID) \leftarrow$$

transformiert.

Alle Entitäten wie beispielsweise Variationspunkte, Variantengruppen und Varianten werden in diese Form transformiert. Sie korrespondieren zu der Menge aller Typen aus der Wissensbasis. Abbildung 4.26 illustriert ein Beispiel.

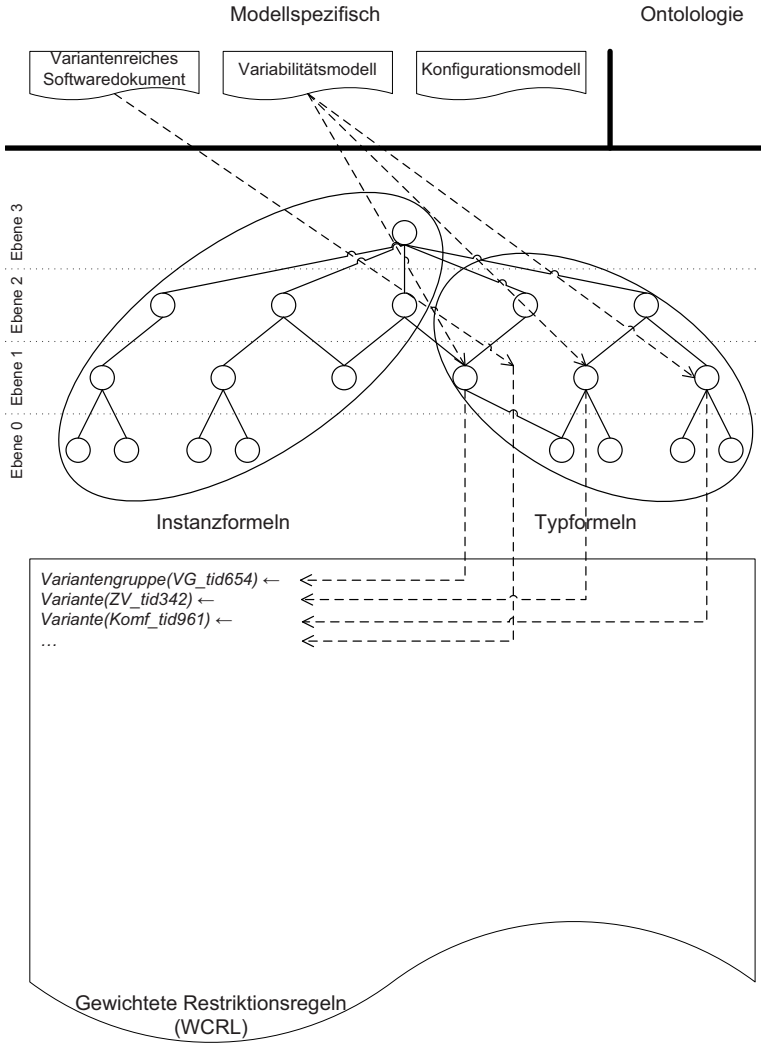


Abbildung 4.26.: Ein Beispiel für die modellspezifische Transformationsregel 1

Modellspezifische Transformationsregel 2 Die Namen aller Entitäten werden zu Fakten der Form

$$W(ATW) \leftarrow$$

transformiert.

Die Namen von Variationspunkten, Varianten usw. werden in diese Form transformiert. Sie korrespondieren zu der Menge aller Werte aus der Wissensbasis. Abbildung 4.27 illustriert ein Beispiel.

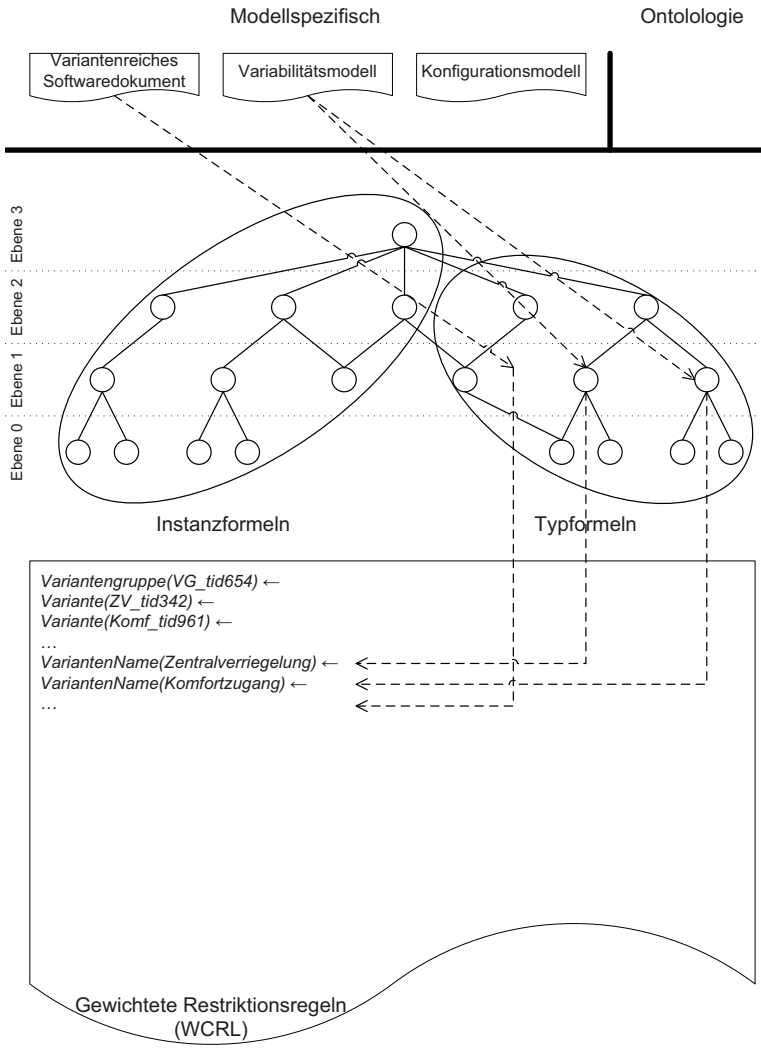


Abbildung 4.27.: Ein Beispiel für die modellspezifische Transformationsregel 2

Modellspezifische Transformationsregel 3 Die hierarchischen Strukturen der Eingabemodelle werden zu Fakten der Form

$$W\_T(ATW, TID) \leftarrow \text{ bzw. } TA(TID_1, \ldots, TID_n) \leftarrow$$

transformiert.

Der Zusammenhang zwischen Variantengruppen und Varianten oder Variantennamen mit Typidentitäten werden auf diese Weise erfasst. Sie korrespondieren also zu der Menge aller Attribute und Typassoziationen aus der Wissensbasis. Abbildung 4.28 illustriert ein Beispiel.

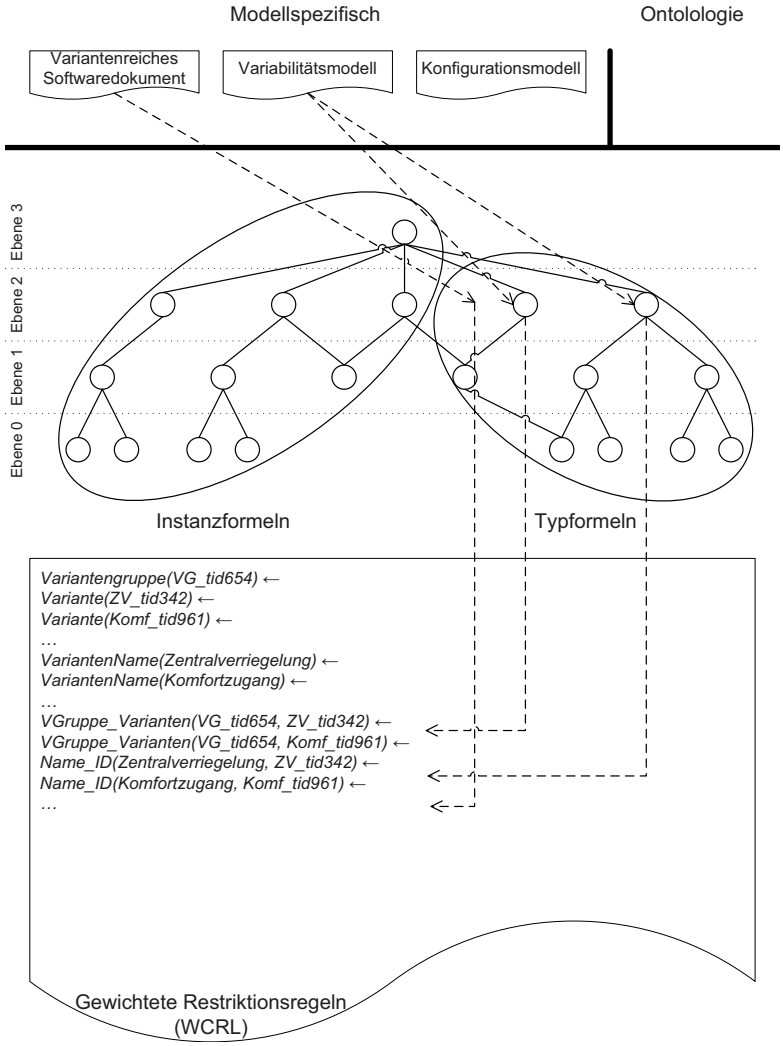


Abbildung 4.28.: Ein Beispiel für die modellspezifische Transformationsregel 3

Modellspezifische Transformationsregel 4 Gruppenkardinalitäten werden zu Fakten der Form

$$L\{TA(TID_1, \ldots, TID_n) : T(TID)\}U \leftarrow$$

transformiert. L und U korrespondieren zu den Grenzen der Gruppenkardinalität und die Typassoziation verknüpft Gruppen mit ihren Varianten.

Auf diese Weise wird der Zusammenhang zwischen Gruppenkardinalitäten und Varianten hergestellt. Sie werden aus der Menge der Typen und Typassoziationen aus der Wissensbasis erstellt. Abbildung 4.29 illustriert ein Beispiel.

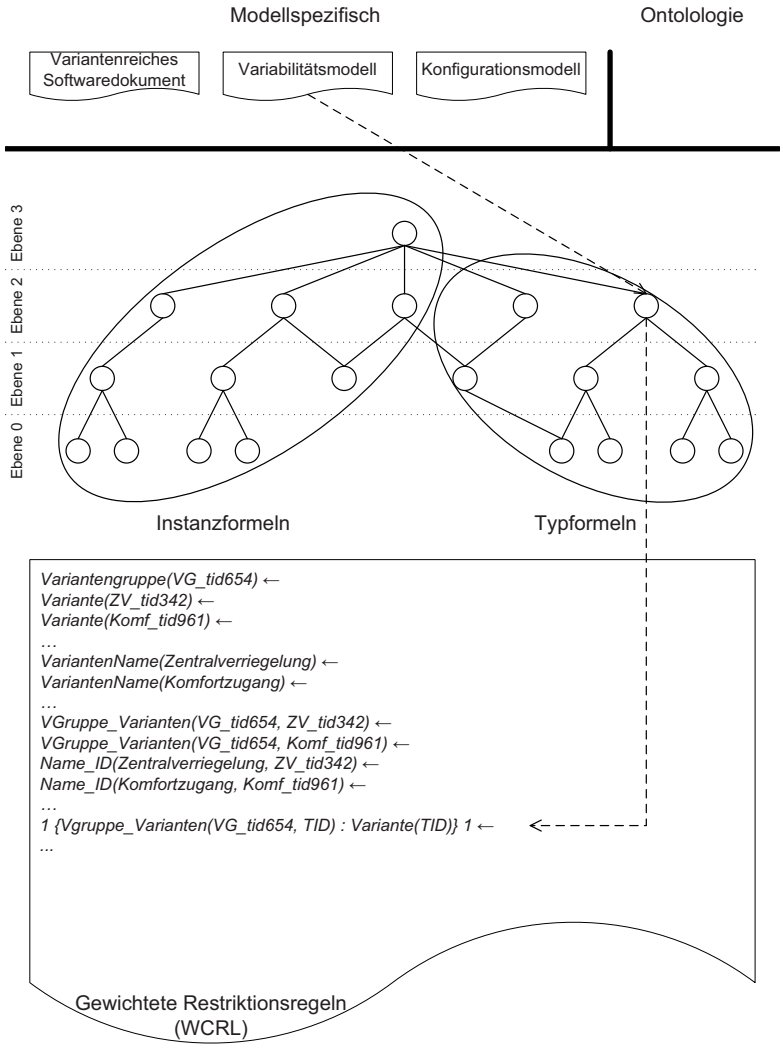


Abbildung 4.29.: Ein Beispiel für die modellspezifische Transformationsregel 4

Modellspezifische Transformationsregel 5 Variantenkradinalitäten werden zu Fakten der Form

$$L\{I\_T(IID, TID) : I(IID)\}U \leftarrow$$

transformiert. L und U korrespondieren zu den Grenzen der Variantenkardinalität. Die Instanz-Typ-Assoziation verknüpft die Varianten mit Instanz- und Typidentitäten.

Auf diese Weise wird der Zusammenhang zwischen Variantenkardinalitäten und Varianten hergestellt. Sie werden aus der Menge der Instanzen und Instanz-Typ-Assoziationen aus der Wissensbasis erstellt. Abbildung 4.30 illustriert ein Beispiel.

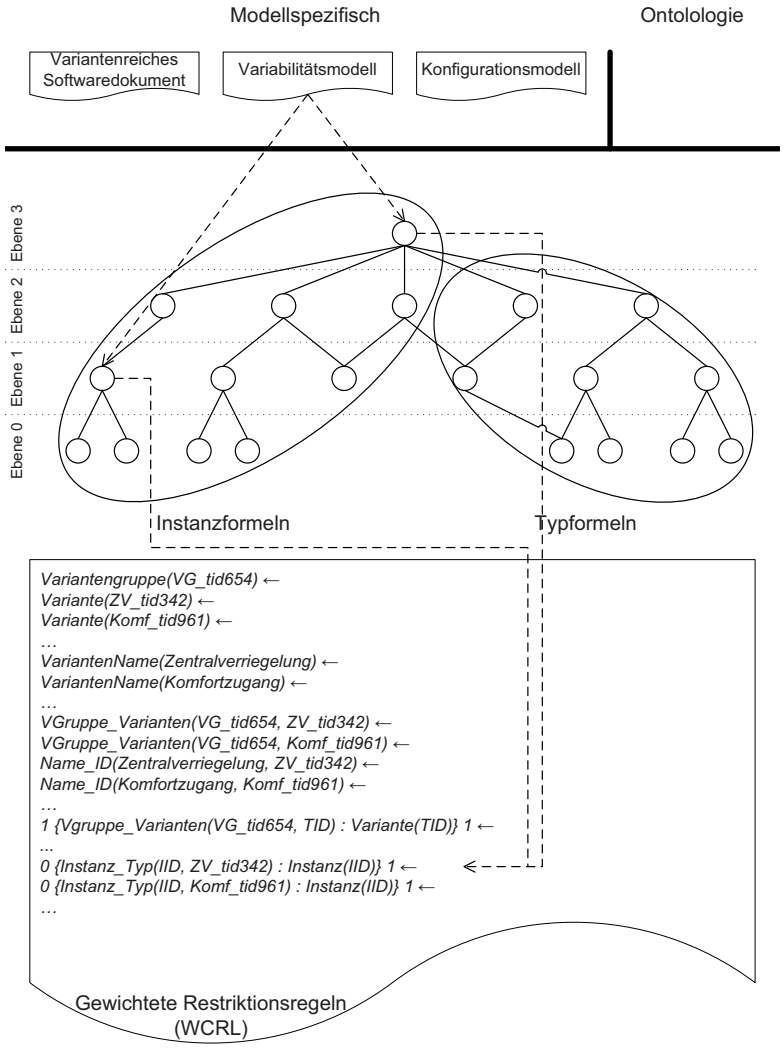


Abbildung 4.30.: Ein Beispiel für die modellspezifische Transformationsregel 5

Modellspezifische Transformationsregel 6 Selektierte Varianten werden zu Fakten der Form

$$L\{I\_T(IID, TID) : I(IID)\}U \leftarrow$$

transformiert. L und U korrespondieren zu den Grenzen der Variantenkardinalität. Die Instanz-Typ-Assoziation verknüpft die Varianten mit Instanz- und Typidentitäten.

Somit wird der Zusammenhang zwischen selektierten Varianten und ihren Kardinalitäten hergestellt. Sie werden durch die Menge der Instanzen und Instanz-Typ-Assoziationen aus der Wissensbasis erstellt. Abbildung 4.31 illustriert ein Beispiel.

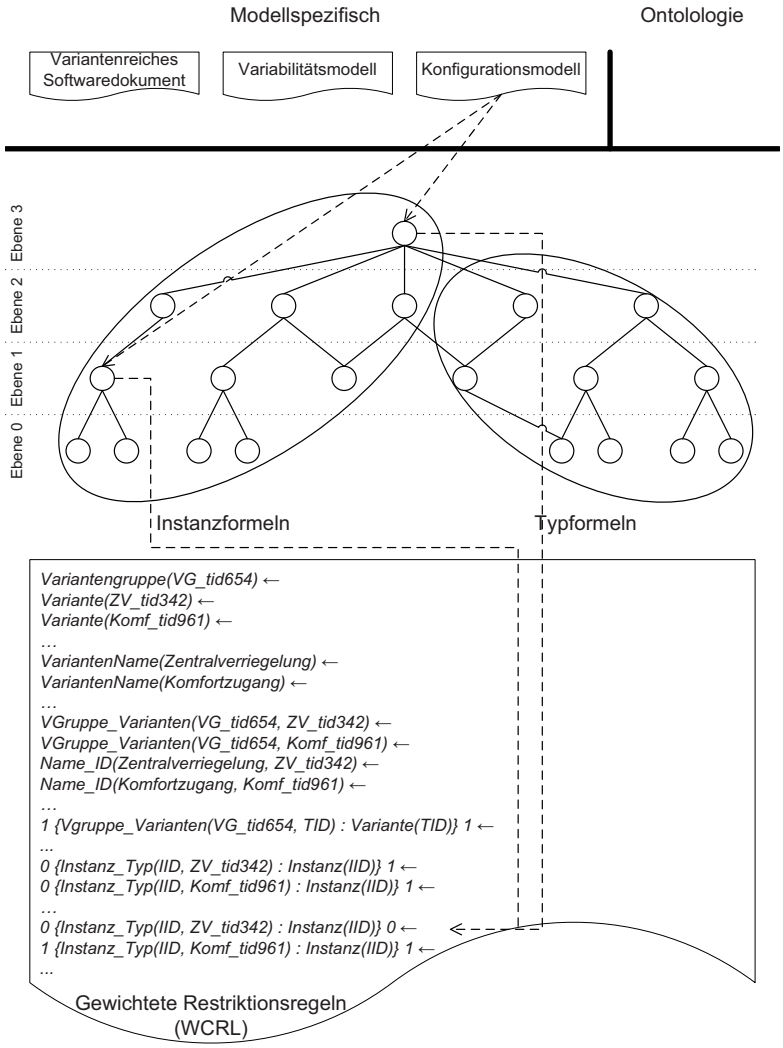


Abbildung 4.31.: Ein Beispiel für die modellspezifische Transformationsregel 6

Ontologische Transformationsregel 1 Es werden n Instanzen, $n \in N$, vor jeder Generierung für die Wissensbasis erzeugt und deklariert. Der Grund hierfür ist, dass es nicht möglich ist, zur Laufzeit neue Atome zur Wissensbasis hinzuzufügen.

$$Instanz(IID_1), Instanz(IID_2), \ldots, Instanz(IID_n)$$

Ontologische Transformationsregel 2 Jede Instanz hat genau einen Typ.

$$1\{Instanz\_Typ(IID, TID) : T(TID)\}1 \leftarrow Instanz\_Typ(IID, TID) \wedge Instanz(IID) \wedge T(TID)$$

Ontologische Transformationsregel 3 Jede Instanz in einer Instanz-Typ-Assoziation hat genau einen korrespondierenden Typ.

$$\begin{aligned}
1\{Instanz_Typ(IID_1, TID_1)\}1 \leftarrow\ & Instanz_Typ(IID_1, \ldots, IID_n, TID_1, \ldots, TID_n) \\
& \wedge Instanz(IID_1) \wedge \ldots \wedge Instanz(IID_n) \\
& \wedge T(TID_1) \wedge \ldots \wedge T(TID_n) \\
1\{Instanz_Typ(IID_2, TID_2)\}1 \leftarrow\ & Instanz_Typ(IID_1, \ldots, IID_n, TID_1, \ldots, TID_n) \\
& \wedge Instanz(IID_1) \wedge \ldots \wedge Instanz(IID_n) \\
& \wedge T(TID_1) \wedge \ldots \wedge T(TID_n) \\
& \ldots \\
1\{Instanz_Typ(IID_n, TID_n)\}1 \leftarrow\ & Instanz_Typ(IID_1, \ldots, IID_n, TID_1, \ldots, TID_n) \\
& \wedge Instanz(IID_1) \wedge \ldots \wedge Instanz(IID_n) \\
& \wedge T(TID_1) \wedge \ldots \wedge T(TID_n)
\end{aligned}$$

4.3.2.3. Inferierung

Die Inferierung ist die Aktivität, in der die zu WCRL transformierten Entitäten zu einem gebundenen Dokument in WCRL abgeleitet werden. Hierzu wird die Inferenzmaschine smodels eingesetzt. Dabei werden Modelle $S_1, \ldots, S_n$ generiert, welche die Menge aller gewichteten Restriktionsregeln erfüllen. Diese Modelle entsprechen jeweils einem gebundenen Modell in WCRL, das für die Rücktransformierung verwendet werden kann. Abbildung 4.32 illustriert mögliche Ergebnisse nach der Inferierung.

Falls die Inferierung keine validen Modelle erzeugt, so deutet dies auf eine ungültige Konfiguration in der Eingabe. Dieser Fall wird allerdings durch die in Abschnitt 4.3.1.3 beschriebene proaktive Validierung verhindert.

Wird genau ein valides Modell erzeugt, kann dieses für die Rücktransformierung übergeben werden.

Bei mehr als einem validen Modell, also $n > 1$, bedarf es an geeigneten Maßnahmen für die Weiterbehandlung. Eine Möglichkeit ist es, die Berechnung nach der

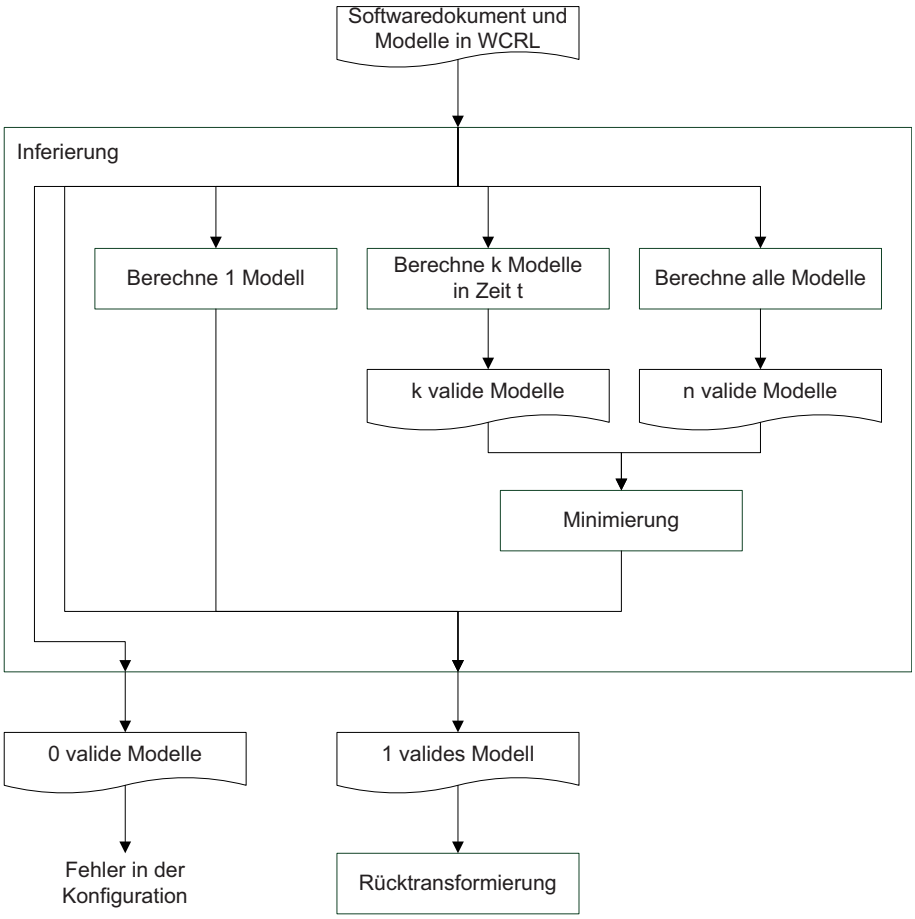


Abbildung 4.32.: Mögliche Ergebnisse der Inferierung und aus diesen resultierende Maßnahmen

Ermittlung des ersten validen Modells sofort abzubrechen. Auf diese Weise können zeitintensive Berechnungen eingespart werden. Diese Maßnahme hat allerdings den Nachteil, dass nicht notwendigerweise ein minimales Modell gefunden wird.

Weiterhin können Modelle in einem vordefinierten Zeitlimit berechnet werden. Auch somit können Berechnungszeiten gekürzt werden. Aus den Ergebnissen kann dann ein minimales Modell ermittelt werden. Auch wenn es kein optimales Ergebnis ist, wird es in vielen Fällen zu einer Verbesserung führen.

Schließlich können alle validen Modelle berechnet werden und aus diesen das optimale Ergebnis ermittelt werden. Gibt es mehrere optimale Ergebnisse, so kann beispielsweise per Zufall ein valides Modell gewählt werden. Diese Maßnahme erfordert die maximale Berechenzeit, liefert allerdings stets ein optimales Ergebnis.

Die Anwendung der Inferierung wird in dieser Arbeit erneut aufgegriffen, wenn neben dem Variabilitätsmodell und Konfigurationsmodell auch variantenreiche Softwaredokumente behandelt werden. Somit kann auch das Resultat der Inferierung besser nachvollzogen werden.

4.3.2.4. Rücktransformierung

Nachdem ein valides Modell durch die Inferierung erzeugt wurde, muss im nächsten Schritt das Modell in das Format des Zieldokuments überführt werden. Dazu muss das WCRL-Modell zunächst gelesen und syntaktisch analysiert werden. Zur syntaktischen Analyse bedarf es nach einem Parser, der die Regeln analysiert. Dadurch wird es möglich, anhand von Transformationsregeln das WCRL-Modell in ein Softwaredokument zu transformieren.

Da diese Aktivität sehr stark vom Zieldokument beeinflusst ist, zum Beispiel die Transformationsregeln, wird wie auch schon in anderen Teilen dieser Arbeit, die Rücktransformierung dann näher erläutert, wenn die entsprechenden Softwaredokumente eingeführt werden.

4.4. Realisierung

Zur Realisierung der in diesem Kapitel vorgestellten Ansätze hinsichtlich der Modellierung und Bindung von Variabilität wurde eine modellgetriebene Entwicklung mit entsprechender Codegenerierung herangezogen [SVEH07]. Als Entwicklungsumgebung wurde Eclipse [Plu03, Val04, Hol04b, Hol04a, Cub05, Bur05, Dau06, CR06] mit den Plug-Ins Eclipse Modeling Framework (EMF) und Graphical Modeling Framework (GMF) aus dem Eclipse Modeling Project (EMP) [Gro09, SBPM08] sowie EMFText [EMF, Dok11] gewählt.

Im Folgenden werden die in Abschnitt 4.2 und Abschnitt 4.3 vorgestellten Modelle hinsichtlich ihrer Realisierungsaspekte beschrieben.

4.4.1. Variabilitätsmodell

Zur Realisierung des Variabilitätsmodells wurde die EMF-Technologie eingesetzt. Dabei wird durch die Modellierung des Metamodells der zugehörige Code zur Erhaltung eines textuellen Baumlisteneditors generiert. Dieser erfüllt alle die zur Modellierung eines Variabilitätsmodells erforderlichen Konzepte.

Das Metamodell wird in Form eines Klassendiagramms realisiert und beschreibt die erforderlichen Konzepte. Sie wurde bereits in Abschnitt 4.2.1 eingeführt (vgl. Abbildung 4.9) und wird daher hier nicht erneut dargestellt.

Das Rahmenwerk generiert aus dem Metamodell Java-Code, der eine Implementierung des Metamodells darstellt. Darüber hinaus wird der entsprechende Code zur Repräsentation und Persistenz generiert. Weiterhin wird ein Editier-Code generiert, mit dem sich das Modell verändern und syntaktisch validieren lässt. Da diese Generierung vollautomatisch stattfindet, wurde hier auf Erläuterungen zu Einzelheiten bzgl. der Codefragmente verzichtet.

Abbildung 4.33 zeigt das entstandene Werkzeug zur Variabilitätsmodellierung. Wie bereits erwähnt, ist der Editor im Mittelpunkt des Werkzeugs und erlaubt das Modellieren von Variabilität. Nicht alle Eigenschaften werden im Editor realisiert. Die Definition von Namen, Kardinalitäten etc. werden in einer Eigenschaftsansicht (Properties) festgelegt.

4.4.2. Restriktionsmodell

Zur Modellierung der Restriktionen wurde zusätzlich zu EMF die EMFText-Technologie herangezogen. Sie ermöglicht die Definition einer textuellen Syntax eines in EMF beschriebenen Metamodells sowie die Codegenerierung mit einem Parser zum Lesen und Speichern der textuellen Ausdrücke, einem Printer zum Laden der Modellinstanzen sowie einem Editor mit automatischer Codevervollständigung, Syntaxhervorhebung, Validierung etc.

Zur Codegenerierung wird das in EMF spezifizierte Metamodell und die konkrete Syntaxdefinition benötigt. Abbildung 4.34 gibt einen Überblick der erforderlichen Aktivitäten, um einen textuellen Editor zu generieren.

Nachdem ein EMFText-Projekt erzeugt wurde, besteht der nächste Schritt aus der Definition der abstrakten Syntax. Dies entspricht dem Metamodell in Form eines Klassendiagramms. Sie wird mithilfe von EMF erstellt. In Abbildung 4.12 und Abbildung 4.13 wurden diese bereits eingeführt, sodass sie hier nicht noch einmal beschrieben werden.

Der nächste Schritt besteht aus der Definition der konkreten Syntax. Listing 4.1 stellt einen Teil dieser Definition dar. Sie besteht aus mehreren Blöcken, die im Folgenden näher erläutert werden:

1. Konfigurationsblock:
   Der Konfigurationsblock umfasst die Zeilen 1-3. Hier wird der Name der Syntax festgelegt (`cl`), der später auch als Dateierweiterung dient. Weiterhin wird der Pfad zum Generatormodell angegeben, welcher in EMF erzeugt wurde, um den

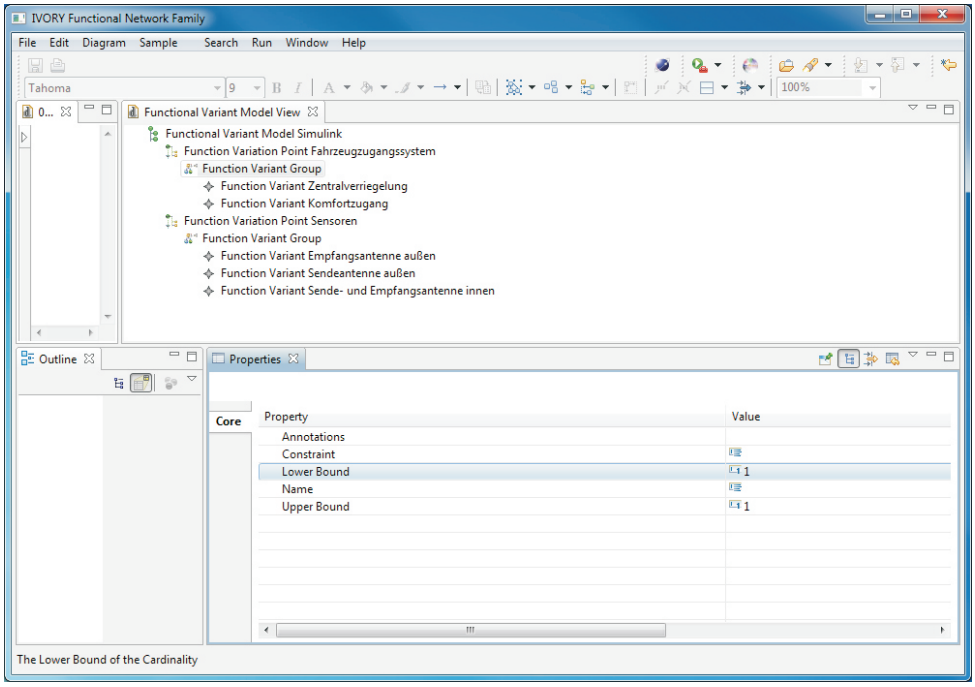


Abbildung 4.33.: Der Editor zur Variabilitätsmodellierung

Modellcode zu generieren. Schließlich wird das Startsymbol festgelegt, welches dem Wurzelelement im Metamodell entspricht. Auf diese Weise können alle weiteren Elemente über den Startpunkt erreicht werden. Die Klasse `ConstraintRule` stellt das Wurzelelement des Metamodells aus Abbildung 4.13 dar. Mit den Zeilen 2 und 3 wird also eine Assoziation zum Metamodell hergestellt.

2. Optionsblock:
   Der Optionsblock umfasst die Zeilen 5-11. Er ist optional und dient primär zum Festlegen von bestimmten Optionen für den Generator. Durch die Option `reloadGeneratorModel = „false“` wird verhindert, dass das Generatormodell des Metamodells für die EMFText-Generierung jedes Mal neu geladen wird. Würde das Metamodell häufig geändert werden, so wäre das Setzen dieser Option auf `true` angebracht. Die Option `tokenspace = „1“` definiert die standardmäßige Anzahl der Leerzeichen zwischen Tokens. Diese beträgt in der Regel den Wert 1. Nichtsdestotrotz ist es möglich, in dem sogenannten Regelblock (wird im Verlauf noch ausführlich behandelt) die Anzahl der Leerzeichen für jedes Token zu redefinieren. Die `override`-Operationen werden zur Anpassung der Editoreigenschaften definiert.

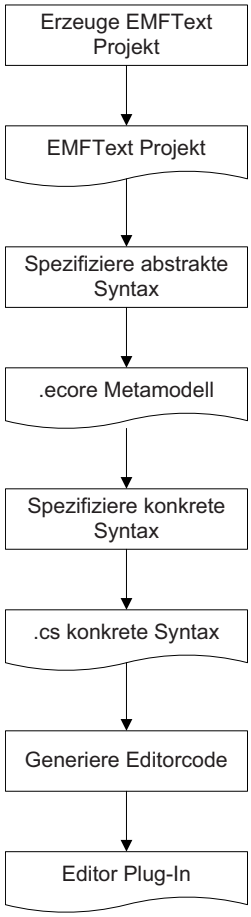


Abbildung 4.34.: Der EMFText-Prozess im Überblick

3. Tokenblock:
   Der Tokenblock umfasst die Zeilen 13-16. Er ist ebenfalls optional und dient zur Festlegung von Bezeichnern und Zahlen. `INTEGER_LITERAL` definiert Ganzzahlen für die Angabe von Kardinalitäten. Über `SIMPLE_NAME` werden die Aussagenvariablen (also die Varianten) notiert.

4. Tokenstyleblock:
   Der Tokenstyleblock umfasst die Zeilen 18-31. Er ist optional und dient zur Definition von Schriftfarben und Schriftarten für Token und Schlüsselwörter. In dem generierten Editor werden somit Schlüsselwörter und Token von anderen Textelementen entsprechend der Definition hervorgehoben.

Zudem wurden zur besseren Übersicht Konventionen zur Bindungsstärke von Entitäten einer Regel festgelegt, damit die Klammerung ausgelassen werden kann.

```
1  SYNTAXDEF cl
2  FOR <http://se.rwth-aachen.de/ivory/constraints/1.0> <constraints.genmodel>
3  START ConstraintRule
4
5  OPTIONS {
6          reloadGeneratorModel = "false";
7          tokenspace = "1";
8          overrideCodeCompletionHelper ="false";
9          overrideAttributeValueProvider = "false";
10         overrideMarkerHelper = "false";
11 }
12
13 TOKENS {
14         DEFINE INTEGER_LITERAL  $('1'..'9') ('0'..'9')* | '0'$;
15         DEFINE SIMPLE_NAME      $('A'..'Z'|'a'..'z'|'_') ('A'..'Z'|'a'..'z
               '|'0'..'9'|'_')*$;
16 }
17
18 TOKENSTYLES {
19         "SIMPLE_NAME" COLOR #058005, BOLD;
20         "INTEGER_LITERAL" COLOR #050080, BOLD;
21         "->" COLOR #800040, BOLD;
22         "!"  COLOR #800040, BOLD;
23         "&"  COLOR #800040, BOLD;
24         "|"  COLOR #800040, BOLD;
25         "^"  COLOR #800040, BOLD;
26         "("  COLOR #800040, BOLD;
27         ")"  COLOR #800040, BOLD;
28         "["  COLOR #050080, BOLD;
29         ".." COLOR #050080, BOLD;
30         "]"  COLOR #050080, BOLD;
31 }
```

Listing 4.1: Schlüsselwörter und Optionen für die konkrete Syntax des Restriktionsmodells

Junktor	Gewicht	Erläuterung
$\neg$	5	bindet am stärksten
$\wedge$	4	
$\vee$	3	
$\oplus$	2	
$\rightarrow$	1	bindet am schwächsten

Tabelle 4.7.: Festlegung der Bindungsstärke für Junktoren

Klammern sowie Literale binden stärker als Junktoren. Bei den Junktoren bindet die einstellige Negationsoperation $\neg$ am stärksten. Darauf folgen die $\wedge$-, $\vee$-, $\oplus$- und $\rightarrow$-Operatoren. Tabelle 4.7 illustriert die Bindungsstärken der Junktoren.

In der Spezifikation der konkreten Syntax werden unter anderem auch diese Bindungsstärken angegeben. Listing 4.2 zeigt den noch ausstehenden Teil zur

```
1  RULES {
2          @Operator(type="primitive", weight="6", superclass="ConstraintExp")
3          NamedLiteralExp ::= ("[" #0 ((LowerBound[INTEGER_LITERAL] #0 ".." #0
               UpperBound[INTEGER_LITERAL]) | LowerBound[INTEGER_LITERAL])   #0 "]")?
               #0  cExp[SIMPLE_NAME];
4
5          @Operator(type="primitive",weight="1", superclass="ConstraintRule")
6          ImpliesRule     ::= (#0 source #0 "->")? #0 target #0;
7
8          @Operator(type="binary_left_associative",weight="2",  superclass="
               ConstraintExp")
9          XorExp      ::= terms #0 "^" #0 terms;
10
11         @Operator(type="binary_left_associative",weight="3",  superclass="
               ConstraintExp")
12         OrExp       ::= terms #0 "|" #0 terms;
13
14         @Operator(type="binary_left_associative",weight="4", superclass="
               ConstraintExp")
15         AndExp      ::= terms #0 "&" #0 terms;
16
17         @Operator(type="unary_prefix", weight="5", superclass="ConstraintExp")
18         NotExp          ::= "!" #0 body ;
19
20         @Operator(type="primitive", weight="6", superclass="ConstraintExp")
21         NestedExp   ::= "(" #0 body #0 ")";
22 }
```

Listing 4.2: Die Regeln der konkreten Syntax des Restriktionsmodells

Definition der konkreten Syntax, dem sogenannten Regelblock.

5. Regelblock:
   Der Regelblock umfasst in Listing 4.2 die Zeilen 1-22. Hier wird die eigentliche konkrete Syntax definiert (ähnlich zu einer EBNF-Form). Zusätzlich wurde in EMFText die spezielle Notation `@Operator` eingeführt. Sie dient primär für einen optimalen Aufbau des abstrakten Syntaxbaumes. Hierfür können folgende Eigenschaften angegeben werde:
   - `type`:
     Die `type`-Eigenschaft erlaubt die Typdefinition einer Regel. Es stehen dabei folgende Werte zur Verfügung:
     - `binary_left_associative` entspricht einem zweiwertigen linksassoziativen Ausdruck.
     - `binary_right_associative` entspricht einem zweiwertigen rechtsassoziativen Ausdruck.
     - `unary_prefix` entspricht einem einwertigen Präfixausdruck (zum Beispiel die Negation).
     - `unary_postfix` entspricht einem einwertigen Postfixausdruck.

  - `primitive` entspricht einem einwertigen Ausdruck (zum Beispiel ein Literal).

  `NamedLiteralExp`, `ImpliesRule` und `NestedExp` sind vom Typ `primitive`. Für die Restriktionsregel wurde eine linksassoziative Auswertung der Ausdrücke vereinbart. `XorExp`, `OrExp` und `AndExp` sind demnach vom Typ `binary_left_association`. Schließlich ist `NotExp` die einzige einstellige Präfixregel vom Typ `unary_prefix`.

- `weight`:
  Die `weight`-Eigenschaft legt die Bindungsstärke der Regeloperation fest. Wie bereits in Tabelle 4.7 dargestellt, werden die Bindungsstärken entsprechend übernommen.
- `superclass`:
  Schließlich definiert die `superclass`-Eigenschaft die Superklasse der Regeloperation aus dem Metamodell. Sie repräsentiert den Typ des Ergebniswertes der Operation.

Die Zeichenfolge `#0` zwischen Schlüsselwörter und Literale ermöglicht das Hinzufügen beliebig vieler Leerzeichen. Die Angabe von Kardinalitäten ist optional. In der Regel `NamedLiteralExp` (Zeile 3) wird dies durch das Fragezeichensymbol `?` festgelegt. Weiterhin ist die Angabe der linken Regelseite mit dem Implikationssymbol ebenfalls optional (Zeile 6). Wird die linke Regelseite ausgelassen, so wird sie als `true` ausgewertet und somit die rechte Seite immer ausgeführt.

Nachdem die konkrete Syntax definiert wurde, kann darauffolgend der Editorcode automatisch generiert werden. Zur Generierung setzt EMFText den Another Tool for Language Recognition (ANTLR)-Generator ein. Durch die Generierung entstehen folgende Pakete:

- `org.emftext.commons.antlr3_2_0`: Dieses Paket enthält die ANTLR-Laufzeitumgebung für den generierten Parser.
- `URI.resources.math`: In diesem Paket sind die generierten Parser, Printer, Scanner, Lexer und weitere Bestandteile der Sprache enthalten.
- `URI.resources.math.ui`: Hier sind die Komponenten enthalten, die mit der Benutzerschnittstelle interagieren, um beispielsweise Textvervollständigungen und Syntaxvalidierungen zu gewährleisten.

Abbildung 4.35 zeigt den entstandenen Editor. Im Variabilitätsmodell wird zur Definition von Restriktionen ein `Audit Conatainer` erzeugt. In diesem Container können dann beliebig viele Restriktionen modelliert werden. In dem Screenshot sind vier Restriktionsregeln modelliert, der Letzte von diesen ist im Editor auf der linken Seite zu sehen. Die Restriktion besagt, dass bei Auswahl von fünf Sende- und Empfangsantennen im Innenraum des Fahrzeugs der Komfortzugang impliziert wird. Außerdem ist die automatische Codevervollständigung zu sehen, die eine Liste möglicher Ausdrücke angibt.

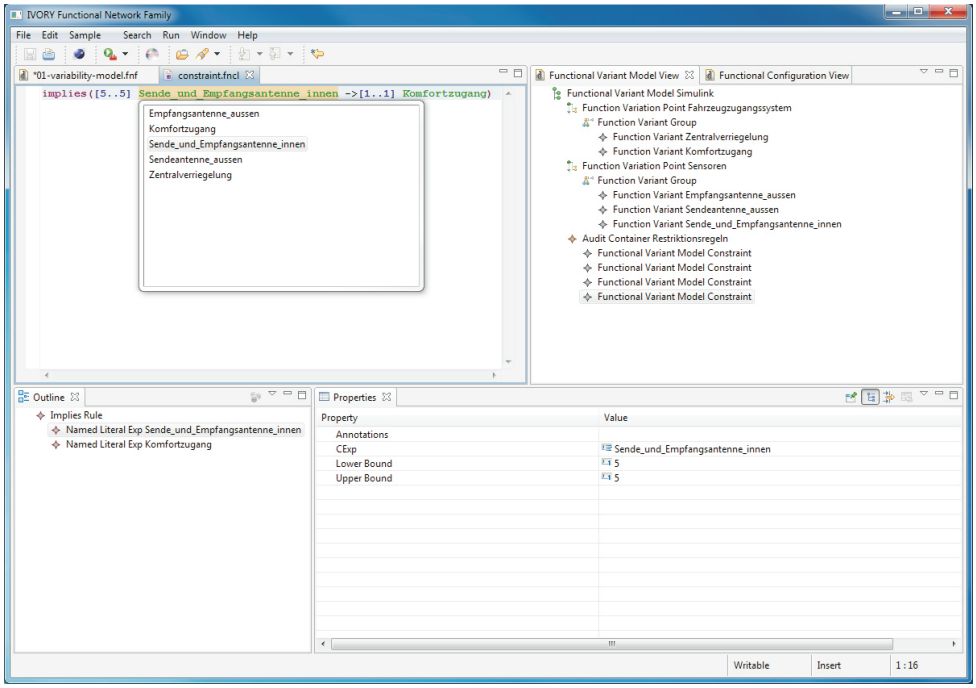


Abbildung 4.35.: Der Editor zur Restriktionsmodellierung

Selektionszustand	Validierungszustand	Repräsentation
UNSELECTED	UNDEFINED	
SELECTED	UNDEFINED	
SELECTED	REQUIRED	
UNSELECTED	EXCLUDED	
UNSELECTED	CONFLICT	
SELECTED	CONFLICT	

Tabelle 4.8.: Grafische Repräsentation der verschiedenen Variantenstatus (Quelle: [Pog10])

4.4.3. Konfigurationsmodell

Wie bereits in Abschnitt 4.3.1 erläutert, ist die Selektion mit einer Konfigurationsmaschine im Hintergrund ein möglicher Bindungsmechanismus zur automatischen Auswertung von Restriktionsregeln sowie zur automatischen Ausführung der Implikationen. Dies wird im Konfigurierungsprozess über eine Validierung gewährleistet. In diesem Abschnitt wird die grafische Repräsentation des Konfigurationsmodells und die Implementierung der Validierung genauer beschrieben.

Das Konfigurationsmodell setzt sich aus dem Variabilitätsmodell zusammen, indem

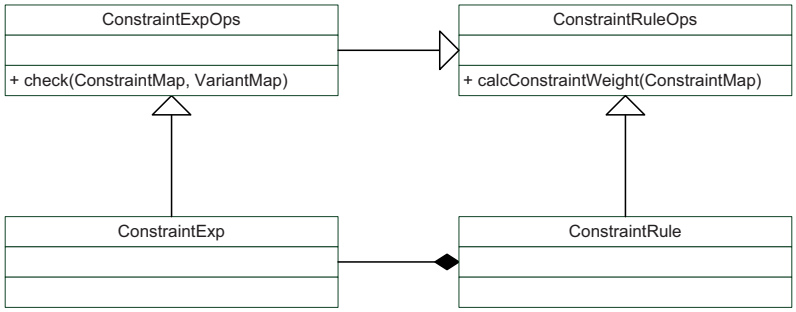


Abbildung 4.36.: Operationen für den Validator

es die Information bezüglich Variationspunkte, Gruppenkardinalitäten, Varianten und Variantenkardinalität übernimmt. Das Metamodell des Konfigurationsmodells wurde bereits in Abbildung 4.15 beschrieben, sodass sie hier nicht erneut aufgeführt wird. Weiterhin wird zur Konfigurierung eine Benutzerschnittstelle erzeugt, die das Selektieren/Deselektieren der Varianten und die Angabe von Kardinalitäten erlaubt. Durch jede Selektion/Deselektion wird die Validierung angestoßen und impliziert möglicherweise weitere Selektionen/Deselektionen. Aus dieser Validierung ergeben sich verschiedene Variantenstatus, für die es jeweils eine entsprechende grafische Repräsentation gibt. Tabelle 4.8 zeigt die möglichen Ergebnisse der Variantenstatus nach der Validierung und die zugehörige grafische Repräsentation. Auf diese Weise kann der Benutzer die Selektionen/Deselektionen leichter nachvollziehen.

Im Gegensatz zu den bisher dargestellten Editoren, die größtenteils automatisch generiert wurden, wird der Validator manuell implementiert. Zu diesem Zweck wurde das Metamodell um weitere Schnittstellen erweitert. Abbildung 4.36 illustriert diese Erweiterungen. Die Klasse `ConstraintExp` implementiert die Schnittstelle `ConstraintExpOps`. Sie beinhaltet zur Überprüfung der Teilausdrücke einer Restriktionsregel eine `check()`-Methode. Als Ergebnis wird entsprechend der Wert `true` oder `false` zurückgeliefert. Als Parameter werden zwei Maps übergeben, die zum einen die Gewichtung jedes Elements und zum anderen den Variantenstatus beinhalten. Die Klasse `ConstraintRule` implementiert des Weiteren die Schnittstelle `ConstraintRuleOps`. Sie beinhaltet die Methode `calcConstraintWeight()` zur Initiierung und Gewichtung der Elemente einer Restriktionsregel. Als Parameter wird die entsprechende Map übergeben.

Weiterhin wird das Metamodell um weitere Schnittstellen erweitert, die für die Konfigurierung erforderlich sind. Abbildung 4.37 zeigt diese Erweiterungen. Die Klasse `Configuration` implementiert die Schnittstelle `ConfigurationOps`. Diese beinhaltet die Methoden `hasConflict()`, `getItemByName()` und `getAllItemsByClass()`. Die Methode `hasConflict()` erkennt jegliche Konflikte bezüglich Kardinalitätsangaben und Restriktionen. Herrscht ein Konflikt vor, so ist die aktuelle Konfiguration solange ungültig, bis der Fehler vonseiten eines Benutzers behoben wird. Durch die Methode `getItemByName()` wird anhand des Parameters das Element mit dem gleichen Namen zurückgeliefert. Schließlich liefert die Methode `getAllItemsByClass()`

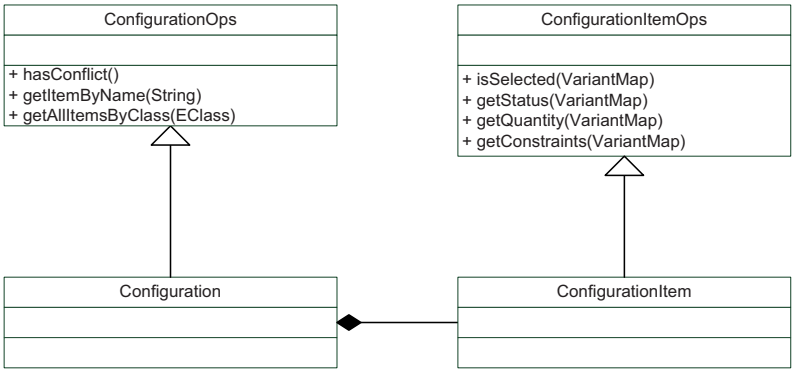


Abbildung 4.37.: Operationen für die Konfigurierung

eine Liste von Elementen, welche die Objekte einer als Parameter übergebenen Klasse speichern. Auf diese Weise können Variationspunkte, Variantengruppen oder Varianten allesamt separat aufgelistet werden. Die Klasse `ConfigurationItem` implementiert die Schnittstelle `ConfigurationItemOps`. Die Methoden `isSelected()`, `getStatus()`, `getQuantity()` und `getConstraints()` ermöglichen, alle Informationen des Variantenstatus einzeln zu extrahieren.

Die eigentliche Validierung findet in der Methode `validate()` statt. Die Implementierung dieser Methode ist in Listing 4.3 dargestellt. Sie beinhaltet im Wesentlichen die in Algorithmus 4.1 und Algorithmus 4.2 beschriebenen Abläufe. In den Zeilen 2-8 werden die Maps initiiert und die Restriktionen gesammelt. Daraufhin folgt die Vorwärtsvalidierung, in der überprüft wird, ob die linke Seite der Restriktionsregel `true` ergibt, woraufhin die rechte Seite ausgeführt wird (Zeilen 14-18). Falls es keine linke Regelseite gibt, wird dies als `true` ausgewertet und somit die rechte Regelseitre ausgeführt. Danach folgt die Rückwärtsvalidierung (Zeilen 26-32). Ist die Vorwärts- und Rückwärtsvalidierung durchgeführt, folgt im Anschluss die Überprüfung der Kardinalitäten (Zeile 35). Schließlich wird das ermittelte Ergebnis in die Map für den Variantenstatus kopiert (Zeile 37).

In Abbildung 4.38 ist das Konfigurationsmodell dargestellt. Wie bereits erwähnt, ergibt sich das Modell aus den Informationen des Variabilitätsmodells. In dem Beispiel wurde manuell die Variante `Komfortzugang` selektiert. Aus dieser Selektion wird die Validierung angestoßen, sodass `Sendeantenne_aussen` und `Sende_und_Empfangsantenne_innen` automatisch gewählt werden und entsprechend den Variantenstatus `(SELECTED,REQUIRED)` erhalten. Aufgrund der Rückwärtsvalidierung wird der Variantenstatus für den `Komfortzugang` ebenfalls auf `(SELECTED,REQUIRED)` gesetzt. Zusätzlich wird die Variante `Empfangsantenne_aussen` manuell gewählt. Diese Auswahl bewirkt keine Implikationen. Schließlich ist in der mittleren Spalte die Angabe der Kardinalität zu sehen.

Abbildung 4.39 illustriert die Konflikterkennung. Hier wurde die Variante `Zentralverriegelung` selektiert. Da aber die Gruppenkardinalität auf `[1..1]` gesetzt ist, darf entweder die `Zentralverriegelung` oder der `Komfortzugang` selektiert werden,

```
1  public void validate(Configuration conf) {
2          initConstraintMap(conf); initVariantMap(conf);
3          Vector<VariantModelConstraint> constraints=new Vector<VariantModelConstraint
               >();
4          for (AuditContainer audit : conf.getVariantModel().getAuditContainers()){
5                  for(VariantModelConstraint constraint : audit.getConstraints()){
6                          constraints.add(constraint);
7                  }
8          }
9          int counter = 0;
10         do{
11                 for(VariantModelConstraint constraint : constraints){
12                         if(constraint.getRule() instanceof ImpliesRule){
13                                 if(((ImpliesRule)constraint.getRule()).getSource()
                                       != null){
14                                         //forward check
15                                         //check left part and if true execute the
                                               right part
16                                         if(((ImpliesRule)constraint.getRule()).
                                               getSource().check(conf, getConstraintMap
                                               (), getVariantMap(), false)){
17                                                 executeConstraintPart(((ImpliesRule)
                                                       constraint.getRule()).getTarget
                                                       (), constraint, conf, true);
18                                         }
19                                 }
20                                 else
21                                         //forward check
22                                         //execute the right part
23                                         executeConstraintPart(((ImpliesRule)
                                               constraint.getRule()).getTarget(),
                                               constraint, conf, true);
24                         }
25                 }
26                 for(VariantModelConstraint constraint : constraints){
27                         if(constraint.getRule() instanceof ImpliesRule)
28                         //check reverse dependencies of Type
29                         //check the right part and if false execute the left part
30                         if(!(((ImpliesRule)constraint.getRule()).getTarget().check(
                               conf, getConstraintMap(), getVariantMap(), true)))
31                                 executeConstraintPart(((ImpliesRule)constraint.
                                       getRule()).getSource(), constraint, conf, false)
                                       ;
32                 }
33                 counter++;
34         }while(isDirtyMap() && (counter < (constraints.size() * getVariantMap().
               keySet().size())));
35         validateGroups(conf);
36         //copy variantMap to Configuration
37         copyVariantMapCommand(conf);
38 }
```

Listing 4.3: Die Methode `validate()` zur Validierung von Konfigurierungsschritten

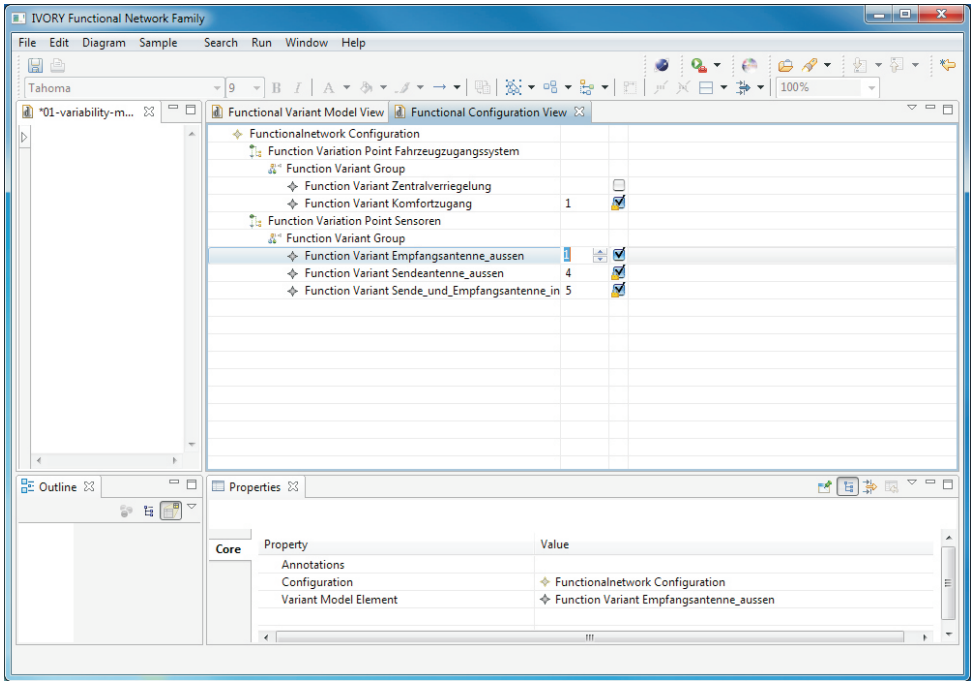


Abbildung 4.38.: Das Konfigurationsmodell und das Ergebnis der Validierung nach Selektion des Komfortzugangs

aber nicht beide gleichzeitig. Dieser Konflikt wird durch das Ausrufezeichensymbol markiert. Dementsprechend ist die gesamte Konfiguration aktuell ungültig.

4.4.4. Generierungsmodell

In Abschnitt 4.3.2 wurde der Generierungsprozess im Detail behandelt. Dabei wurde der Bedarf an einer Transformierung zu WCRL, die Inferierung dieses WCRL-Modells sowie die Rücktransformierung in ein Zieldokument erläutert. In diesem Abschnitt werden die zur Realisierung erforderlichen Technologien und ihre Anwendung genauer beschrieben.

Für die Transformierung wurde das Xpand Rahmenwerk von Eclipse verwendet, welches Teil des EMP ist [Gro09]. Das Rahmenwerk verfügt über textuelle Sprachen, die u.a. zur Codegenerierung und Modelltransformationen eingesetzt werden können. Diese Sprachen sind Xtend und Xpand. Mit diesen Sprachen wird der Generierungsprozess realisiert. Dabei sind folgende Module erforderlich:

- xmiReader zum Lesen der Modelle im XML Metadata Interchange (XMI)-Format
- xpandGenerator zur Transformierung der Eingabemodelle zu WCRL-Regeln

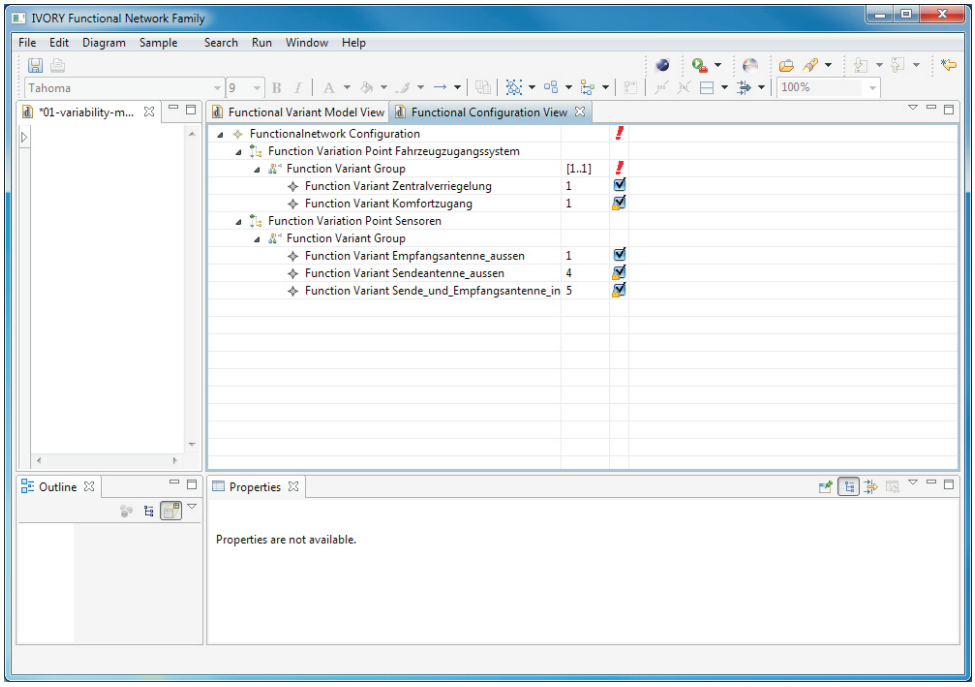


Abbildung 4.39.: Die Erkennung eines Konflikts bei zusätzlicher Selektion der Zentralverriegelung

- xtextGenerator zur Generierung eines Metamodells und Parsers für WCRL
- wcrlParser zur syntaktischen Analyse von WCRL-Regeln zu WCRL-Modellen
- xtendGenerator zur Transformierung von WCRL-Modellen zu gewünschten Zieldokumenten
- xmiWriter zum Speichern der Zieldokumente im XMI-Format

EMP verwendet zur Serialisierung ihrer Modelle den XMI-Standard. Da das Variabilitätsmodell mit EMF erzeugt wurde (ein Teilprojekt von EMP), ist XMI ebenfalls das Standardformat zur Serialisierung des Variabilitätsmodells. Somit kann das Xpand Rahmenwerk durch einen xmiReader die Eingabemodelle einlesen.

Der xpandGenerator ist ein template-basierter Codegenerierungsmechanismus. Es überführt ein Eingabemodell durch die Anwendung der in einem Template definierten Anweisungen in eine textuelle Form. Es findet also eine Modell-zu-Text-Transformation statt. Listing 4.4 zeigt einen Ausschnitt aus einem solchen Template. Ein Template beginnt mit `IMPORT`-Anweisungen, gefolgt von einer Menge von `EXTENSION`-Anweisungen, die auf Erweiterungen verweisen, wie beispielsweise

```
1  «IMPORT variabilitymodel»
2  «IMPORT infrastructure»
3  «IMPORT xmi»
4
5  «EXTENSION Extensions»
6
7  «DEFINE root FOR VariabilityModel»
8          «FILE "rules.wcrl"»
9          «EXPAND vmodel FOR VariantModel»
10         «ENDFILE»
11 «ENDDEFINE»
12
13 «DEFINE vmodel FOR variabilityModel::VariabilityModel»
14         «EXPAND var FOREACH variationPoints»
15         «EXPAND acons FOREACH auditConrainers»
16 «ENDDEFINE»
17
18 «DEFINE var FOR variabilityModel::VariationPoint»
19         vp(«this.getId().replaceAll("-", "___")»).
20         vpName(«this.name.toFirstLower()»).
21         name_id(«this.name.toFirstLower()», «this.getId().replaceAll("-", "___")»).
22 «ENDDEFINE»
```

Listing 4.4: Ein Ausschnitt aus einem Xpand-Template

Java-Quellcode, um Operationen ausführen zu können, die mit Xpand alleine nicht möglich wären. Das wichtigste Konzept ist der `DEFINE`-Block.

```
«DEFINE templateName(formalParameterList) FOR MetaClass »
        a sequence of statements
«ENDDEFINE»
```

Er wird im Kontext der Metaklasse definiert und greift auf Werte oder Objekte in der Parameterliste zu. `DEFINE`-Blöcke werden durch `EXPAND`-Anweisungen um Objekte der Metaklasse erweitert.

```
«EXPAND my::templates::TemplateFile::definitionName FOR myModelElement»
```

Das Metamodell des zu syntaktisch analysierenden Variabilitätsmodells wird registriert, wonach die Ausführung des xpandGenerator stattfindet. Das Resultat dieser Ausführung ist schließlich die Menge der textuellen WCRL-Regeln.

Damit die textuellen WCRL-Regeln nach der Inferierung rücktransformiert werden können, bedarf es an einem Metamodell für WCRL, damit eine Modell-zu-Modell-Transformation durchgeführt werden kann. Um dies zu realisieren, wurde Xtext eingesetzt. Xtext ist ähnlich wie EMFText ein Rahmenwerk zur Erzeugung domänenspezifischer Sprachen. Durch die Definition der gewünschten Sprache in einer EBNF-ähnlichen Notation erzeugt der xtextGenerator das entsprechende Metamodell, einen Parser und einen Texteditor für die Sprache.

Schließlich wird der xtendGenerator eingesetzt, um die Transformierung von WCRL-Modellen zu den Zieldokumenten zu gewährleisten.

4.5. Verwandte Arbeiten

Variabilität ist ein Aspekt der Softwareentwicklung, der aus diversen Gründen entsteht. Verschiedene Kundenanforderungen sind eine mögliche Quelle der Variabilität. So bieten Automobilhersteller ihren Kunden die Möglichkeit zusätzlich zur Grundausstattung weitere Sonderausstattungen auszuwählen. Der Einsatz verschiedener Technologien, wie Bussysteme und Steuergeräte, führt ebenfalls zu Variabilität. Wettbewerb ist eine weitere wichtige Quelle. Um Kunden für eine bestimmte Automobilmarke zu binden, müssen vielfältige Angebote offeriert werden, die von Kleinwagen bis hin zu Premiumklassefahrzeugen reichen. Schließlich führt die Präsenz der Hersteller in verschiedenen Märkten zu einer weiteren Quelle. Die explizite Erfassung und systematische Wiederverwendung von variablen Aspekten im Softwareentwicklungsprozess ist daher essenziell, um diese Dimensionen der Variabilitätskomplexität zu beherrschen.

In der Literatur sind nun seit knapp 20 Jahren verschiedene Modellierungstechniken und Prozesse zur Unterstützung der systematischen Wiederverwendung definiert. Die Ansätze legen dabei ihren Schwerpunkt auf verschiedene Faktoren. Im Folgenden werden diese Ansätze genauer vorgestellt. Darüber hinaus werden in diesem Abschnitt alle beschriebenen Ansätze bewertet. Zu diesem Zweck werden die bereits aus Abschnitt 4.1.2 ermittelten Anforderungen als Bewertungskriterien herangezogen. Abschließend werden alle Ansätze mit dem in dieser Arbeit vorgestellten Ansatz verglichen.

Die betrachteten Kriterien umfassen somit (1) Konzepte der Modellierung und (2) Konzepte der Bindung. Nachfolgend werden die entsprechenden Kriterien aufgelistet.

1. **Konzepte der Modellierung**
   a) Variabilitätsmodellierung
      - *Art der Modellierung:* Dieses Kriterium betrachtet die Art der Variabilitätsmodellierung. Hier stehen typischerweise zwei Arten zur Verfügung: (1) Hierarchisch strukturierte Modelle und (2) Auswahlmodelle.
      - *Strukturierungsmaßnahmen:* Variabilitätsmodellierung kann durch Konzepte der Strukturierung geeigneter organisiert und detailliert werden. Es stehen verschiedene Möglichkeiten zur Strukturierung zur Verfügung, zum Beispiel die Einführung absrakter Features oder Gruppierungskonzepte.
      - *Berücksichtigung von Abstraktionsebenen:* Dieses Kriterium bezieht die Abstraktionsebenen des Entwicklungsprozesses in die Strukturierung des Variabilitätsmodells ein. Komplexitätsreduzierende Maßnahmen, wie etwa Hierarchisierungs- bzw. Schichtenkonzepte mit Referenzierung, können das Modell geeignet teilen.
      - *Unterstützung bei der Modellierung:* Damit der Aufwand bei der Identifikation und Dokumentation von Variabilität so gering wie möglich

gehalten wird, können unterstützende Maßnahmen getroffen werden, die ungültige Aktionen verhindern oder Aktionen vorschlagen. Hierbei gibt es zwei Arten der Unterstützung: (1) proaktiv und (2) reaktiv.

- *Modellierung von Variabilitätsmechanismen:* Dieses Kriterium untersucht, ob die Variabilitätsmodellierung ein Konzept beinhaltet, das die Modellierung von Variablitätsmechanismen unterstützt.
- *Modellierung von Variantenarten:* Die Unterscheidung zwischen integrierten und separierten Varianten unterstützt bei der Aufgabenplanung im Entwicklungsprozess und stellt eine wichtige Entscheidungsgrundlage für weitere Fragestellungen, wie etwa das Deployment von Software auf Steuergeräte, dar. Anhand dieses Kriterium wird überprüft, ob das Variabilitätmodell die Möglichkeit zur Modellierung integrierter bzw. separierter Varianten unterstützt.

b) Restriktionsmodellierung

- *Definition variabler Eigenschaften:* Die Vorteile einer Variabilitätsmodellierung zeigen sich erst durch die Möglichkeit, variable Eigenschaften für die modellierten Entitäten festzulegen. Typische Eigenschaften sind Verbindlichkeit, Optionalität, Alternativität etc. Durch dieses Kriterium wird überprüft, welche Möglichkeiten die Variabilitätsmodellierung zur Festlegung variabler Eigenschaften bietet.
- *Restriktionssprachen:* Komplexere Beziehungen zwischen variablen Entitäten im Variabilitätsmodell werden durch Restriktionssprachen ausgedrückt. Die Existenz einer Restriktionssprache erhöht somit die Ausdrucksstärke in einem Variabilitätsmodell. Es wird untersucht, welche Variabilitätsmodelle Restriktionssprachen anbieten.

2. **Konzepte der Bindung**

a) Konfigurierung

- *Konfiguration des Variabilitätsmodells:* Die Konfiguration eines Variabilitätsmodells ist das Ergebnis einer Konfigurierungsaktivität. In diesem Fall werden also Varianten in ein Softwaresystem gebunden. Dies wird in der Regel durch die Selektion als Bindungsmechanismus durchgeführt. Besonders wichtig ist hierbei die Verknüpfung zwischen Variabilitätsmodell und der Konfiguration. In einem Fall entstehen zwei separate Modelle, die nach dem Konfigurierungsprozess keinerlei Beziehungen zueinander besitzen. Somit können Änderungen, die im Variabilitätsmodell durchgeführt werden, nicht an die Konfiguration propagiert werden. Im anderen Fall besteht im Gegensatz dazu eine enge Bindung zwischen Variabilitätsmodell und der Konfiguration. Auf diese Weise werden Änderungen im Variabilitätsmodell weitergeleitet. In der Literatur werden Konfigurationen mit fester Bindung als Entscheidungsmodelle (engl. Decision Model) bezeichnet.

- *Unterstützung bei der Konfigurierung:* Während der Konfigurierung implizieren bestimmte Selektionen weitere Konfigurierungsschritte, die notwendig werden. Das Werkzeug kann diese Implikationen erkennen und automatische Aktionen durchführen. Zum Beispiel kann es sein, dass die Auswahl einer Variante die Auswahl oder den Ausschluss einer anderen Variante impliziert.

b) Generierung

- *Inferenz konkreter Produkte:* Die Konfigurierung führt dazu, dass Varianten an Softwaredokumente gebunden werden, sodass konkrete Dokumente entstehen. Das Inferieren dieser kann durch Ersetzung von Variationspunkten, die als Platzhalter dienen, oder durch Generierung realisiert werden. Die Automatisierung dieses Prozesses reduziert die Fehlerwahrscheinlichkeit.

Die folgenden sieben Arbeiten werden in den nächsten Abschnitten zunächst allgemein beschrieben und anschließend anhand der eingeführten Kriterien bewertet: (1) Featuremodelle nach der FODA-Methode, (2) FeatuRSEB, (3) Kardinalitätsbasierte Featuremodelle, (4) Variability Specification Language (VSL), (5) Orthogonale Variabilitätsmodelle, (6) ConIPF Variability Modeling Framework (COVAMOF) und (7) Configuration Support Library (CONSUL) bzw. pure::variants.

4.5.1. Featuremodelle nach der FODA-Methode

Kang et al. haben in ihrem technischen Bericht die FODA-Methodik zur Domänenanalyse eingeführt [KCH+90]. Featuremodelle sind ein Bestandteil dieser Methodik. Im Folgenden wird basierend auf [KCH+90] ein Überblick der FODA-Methodik gegeben, sodass die Featuremodelle eingeordnet werden können. Im Anschluss werden sie dann genauer beschrieben und bewertet.

Die Domänenanalyse ist eine Aktivität zur systematischen Identifizierung gemeinsamer Aspekte einer Menge ähnlicher Softwaresysteme. FODA ist ein Vorschlag von *Kang et al.* zur Domänenanalyse. Es sei an dieser Stelle betont, dass der Begriff FODA oft mit Featuremodell gleichgesetzt oder behandelt wird, was allerdings nicht zutreffend ist. Dies wird im Verlauf dieses Abschnitts noch genauer beleuchtet. Für die Methode werden drei Phasen definiert: (1) die Kontextanalyse, (2) die Domänenmodellierung und die (3) Architekturmodellierung.

In der Kontextanalysephase wird der Umfang der Domäne definiert. Hier werden also die Schnittstellen der Softwaresysteme klar identifiziert. Aus dieser Phase entstehen Strukturdiagramme und Kontextdiagramme als Ergebnis. In der Domänenmodellierungsphase werden alle aus der Kontextanalysephase ermittelten Aspekte genauer beschrieben. Hierzu gehören die Beschreibung der Entitäten mit ihren Abhängigkeiten, die Definition der Features, also der benutzersichtbaren Eigenschaften der Softwaresysteme, die Spezifikation von Kontroll- und Datenflüssen sowie die Auflistung der Terminologie. Als Ergebnis entstehen Entity-Relationship-Modelle, Featuremodelle, Funktionsmodelle und ein Glossar mit der Terminologie für die

Domäne. An dieser Stelle wird also klar, dass Featuremodelle nur ein Teilergebnis des Gesamtprozesses sind und somit nicht undifferenziert zu FODA betrachtet werden können. Schließlich werden in der Phase zur Architekturmodellierung weitere Modelle entworfen, die zur Realisierung der Softwaresysteme verwendet werden können. Als Ergebnis entstehen hier Prozessinteraktionsmodelle und Modulstrukturmodelle. Die Gesamtheit der Ergebnisse in der Domänenanalyse dient letztlich als Referenzmodell für zukünftige Entwicklungen.

Nach Einordnung der Featuremodelle in den Gesamtprozess von FODA können nun die Details genauer behandelt werden. Den Kern des Featuremodells bildet eine Baumstruktur. Alle Knoten in einem derartigen Baum stellen ein Feature dar. Jedes Feature hat einen eindeutigen Namen. Die Kanten zwischen Features dienen zur Strukturierung des Baums (Vaterknoten-Kindknoten). Die Semantik dieser Kanten drückt eine `consists_of`-Relation aus.

Es können Features weitere Eigenschaften zugewiesen werden. So können sie als alternativ oder optional markiert werden. Bei alternativen Features kann stets genau ein Feature aus der Menge in ein Softwaresystem integriert werden ([1..1]-Kardinalität). Optionale Features sind Features, die in einem System vorhanden sein können oder auch nicht ([0..1]-Kardinalität).

Zusätzlich können zwischen beliebig verteilten Features sogenannte Kompositionsregeln (auch Restriktionen oder Constraints genannt) definiert werden. Hierfür wurden in FODA zwei Regeln eingeführt: `mutually exclusive with` und `requires`. Erste Regel drückt den Ausschluss eines Features aus, wenn ein in der Regel angegebenes Feature für das Softwaresystem selektiert ist. Zweite Regel drückt den Bedarf eines weiteren Features aus, wenn ein in der Regel angegebenes Feature selektiert ist. Die Menge aller Kompositionsregeln schränkt somit die Auswahlmöglichkeiten für das System ein. Weiterhin können Beschreibungen zum Featuremodell hinzugefügt werden, die einem Benutzer bei der Auswahl der Features unterstützen können. Schließlich werden in FODA für Features drei Bindezeiten definiert: (1) Compile-Time, (2) Load-Time und (3) Runtime. Die Compile-Time-Bindung führt dazu, dass die Features mit dieser Eigenschaft beim Kompilieren der Software gebunden und somit im Objektcode fixiert werden. Die Load-Time-Bindung ergibt, dass Features vor der Ausführung gebunden werden und auf diese Weise die Ausführungspfade der Software für diese Features festgelegt werden. Schließlich sind Runtime Bindungen solche, die interaktiv oder automatisch während der Ausführung verändert werden können.

Im Folgenden werden die Kriterien herangezogen, um die Arbeit von *Kang et al.* detaillierter zu bewerten.

1. **Konzepte der Modellierung**

   a) Variabilitätsmodellierung

      - *Art der Modellierung:* Featuremodelle nach der FODA-Methode sind hierarchisch strukturierte Modelle, die alle Features der Domäne erfassen und modellieren.

- *Strukturierungsmaßnahmen:* Zur Strukturierung werden abstrakte Features verwendet. Abstrakte Features haben keine Manifestation im Softwaredokument. Sie dienen nur der Strukturierung.
- *Berücksichtigung von Abstraktionsebenen:* Featuremodelle werden hierarchisiert, es gibt kein Schichtenkonzept mit Referenzierung, um die Komplexität des Entwicklungsprozesses zu beherrschen. Demnach gibt es ein zentrales Featuremodell, das alle Features der Domäne beinhaltet.
- *Unterstützung bei der Modellierung:* Keine Unterstützung.
- *Modellierung von Variabilitätsmechanismen:* Keine explizite Unterstützung. Variabilitätsmechanismen können allerdings implizit durch Features modelliert werden.
- *Modellierung von Variantenarten:* Keine Unterstützung.

b) Restriktionsmodellierung

- *Definition variabler Eigenschaften:* Es können verbindliche, optionale und alternative Features festgelegt werden.
- *Restriktionssprachen:* Hierfür werden in FODA zwei Regeln eingeführt: `mutually exclusive with` und `requires`. Ein Formalismus für diese Restriktionen gibt es allerdings nicht.

2. **Konzepte der Bindung**

a) Konfigurierung

- *Konfiguration des Variabilitätsmodells:* Keine Unterstützung.
- *Unterstützung bei der Konfigurierung:* Keine Unterstützung.

b) Generierung

- *Inferenz konkreter Produkte:* Keine Unterstützung.

4.5.2. FeatuRSEB

Griss et al. haben in ihrem Aufsatz aus [GFd98] beschrieben, wie sie Featuremodelle aus der FODA-Methodik als Basis verwenden, diese für ihre Bedürfnisse anpassen und in ihre Reuse-Driven Software Engineering Business (RSEB)-Methodik integrieren. RSEB ist dabei ein systematischer und modellgetriebener Ansatz zur Domänenanalyse. Dabei werden überwiegend objektorientierte Konzepte eingesetzt. So sind Spracheinheiten aus der Unified Modeling Language (UML) [Rum11, Gro10b, Rum04, FS00] Kernbestandteile der RSEB-Methodik. Insbesondere werden Use-Case-Modelle in allen Phasen der Analyse eingesetzt. Für detaillierte Informationen zur Vorgehensweise in RSEB sei auf [JGJ97] verwiesen.

Nachdem die Autoren in einem Projekt für die Telekommunikationsbranche Erfahrungen mit der FODA-Methodik gewonnen haben, stellten sie fest, dass ihre RSEB-Methodik Unvollständigkeiten in Bezug auf die Featureanalyse aufwies. So wurden

in RSEB Features nur informell gehandhabt. Featuremodelle werden dementsprechend gar nicht eingesetzt. Daher wird in dem Beitrag von *Griss et al.* das Konzept der Featuremodelle in den bereits bestehenden Prozess RSEB integriert [GFd98]. Das Resultat ist FeatuRSEB.

Für ihren Ansatz wurden Konzepte der Featuremodelle aus der FODA-Methodik für RSEB angepasst und erweitert. Zunächst wurden für Featuremodelle zwei Sichten eingeführt: (1) eine grobgranulare Sicht und (2) eine feingranulare Sicht. Features können in Subfeatures gegliedert werden. Dafür wird die `composed_of`-Relation verwendet. Neben verbindlichen Features können optionale Features modelliert werden. Alle Features, die weiter geteilt werden können, werden als Variationspunkte (`vp-Feature`) bezeichnet, ihre Kinder sind somit Varianten (`Variant Feature`) des Variationspunktes. Varianten können wiederum selbst Variationspunkte sein. Variationspunkte werden dabei abhängig von der Bindezeit in zwei Arten unterschieden:

1. Reuse-Time. Diese Bindezeit beinhaltet alle Bindezeiten vor dem Laden des Softwaresystems. `vp-Features` können also in der Entwicklungsphase (zum Beispiel beim Entwurf der Softwarearchitektur) gebunden werden. Die Varianten dieser Features können alternativ selektiert werden, d.h., genau eine Variante kann zur Konfigurierung eines Softwaredokuments ausgewählt werden.

2. Use-Time. In diesem Fall können Features zur Ladezeit und Laufzeit gebunden werden. Hierbei kann eine Teilmenge der Varianten eines Variationspunktes für das Softwaresystem gewählt werden.

In der feingranularen Sicht können weitere Details spezifiziert werden, wie beispielsweise Restriktionen, detailliertere Beschreibungen und Notizen. Die Ideen von *Griss et al.* sind in bereits vorhandene objektorientierte Werkzeuge eingeflossen. Dabei wurden die Modellierungssprachen für die Bedürfnisse der Featuremodellierung angepasst.

Nachfolgend wird anhand der eingeführten Kriterien die Arbeit von *Griss et al.* evaluiert.

1. **Konzepte der Modellierung**
   a) Variabilitätsmodellierung
      - *Art der Modellierung:* FeatuRSEB sind hierarchisch strukturierte Modelle. Features können entweder Variationspunkte oder Varianten sein. Dabei sind alle Blätter des Modells Varianten, alle anderen Knoten sind Variationspunkte.
      - *Strukturierungsmaßnahmen:* Durch Variationspunkte wird das Modell in mehrere Hierarchieebenen strukturiert.
      - *Berücksichtigung von Abstraktionsebenen:* Ähnlich wie bei Featuremodellen nach FODA wird in FeatuRSEB kein Schichtenkonzept mit Referenzierung eingesetzt, um den Entwicklungsprozess zu berücksichtigen. Hier ist demnach auch ein zentrales Modell im Einsatz.

- *Unterstützung bei der Modellierung:* Keine Unterstützung.
- *Modellierung von Variabilitätsmechanismen:* Keine explizite Unterstützung. Variabilitätsmechanismen können allerdings implizit durch Features modelliert werden.
- *Modellierung von Variantenarten:* Keine Unterstützung.

b) Restriktionsmodellierung

- *Definition variabler Eigenschaften:* Es gibt verbindliche und optionale Features sowie alternative als auch oder-Featuregruppen.
- *Restriktionssprachen:* In der Detailsicht können beliebige Restriktionen formuliert werden (z.B. `requires`). Es gibt allerdings keinen Formalismus.

2. **Konzepte der Bindung**

a) Konfigurierung

- *Konfiguration des Variabilitätsmodells:* Keine Unterstützung.
- *Unterstützung bei der Konfigurierung:* Keine Unterstützung.

b) Generierung

- *Inferenz konkreter Produkte:* Keine Unterstützung.

4.5.3. Kardinalitätsbasierte Featuremodelle

Czarnecki et al. haben in einer Reihe von Aufsätzen ihren Ansatz zur Modellierung und Verwaltung von Variabilität in einer Domäne, den sogenannten kardinalitätsbasierten Featuremodellen (engl. Cardinality-Based Feature Model (CBFM)), vorgestellt [CHE04, CA04, CHE05b, CHE05a, CK05]. CBFMs erweitern dabei Featuremodelle durch weitere hilfreiche Konzepte, die im Folgenden genauer beschrieben werden.

Genau wie Featuremodelle aus der FODA-Methodik, werden auch hier Features in eine hierarchische Baumstruktur angeordnet. Jedes Feature wird durch eine Featurekardinalität der Form [m..n] gekennzeichnet. Die Kardinalität [1..1] bedeutet dabei, dass das Feature verbindlich ist, also in jedem Softwaresystem der Familie vorhanden ist. Die Kardinalität [0..1] deutet auf ein optionales Feature. Featurekardinalitäten mit einer Obergrenze größer als eins sind replizierbare Features, d.h., diese können in einem Softwaresystem mehrfach vorhanden sein.

Weiterhin sind in CBFM Featuregruppen eingeführt. Jede Featuregruppe kann durch eine Gruppenkardinalität markiert werden. Anhand der Gruppenkardinalität wird festgelegt, wie viele der gruppierten Features für ein konkretes System selektiert werden können. Um die Größe der Modelle überschaubar zu halten, zur Arbeitsteilung zu verwenden oder die zur Verfügung stehenden Ressourcen nicht auszulasten, werden Referenzen eingeführt, die auf weitere Featuremodelle referenzieren. Schließlich können auch in CBFM beliebige Regeln definiert werden, die Features in Abhängigkeiten setzen. Hierfür stellt CBFM die Sprachen Object

Constraint Language (OCL) [Gro10a] und XML Path Language (XPath) [W3C10] zur Verfügung. Alle beschriebenen Konzepte sind in das Werkzeug *fmp* eingeflossen [CA04, CK05].

Auch für CBFM wird im Folgenden eine genauere Bewertung durchgeführt.

1. **Konzepte der Modellierung**
   a) Variabilitätsmodellierung
      - *Art der Modellierung:* CBFMs sind hierarchisch strukturierte Modelle in Form einer Baumstruktur. Jeder Knoten stellt dabei ein Feature dar.
      - *Strukturierungsmaßnahmen:* Durch abstrakte Features wird das Modell strukturiert.
      - *Berücksichtigung von Abstraktionsebenen:* Jedes Feature in CBFM kann eine Referenz beinhalten, die auf weitere Featuremodelle verweist. Auf diese Weise können CBFMs für den Entwicklungsprozess geeignet eingeteilt werden.
      - *Unterstützung bei der Modellierung:* Keine Unterstützung.
      - *Modellierung von Variabilitätsmechanismen:* Keine explizite Unterstützung. Variabilitätsmechanismen können allerdings implizit durch Features modelliert werden.
      - *Modellierung von Variantenarten:* Keine Unterstützung.
   b) Restriktionsmodellierung
      - *Definition variabler Eigenschaften:* Durch Feature- und Gruppenkardinalitäten können verbindliche, optionale Features sowie alternative als auch oder-Featuregruppen ausgedrückt werden.
      - *Restriktionssprachen:* Restriktionen können durch die Sprachen OCL und XPath beschrieben werden.

2. **Konzepte der Bindung**
   a) Konfigurierung
      - *Konfiguration des Variabilitätsmodells:* In CBFM werden Entscheidungsmodelle eingesetzt, um eine Konfiguration der modellierten Variabilität zu bilden.
      - *Unterstützung bei der Konfigurierung:* Während der Konfigurierung kann das Werkzeug den Benutzer aktiv unterstützen, indem es feststehende Auswahlmöglichkeiten (z.B. aufgrund von Restriktionen wie `requires` oder `exclude`) automatisch selektiert oder deselektiert. So wird die Fehlerwahrscheinlichkeit bei der Konfigurierung reduziert.
   b) Generierung
      - *Inferenz konkreter Produkte:* Keine Unterstützung.

4.5.4. Variability Specification Language

Martin Becker beschreibt in seinem Aufsatz [Bec03] ein Modell, das Variabilität über alle Abstraktionsebenen eines Softwareentwicklungsprozesses hinweg realisiert und verwaltet. Die Motivation für diesen Ansatz begründet er wie folgt: Zur Realisierung und Verwaltung von Variabilität in den verschiedenen Abstraktionsebenen werden vielfältige Ansätze und Denkweisen verwendet, die letztlich zu unterschiedlichen Semantiken in der Begriffsverwendung führen. Dies verhindert aber, dass Synergien entstehen und erschwert somit die konsistente Verwaltung. Die Einführung eines generischen Modells zur Realisierung und Verwaltung von Variabilität kann daher zur Entstehung von Synergien beitragen.

Damit dieses Ziel erreicht werden kann, werden, ausgehend von festgestellten Problemen in Bezug auf Variabilität, Anforderungen für ein generisches Modell abgeleitet. Zunächst kann Variabilität nicht ohne Weiteres lokalisiert werden, da sie in den Realisierungsdokumenten verstreut ist. Variabilität hat also weitreichende Auswirkungen. Insbesondere beeinflusst sie variierende Qualitätsattribute wie beispielsweise Performanz und Ressourcenzuteilung. Es muss also Variabilität auf allen Abstraktionsebenen erfasst werden. Dies entspricht einer vertikalen Realisierung und Verwaltung von Variabilität. Zudem muss Variabilität auf derselben Abstraktionsebene über verschiedene und verteilte Realisierungsdokumente hinweg behandelt werden. Dies entspricht einer horizontalen Realisierung und Verwaltung von Variabilität. Des Weiteren ist Variabilität mit Variationspunkten innerhalb der Realisierungsdokumente assoziiert. Diese können komplexer Natur sein sowie vertikal und horizontal verstreut auftreten. Variabilität muss demnach auf allen Ebenen verfolgbar sein. Schließlich können Varianten gegenseitige Abhängigkeiten aufweisen. Zum Beispiel können sich Varianten gegenseitig ausschließen oder benötigen. Also müssen Abhängigkeiten zwischen Varianten explizit modelliert werden, um konsistente Softwaresysteme ableiten zu können.

Die Arbeit von *Martin Becker* verfolgt einen Ansatz bestehend aus zwei Abstraktionsebenen, die Spezifikationsebene und die Realisierungsebene.

Auf der Spezifikationsebene werden extern sichtbare Eigenschaften der Variabilität betrachtet. Hier werden also Realisierungsdetails vernachlässigt. Die Informationen werden durch Featuremodelle erfasst und repräsentiert. Varianten werden in sogenannten Profilen beschrieben, die das Binden von Variabilität steuern. Auf dieser Ebene werden diese Informationen u.a. für die Analyse und Spezifikation der Anforderungen, zur Dokumentation der variablen Features und schließlich zur Konfiguration und Ableitung konkreter Systeme verwendet.

Die Implementierungsebene beinhaltet eine Menge von wiederverwendbaren Softwaredokumenten verschiedener Abstraktionsebenen, wie beispielsweise Systemarchitekturen, Softwarearchitekturen, Code, Testfälle etc. Auf dieser Ebene wird die Variabilität aus der Spezifikationsebene realisiert und verwaltet. Das wichtigste Konzept, um Variabilität zu repräsentieren, ist die Verwendung von Variationspunkten. Schließlich ist es nach der Realisierung möglich, konkrete Lösungen abzuleiten.

Die Anforderungen zur Realisierung und Verwaltung von Variabilität auf den zwei Abstraktionsebenen wurde durch ein Metamodell beschrieben (vgl. [Bec03]). Die

wesentlichen Elemente auf der Spezifikationsebene sind die Konzepte zur Variabilität (`Variability`) und Profilierung (`Profile`). `Variability` besteht aus einer Menge von Varianten (`Range` und `Variant`) und einer Erklärung für die Variabilität (`Rationale`). Abhängigkeiten zwischen Varianten können über das Konzept `Dependency` ausgedrückt werden. Zudem können Informationen bezüglich unterstützter Bindezeiten angegeben werden (`BindingTime`). In einem `Profile` werden Varianten spezifiziert. Es umfasst eine Reihe von Zuweisungen (`Assignment`), von denen jede einer Entscheidung zur Bindung der Varianten entspricht.

Die Realisierungsebene adressiert als grundlegendes Konzept den Variationspunkt (`VariationPoint`). Jedes Softwaredokument (*Asset*) einer Produktfamilie kann Variationspunkte enthalten. Mit einem Variationspunkt wird also die Variabilität aus der Spezifikationsebene umgesetzt. Dabei herrscht eine [n..m]-Beziehung vor. Lokale Abhängigkeiten (`LocalDependency`) sind Abhängigkeiten zwischen Variationspunkten, die nicht auf der Spezifikationsebene ausgedrückt werden können und somit in die Realisierungsebene propagiert werden. Jeder Variationspunkt wird über einen bestimmten Mechanismus (`Mechanism`) realisiert. Die Mechanismen sind in drei Kategorien unterteilt:

- **Selektion**: Eine existierende Variante wird selektiert. Über die Spezifikation (`Specification`) werden die entsprechenden Konstrukte zur Selektion angegeben. In Programmiersprachen sind derartige Konstrukte zum Beispiel Kontrollstrukturen wie if/else oder switch. In objektorientierten Programmiersprachen kommt zusätzlich die Vererbung als eine mögliche Spezifikation hinzu.

- **Generierung:** Eine Variante wird generiert. Die Spezifikation beinhaltet die Eingabeinformationen für den Generator. Die generierte Ausgabe ist schließlich die Spezialisierung des Variationspunktes.

- **Substitution:** Der Variationspunkt wird durch eine eindeutige Variante ersetzt. Somit dienen Variationspunkte als Platzhalter für Varianten.

Variationspunkte werden in zwei Subtypen unterschieden, dynamische Variationspunkte (`DynamicVariationPoint`) und statische Variationspunkte (`StaticVariationPoint`). Dynamische Variationspunkte werden erst zur Laufzeit gebunden. Sie werden also während der Entwicklungsphase nicht weiter spezialisiert. Statische Variationspunkte hingegen werden zur Entwicklungszeit gebunden. Sie besitzen somit eine Spezifikation (`Specification`), die eine Beschreibung (`Rationale`) und eine Regel zur Auflösung der Variation (`ResolutionRule`) beinhaltet. Hierbei können natürlich oben definierte Mechanismen eingesetzt werden, um die Bindung zu automatisieren. Das Ergebnis der Bindung der Variation zu einer bestimmten Variante ergibt einen `ResolvedVariationPoint`.

Softwaredokumente werden in zwei Typen unterschieden, Statische (`StaticAsset`) und Generische (`GenericAsset`). Statische Softwaredokumente enthalten keine variablen Stellen. Sie besitzen somit keine statischen Variationspunkte. Generische Softwaredokumente wiederum besitzen variable Stellen, sodass sie mit statischen Variationspunkten durchaus assoziiert sind. Die Instanziierung generischer

Softwaredokumente zu konkreten führt zu einem abgeleiteten Softwaredokument (`DerivedAsset`). Somit besitzen abgeleitete Softwaredokumente nur noch gebundene Variationspunkte.

Die Abhängigkeiten zwischen der Spezifikationsebene und der Realisierungsebene wird über eine `implements`-Assoziation zum Ausdruck gebracht. Diese Assoziation wird manuell erstellt, wenn ein Softwaredokument implementiert wird. Die Konsistenz dieser Assoziation muss zudem über den gesamten Lebenszyklus erhalten bleiben.

Des Weiteren geht *Martin Becker* in seinem Aufsatz auf die konkrete Syntax (Instanziierung des Metamodells) ein. Sie wird als *VSL* bezeichnet und basiert auf Extensible Markup Language (XML). Mittels VSL können Variationspunkte spezifiziert werden und zudem ein Mechanismus zur Realisierung des Variationspunktes angegeben werden. Der Mechanismus zur Generierung von Varianten wird über Extensible Stylesheet Language Transformation (XSLT) und JScript realisiert. Über ein Profil wird die Bindung und somit die Auflösung von Variationspunkten gesteuert.

Abschließend werden die beschriebenen Konzepte von *Martin Becker* anhand der Bewertungskriterien analysiert.

1. **Konzepte der Modellierung**
   a) Variabilitätsmodellierung
      - *Art der Modellierung:* VSL definiert Auswahlmodelle, die Variabilität, Variationspunkte und Varianten spezifizieren.
      - *Strukturierungsmaßnahmen:* Keine weiteren Strukturierungsmaßnahmen.
      - *Berücksichtigung von Abstraktionsebenen:* In VSL gibt es zwei Abstraktionsebenen, die Spezifikationsebene und die Realisierungsebene. Insofern gibt es in VSL Ansätze zur Komplexitätsbeherrschung. Allerdings fehlt es an feingranulareren Konzepten zur Berücksichtigung des Entwicklungsprozesses.
      - *Unterstützung bei der Modellierung:* Keine Unterstützung.
      - *Modellierung von Variabilitätsmechanismen:* Keine Unterstützung.
      - *Modellierung von Variantenarten:* Keine Unterstützung.
   b) Restriktionsmodellierung
      - *Definition variabler Eigenschaften:* Siehe Restriktionssprachen.
      - *Restriktionssprachen:* Es gibt im Metamodell das Element `Dependency` zur Formalisierung von Restriktionen. Es wird allerdings nicht weiter spezifiziert wie derartige Restriktionen aussehen.

2. **Konzepte der Bindung**
   a) Konfigurierung
      - *Konfiguration des Variabilitätsmodells:* Konfigurationen in VSL werden durch Profile spezifiziert. Profile sind mit der Variabilität verknüpft, sodass sie somit Entscheidungsmodelle darstellen.
      - *Unterstützung bei der Konfigurierung:* Keine Unterstützung.

   b) Generierung
      - *Inferenz konkreter Produkte:* Anhand des Profils und über XSLT sowie JScript Programmen werden XML-basierte Ergebnisdokumente erzeugt, die jeweils einer Variante entsprechen. Das Resultat muss allerdings noch in die konkreten Softwaredokumente manuell übertragen werden.

4.5.5. Orthogonale Variabilitätsmodelle

Pohl et al. haben in ihrem Buch [PBvdL05] das Paradigma der Softwareproduktlinien beschrieben und mögliche Methoden sowie Techniken eingeführt. Unter anderem wird in dem Buch das Konzept zu den sogenannten orthogonalen Variabilitätsmodellen erläutert. Sie definieren orthogonale Variabilitätsmodelle als separate Modelle, die im gesamten Domain Engineering der Softwareproduktlinienentwicklung Variabilität identifizieren und definieren sowie die Assoziationen zu den im Prozess entstehenden Softwaredokumenten herstellen. Sie motivieren den Ansatz der separaten Variabilitätsmodellierung, indem sie die Nachteile der integrierten Variabilitätsmodellierung, also der direkten Variabilitätsmodellierung in den Softwaredokumenten, aufzeigen. Bei der integrierten Variabilitätsmodellierung sind die Informationen bzgl. der Variabilität einer bestimmten Prozessphase über die verschiedenen Softwaredokumente hinweg verstreut. Dies entspricht also einer horizontalen Verteilung der Variabilitätsinformationen. Die manuelle Kollektion und Konsistenzsicherung dieser Informationen ist daher enorm schwierig wenn nicht unmöglich. Darüber hinaus entsteht Variabilität zu jeder Entwicklungsphase und muss somit in den über die Zeit hinweg entstehenden Softwaredokumenten modelliert werden. Bei integrierter Variabilitätsmodellierung bewirkt diese somit eine Verstreuung der Variabilitätsinformationen über die verschiedenen Abstraktionsebenen. In diesem Fall herrscht also eine vertikale Verteilung vor. Die manuelle Ermittlung aller Implikationen über Abstraktionsebenen hinweg ist allerdings sehr komplex. Weiterhin argumentieren die Autoren, dass die Softwaredokumente für sich betrachtet bereits sehr komplex sind. Eine integrierte Variabilitätsmodellierung fügt eine weitere Komplexitätsdimension hinzu. Zudem ist es besonders schwierig, Variabilitätsinformationen zu sammeln, da oftmals unterschiedliche Konzepte bei der Variabilitätsmodellierung verwendet werden.

In [PBvdL05] wird daher ähnlich wie im Ansatz von *Martin Becker* [Bec03] ein allgemeines Metamodell vorgestellt. Es beinhaltet zwei Kernelemente: `Variation Point` und `Variant`. Mit `Variation Point` können variable Stellen in den verschiedenen Softwaredokumenten repräsentiert werden. `Variant` umfasst alle mögli-

chen Ausprägungen der Variationspunkte. Variation Point ist eine abstrakte Klasse, die in zwei konkrete Unterklassen External Variation Point und Internal Variation Point spezialisiert wird. External Variation Point bezeichnet kundensichtbare Varianten, während Internal Variation Point entwicklersichtbare Varianten umfasst. Die abstrakte Assoziationsklasse Variability Dependency modelliert die Beziehungen zwischen Variation Point und Variant. Sie kann entweder optionale (Optional) oder verbindliche (Mandatory) Beziehungen ausdrücken. Zudem kann eine Gruppierung optionaler Varianten mit Festlegung einer Kardinalität über die Klasse Alternative Choice festgelegt werden. Die Kardinalität spezifiziert die mögliche Anzahl der selektierbaren Varianten aus der definierten Gruppe. Weiterhin umfasst das Metamodell die Möglichkeit, Regeln zur Restriktion (Constraints) des Variabilitätsmodells auszudrücken. Hiervon gibt es drei verschiedene Typen:

1. Constraints zwischen Varianten. Eine Variante kann eine oder mehrere andere Varianten erfordern oder ausschließen. Genauso kann eine Variante von einer oder mehrerer Varianten erfordert bzw. ausgeschlossen werden. Hierfür wird eine abstrakte Assoziationsklasse Variant Constraint Dependency mit den Spezialisierungen Requires V_V und Excludes V_V eingeführt.

2. Constraints zwischen Varianten und Variationspunkten. Eine Variante kann einen oder mehrere Variationspunkte erfordern oder ausschließen. Genauso kann ein Variationspunkt von einer oder mehrerer Varianten erfordert bzw. ausgeschlossen werden. Hierfür wird eine abstrakte Assoziationsklasse Variation Point to Variant Constraint Dependency mit den Spezialisierungen Requires V_VP und Excludes V_VP eingeführt.

3. Constraints zwischen Variationspunkten. Ein Variationspunkt kann einen oder mehrere Variationspunkte erfordern oder ausschließen. Genauso kann ein Variationspunkt von einem oder mehreren Variationspunkten erfordert bzw. ausgeschlossen werden. Hierfür wird eine abstrakte Assoziationsklasse Variation Point Constraint Dependency mit den Spezialisierungen Requires VP_VP und Excludes VP_VP eingeführt.

Damit Assoziationen zwischen Variabilitätsmodell und Softwaredokumenten hergestellt werden können, wird im Metamodell eine abstrakte Klasse Development Artefact modelliert. Die Spezialisierungen dieser Klasse entsprechen den konkreten Softwaredokumenten der verschiedenen Abstraktionsebenen. Die Assoziationen zwischen Varianten und Softwaredokumenten wird durch die Assoziationsklasse Artefact Dependency realisiert. Zudem ist es möglich, Variationspunkte mit Softwaredokumenten zu assoziieren. Für diesen Fall wird die Assoziationsklasse VP Artefact Dependency eingeführt. Auf diese Weise ist es möglich, Variationspunkte und Varianten in Softwaredokumenten über beliebige Granularitätsstufen zu modellieren.

Neben der Formalisierung des Variabilitätsmodells als Metamodell wurde in [PBvdL05] auch eine konkrete grafische Notation eingeführt. Hierbei sind Variationspunkte, Varianten, die Beziehungen von Variationspunkten und Varianten, die

Gruppierungsmöglichkeit mit Kardinalitätsangabe, die Constraintarten sowie die Assoziationen zu den Softwaredokumenten enthalten.

Schließlich werden erneut die Kriterien zur Bewertung der Konzepte orthogonaler Variabilitätsmodelle herangezogen.

1. **Konzepte der Modellierung**
   a) Variabilitätsmodellierung
      - *Art der Modellierung:* Orthogonale Variabilitätsmodelle sind Auswahlmodelle, die aus Variationspunkten und Varianten bestehen.
      - *Strukturierungsmaßnahmen:* Keine weiteren Strukturierungsmaßnahmen.
      - *Berücksichtigung von Abstraktionsebenen:* Mit orthogonalen Variabilitätsmodellen können beliebige Softwaredokumente des Entwicklungsprozesses assoziiert werden. Allerdings ist dieses Schichtenkonzept isoliert, sodass vertikale Beziehungen nicht modelliert werden können.
      - *Unterstützung bei der Modellierung:* Keine Unterstützung.
      - *Modellierung von Variabilitätsmechanismen:* Keine Unterstützung.
      - *Modellierung von Variantenarten:* Keine Unterstützung.

   b) Restriktionsmodellierung
      - *Definition variabler Eigenschaften:* Verbindliche, optionale und alternative Varianten können definiert werden.
      - *Restriktionssprachen:* Es gibt drei Formen von Restriktionen: (1) zwischen Varianten, (2) zwischen Varianten und Variationspunkten und (3) zwischen Variationspunkten. Für jede Form gibt es `requires`- und `excludes`-Relationen.

2. **Konzepte der Bindung**
   a) Konfigurierung
      - *Konfiguration des Variabilitätsmodells:* Keine Unterstützung.
      - *Unterstützung bei der Konfigurierung:* Keine Unterstützung.

   b) Generierung
      - *Inferenz konkreter Produkte:* Keine Unterstützung.

4.5.6. COVAMOF

Sinnema et al. beschreiben in ihrem Aufsatz aus [SDNB04] ihren Ansatz zur Variabilitätsmodellierung - ConIPF Variability Modeling Framework (COVAMOF). COVAMOF ist dabei im Rahmen des Projekts Configuration of Industrial Product Families (ConIPF) entstanden [HKW+06]. COVAMOF stellt neben der Variabilitätsmodellierung auf allen Abstraktionsebenen insbesondere die Abhängigkeiten zwischen Variationspunkten in den Fokus. So werden zwei Modelle in der sogenannten COVAMOF

Varaibility View (CVV) definiert: (1) Variation Point View und (2) Dependency View. Der Grund für diese zweigeteilte Sicht ist, dass nicht nur Variabilität als primär wichtigstes Konzept betrachtet wird, sondern auch die Abhängigkeiten zwischen Variationspunkten. Insbesondere ist es auf diese Weise möglich, komplexere Abhängigkeiten zu modellieren ohne den Überblick zu verlieren. Damit die Beziehungen zwischen beiden Modellen erfasst werden können, beinhalten beide Sichten jeweils die Elemente der anderen Sicht als Attribute.

Die zentralen Elemente im Metamodell sind `Variation Point` und `Dependency`. Ein `Variation Point` realisiert dabei (über eine `Realization` Klasse) eine oder mehrere `Variation Points` der nächsthöheren Abstraktionsebene. Zudem ist `Variation Point` mit einer Klasse `Dependency` mit einer [0..n]-Kardinalität assoziiert. Umgekehrt ist jede `Dependency` mit der Klasse `Variation Point` mit einer [1..n]-Kardinalität assoziiert. Auf diese Weise kann stets eine konsistente Beziehung zwischen beiden Entitäten hergestellt werden. Schließlich beschreibt die `Dependency Interaction` die Interaktionen zwischen modellierten Abhängigkeiten.

Mit einem Variationspunkt werden alle variablen Stellen in den Softwaredokumenten markiert. Jeder Variationspunkt besteht aus beliebig vielen Varianten. *Sinnema et al.* definieren fünf verschiedene Typen von Variationspunkten:

- `optional`
  Ein Variationspunkt vom Typ `optional` drückt aus, dass aus der Menge der assoziierten Varianten null oder eine selektiert werden kann ([0..1]-Kardinalität).

- `alternative`
  Ein Variationspunkt vom Typ `alternative` drückt aus, dass aus der Menge der assoziierten Varianten genau eine selektiert werden muss ([1..1]-Kardinalität).

- `optional variant`
  Ein Variationspunkt vom Typ `optional variant` drückt aus, dass aus der Menge der assoziierten Varianten null oder n Varianten selektiert werden können ([0..n]-Kardinalität).

- `variant`
  Ein Variationspunkt vom Typ `variant` drückt aus, dass aus der Menge der assoziierten Varianten eine oder n Varianten selektiert werden können ([1..n]-Kardinalität).

- `value`
  Ein Variationspunkt vom Typ `value` drückt aus, dass ein Wert aus dem Bereich `range` dem Variationspunkt zugewiesen werden kann.

Des Weiteren wird in [SDNB04] das Konzept der Realisierung von Variationspunkten über Abstraktionsebenen eingeführt. Demnach können Variationspunkte aus einer höheren Abstraktionsebene durch einen Variationspunkt aus einer niedrigeren Abstraktionsebene realisiert werden. Hierfür wird die `Realization`-Beziehung eingeführt.

Das zweite wesentliche Element in COVAMOF sind Abhängigkeiten. Sie werden mit Variationspunkten assoziiert und beschreiben Restriktionen bei der Selektion von Varianten. So können einfache Restriktionen wie etwa der gegenseitige Ausschluss aber auch komplexe Einschränkungen ausgedrückt werden. *Sinnema et al.* unterscheiden dabei drei Arten von Assoziationstypen:

1. `predictable`
   Eine Assoziation vom Typ `predictable` wird verwendet, wenn vollständig bekannt ist, welchen Einfluss eine Variantenselektion haben wird.

2. `directional`
   Eine Assoziation vom Typ `directional` wird verwendet, wenn nicht vollständig bekannt ist, welchen Einfluss eine Variantenselektion haben wird, aber zumindest richtungsweisende Aussagen (positiver oder negativer Einfluss) getroffen werden können.

3. `unknown`
   Eine Assoziation vom Typ `unknown` wird verwendet, wenn nicht bekannt ist, welchen Einfluss eine Variantenselektion haben wird.

Neben Assoziationstypen werden verschiedene Abhängigkeitstypen definiert:

1. `logical`
   Eine Abhängigkeit vom Typ `logical` wird über eine Funktion `valid` spezifiziert und beschreibt so die Restriktion für die assoziierten Variationspunkte. Hierbei können Variationspunkte nur über den Assoziationstypen `predictable` angebunden werden.

2. `numerical`
   Eine Abhängigkeit vom Typ `numerical` wird über einen numerischen Wert `N` definiert und beschreibt so die Restriktion für die assoziierten Variationspunkte. Hierbei können Variationspunkte über alle drei Assoziationstypen angebunden werden.

3. `nominal`
   Eine Abhängigkeit vom Typ `nominal` wird über eine Menge von Kategorien `categories` spezifiziert. Jeder Variationspunkt gehört zu einer dieser Kategorien und beschreibt auf diese Weise die Restriktion für die assoziierten Variationspunkte. Hierbei können Variationspunkte nur über den Assoziationstyp `unknown` angebunden werden.

Weiterhin wurde in [SDNB04] die Interaktion zwischen Abhängigkeiten definiert. Hierfür wird das Element `Dependency Interaction` eingeführt und verbindet zwei oder mehrere Abhängigkeiten miteinander.

Schließlich wurde für die oben beschriebenen Konzepte eine textuelle Sprache, die CVV Language (CVVL), entwickelt. CVVL ist eine XML-basierte Sprache und repräsentiert CVV in Textform. Sie besteht aus zwei Teilen, (1) die Notation für die `Variation Point View` und (2) die Notation für die `Dependency View`. Es sind noch

weitaus mehr Konzepte in COVAMOF vorhanden. Sie sind aber hier nicht genauer aufgeführt. So kann ein Variationspunkt einen Zustand haben kann (`state`), einen Realisierungsmechanismus definieren (`mechanism`), die Bindezeiten spezifizieren (`bindingtime`) und eine Beschreibung (`rationale`) beinhalten. Für weitere Details wird auf [SDNB04] verwiesen.

Zur genaueren Bewertung von COVAMOF werden im Folgenden anhand der eingeführten Kriterien die Konzepte evaluiert.

1. **Konzepte der Modellierung**

   a) Variabilitätsmodellierung

      - *Art der Modellierung:* COVAMOF stellt Auswahlmodelle mit Variationspunkten und Varianten zur Verfügung.
      - *Strukturierungsmaßnahmen:* Keine weiteren Strukturierungsmaßnahmen.
      - *Berücksichtigung von Abstraktionsebenen:* Variationspunkte werden hierarchisch über mehrere Abstraktionsebenen hinweg modelliert. Hierfür wird im Metamodell das Element `Realization` eingeführt. Auf diese Weise kann das Variabilitätsmodell geeignet unterteilt werden.
      - *Unterstützung bei der Modellierung:* Keine Unterstützung.
      - *Modellierung von Variabilitätsmechanismen:* Keine Unterstützung.
      - *Modellierung von Variantenarten:* Keine Unterstützung.

   b) Restriktionsmodellierung

      - *Definition variabler Eigenschaften:* Es werden fünf Arten der Restriktionen beschrieben.
      - *Restriktionssprachen:* Durch die `Dependency View` werden weitere Restriktionsarten realisiert.

2. **Konzepte der Bindung**

   a) Konfigurierung

      - *Konfiguration des Variabilitätsmodells:* Entscheidungsmodelle entstehen nach dem Konfigurierungsprozess.
      - *Unterstützung bei der Konfigurierung:* COVAMOF wertet die einzelnen Konfigurierungsschritte aus und unterstützt auf diese Weise den Benutzer bei der Selektion oder Deselektion von Varianten.

   b) Generierung

      - *Inferenz konkreter Produkte:* Keine Unterstützung.

4.5.7. CONSUL und pure::variants

Beuche et al. präsentieren in ihrem Artikel [BPSP04] einen Ansatz zur Variabilitätsmodellierung in allen Phasen der Softwareentwicklung mit der entsprechenden Werkzeugunterstützung CONSUL. Dieses Werkzeug ist die Basis für das heute im industriellen Einsatz verwendete Werkzeug pure::variants der Pure-Systems GmbH [psG03]. Im Folgenden wird CONSUL genauer erläutert. Für weitere Informationen zu pure::variants sei auf [psG03] verwiesen.

Der Grundgedanke hinter CONSUL ist die systematische Wiederverwendung von Softwareentwicklungsdokumenten in allen Phasen zu unterstützen. Es werden insbesondere im Vergleich zu anderen Ansätzen Implementierungsdokumente genau behandelt. Auf diese Weise ist es möglich, den Übergang von kundensichtbaren Eigenschaften der Softwaresysteme, die in der Domänenanalyse ermittelt werden, zu Realisierungsdetails zu gewährleisten. Hierfür kommen in CONSUL mehrere Modelle zum Einsatz:

1. **Featuremodelle**. Durch Featuremodelle werden Gemeinsamkeiten und Unterschiede einer Domäne erfasst und repräsentiert. In CONSUL werden viele Konzepte aus der FODA-Methodik als Basis verwendet und um weitere Aspekte erweitert.

2. **Featurekonfiguration**. Die Featurekonfiguration repräsentiert eine spezifische Ausprägung eines Featuremodells, also die Menge der Features, die selektiert wurden.

3. **Familienmodell**. Das Familienmodell beinhaltet die Beschreibungen der Aktionen zur Bindung entsprechender Implementierungsdateien oder zur Generierung von Quelltext. Auf diese Weise werden konkrete Varianten zur Verfügung gestellt.

Ein Featuremodell besteht aus hierarchisch strukturierten Features mit verbindlichen, optionalen und alternativen Relationen zwischen Features. Jedes Feature hat einen Typ und einen Wert. Zusätzlich können Featurebeschreibungen existieren, die Definitionen und Erläuterungen beinhalten. Constraints können in CONSUL in zwei Sprachen ausgedrückt werden, (1) Prolog und (2) XMLTrans. Dadurch ist es möglich, beliebig komplexe Constraints zu beschreiben.

Das CONSUL Familienmodell besteht aus einer hierarchischen Struktur, die drei Elemente beinhaltet: (1) Komponente (engl. `Component`), (2) Bestandteil (engl. `Part`) und (3) Quelle (engl. `Source`). `Components` haben einen Namen und bestehen aus `Parts`, die wiederum aus `Sources` bestehen. Somit ist ein `Component` ein Element mit höchster Abstraktionsstufe, die alle Konfigurationsinformationen kapselt. Das Element `Part` ist eine logische Einheit, die vielfältige Aspekte der Softwaresysteme beschreibt. Hierfür hat ein `Part` zusätzlich zu einem Namen weiterhin einen Typ, der diesen Sachverhalt beschreibt. Zum Beispiel können auf diese Weise Klassen oder Schnittstellen einer Programmiersprache beschrieben werden. Folgende Typen wurden von *Beuche et al.* in [BPSP04] eingeführt:

- `interface(x)`: Repräsentiert die Schnittstelle `x` einer Softwarekomponente.

- `class(x)`: Repräsentiert eine Klasse `x` mit Attributen und Methoden.
- `object(x)`: Repräsentiert ein Objekt `x`, also eine Instanz einer Klasse.
- `classalias(x)`: Repräsentiert einen abstrakten Typnamen `x`, das zu einer konkreten Klasse gebunden wird.
- `flag(x)`: Repräsentiert eine Konfigurationsentscheidung durch Festlegung eines bestimmten Wertes `x`.
- `variable(x)`: Wie `flag(x)`, es sollte allerdings nicht für Konfigurationszwecke verwendet werden.
- `project(x)`: Repräsentiert alles, was nicht mit den vorherigen Typen beschrieben werden kann.

Schließlich sind `Sources` die physikalischen Repräsentationen von `Parts`. Ein `Source` hat nur einen Typ, aber keinen Namen. Durch den Typ wird bestimmt, wie Programmelemente eingebunden werden. Folgende Typen wurden in [BPSP04] eingeführt:

- `file`: Repräsentiert eine Datei, die unverändert verwendet wird.
- `flagfile`: Repräsentiert ein C/C++ Präprozessor Flag.
- `makefile`: Repräsentiert eine `Makefile` Variable.
- `classalias`: Repräsentiert eine C/C++ `typedef` Variable.

Alle beschriebenen Elemente können durch Constraints angereichert werden und auf diese Weise bestimmte Regeln bei der Zusammensetzung der Softwarekomponenten beschrieben werden. Hierfür wird, wie bereits erwähnt, entweder Prolog oder XMLTrans angewendet. Das CONSUL Familienmodell wird in Form einer textuellen Beschreibung modelliert. Hierfür wurde die CONSUL Family Description Language (CFDL) entwickelt.

CONSUL wird im Folgenden durch die bereits eingeführten Bewertungskriterien genauer analysiert.

1. **Konzepte der Modellierung**
   a) Variabilitätsmodellierung
      - *Art der Modellierung:* In CONSUL werden hierarchisch strukturierte Featuremodelle und Familienmodelle eingesetzt. Während CONSUL Featuremodelle ähnlich zu den Featuremodellen aus FODA sind, ist das Familienmodell ein erweiterter Ansatz, das die Beschreibung konkreter Varianten ermöglicht.
      - *Strukturierungsmaßnahmen:* Es können abstrakte Features eingesetzt werden.

- *Berücksichtigung von Abstraktionsebenen:* Variabilität kann zu jeder Abstraktionsebene erfasst werden, es gibt allerdings nur ein zentrales Featuremodell ohne die Möglichkeit, auf weitere Featuremodelle zu referenzieren.
- *Unterstützung bei der Modellierung:* Keine Unterstützung.
- *Modellierung von Variabilitätsmechanismen:* Keine explizite Unterstützung. Variabilitätsmechanismen können allerdings implizit durch Features modelliert werden.
- *Modellierung von Variantenarten:* Durch die Angabe von `Sources` können verschieden Variantenarten spezifiziert werden.

b) Restriktionsmodellierung

- *Definition variabler Eigenschaften:* Features können als verbindlich, optional und alternativ markiert werden.
- *Restriktionssprachen:* Durch Prolog oder XMLTrans können in CONSUL weitere Restriktionen im Familienmodell angegeben werden.

2. **Konzepte der Bindung**

a) Konfigurierung

- *Konfiguration des Variabilitätsmodells:* Die Konfiguration eines Featuremodells ist ein Entscheidungsmodell.
- *Unterstützung bei der Konfigurierung:* CONSUL unterstützt den Konfigurierungsprozess durch Auswertung der Restriktionen und der automatischen Implikation der notwendigen Aktionen.

b) Generierung

- *Inferenz konkreter Produkte:* CONSUL ermöglicht die Ableitung konkreter Quelltextdokumente der Programmiersprachen C/C++.

4.5.8. Vergleich

Die wichtigsten Ansätze wurden in den vorangegangenen Abschnitten erläutert und bewertet. Im Folgenden werden abschließend diese Ansätze mit dem dieser Arbeit zugrundeliegenden Ansatz verglichen. Tabelle 4.9 zeigt das Resultat dieses Vergleichs.

In dieser Arbeit wurde als primäre Modellierungsart das Konzept der Auswahlmodelle gewählt, da diese sich besonders gut zur Repräsentation verschiedener Softwaredokumente, die im Verlauf des Entwicklungsprozess entstehen, eignen. VSL, Orthogonales Variabilitätsmodell (OVM) und COVAMOF basieren ebenfalls auf Auswahlmodellen. Der wesentliche Nachteil von Auswahlmodellen ist, das es kein Strukturierungskonzept beinhaltet. In dieser Arbeit wurden daher Auswahlmodelle durch das Gruppierungskonzept erweitert, um diesem Nachteil entgegenzuwirken. Die anderen Ansätze basieren auf hierarchisch strukturierten Modellen. Der Vorteil

dieser Modelle ist, dass ein Strukturierungskonzept bereits enthalten ist. Hierfür werden zum Beispiel abstrakte Features oder Variationspunkte eingesetzt. Der Bezug zu Softwaredokumenten kann allerdings durch diese Modelle nicht immer intuitiv hergestellt werden.

Damit Abstraktionsebenen im Referenzprozess geeignet erfasst werden, wurde im Rahmen dieser Arbeit ein schichtenbasierter Ansatz gewählt, der es erlaubt, Variabilität in mehreren Variabilitätsmodellen zu dokumentieren. Diese Teilung entspricht den beschriebenen Abstraktionsebenen. Durch einen Referenzierungsmechanismus ist es möglich, alle geteilten Modelle zueinander zu assoziieren. Dieses Konzept wurde auch in CBFM fovorisiert. FODA, FeatuRSEB und CONSUL hingegen basieren vollständig auf Hierarchisierung. Dies führt allerdings zu einem zentralen, aber enorm großen Variabilitätsmodell, das an Übersichtlichkeit und Verständlichkeit verliert. In VSL werden zwei Ebenen eingeführt, die Spezifikations- und Realisierungsebene. Diese Unterteilung eignet sich Komplexität bis zu bestimmten Grenzen zu reduzieren. Für feingranularere Abstraktionsebenen ist es jedoch ungeeignet. In COVAMOF wird zur Unterteilung in Abstraktionsebenen das `Realization`-Konzept verwendet, das Variationspunkte bzw. Varianten über eine Realisierungsassoziation miteinander verbindet. Dieses Konzept eignet sich zwar sehr gut zur Bestimmung der realisierenden Elemente in den verschiedenen Abstraktionsebenen, allerdings kann hiermit ebenenspezifische Variabilität nicht erfasst werden.

Erweiterte Konzepte, wie etwa die Unterstützung bei der Variabilitätsmodellierung, das Berücksichtigen von Variabilitätsmechanismen oder die Erfassung der Variantenart, werden in wenigen Arbeiten behandelt. So ist diese Arbeit die Einzige, welche die proaktive Modellierung von Variabilität unterstützt. Dies wird dadurch erreicht, dass dieser Ansatz über ein Assoziationskonzept verfügt, um mit Entitäten der Softwaredokumente zu verknüpfen. Darüber hinaus werden in dieser Arbeit Variabilitätsmechanismen explizit berücksichtigt. Sie sind ein wichtiger Bestandteil in der Realisieurung der Variabilität und sollten daher auch explizit modelliert werden. In den Ansätzen FODA, FeatuRSEB, CBFM und CONSUL ist diese nur implizit durch Einführung entsprechender Features möglich. Die Variantenarten werden bei CONSUL durch das `Source`-Konzept unterstützt. Diese sind stets separierte Varianten. In dieser Arbeit wird der Ansatz um integrierte Varianten erweitert.

Die Definition variabler Eigenschaften stellt eines der Kernkonzepte in der Variabilitätsmodellierung dar und wird daher auch von jedem Ansatz unterstützt. Die meisten Ansätze realisieren die Angabe von verbindlichen, optionalen und alternativen Varianten. Die Gruppen- und Variantenkardinalitäten gehen über diese einfachen Eigenschaften hinaus und ermöglichen die Angabe beliebiger Restriktionen. Daher wurde in dieser Arbeit, wie in CBFM auch, dieser Ansatz gewählt. Restriktionssprachen hingegen sind nicht in jeder Arbeit enthalten. So sind in FODA, FeatuRSEB und OVM einfache requires- und excludes-Restriktionen erlaubt. In VSL und COVAMOF können Restriktionen durch ein `Dependency`-Konzept formuliert werden. Formale Restriktionssprachen werden lediglich in dieser Arbeit, in CBFM und in CONSUL unterstützt. In diesen Ansätzen können beliebige Restriktionen formuliert werden.

FODA, FeatuRSEB und OVM definieren kein Konzept zur Bindung von Varianten.

	Mengi	FODA	FeatuRSEB	CBFM	VSL	OVM	COVAMOF	CONSUL
Variablität								
Art der Modellierung	Auswahl-modell	Hierarchisch strukturiertes Modell	Hierarchisch strukturiertes Modell	Hierarchisch strukturiertes Modell	Auswahl-modell	Auswahl-modell	Auswahl-modell	Hierarchisch strukturiertes Modell
Strukturierung	Gruppen	abstrakte Features	Variations-punkte	abstrakte Features	×	×	×	abstrakte Features
Abstraktionsebenen	Schichten mit Referenzierung	Hierarchie	Hierarchie	Schichten mit Referenzierung	Spezifikations- und Realisierungsebene	×	Realization-Entität	Hierarchie
Unterstützung	proaktiv	×	×	×	×	×	×	×
Variabilitätsmechanismen	explizit	implizit	implizit	implizit	×	×	×	implizit
Variantenarten	integriert und separiert	×	×	×	×	×	×	Sources
Restriktionen								
Variable Eigenschaften	Gruppen- und Variantenkardinalitäten	verbindliche, optionale und alternative Features	verbindliche, optionale, alternative und oder-Features	Gruppen- und Featurekardinalitäten	Dependency	verbindliche, optionale und alternative Varianten	optional, alternative, optional variant, variant, value	verbindliche, optionale und alternative Features
Restriktionssprachen	Erweiterte Aussagenlogik oder WCRL	mutually exclusive with, requires (informell)	Relationen mit beliebiger Semantik (informell)	OCL, XPath	Dependency	requires, exclude	Dependency View	Prolog, XMLTrans
Konfigurierung								
Konfiguration	Entschei-dungsmodell	×	×	Entschei-dungsmodell	Entschei-dungsmodell	×	Entschei-dungsmodell	Entschei-dungsmodell
Unterstützung	proaktive und reaktive Validierung	×	×	proaktive Validierung	×	×	proaktive Validierung	proaktive Validierung
Generierung								
Inferenz konkreter Produkte	WCRL-Dokumente durch smodels	×	×	×	XML-Dokumente durch XSLT, Jscript	×	×	C/C++ Quelltext-dokumente

Tabelle 4.9.: Vergleichsaufstellung mit den Konzepten dieser Arbeit und verwandter Arbeiten

In dieser Arbeit und in CBFM, VSL, COVAMOF und CONSUL werden Entscheidungsmodelle eingesetzt. Diese haben den Vorteil, dass sie stets synchron zum Variabilitätsmodell sind. Der Konfigurierungsprozess wird in CBFM, COVAMOF und CONSUL durch proaktive Validierung unterstützt. In dieser Arbeit wird dieser Ansatz durch eine reaktive Validierung ergänzt.

Die Inferierung konkreter Produkte nach der Konfigurierung wird durch diese Arbeit und durch die Arbeiten in VSL und CONSUL unterstützt. Während CONSUL speziell für C/C++-Quelltextdokumente zugeschnitten ist, wird in VSL XML-Dokumente generiert, die durch einen weiteren Transformierungsschritt in beliebige Sofwaredokumente überführt werden können. Einen ähnlichen Ansatz verfolgt diese Arbeit, allerdings mit dem Unterschied, dass statt XML-Dokumente WCRL-Dokumente generiert werden. Der Grund hierfür liegt in der leistungsstarken Inferenzmaschine smodels.

4.6. Zusammenfassung

Variabilität ist ein Aspekt, der im gesamten Entwicklungsprozess berücksichtigt werden muss. In diesem Kapitel wurden in diesem Zusammenhang diverse Konzepte hinsichtlich der Modellierung und Bindung von Variabilität beschrieben. Zur expliziten Erfassung der Variabilität wurde ein Auswahlmodell entwickelt, das durch ein Variationpunkt- und Variantenkonzept auf allen Abstraktionsebenen anwendbar ist. Das Auswahlmodell wurde durch Strukturierungsmaßnahmen geeignet erweitert. Durch ein sogenanntes Gruppierungskonzept kann somit das Auswahlmodell geeignet strukturiert werden. Besonders hervorzuheben ist die explizite Modellierung von Variabilitätsmechanismen. Hierdurch wird die Trennung zwischen der funktionalen Logik einer Software und der Mechanismen zur Realisierung von Variabilität gewährleistet. Außerdem wird die Modellierung verschiedener Variantenarten unterstützt. Hierbei wurden zwei Arten identifiziert, integrierte und separierte Varianten. Weiterhin ist das Variabilitätsmodell in dieser Arbeit das Einzige, das die proaktive Modellierung von Variabilität durch Integration mit Softwaredokumenten der verschiedenen Abstraktionsebenen unterstützt.

Außerdem wurde in diesem Kapitel erläutert, dass Relationen zwischen variablen Entitäten in Form von Restriktionen existieren. Grundlegenden Eigenschaften, wie Verbindlichkeit, Optionalität und Alternativität können durch Gruppen- und Variantenkardinalitäten ausgedrückt werden. Diese Konzepte erlauben die Festlegung variabler Eigenschaften auf sehr flexible Weise. Komplexere Restriktionen über Gruppengrenzen hinweg werden in dieser Arbeit durch Restriktionssprachen formuliert. Diesbezüglich wurde eine erweiterte Form der Aussagenlogik realisiert. Weiterhin wurde mit WCRL eine aus der Literatur bekannte Restriktionssprache in die Werkeugumgebung integriert.

Neben der Modellierung von Variabilität wurde zudem die Bindung als ein weiterer wichtiger Punkt identifiziert. Als Bindungsmechanismus wurde die Selektion mit einer Konfigurationsmaschine im Hintergrund realisiert. Besondere Eigenschaften dieses Mechanismus sind die zur Variabilitätsmodellierung separate aber dennoch

synchrone Aktivität. Zusätzlich wird die Konfigurierung durch eine proaktive und reaktive Validierung in jedem Konfigurierungsschritt unterstützt.

Als Erweiterung der Selektion als Bindungsmechanismus wurde die Generierung mit einer Inferenzmaschine im Hintergrund realisiert, das die Erzeugung gebundener Softwaredokumente unterstützt.

Teil III.

Modelle und Variabilität im Referenzprozess

Kapitel 5.

Funktionsebene

5.1. Einleitung und Motivation

Funktionsnetze sind ein Beschreibungsmittel für Fahrzeugfunktionen und ihrer gegenseitigen Kommunikation. Sie werden in der frühen Phase im Entwicklungsprozess eingesetzt. Funktionsnetze konkretisieren die in der Anforderungsspezifikation definierten Features. Auf diese Weise bilden sie die erste virtuelle Realisierung der systemweiten Funktionalitäten.

Der Bedarf für Funktionsnetze entsteht aus der zu großen Lücke zwischen der Anforderungsspezifikation und der E/E-Architektur. Oftmals sind aufgrund dessen Entwurfsentscheidungen nicht nachvollziehbar. Funktionsnetze sollen an dieser Stelle diese Lücke schließen, indem sie eine Brücke zwischen der Anforderungsspezifikation und der E/E-Architektur bilden. Ihr primärer Zweck ist damit, als *Verständnismodell* für Entwickler zu dienen als auch eine *Kommunikationsgrundlage* zwischen Ingenieuren herzustellen. Abbildung 5.1 veranschaulicht den erläuterten Bedarf an Funktionsnetzen.

Derzeit besteht bei der Anwendung von Funktionsnetzen immer noch kein Konsens zwischen verschiedenen Automobilherstellern. Aspekte, die hierunter leiden sind u.a. folgende:

- der zugrunde liegende Formalismus ist unterschiedlich
- die verwendete Notation ist verschieden
- es fehlt an weiteren Abstraktionsschritten
- die Wiederverwendung ist nur rudimentär gegeben
- Varianten werden weder formal noch explizit erfasst

Für eine übergreifende und erfolgsversprechende Anwendung von Funktionsnetzen ist es daher zwingend erforderlich, diese Hindernisse zu bewältigen. Dieses Kapitel wird hierzu entsprechende Lösungsvorschläge geben. Alternativ zu dieser Arbeit sind weitere Ergebnisse aus der selben Forschungsgruppe entstanden, allerdeings mit einem anderen Schwerpunkt [GHK$^+$07, GHK$^+$08a, GKPR08, GHK$^+$08b].

Werden Entwicklungsprozesse verschiedener Automobilhersteller bezüglich der Anwendung von Funktionsnetzen genauer analysiert, kann festgestellt werden, dass

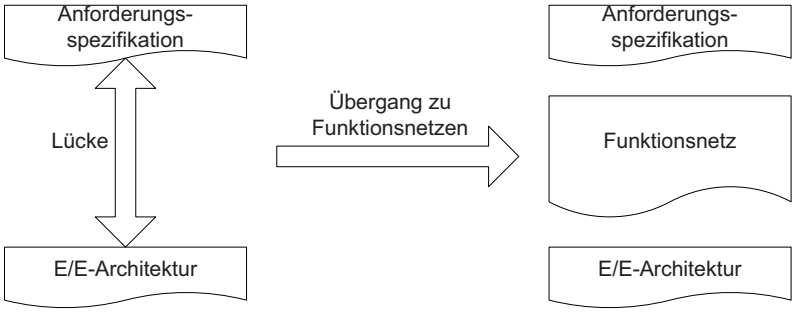


Abbildung 5.1.: Illustration für den Zweck von Funktionsnetzen

es sehr unterschiedliche Interpretationen von Funktionsnetzen existieren. Einige Automobilhersteller verstehen unter einem Funktionsnetz eine an die Hardwareplattform zugeschnittene Beschreibung. Diese wird auch oft als *Kommunikationsmatrix* (K-Matrix) bezeichnet [WH06, SZ06]. Sie ist motiviert durch den in der Automobilindustrie vorherrschenden klassischen Hardware-getriebenen Ansatz. Hierbei sind das Steuergerätenetz, sämtliche Bustopologien und Deploymententscheidungen frühzeitig bekannt. Funktionsnetze werden somit bereits spezifischen Steuergeräten zugeordnet und modelliert. Auch wenn dieser Ansatz eine Reihe von Vorteilen bietet, wie etwa den einfachen Aufbau virtueller Prototypen, die durch Simulationen, Validierungs- und Verifikationsmethoden überprüft werden können, hat er den wesentlichen Nachteil, dass Funktionsnetze in der Regel sehr groß werden, sodass sie schwierig zu verstehen und zu ändern sind. Darüber hinaus ist es aufgrund des Deployments auch nicht ohne Weiteres möglich, Abhängigkeiten zu erkennen. Zudem entspricht der Hardware-getriebene Ansatz nicht mehr den zukünftigen Anforderungen der Automobilhersteller, die zu einem Paradigmenwechsel durch Einführung eines funktionsgetriebenen Entwicklungsprozesses tendieren. Dies wird insbesondere durch die Standardisierungsbemühungen im Rahmen des AUTOSAR-Konsortiums deutlich [AUT10d, AUT10c].

Abbildung 5.2 zeigt ein Beispiel eines derartigen Hardware-nahen Funktionsnetzes. Die schwarze Umrandung soll dabei das Bussystem darstellen. Alle Verbindungen, die mit dieser schwarzen Umrandung verbunden sind, senden bzw. empfangen Signale über den Bus. Alle weiteren Verbindungen stellen die intern ablaufende Kommunikation dar. Aus der Abbildung können folgende Daten erhoben werden:

- 41 Funktionen
- 275 eingehende Signale
  - 206 Bussignale
  - 69 interne Signale
- 134 ausgehende Signale
  - 134 Bussignale

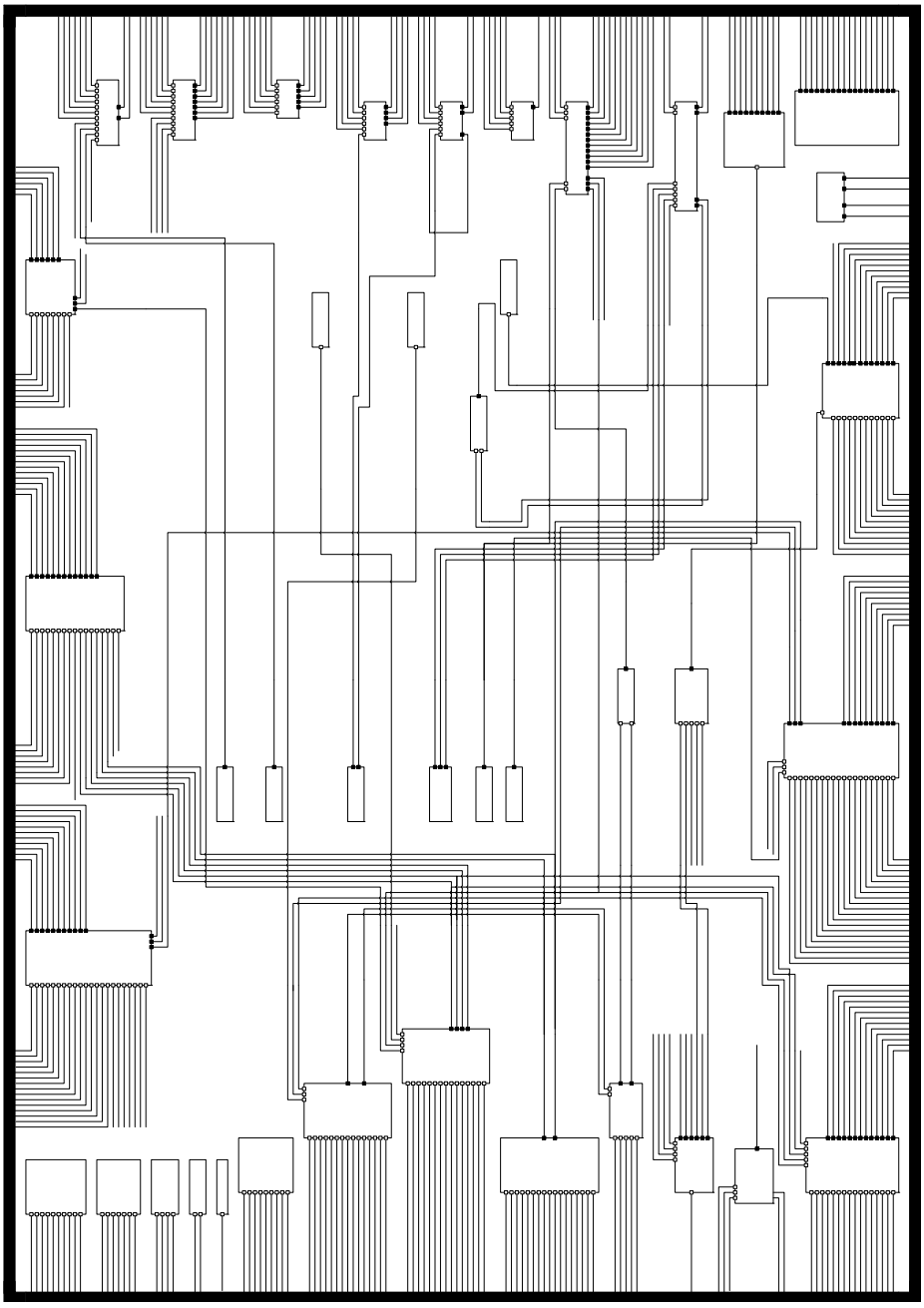


Abbildung 5.2.: Ein Funktionsnetz in Form einer visualisierten Kommunikationsmatrix

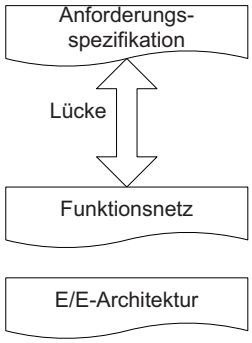


Abbildung 5.3.: Ein technisch nahes Funktionsnetz, das nur einen Teil der Lücke schließen kann

– 45 interne Signale

Auffallend dabei ist, dass der Großteil der Kommunikation über das Bussystem verläuft. Ein derartiges Funktionsnetz ist keine Seltenheit in der Automobilindustrie. Es gibt durchaus größere Funktionsnetze mit mehr Funktionen und Signalen. Dieses Beispiel hat eher Durchschnittsgröße.

Es stellt sich an dieser Stelle nun die wesentliche Frage, ob diese Art von Funktionsnetz alleine ausreicht, die Lücke zwischen der Anforderungsspezifikation und E/E-Architektur zu schließen. Die Frage kann hier verneint werden, da ein derartiges Funktionsnetz eher an die technische Umgebung angepasst ist, eine enorme Komplexität aufweist und jegliche Form einer logischen Betrachtungsweise auslässt. Es kann daher nicht als Verständnismodell für Entwickler dienen und auch keine Kommunikationsgrundlage zwischen Ingenieuren bilden. Vielmehr kann es im unteren Bereich der Lücke zwischen der Anforderungsspezifikation und der E/E-Architektur angesiedelt werden. Abbildung 5.3 illustriert diese Situation.

Eine andere Form von Funktionsnetzen, welche ebenfalls häufig Beachtung und Anwendung in der Automobilindustrie findet, basiert auf einer Hardware-unabhängigen *logischen Beschreibung*. Abbildung 5.4 stellt ein Beispiel dar. Funktionen sind durch Rechtecke visualisiert. Eine Kante zwischen zwei Funktionen zeigt, dass es eine Beziehung zwischen den Funktionen existiert. In welcher Form diese Beziehung manifestiert, wird allerdings nicht weiter spezifiziert.

Derartige logische Funktionsnetze werden oftmals in funktionsgetriebenen Ansätzen eingesetzt. Sie sind im Vergleich zum Hardware-getriebenen Ansatz einfacher zu verstehen. Zudem sind Abhängigkeiten leichter nachvollziehbar. So sind sie eher als Verständnismodell geeignet. Allerdings ist der Abstraktionsgrad sehr hoch, sodass keinerlei Realisierungsnähe vorhanden ist. Weiterhin sind konkrete Kommunikationen zwischen Funktionen nicht erkennbar. Schließlich können derartige Funktionsnetze nicht ausreichend genug überprüft werden. Auch diese Funktionsnetze schließen nur teilweise die Lücke zwischen Anforderungssepzifikation und E/E-Architektur. Sie sind im oberen Bereich der Lücke wiederzufinden. Abbildung 5.5

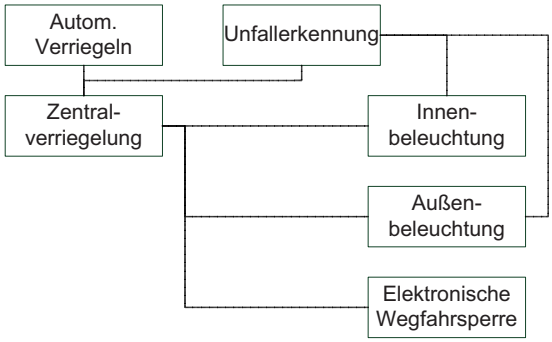


Abbildung 5.4.: Ein logisches Funktionsnetz, das vollständig von der Hardware abstrahiert

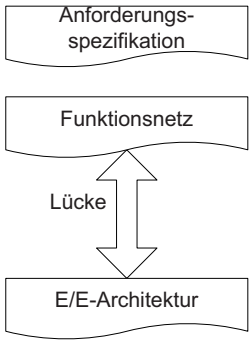


Abbildung 5.5.: Ein abstraktes Funktionsnetz, das ebenfalls nur einen Teil der Lücke schließen kann

veranschaulicht diese Situation.

Beide vorgestellten Ansätze haben also ihre spezifischen Vor- und Nachteile. Ihnen ist allerdings gemeinsam, dass weder der eine noch der andere Ansatz alleine ausreichen wird, um die Lücke zwischen der Anforderungsspezifikation und E/E-Architektur zu schließen.

Eine Lösung kann also nur durch *Integration beider Ansätze* in ein umfassendes Konzept erreicht werden. Dieses Konzept muss zudem um weitere *Abstraktionsebenen* erweitert werden, damit die Lücke vollständig geschlossen werden kann. Die zu abstrahierenden Merkmale müssen dabei für jede Ebene klar definiert werden. Diese können in Form von *Abstraktionsregeln* beschrieben werden. Jede Regel gibt dabei das zu abstrahierende Merkmal an und beschreibt entsprechende Handlungsempfehlungen. Auf diese Weise können die Regeln durch Werkzeugunterstützung ausgeführt werden. Abbildung 5.6 illustriert die beschriebene Lösungsidee. Die Lösungsansätze sind dabei aus den Arbeiten [MA09b, MPF09] und im Rahmen der Masterarbeit von *Jan Pojer* [Poj11] sowie der Diplomarbeit von *Antonio Navarro Perez* [Per09] entstanden. Sie werden in diesem Kapitel aufgegriffen und beschrieben.

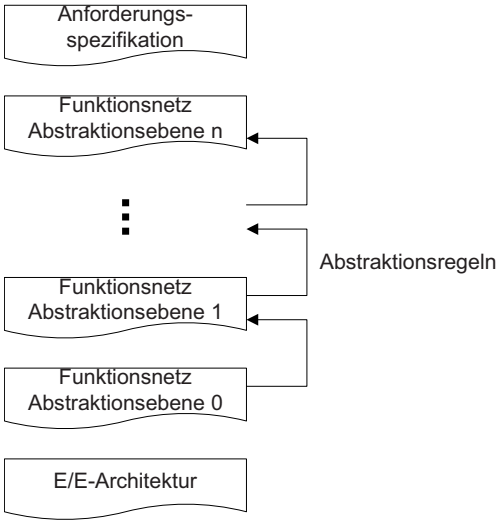


Abbildung 5.6.: Funktionsnetze auf mehreren Abstraktionsebenen zur Schließung der Lücke zwischen der Anforderungsspezifikation und E/E-Architektur

Damit Funktionsnetze über mehrere Abstraktionsebenen eingesetzt werden können, ist es von essenzieller Bedeutung ein *formales Basiskonzept* der Beschreibungssprache zu definieren, welches die Grundlage aller Abstraktionsebenen darstellt. Auf diese Weise wird der Übergang zwischen den verschiedenen Ebenen auf einer einheitlichen Weise gewährleistet. Als Formalismus kann hierzu die Metamodellierung verwendet werden. Im Metamodell werden somit sämtliche Beschreibungselemente definiert, die in allen Funktionsnetzen wiederzufinden sind. Beispielsweise sind dies Konzepte, wie etwa Funktionen, Schnittstellen und Verbindungen. Besonders wichtig ist auch in diesem Kontext die geeignete *Definition einer homogenen Notation*. Dies fördert insbesondere die Verständlichkeit von Funktionsnetzen in einem hohen Maß. *Antonio Navarro Perez* hat in seiner Diplomarbeit ein derartiges Metamodell entworfen [Per09]. Die Erweiterungen des Metamodells für die verschiedenen Abstraktionsebenen sind dabei in der Masterarbeit von *Jan Pojer* umgesetzt [Poj11].

Formal definierte Funktionsnetze in mehreren Abstraktionsebenen können den Entwicklungsprozess in der frühen Phase geeignet unterstützen. Ein weiterer Aspekt, der in diesem Zusammenhang ebenfalls Beachtung finden sollte, ist, die Möglichkeit wiederverwendbare bzw. generische Funktionen/Funktionsnetze zu ermitteln und zu verwalten. Funktionsnetze werden oftmals in Korrespondenz zu Hardwarekomponenten modelliert. So werden für vier Drehratensensoren an Reifen ebenfalls vier Sensorfunktionen modelliert. Die Gemeinsamkeiten dieser Funktionen werden an dieser Stelle typischerweise nicht erkannt, da es an Wiederverwendungsstrategien fehlt. Es existieren für Funktionsnetze beispielsweise keine Bibliotheken, in denen wiederverwendbare Bausteine abgelegt und instanziiert werden können.

Die Erfassung von wiederverwendbaren Funktionsnetzen in Form von Funktionen und Schnittstellenbeschreibungen in einem vorgezogenen Prozessschritt ist eine Möglichkeit die *systematische Wiederverwendung* zu fördern. Dies kann durch *Einführung eines Domänenmodells*, welches Funktionen klassifiziert und beschreibt, umgesetzt werden. Das Domänenmodell wird dann herangezogen, um Funktionen für die Funktionsnetzmodellierung zu instanziieren. In der Masterarbeit von *Jan Pojer* wurde diesbezüglich ein Konzept vorgeschlagen und realisiert [Poj11]. Die Ideen werden in diesem Kapitel näher erläutert.

Schließlich sei als letzter Punkt die Variabilität in Funktionsnetzen betrachtet. Variabilität wird dabei durch Modellierung des maximalen Funktionsnetzes erfasst (oftmals auch als 150%-Modell bezeichnet). Das bedeutet, dass alle Varianten in einem Funktionsnetz modelliert und spezifische Ausprägungen durch Wegstreichen nicht erforderlicher Anteile erzeugt werden. Variationspunkte werden dabei kaum formal erfasst. Es existieren keine Variabilitätsmechanismen zur Realisierung von Variationspunkten. Varianten werden daher auch nicht systematisch strukturiert.

Es besteht also der Bedarf nach einem *Variabilitätsmechanismus* für Funktionsnetze, mit dem Variationspunkte durch Kapselung der Varianten realisiert werden können. Zudem ist ein *Variabilitätsmodell* erforderlich mit dem Variationspunkte dokumentiert und repräsentiert werden können. Hiermit wird auch gewährleistet, dass Ausprägungen durch ein Konfigurationsmodell erstellt werden können. In den Arbeiten [Men08, Men09, MA09b, MA09a, MPF09, Men10] sowie in der Diplomarbeit von *Antonio Navarro Perez* [Per09], in der Masterarbeit von *Önder Babur* [nB10] als auch in der Bachelorarbeit von *Maxim Pogrebinski* [Pog10] wurden die erforderlichen Konzepte diskutiert, Lösungsansätze vorgestellt und durch Werkzeuge umgesetzt. In diesem Kapitel werden diese Arbeiten aufgegriffen und erläutert.

Aus den obigen Beschreibungen sind drei Kernbereiche identifiziert, die für die Anwendung von Funktionsnetzen besonders wichtig sind. Diese sind (1) die *Funktionsnetzmodellierung*, (2) die *Domänenmodellierung* und (3) die *Variabilitätsmodellierung*. Hierbei wurden jeweils zwei wesentliche Anforderungen ermittelt. Für die Funktionsnetzmodellierung hat sich die formale Definition eines Basiskonzepts in Form eines Metamodells als auch die einheitliche Notation als wichtige Anforderungen herauskristallisiert. Bei der Domänenmodellierung sind die Ermittlung der erforderlichen Abstraktionsebenen und den zugehörigen Abstraktionsregeln zwei wichtige Aspekte, um die Verständlichkeit, die Übersichtlichkeit und insbesondere die Wiederverwendung von Funktionsnetzen zu unterstützen. Schließlich sind die Einführung von Variabilitätsmechanismen und der Variabilitätsmodellierung zwei erforderliche Anforderungen, um die Variabilität zu beherrschen.

Das Zusammenspiel dieser drei Bereiche wird in Abbildung 5.7 dargestellt. Dabei ist die Domänenmodellierung die erste Aktivität vor der eigentlichen Funktionsnetzmodellierung. Diese Phase dient zur Klassifikation der Domäne hinsichtlich wiederverwendbarer Eigenschaften. So können Funktionen und ihre Schnittstellen beschrieben werden, um sie später für die Funktionsnetzmodellierung zu verwenden. Die Phase muss dabei keineswegs vollständig abgeschlossen werden. Sie kann auch inkrementell entworfen werden. Insbesondere können Rückgriffe aus der Funkti-

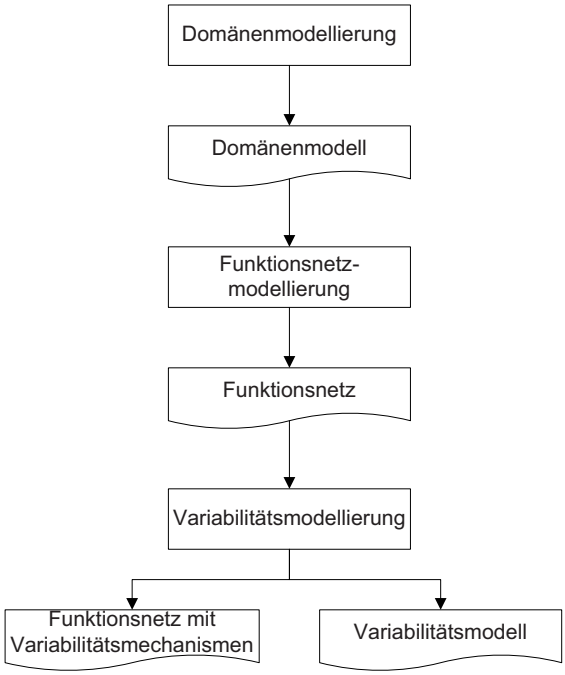


Abbildung 5.7.: Der Prozess auf Funktionsebene

onsnetzmodellierung ohne Weiteres durchgeführt werden. Zur Funktionsnetzmodellierung werden dann die Elemente aus dem Domänenmodell herangezogen und instanziiert. Dabei werden sowohl Top-Down als auch Bottom-Up jegliche Abstraktionsebenen durchlaufen. So wird ein integrierter Ansatz geschaffen, der die Lücke zwischen der Anforderungsspezifikation und E/E-Architektur vollständig schließt. Hierfür werden stets definierte Abstraktionsregeln angewendet. Schließlich können Funktionsnetze zur Handhabung aller Varianten durch entsprechende Variabilitätsmechanismen und einem Variabilitätsmodell angereichert werden. Auf diese Weise wird somit auch die Variantenproblematik beherrscht.

Im Folgenden werden die beschriebenen Aktivitäten genauer erläutert. Dabei wird zunächst die Funktionsnetzmodellierung behandelt (vgl. Abschnitt 5.2), da hier das Metamodell zur Beschreibung von Funktionsnetzen als auch die Notation beschrieben wird. Es dient als Basis für die Erläuterungen der anderen Aktivitäten. Weiterhin werden in Abschnitt 5.3 die Domänenmodellierung detailliert erklärt. Hier werden insbesondere die ermittelten Abstraktionsebenen als auch die hierfür erforderlichen Abstraktionsregeln vorgestellt. Eine entsprechende Erweiterung des Metamodells ist ebenfalls Bestandteil dieses Abschnitts. Schließlich wird in Abschnitt 5.4 das Konzept zur Realisierung von Variationspunkten beschrieben und der Zusammenhang zum Variabilitätsmodell hergestellt.

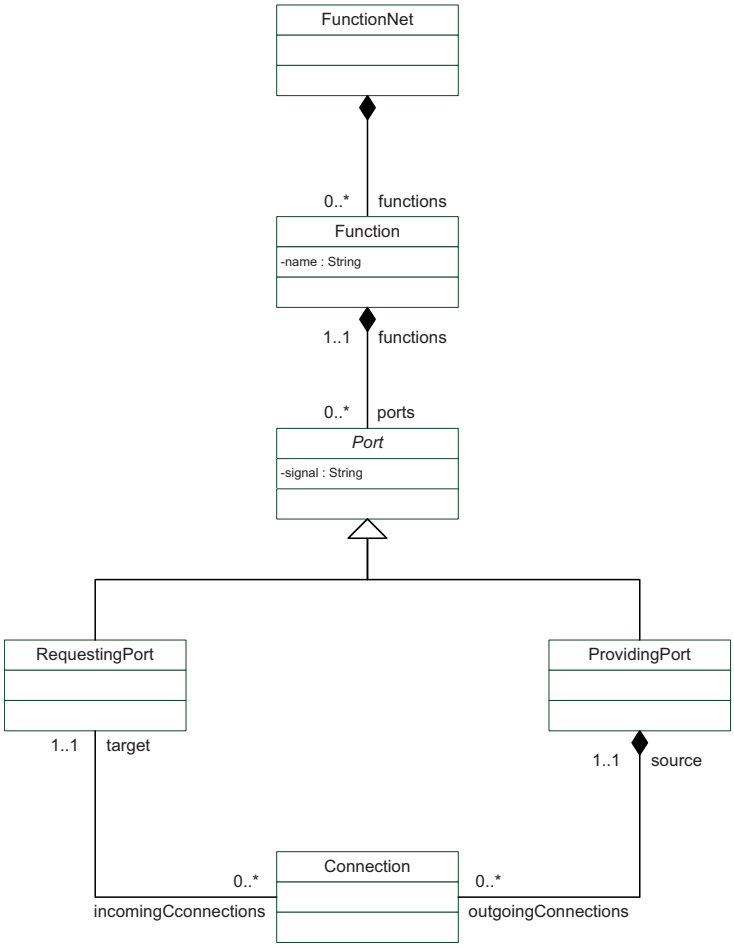


Abbildung 5.8.: Ein Metamodell für logische Funktionsnetze

5.2. Funktionsnetzmodellierung

In diesem Abschnitt wird das zugrunde liegende Metamodell für Funktionsnetze vorgestellt. Außerdem wird eine grafische Notation zur Modellierung von Funktionsnetzen vorgeschlagen. Es stellt somit die Basis für alle weiteren Konzepte in diesem Kapitel dar. Der Abschnitt basiert auf den Arbeiten [MPF09] und [Per09].

5.2.1. Metamodell

Abbildung 5.8 illustriert das grundlegende Metamodell zur Beschreibung von logischen Funktionsnetzen. Dieses Metamodell wird noch im Verlauf dieses Kapitels an vielen Stellen erweitert. Ein wesentliches Element ist die Klasse `FunctionNet`. In

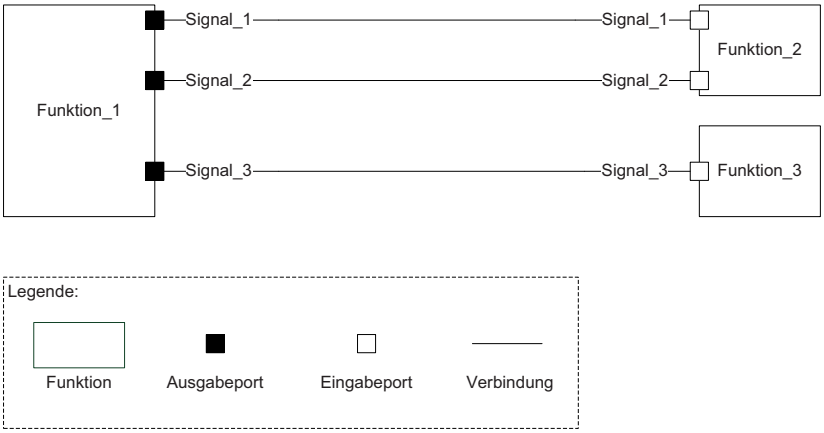


Abbildung 5.9.: Eine grafische Notation für logische Funktionsnetze

diesem Kontext bezeichnet ein logisches Funktionsnetz eine statische Beschreibung von Funktionen und Interaktionen untereinander, die als Brücke zwischen der Featureebene und Architekturebene dient. Funktionsnetze sind insofern *logisch*, da sie von Deploymententscheidungen, Softwarestrukturen, Datentypen sowie technischen Schnittstellen abstrahieren. Die zugrunde liegende Hardwareplattform wird also nicht berücksichtigt. Darüber hinaus sind Funktionsnetze *statisch*, da sie von Echtzeitverhalten und zeitlichen Abläufen abstrahieren. Auf diese Weise sind lediglich Datenflüsse nachvollziehbar, aber keinerlei zeitliche Aspekte, welche die Dynamik des Systems ausmachen.

Ein Funktionsnetz besteht aus einer Menge von Funktionen (in der Abbildung durch die Klasse `Function` beschrieben). Sie repräsentiert eine abgeschlossene Einheit mit funktionalem Charakter. Ein wichtiges Merkmal einer Funktion ist sein ausdrucksstarker Name. Jede Funktion besitzt darüber hinaus eine Schnittstelle zur Kommunikation mit weiteren Funktionen.

Die Schnittstelle einer Funktion wird durch Ports beschrieben (Klasse `Port` aus der Abbildung). Ports kommunizieren Signale. Dabei gibt es zwei Arten von Ports: (1) Eingabeports (`RequestingPort`) und (2) Ausgabeports (`ProvidingPort`). Erstere fordern ein bestimmtes Signal an. Letztere hingegen stellen bestimmte Signale bereit.

Kommunikationen zwischen Ports werden durch Verbindungen kenntlich gemacht (die Klasse `Connection` aus der Abbildung). Eine Verbindung vernetzt einen Eingabeport mit einem Ausgabeport. Dabei können nur Ports mit dem gleichen Signal verbunden werden.

5.2.2. Grafische Notation

Abbildung 5.9 zeigt anhand eines abstrakten Beispiels die in dieser Arbeit vorgeschlagene grafische Notation. Funktionen werden durch Rechtecke dargestellt. Aus-

gabeports werden durch kleine schwarz gefüllte Quadrate visualisiert. Ausgabeports werden an die rechte oder untere Seite der Funktion angehangen. Eingabeports sind hingegen weiß gefüllte Quadrate, die an die linke oder obere Seite der Funktion angefügt werden. Schließlich werden Verbindungen durch eine durchgezogene schwarze Linie dargestellt.

5.3. Domänenmodellierung

Die Domänenmodellierung ist die Aktivität zur Klassifizierung der Domäne, um wiederverwendbare Bausteine geeignet zu identifizieren und zu verwalten [RRE91]. Dabei werden Funktionen, ihre Schnittstellen und Verbindungen erfasst. Außerdem ist das Domänenmodell wichtiger Bestandteil bei der Einführung der verschiedenen Abstraktionsebenen. Es ist die zentrale Verwaltungsstelle, um die Korrespondenzen zwischen allen Elementen im Funktionsnetz über die verschiedenen Abstraktionsebenen hinweg zu steuern. Die Identifikation geeigneter Abstraktionsebenen werden durch Analyse des Hardware-getriebenen Funktionsnetzes, also der Kommunikationsmatrix, begonnen und schrittweise generalisiert. Die Generalisierung wird in Abstraktionsregeln beschrieben. Im Folgenden werden, ausgehend von der Beschreibung der Abstraktionsregeln (vgl. Abschnitt 5.3.1), die identifizierten Abstraktionsebenen beschrieben sowie die erforderlichen Erweiterungen am Metamodell dargestellt (vgl. Abschnitt 5.3.2). Die Erläuterungen stammen hauptsächlich aus der Masterarbeit von *Jan Pojer* [Poj11].

5.3.1. Abstraktionsregeln

Abstraktionsregeln sind in drei Bereiche unterteilt. Der Ausgangspunkt ist die Kommunikationsmatrix. Diese wird zunächst durch zwei Abstraktionsregeln in ein logisches Funktionsnetz überführt. Die restlichen Regeln erfolgen auf dieser logischen Sicht. Dabei werden zwischen Regeln für Ports und Verbindungen (sieben Regeln) und Regeln für Funktionen unterschieden (fünf Regeln).

5.3.1.1. Ausgangspunkt

Abstraktionsregel 1: Deployment Eine Kommunikationsmatrix ist die unterste Ebene der Beschreibungsstufen für Funktionsnetze, also die Abstraktionsebene 0. Sie ist also vom Detailgrad her sehr nah an der Hardwareplattform und wird daher auch als technisches Funktionsnetz bezeichnet. Eine Kommunikationsmatrix hat die Eigenschaft, dass Deploymententscheidungen bereits durch die Matrix gegeben sind. Jede Kommunikationsmatrix beinhaltet also Funktionen und Signale, die mit weiteren Funktionen innerhalb bzw. außerhalb eines bestimmten Steuergeräts kommunizieren. Demnach gibt es für jedes Steuergerät eine Kommunikationsmatrix.

Ziel dieser Abstraktionsregel ist es, von diesen Deploymententscheidungen zu abstrahieren. Eine entsprechende Regel wird wie folgt formuliert:

> Eliminiere aus allen Kommunikationsmatrizen alle Steuergeräte-spezifischen Beschreibungselemente. Erzeuge stets direkte Verbindungen zwischen Quell- und Zielfunktionen.

Auf diese Weise wird die Kommunikation zwischen Funktionen in ein logisches Funktionsnetz überführt. Sie wird in diesem Zusammenhang in die Abstraktionsebene 1 eingestuft. Abbildung 5.10 illustriert ein Beispiel. Die untere Hälfte der Abbildung zeigt zwei kommunizierende Funktionen, die auf verschiedenen Steuergeräten verteilt sind (Abstraktionsebene 0). Es handelt sich dabei um die Funktionalität der elektronischen Wegfahrsperre, die bei Betätigung der Verriegelung/Entriegelung des Fahrzeugs aktiviert wird. Die auslösende Funktion `EWS_Master` ist im CAS deployed. Die zugehörige Funktion `EWS_Motor` ist wiederum im DME integriert. Sie ist die Funktion, die nach erfolgreicher Authentifikation, die Wegfahrsperre für den Motor aktiviert bzw. deaktiviert. Der Abstraktionsschritt ist in der oberen Hälfte der Abbildung zu sehen (Abstraktionsebene 1). Die Verteilung der Funktionen auf verschiedene Steuergeräte ist hierbei nicht mehr zu sehen. Stattdessen findet die Kommunikation zwischen beiden Funktionen auf der gleichen logischen Ebene statt.

Abstraktionsregel 2: Signalträger Die Kommunikationsmatrix (Abstraktionsebene 0) beinhaltet neben der Deploymententscheidung auch die Spezifikation der Signalträger aller empfangenen bzw. gesendeten Signale. So kann ein Signal zum Beispiel über Bussysteme, wie etwa CAN, LIN oder MOST, direkte Verkabelungen oder Steuergeräte-intern versendet werden.

Durch diese Abstraktionsregel werden die unterschiedlichen Signalträger verallgemeinert, sodass nur ein logischer Signalträger existiert. Eine entsprechende Regel wird wie folgt formuliert:

> Eliminiere aus den in einer Kommunikationsmatrix enthaltenen Signaleigenschaften das Attribut, das den Signalträger spezifiziert.

Das Ergebnis dieser Regel ist ein weiterer Beitrag, die Kommunikationsmatrix in ein logisches Funktionsnetz zu überführen (Abstraktionsebene 1). Abbildung 5.10 veranschaulicht diese Regel anhand eines Beispiels. Während in der unteren Hälfte der Abbildung (Abstraktionsebene 0) die Signale zwischen den beiden Funktionen der elektronischen Wegfahrsperre über das CAN-Bussystem verlaufen, zeigt der Abstraktionsschritt in der oberen Hälfte (Abstraktionsebene 1) ausschließlich logische Signalträger, die direkt mit den Funktionen verbunden werden.

5.3.1.2. Ports und Verbindungen

Abstraktionsregel 3: Sensorik-/Aktoriksignale Das bisher erstellte logische Funktionsnetz (Abstraktionsebene 1) beinhaltet noch viele Stellen, die weiter behandelt werden müssen, um weitere Abstraktionsschritte zu ermöglichen. Eine dieser Stellen wird nachfolgend erläutert.

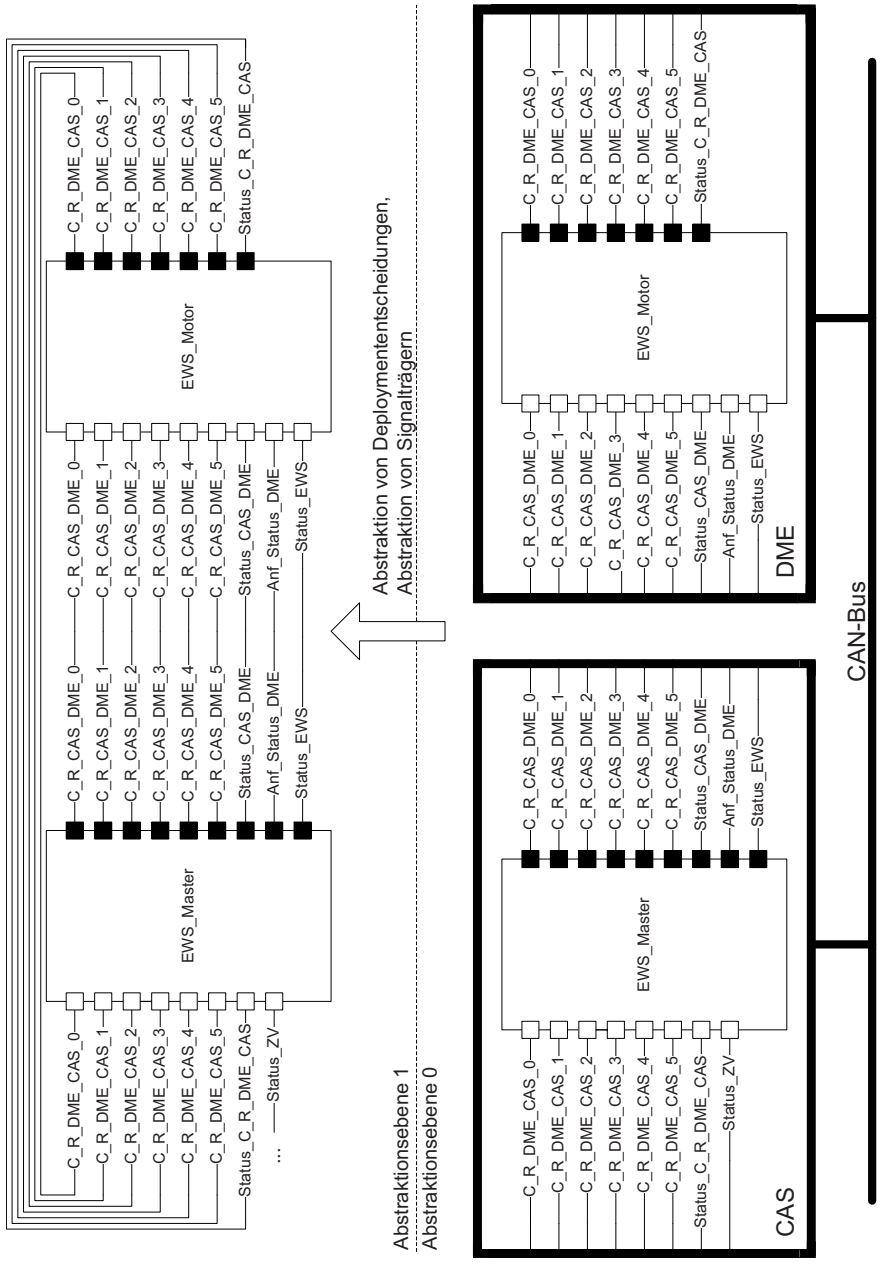


Abbildung 5.10.: Anwendung der Abstraktionsregeln 1 und 2

Das logische Funktionsnetz enthält sehr viele semantisch ähnliche Signale, die aufgrund der mehrfach installierten Sensorik bzw. Aktorik auch mehrfach empfangen bzw. gesendet werden.

Der Zweck dieser Abstraktionsregel ist es genau von derartigen Sensorik- / Aktoriksignalen zu abstrahieren, sodass die Multiplizität dieser Signale reduziert werden kann. Eine entsprechende Regel wird wie folgt formuliert:

> Generalisiere alle semantisch ähnlichen Sensorik-/Aktoriksignale zu einem gemeinsamen Signal. Füge dem Signal eine Parametrisierung hinzu, die es erlaubt, zwischen den verschiedenen Sensorik-/Aktoriksignalen zu unterscheiden.

Auf diese Weise wird die bisherige Betrachtung der logischen Funktionsnetze in eine weitere Form überführt (Abstraktionsebene 2). Im Gegensatz zur bisherigen visuellen Darstellung werden die Ports in dieser Ebene in Form eines Diamanten dargestellt.

In Abbildung 5.11 wird zu dieser Regel ein Beispiel dargestellt. In der Abstraktionsebene 1 ist die Masterfunktion `ZV_Master` der Zentralverriegelung enthalten. Sie steuert die Aktuatoren an den Türen des Fahrzeugs. Dementsprechend werden vier Signale gesendet. Durch den beschriebenen Abstraktionsschritt können die vier semantisch ähnlichen Signale zusammengefasst werden und über einen Parameter zugeordnet werden. Das Resultat ist in der Abbildung auf der Abstraktionsebene 2 zu sehen.

Der aufmerksame Leser wird an dieser Stelle wohl bemerkt haben, dass es eine weitere Abstraktionsmöglichkeit existiert. Die vier Funktionen `Stellmotor_ZV_FT`, `Stellmotor_ZV_BFT`, `Stellmotor_ZV_FTH` und `Stellmotor_ZV_BFTH`, welche die Aktuatoren abbilden, könnten durchaus zusammengefasst werden. Da derartige Abstraktionen primär Funktionen betreffen, werden sie gesondert in Abschnitt 5.3.1.3 behandelt.

Abstraktionsregel 4: Zweiwertige Signale Logische Funktionsnetze der Abstraktionsebene 1 bieten noch weiteres Potenzial für Abstraktionsmöglichkeiten. Eine dieser Möglichkeiten wird im Folgenden erläutert.

In den Funktionsnetzen treten oftmals Signale paarweise auf, wie zum Beispiel Signale zum Öffnen/Schließen, Verriegeln/Entriegeln oder Aktivieren/Deaktivieren. Diese werden immer über separate Ports modelliert.

Die Absicht dieser Abstraktionsregel ist die Zusammenfassung aller separat modellierten Signale, die einen Boolschen Charakter besitzen. Eine entsprechende Regel wird wie folgt formuliert:

> Fasse alle Signale mit Boolschem Charakter zusammen, die über getrennte Ports empfangen bzw. gesendet werden. Benenne das neu erzeugte Signal so um, dass aus dem Namen die Bedeutung und Zweiwertigkeit des Signals erkennbar wird.

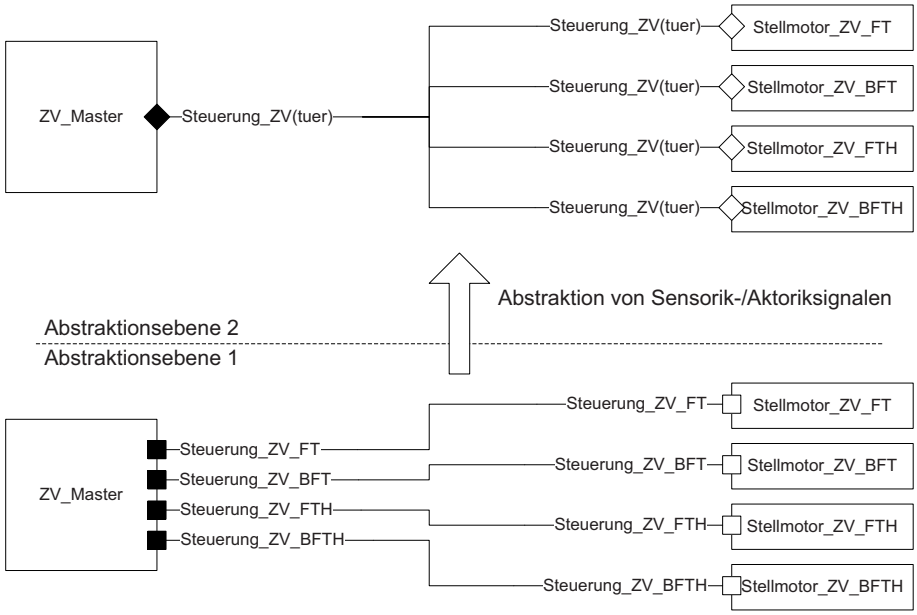


Abbildung 5.11.: Anwendung der Abstraktionsregel 3

Mit dieser Regel wird ein weiterer Beitrag geleistet, die Komplexität des logischen Funktionsnetzes auf Abstraktionsebene 2 zu reduzieren. Abbildung 5.12 zeigt hierzu ein Beispiel. Die Funktion `ZV_Aussenbedienung` bearbeitet sämtliche Signale, die außerhalb des Fahrzeugs empfangen werden, also über die Außenantenne oder Türschlösser. Einige dieser Signale werden dann an die Masterfunktion `ZV_Master` gesendet. Unter anderem sind diese die Anforderung zum Öffnen (`Anf_Oeffnen`) bzw. Schließen (`Anf_Schliessen`) der Fahrzeugtüren. Da diese beiden Signale den oben beschriebenen Boolschen Charakter besitzen, können sie über einen Abstraktionsschritt zusammengefasst werden. In der Abstraktionsebene 2 wird das Ergebnis dargestellt. Aus dem neuen Namen des Signals ist sowohl die Bedeutung als auch die Zweiwertigkeit erkennbar.

Abstraktionsregel 5: Protokollabläufe Die letzte Abstraktionsregel für logische Funktionsnetze auf Abstraktionsebene 1 kann erneut die Komplexität der Ports bzw. Signale stark minimieren. Sie wird im Folgenden erläutert.

Viele Signale, die eine Funktion im Funktionsnetz empfängt bzw. sendet, sind Teil eines bestimmten Protokollablaufs.

Das Ziel dieser Abstraktionsregel ist genau von derartig detaillierten Protokollabläufen zu abstrahieren. Eine entsprechende Regel wird wie folgt formuliert:

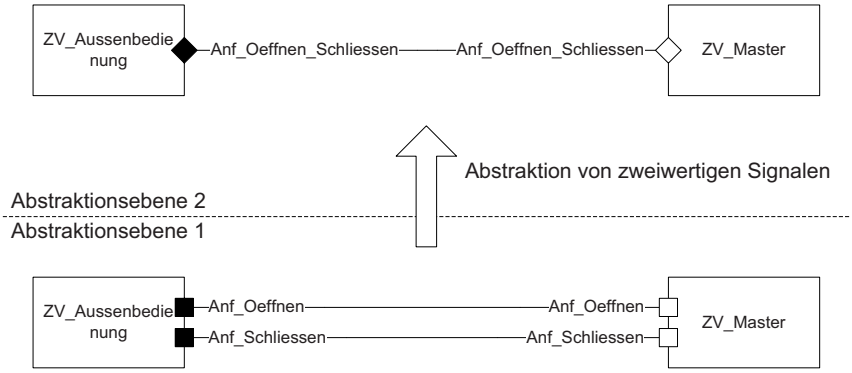


Abbildung 5.12.: Anwendung der Abstraktionsregel 4

> Gruppiere alle Signale, die Teil eines Protokollablaufs sind, zu einem gemeinsamen Signal zusammen. Benenne das Signal so um, dass der Zweck des neu entworfenen Signals verständlich bleibt. Parametrisiere das Signal so, dass verschiedene Protokollschritte nachvollziehbar bleiben.

Durch diese Regel wird das logische Funktionsnetz der Abstraktionsebene 1 mit beachtlicher Komplexitätsreduzierung in ein logisches Funktionsnetz der Abstraktionsebene 2 überführt. Abbildung 5.13 illustriert dies an einem Beispiel. Hier sind in der Abstraktionsebene 1 zwei Funktionen der elektronischen Wegfahrsperre dargestellt: (1) die Masterfunktion `EWS_Master` steuert die Initiierung der Wegfahrsperre und (2) die Funktion zur Steuerung der Wegfahrsperre des Motors (`EWS_Motor`). Zwischen diesen beiden Funktionen wird ein Challenge-Response-Verfahren zur Authentifikation eingeleitet. Das Verfahren erfordert einen sechsfachen Signalaustausch, bis die Funktion authentifiziert ist und der Motor aktiviert/deaktiviert werden kann. In der Abbildung sind zur Vereinfachung nur die ersten vier Signale dargestellt und zudem auch nur in einer Richtung, von `EWS_Master` zu `EWS_Motor`. Eine Abstraktion dieses Challange-Response-Verfahrens ist in der Abstraktionsebene 2 dargestellt. Hier sind die Signale gruppiert und können über den Parameter `schritt` klassifiziert werden.

Abstraktionsregel 6: Kommunikationsparadigmen Nachdem nun drei Regeln zur Überführung von logischen Funktionsnetzen der Abstraktionsebene 1 in die Abstraktionsebene 2 vorgestellt wurden, wird nun die Abstraktionsebene 2 als Basis genommen.

In einem Funktionsnetz sind typischerweise zwei Kommunikationsparadigmen enthalten: (1) Sender-Receiver und (2) Client-Server. Letzteres wird in der Regel als eine bidirektionale Sender-Receiver-Kommunikation realisiert. Dies führt allerdings oftmals zu einer Überflutung an Interaktionen.

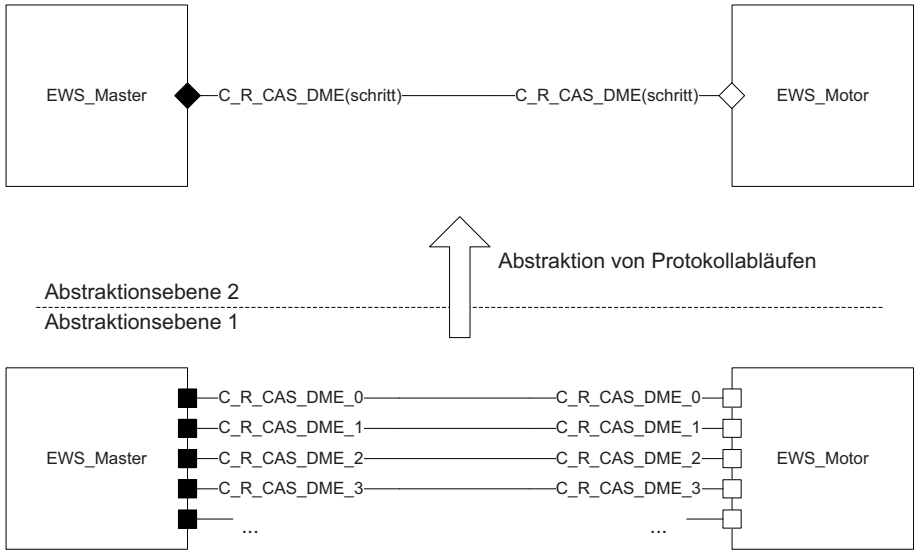


Abbildung 5.13.: Anwendung der Abstraktionsregel 5

Durch diese Regel soll daher vom zugrunde liegendem Kommunikationsparadigma abstrahiert werden. Eine entsprechende Regel wird wie folgt formuliert:

> Jedes Sender-Receiver-Kommunikationspaar, das eine Client-Server-Kommunikation realisiert, wird in eine einfache Sender-Receiver-Kommunikation überführt. Mögliche Parameter an den Signalen werden eliminiert. Die eigentlichen Sender-Receiver-Signale bleiben wie üblich erhalten.

Hiermit werden primär alle Client-Server-Kommunikationen, die über bidirektionale Sender-Receiver-Verbindungen realisiert wurden, reduziert. Das logische Funktionsnetz wird also in die Abstraktionsebene 3 überführt. Auf dieser Ebene werden alle Ports dreieckig dargestellt.

Abbildung 5.14 zeigt diesbezüglich ein Beispiel. Dargestellt sind die bekannten Funktionen der elektronischen Wegfahrsperre `EWS_Master` und `EWS_Motor`. Das bereits erwähnte Challenge-Response-Verfahren zur Authentifikation der Masterfunktion ist eine Client-Server-Kommunkation. Zu jedem Challenge wird ein Response gesendet. Die beiden dargestellten Sender-Receiver-Signale `C_R_CAS_DME(schritt)` realisieren dabei den Client-Server-Mechanismus. Darüber hinaus ist auch eine einfache Sender-Receiver-Kommunikation dargestellt. Hierbei sendet `EWS_Master` ein Statussignal `Status_C_R_CAS_DME` an die Funktion `EWS_Motor`. Die bidirektionale Kommunikation kann nun durch den beschriebenen Abstraktionsschritt in einen einfachen Signalaustausch zusammengefasst werden. In der Abbildung werden also die beiden Signale `C_R_CAS_DME(schritt)` zu einem Signal `C_R_CAS_DME` verschmolzen.

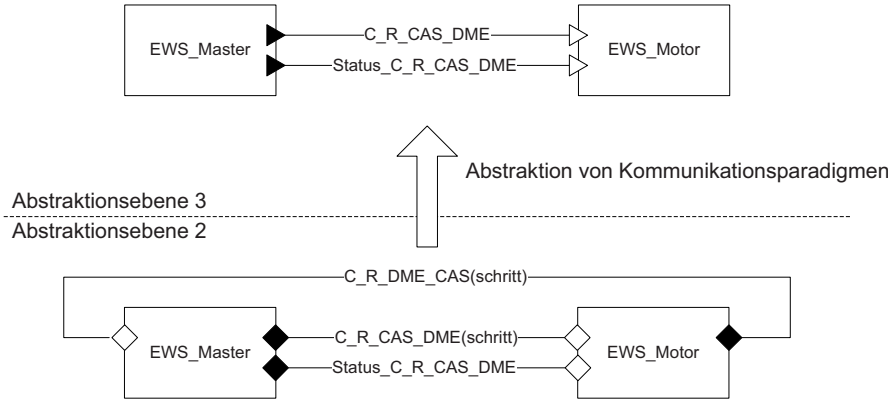


Abbildung 5.14.: Anwendung der Abstraktionsregel 6

Abstraktionsregel 7: Funktional zusammenhängende Signale Ausgangspunkt für diese Regel ist das logische Funktionsnetz der Abstraktionsebene 3. Auch in dieser Ebene herrscht immer noch Potenzial zur Abstraktion, welches im Folgenden erläutert wird.

Oftmals werden verschiedene Signale, wie etwa Status- oder Steuerungsinformationen, zur Erfüllung genau einer bestimmten Funktionalität gesendet. Der Zweck dieser Abstraktionsregel ist, diese funktional zusammenhängeden Signale durch Gruppierung zu abstrahieren. Eine entsprechende Regel wird wie folgt formuliert:

> Fasse alle Signale zusammen, die funktional zusammenhängen. Benenne das neu erzeugte Signal geeignet um.

Aus dieser Regel heraus entsteht für logische Funktionsnetze die neue Abstraktionsebene 4. Die Ports werden in dieser neuen Ebene sechseckig dargestellt. Abbildung 5.15 veranschaulicht diese Regel anhand eines Beispiels. In der Abstraktionsebene 3 sind zwei Funktionen zu sehen, die für die Steuerung der Fensterheber dienen. Die Masterfunktion `FH_Master` sendet dabei drei verschiedene Signale an die Fensterheberaktuatorik `FH_Ansteuerung`: (1) `Steuerung_FH`, (2) `Status_FH` und (3) `Authentisierung_FH`. Da diese drei Signale für die Ansteuerung der Fensterheber existieren, können sie durch die beschriebene Abstraktionsregel zusammengefasst werden. In der Abstraktionsebene 4 ist das Resultat dieser Zusammenfassung dargestellt. Es existiert jetzt nur noch ein Signal mit dem Namen `Ansteuerungsdaten_FH`.

Abstraktionsregel 8: Funktional verschiedene Signale Auch das logische Funktionsnetz der Abstraktionsebene 4 besitzt Möglichkeiten zur Abstraktion. Diese wird nachfolgend beschrieben.

Oftmals werden ausgehend von einer Funktion verschiedene Signale an unterschiedliche Funktionen gesendet. Durch diese Abstraktionsregel werden diese ver-

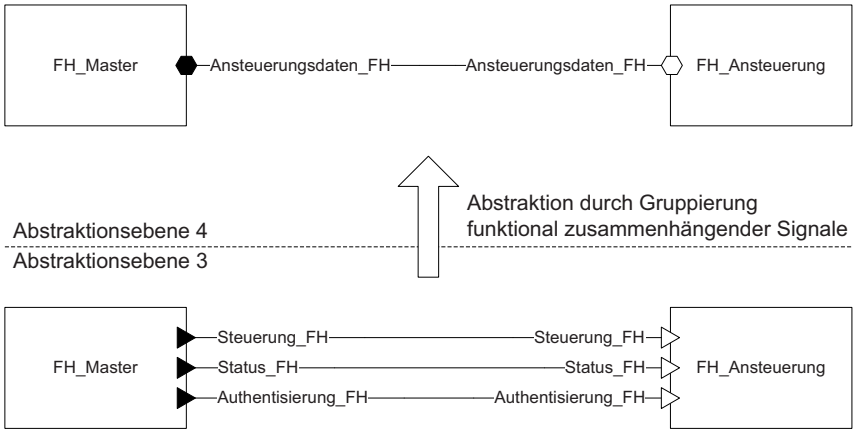


Abbildung 5.15.: Anwendung der Abstraktionsregel 7

schiedenen funktional-unabhängigen Signale durch Gruppierung abstrahiert. Eine entsprechende Regel wird wie folgt formuliert:

> Gruppiere alle gesendeten Signale einer Funktion, die funktional verschieden sind. Benenne das neu erzeugte Signal geeignet um.

Auf diese Weise entsteht ein logisches Funktionsnetz in der Abstraktionsebene 5. Die Ports in dieser Ebene werden kreisförmig dargestellt. Abbildung 5.16 zeigt ein Beispiel. Hier sind drei kommunizierende Funktionen dargestellt: (1) `Funkfernbedienung`, (2) `ZV_Authetifikation` und (3) `ZV_Aussenbedienung`. Die Funktion `Funkfernbedienung` sendet jeweils ein Signal an die zwei anderen Funktionen. Diese sind `Zugangsdaten_ZV` und `Bedienungsdaten_ZV`. Diese Signale können nun durch die oben beschriebene Abstraktionsregel gruppiert werden. In der Abstraktionsebene 5 wird das Ergebnis dieser Abstraktion dargestellt. Die beiden Signale wurden zu einem Signal `Funkschluesseldaten_ZV` zusammengefasst.

Abstraktionsregel 9: Signalflussrichtungen Das logische Funktionsnetz der Abstraktionsebene 5 bietet ebenfalls die Möglichkeit, einige Merkmale zu abstrahieren. Im Folgenden wird dies erläutert. Alle Funktionen besitzen Ein- und Ausgabeports (weiß und schwarz gefüllt). Durch die Existenz dieser Ports ist stets die Richtung aller Signale bekannt. Das Ziel dieser Abstraktionsregel ist genau von dieser Erkenntnis der Signalflussrichtung zu abstrahieren. Eine entsprechende Regel wird wie folgt formuliert:

> Eliminiere alle Ports aus den Funktionen. Eliminiere alle Signalnamen. Fasse alle Signale zwischen zwei Funktionen zu einem Signal zusammen.

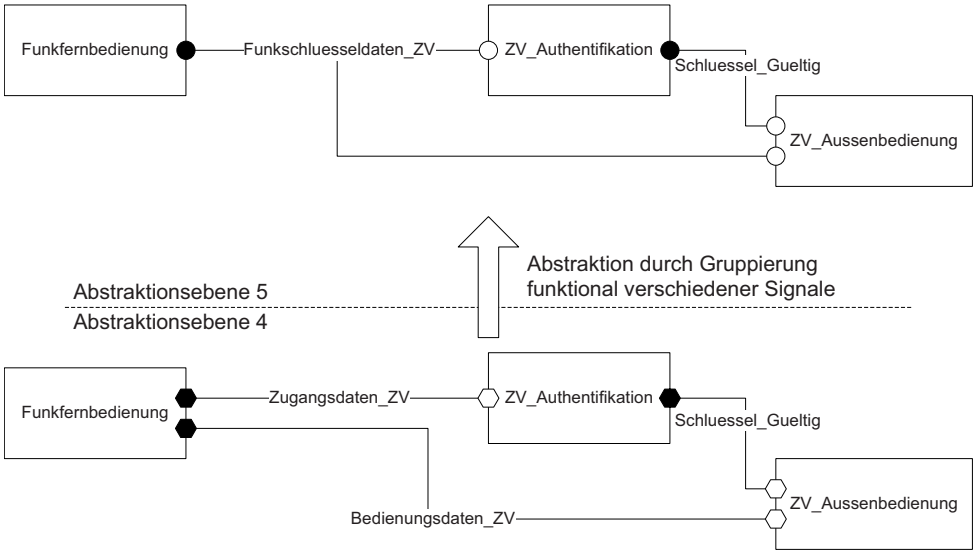


Abbildung 5.16.: Anwendung der Abstraktionsregel 8

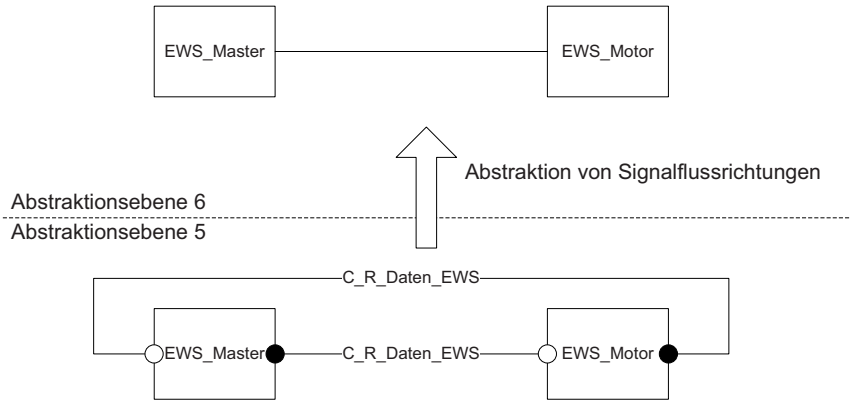


Abbildung 5.17.: Anwendung der Abstraktionsregel 9

Durch diese Regel entsteht nun die letzte Abstraktionsebene des logischen Funktionsnetzes, die Abstraktionsebene 6. In Abbildung 5.17 ist hierzu ein Beispiel dargestellt. In der Abstraktionsebene 5 sind die beiden bekannten Funktionen `EWS_Master` und `EWS_Motor` visualisiert. Sie senden sich jeweils ein Signal zu, `C_R_Daten_EWS`. Durch diese Abstraktionsregel können nun beide Signale zu einem Signal zusammengefasst werden, indem die Signalrichtung abstrahiert wird. Die Abstraktionsebene 6 zeigt das Ergebnis nach Anwendung der Regel. Es besteht nun nur noch eine Verbindung zwischen beiden Funktionen.

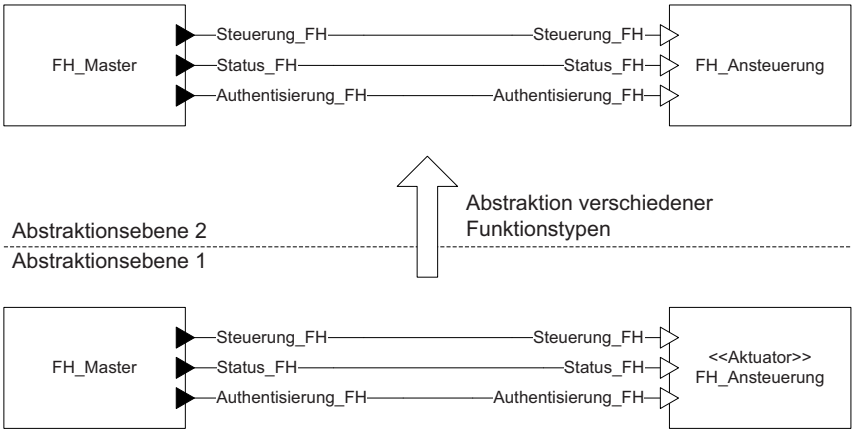


Abbildung 5.18.: Anwendung der Abstraktionsregel 10

5.3.1.3. Funktionen

Abstraktionsregel 10: Funktionstypen Das betrachtete Funktionsnetz ist auf Abstraktionsebene 1 angesiedelt. Auf dieser Ebene sind Funktionen durch verschiedene Typen definiert. So gibt es Funktionen vom Typ Sollwertgeber, Sensor oder Aktuator. Der Zweck dieser Abstraktionsregel ist es, von diesen Typen zu abstrahieren. Eine entsprechende Regel wird wie folgt formuliert:

Eliminiere alle Funktionstypdefinitionen aus den Funktionsbeschreibungen.

Bei Anwendung dieser Regel entsteht ein logisches Funktionsnetz der Abstraktionsebene 2. Abbildung 5.18 veranschaulicht hierzu ein Beispiel. In der Abstraktionsebene 1 sind zwei Funktionen der Fensterhebersteuerung zu sehen. Die Funktion `FH_Ansteuerung` stellt den Aktuator dar, um die Fenster zu öffnen oder zu schließen. Erkennbar ist er neben dem ausdrucksstarken Namen aufgrund der Markierung «`Aktuator`». Derartige Markierungen werden durch diese Abstraktionsregel nun eliminiert. Auf Abstraktionsebene 2 sind daher alle Funktionen vom gleichen Typ.

Abstraktionsregel 11: Funktionsmultiplizität Das logische Funktionsnetz der Abstraktionsebene 2 ist nun von verschiedenen Funktionstypen abstrahiert. Dies ist ein weiterer Schritt von der Hardwareplattform zu abstrahieren. Es gibt allerdings immer noch Harwdware-abhängige Funktionen, die ebenfalls abstrahiert werden können. Im Funktionsnetz gibt es typischerweise sehr viele Funktionen, die aufgrund von Sensorik oder Aktorik mehrfach modelliert sind. Durch diese Abstraktionsregel soll von dieser Hardwaremultiplizität abstrahiert werden. Eine entsprechende Regel wird wie folgt formuliert:

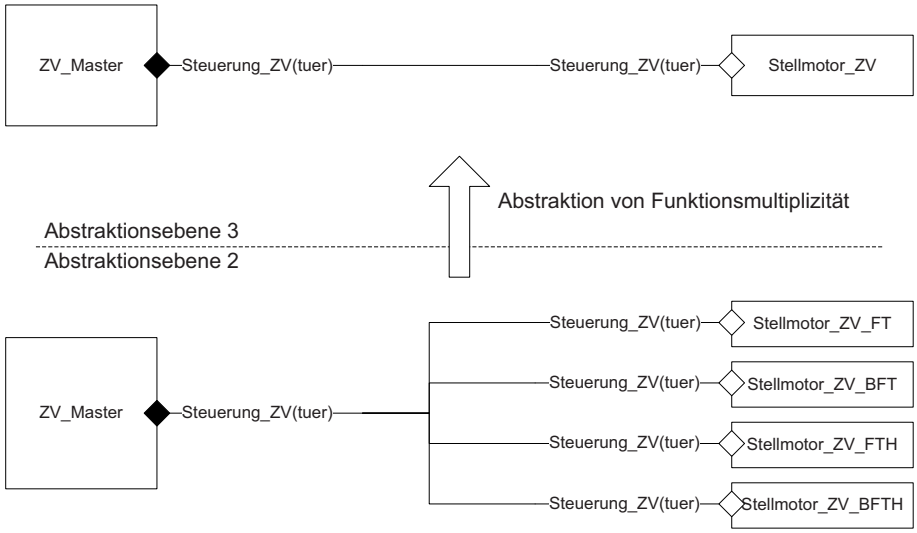


Abbildung 5.19.: Anwendung der Abstraktionsregel 11

> Gruppiere alle erfassten multiplen Funktionen zu einer Funktion. Benenne die Funktion geeignet um. Alle Ports und Signale, die genau aus einer Quelle stammen werden gleichermaßen zusammengefasst.

Anhand dieser Abstraktionsregel entsteht das logische Funktionsnetz der Abstraktionsebene 3. Abbildung 5.19 visualisiert ein Beispiel. Hier sind vier Funktionen dargestellt, die jeweils einen Aktuator repräsentieren: (1) `Stellmotor_ZV_FT`, (2) `Stellmotor_ZV_BFT`, (3) `Stellmotor_ZV_FTH` und (4) `Stellmotor_ZV_BFTH`. Diese multiplen Funktionen können anhand der oben beschriebenen Abstraktionsregel zusammengefasst werden. Das Ergebnis ist in der Abstraktionsebene 3 zu sehen. Alle Funktionen sind zu einer Funktion `Stellmotor_ZV` zusammengefasst. Gleiches gilt auch für die Ports und Signale. Alle Signale besitzen die gleiche Quelle. Also können sie zu einem gemeinsamen Signal zusammengefasst werden.

Abstraktionsregel 12: Anfangs- und Endpunkte Nachdem nun durch die vorherige Abstraktionsregel das logische Funktionsnetz in die Abstraktionsebene 3 überführt wurde, wird nachfolgend für diese Ebene eine weitere Abstraktionsmöglichkeit beschrieben. In Funktionsnetzen sind benachbarte Funktionen von Quell- bzw. Zielfunktionen in der Regel mit diesen eng verknüpft. Zum Beispiel sind Sensorikfunktionen mit Datenverarbeitungsfunktionen gekoppelt oder Aktorikfunktionen mit Statusinformationsfunktionen. Durch diese Abstraktionsregel werden durch Kompositionen an Anfangs- bzw. Endpunkten eines Funktionsnetzes derartige Strukturen abstrahiert. Eine entsprechende Regel wird wie folgt formuliert:

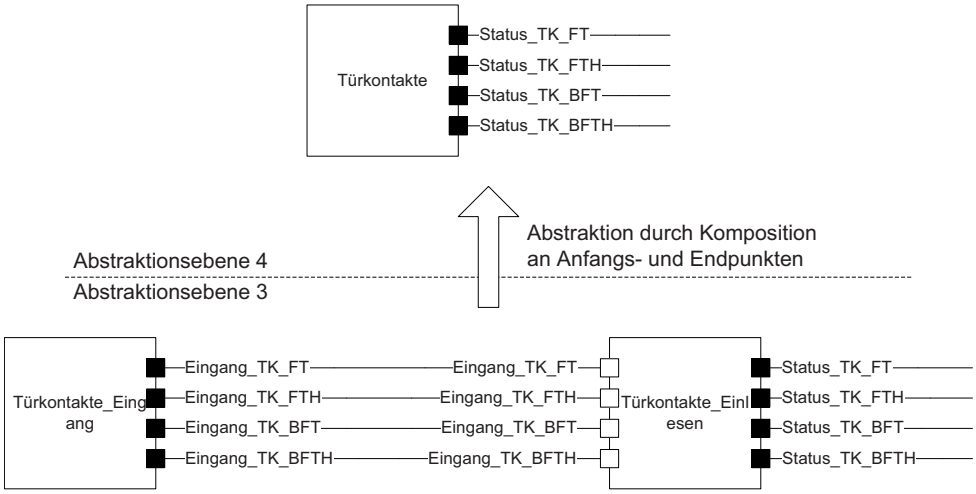


Abbildung 5.20.: Anwendung der Abstraktionsregel 12

> Komponiere, wenn sinnvoll, Quell- bzw. Zielfunktionen mit ihren benachbarten Funktionen. Alle Signale zwischen den komponierten Funktionen werden eliminiert. Benenne die so entstehende Funktion geeignet um.

Durch Anwendung dieser Regel wird das logische Funktionsnetz auf die Abstraktionsebene 4 transformiert. Abbildung 5.20 zeigt diesbezüglich ein Beispiel. Zu sehen sind zwei Funktionen, `Türkontakte_Eingang` und `Türkontakte_Einlesen`, die den Status der Türen ermitteln, also ob sie geöffnet oder geschlossen sind. Wenn die beschriebene Regel nun auf diese beiden Funktionen angewendet wird, resultiert die Komponierung in einer Funktion. Diese ist in der Abstraktionsebene 4 zu sehen. Beide Funktionen wurden also zu einer Funktion `Türkontakte` zusammengefasst. Die Signale, die zwischen beiden Funktionen existierten, fallen in dieser Abstraktionsebene aus.

Abstraktionsregel 13: Client- und Serverfunktionen Auf Abstraktionsebene 3 gibt es noch eine weitere Möglichkeit, das Funktionsnetz zu abstrahieren. Typischerweise treten viele Funktionspaare in Form von Client- und Serverrollen auf. Das Ziel dieser Abstraktionsregel ist es, von derartigen Client- und Serverfunktionen zu abstrahieren. Eine entsprechende Regel wird wie folgt formuliert:

> Fasse alle Funktionspaare zusammen, die in einer Client-Server-Beziehung zueinanderstehen. Eliminiere alle Signale, die im Rahmen der Client-Server-Kommunikation gesendet werden. Benenne die neue Funktion geeignet um.

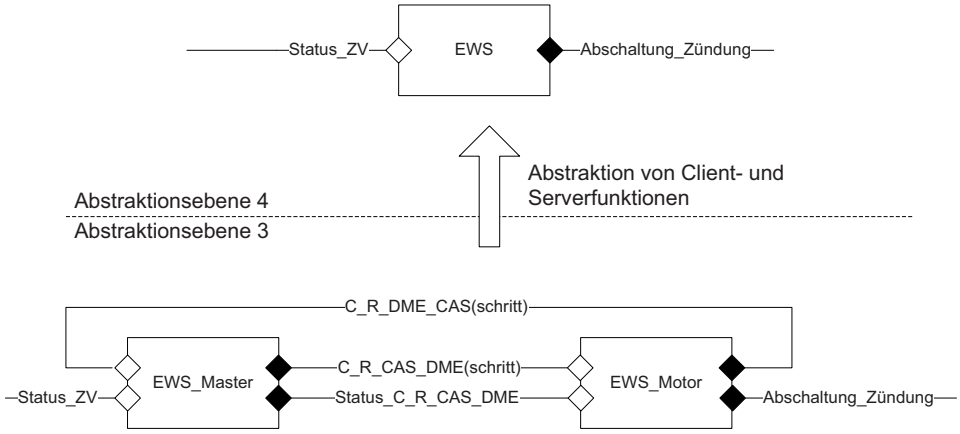


Abbildung 5.21.: Anwendung der Abstraktionsregel 13

Somit entsteht ein logisches Funktionsnetz der Abstraktionsebene 4. Abbildung 5.21 illustriert ein Beispiel. Hier sind auf Abstraktionsebene 3 die beiden Funktionen `EWS_Master` und `EWS_Motor` dargestellt. Sie stehen in einer Client-Server-Beziehung zueinander. Durch Anwendung der erläuterten Abstraktionsregel werden beide Funktionen zusammengefasst. Die gemeinsam kommunizierten Signale werden ebenfalls verborgen. Lediglich Signale, die von anderen Funktionen empfangen bzw. an weitere Funktionen gesendet werden, sind noch erhalten.

Abstraktionsregel 14: Features Das logische Funktionsnetz der Abstraktionsebene 4 bietet ebenfalls noch die Möglichkeit bestimmte Aspekte zu abstrahieren. So realisiert in der Regel eine Gruppe von Funktionen ein bestimmtes Feature, wie etwa die Zentralverriegelung, die Fensterheber oder die Wegfahrsperre. Der Zweck dieser Abstraktionsregel ist es, von diesen Featurerealisierungen zu abstrahieren, sodass lediglich das Feature modelliert wird. Eine entsprechende Regel wird wie folgt formuliert:

> Verschmelze alle Funktionen, die ein bestimmtes Feature realisieren zu einer gemeinsamen Funktion. Eliminiere alle Signale, die innerhalb dieser Featurefunktionen gesendet/empfangen werden. Alle Signale, die von Funktionen empfangen werden, die nicht Teil des Features sind, bilden die Eingabeschnittstelle der neuen Funktion. Alle Signale, die an Funktionen gesendet werden, die nicht Teil des Features sind, bilden die Ausgabeschnittstelle der neuen Funktion. Benenne die neue Funktion geeignet um.

Durch die Anwendung dieser Regel wird das logische Funktionsnetz in die Abstraktionsebene 5 überführt. In Abbildung 5.22 wird ein Beispiel dargestellt. In der Abstraktionsebene 4 ist ein Funktionsnetz dargestellt, das eine Reihe von Funktionen beinhaltet, die verschiedene Features realisieren. Dabei sind Funktionen

für die Zentralverriegelung, den Fensterheber und die Wegfahrsperre enthalten. Wenn nun die oben beschriebene Regel angewendet wird, können einige Funktionen zusammengefasst werden. So werden die Funktionen ZV_Aussenbedienung, ZV_Innenbedienung und ZV_Master zu einer gemeinsamen Funktion ZV zusammengefasst. Weiterhin werden die beiden Funktionen FH_Master und FH_Ansteuerung ebenfalls zu einer Funktion FH verschmolzen. Dies gilt auch für die Funktion EWS und weiteren Funktionen, die nicht in der Abbildung dargestellt sind. Das Resultat ist in der Abstraktionsebene 5 zu sehen. Die Ausgabeschnittstelle der neuen Funktion ZV setzt sich aus den Signalen zusammen die an Funktionen gesendet wurden, die nicht Teil des Features sind. Zum Beispiel sind dies die Signale Anf_Oeffnen und Anf_Schliessen, die an die Funktion FH_Master gesendet werden (vgl. Abstraktionsebene 4).

5.3.2. Abstraktionsebenen

Nachdem nun die Abstraktionsschritte durch Regeln definiert wurden, können die hieraus resultierenden Abstraktionsebenen erläutert werden. Zu diesem Zweck wird das Metamodell aus Abschnitt 5.2 durch entsprechende Konzepte erweitert. Die Vererbung ist dabei das wesentliche Konzept, um Abstraktionsebenen zu ermöglichen.

5.3.2.1. Ports und Verbindungen

Abbildung 5.23 illustriert die Erweiterungen am Metamodell aus Abbildung 5.8. Sie erweitern das Konzept der Verbindungen. Im Wesentlichen werden hierdurch die Signalträger generalisiert. Es existieren dabei drei verschiedene Arten: (1) Verbindungen zwischen Funktionen innerhalb eines Steuergeräts (LocalConnection), (2) direkte Verkabelungen zu Hardwarekomponenten (HWConnection) und (3) Verbindungen über verschiedene Bussysteme (BusConnection). Die Klasse Connection ist die Generalisierung dieser verschiedenen Verbindungsarten. Sie wird ab Abstraktionsebene 1 permanent verwendet.

In Abbildung 5.24 sind die Erweiterungen für Ports dargestellt. Aus den Abstraktionsregeln in Abschnitt 5.3.1 wurden insgesamt sechs Abstraktionsebenen identifiziert. Diese sind durch die Vererbungshierarchie auch hier abgebildet. Die Abstraktionsebene 1 wird durch die unterste Ebene der Vererbungshierarchie beschrieben. Hier sind die Kommunikationsparadigmen durch die Ports festgehalten. Das Sender-Receiver-Paradigma wird durch die Klassen RealizationRP und RealizationPP abgedeckt. Das Client-Server-Paradigma wird hingegen durch die Klassen RealizationClientRP, RealizationServerRP, RealizationClientPP und RealizationServerPP beschrieben. Da die auf dieser Ebene aufgestellten Abstraktionsregeln nicht die unterschiedlichen Kommunikationsparadigmen beeinflussen, gibt es auch in der nächsthöheren Vererbungshierarchie eine eins-zu-eins Generalisierung der untersten Portklassen. In der Abstraktionsebene 1 wurden bekanntlich Sensorik-/Aktoriksignale, zweiwertige Signale und Protokollabläufe abstrahiert, welche nicht die Kommunikationsparadigmen betreffen.

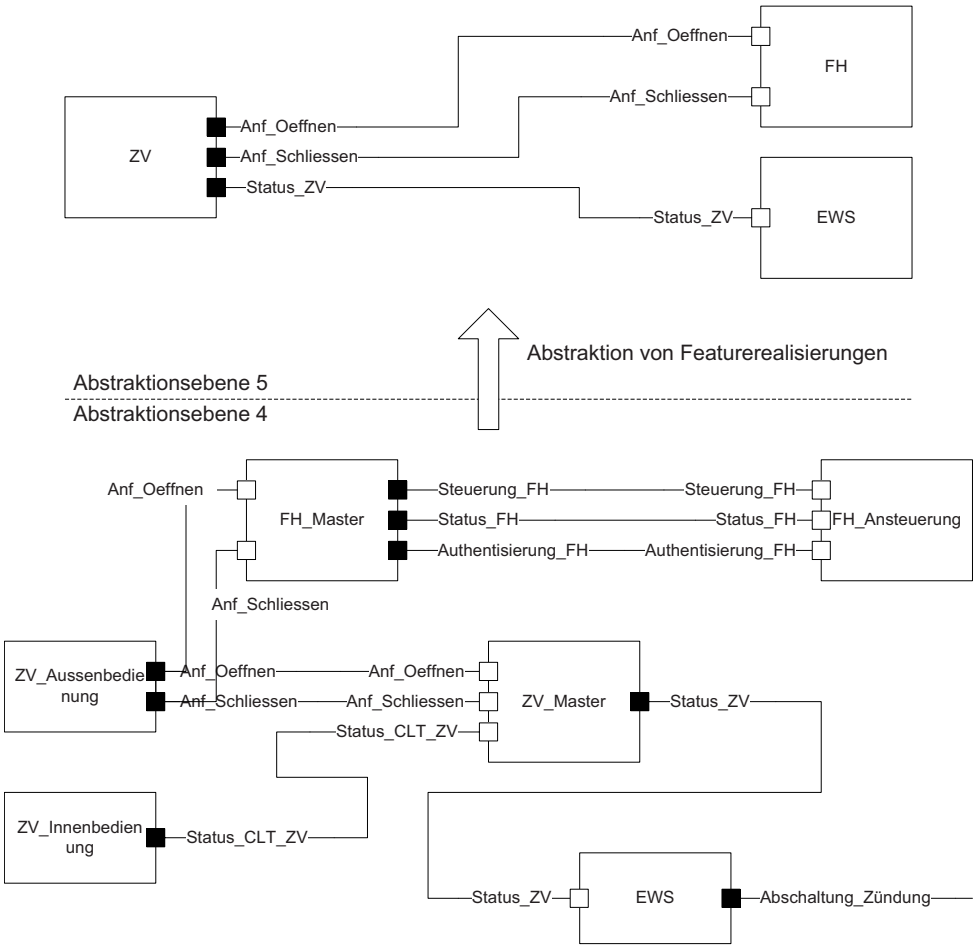


Abbildung 5.22.: Anwendung der Abstraktionsregel 14

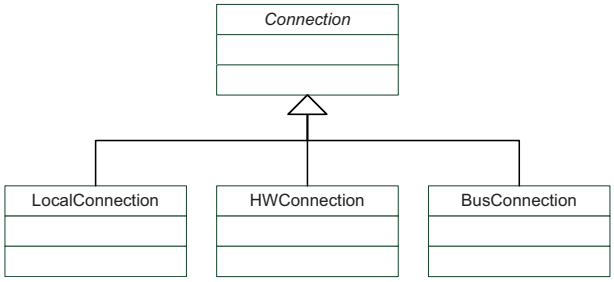


Abbildung 5.23.: Das Domänenmetamodell für Verbindungen (Quelle: [Poj11])

In Abstraktionsebene 2 wurden schließlich Kommunikationsparadigmen behandelt. Hierbei wurden ausschließlich Client-Server-Kommunikationen abstrahiert. Diese bestehen typischerweise aus zwei Sender-Receiver-Signalen, eine Anfrage und eine Antwort. Genau dies wurde mit der Regel abstrahiert. Im Metamodell aus Abbildung 5.24 ergibt dies die beiden Klassen `ClientServerRP` und `ClientServerPP`. Sie sind somit auf ein typisches Sender-Receiver-Muster reduziert. Eine Verschmelzung mit den entsprechenden Klassen `SenderReceiverRP` und `SenderReceiverPP` wäre an dieser Stelle ebenfalls vorstellbar. Hiervon wurde aber aus Verständlichkeits- und Übersichtlichkeitsgründen abgesehen.

Des Weiteren wurden in Abstraktionsebene 3 funktional zusammenhängende Signale gruppiert. Das Resultat hiervon ist letztlich die Verschmelzung der eben beschriebenen Klassen. Der wesentliche Faktor hierbei ist die Identifizierung der funktional zusammenhängenden Signale. In Abbildung 5.24 sind die entsprechenden repräsentativen Klassen `GeneralFunctionalityRP` und `GeneralFunctionalityPP`.

In Abstraktionsebene 4 wurde dann ein Schritt weiter gegangen und funktional verschiedene Signale gruppiert. Hiermit wurde eine weitere Maßnahme getroffen, die Überflut an Signalen zu reduzieren. Auffalend hierbei ist, dass primär Ausgabeports gebündelt und an verschiedene Funktionen gesendet werden. Die entsprechenden beschreibenden Klassen sind in der Vererbungshierarchie durch die beiden Klassen `RequestingPort` und `ProvidingPort` dargestellt.

Schließlich wurden in der Abstraktionsebene 5 von Signalflussrichtungen abstrahiert. Somit wurde lediglich kenntlich gemacht, dass es ein Informationsaustausch zwischen Funktionen vorliegt. In welche Richtung dieser Informationsfluss stattfindet wird aber dabei verborgen. In der Abbildung wird dies durch die Wurzelklasse `Port` beschrieben.

5.3.2.2. Funktionen

Nachdem nun die Erweiterungen für Ports und Verbindungen erläutert wurden, werden in diesem Abschnitt analog die Erweiterungen für Funktionen beschrieben. Abbildung 5.25 stellt das Metamodell dar, das die Erweiterungen beinhaltet. In Abschnitt 5.3.1 wurden insgesamt fünf Abstraktionsregeln vorgestellt, die sich durch fünf Vererbungshierarchieebenen in Abbildung 5.25 widerspiegeln.

Abstraktionsebene 1 besteht aus verschiedenen Funktionstypen, so gibt es neben Regelungsfunktionen weiterhin Sensorik- und Aktorikfunktionen, Sollwertgeber als auch Hardwarefunktionen. In der Abbildung sind diese Funktionstypen durch die Klassen `ControlFunction`, `Sensor`, `Actuator`, `SetPointDevice` und `Hardware` beschrieben. Die Regel für diese Ebene besagt, dass die verschiedenen Funktionstypen abstrahiert werden sollten, sodass ein gemeinsamer logischer Funktionstyp entsteht. Im Metamodell wird dies durch die Generalisierung der verschiedenen Funktionstypklassen mit der Klasse `Function` beschrieben.

Als Nächstes wurde in der Abstraktionsebene 2 die Funktionsmultiplizität behandelt. Die entsprechende Abstraktionsregel gruppiert semantisch ähnliche Funktionen, wie beispielsweise vier Aktuatorfunktionen zur Steuerung der vier Fahrzeugtüren, zu einer gemeinsamen Funktion. In Abbildung 5.25 wird dies durch die Klasse

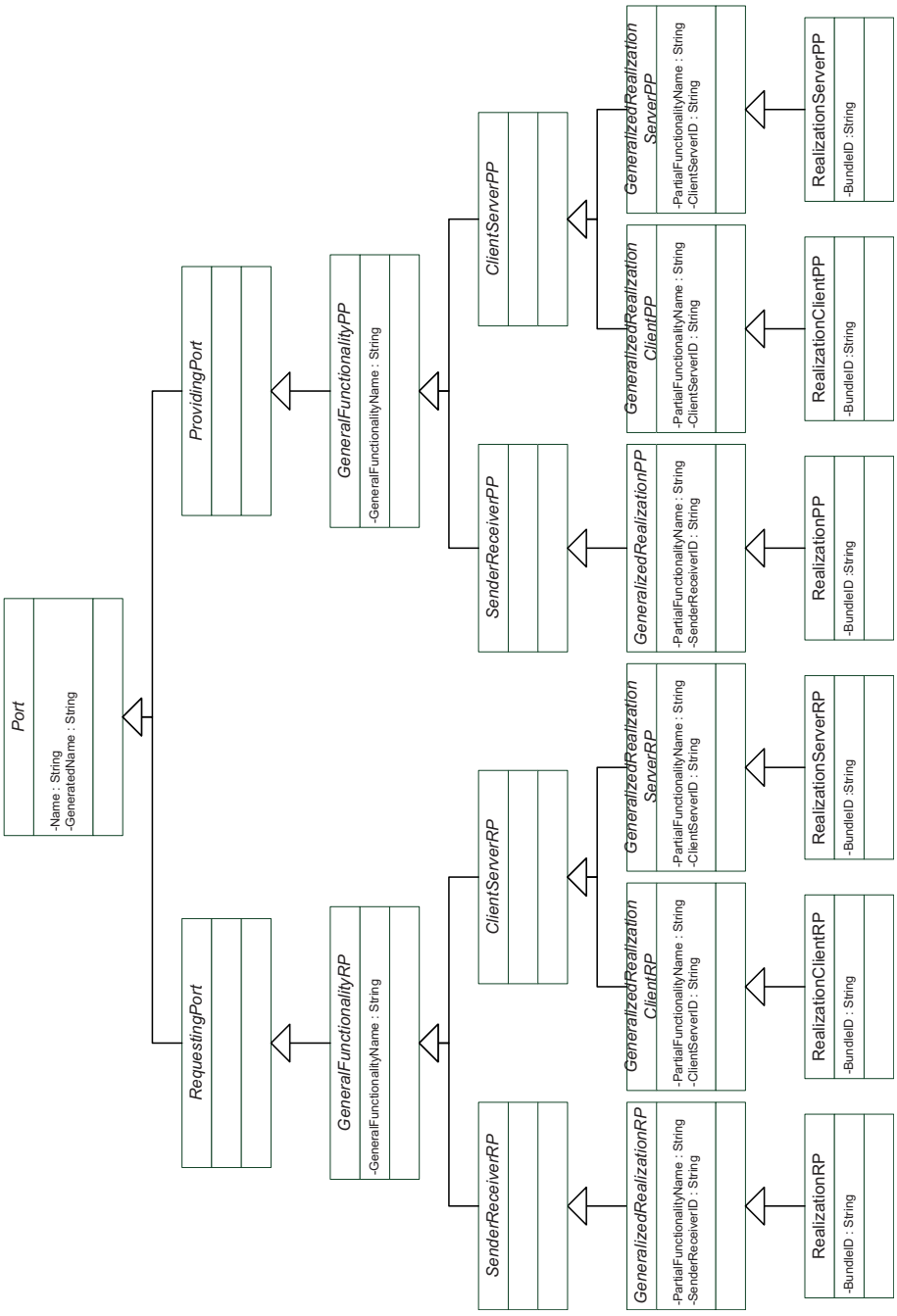


Abbildung 5.24.: Das Domänenmetamodell für Ports (Quelle: [Poj11])

GeneralLogicalFn abgedeckt.

Weiterhin wurden in der Abstraktionsebene 3 zwei Regeln vorgeschlagen, die zum einen Anfangs- und Endpunkte in einem Funktionsnetz komponieren und zum anderen Client- und Serverfunktionen verschmelzen. Daher wurde im Metamodell eine weitere Vererbungshierarchieebene eingeführt, die dies abbildet. Die entsprechende Klasse ist CompundFnWrapper.

Schließlich wurde in der Abstraktionsebene 4 eine Regel eingeführt, die Funktionen zusammenfassen, wenn sie Teil eines Features sind. Im Metamodell wurde daher die Klasse FeatureFusionFn als Wurzelelement der Vererbungshierarchie realisiert.

Die Vererbungshierarchien der Ports, Verbindungen und Funktionen zusammen stellen insgesamt das Domänenwissen auf Funktionsebene dar. Sie kann dazu verwendet werden, die Domäne geeignet zu klassifizieren. Daher wird das Metamodell um eine Klasse DomainKnowledge erweitert, die zur Klassifizierung dient. Sie ist an die Wurzelklasse FeatureFusionFn gebunden. Von hier aus sind sowohl Ports als auch Verbindungen erreichbar.

5.4. Variabilitätsmodellierung

Abstraktionsregeln ermöglichen, die Komplexität eines Funktionsnetzes zum einen durch kompaktere Ausdrucksmittel zu reduzieren und zum anderen durch Informationsverbergung zu abstrahieren. Bislang umfassen die Regeln aber noch nicht die Handhabung verschiedener Varianten. Diese koexistieren in Funktionsnetzen in einer informellen und unstrukturierten Form. Funktionsnetze müssen also noch um ein Konzept zur Anwendung von Variabilitätsmechanismen erweitert werden. Auf diese Weise werden Variationspunkte durch Kapselung der Varianten geeignet realisiert. Die Modellierung der Variabilität kann dabei durch das Variabilitätsmodell aus Kapitel 4 durchgeführt werden. Sowohl der Variabilitätsmechanismus als auch die Anwendung des Variabilitätsmodells werden in den folgenden beiden Abschnitten erläutert.

5.4.1. Variabilitätsmechanismus

Sind Variationspunkte in einem Funktionsnetz identifiziert (vgl. Kapitel 6), müssen sie durch einen geeigneten Variabilitätsmechanismus realisiert werden. Im Rahmen dieser Arbeit wird hierfür das Konzept der *Funktionsvarianten* eingeführt. Eine Funktionsvariante ist dabei eine spezielle Funktion, die Variationspunkte ausdrücken kann. So wie eine Funktion im bisherigen Sinne besitzt sie einen Namen und eine Schnittstelle, auf die durch Ports zugegriffen wird. Funktionsvarianten sind aber zudem in der Lage Varianten zu kapseln. Zu diesem Zweck wird eine Erweiterung des bisher vorgestellten Variabilitätsmodells herangezogen (vgl. Abschnitt 5.4.2).

Abbildung 5.26 illustriert ein Funktionsnetz, das die beiden Varianten Zentralverriegelung und Komfortzugang beinhaltet. Den wesentlichen Unterschied im Komfortzugang machen die zusätzlichen Sensoren an allen Türaußengriffen aus (Sensor_X; X steht dabei für FT, BFT, FTH und BFTH). Die Sensoren erkennen, ob ein

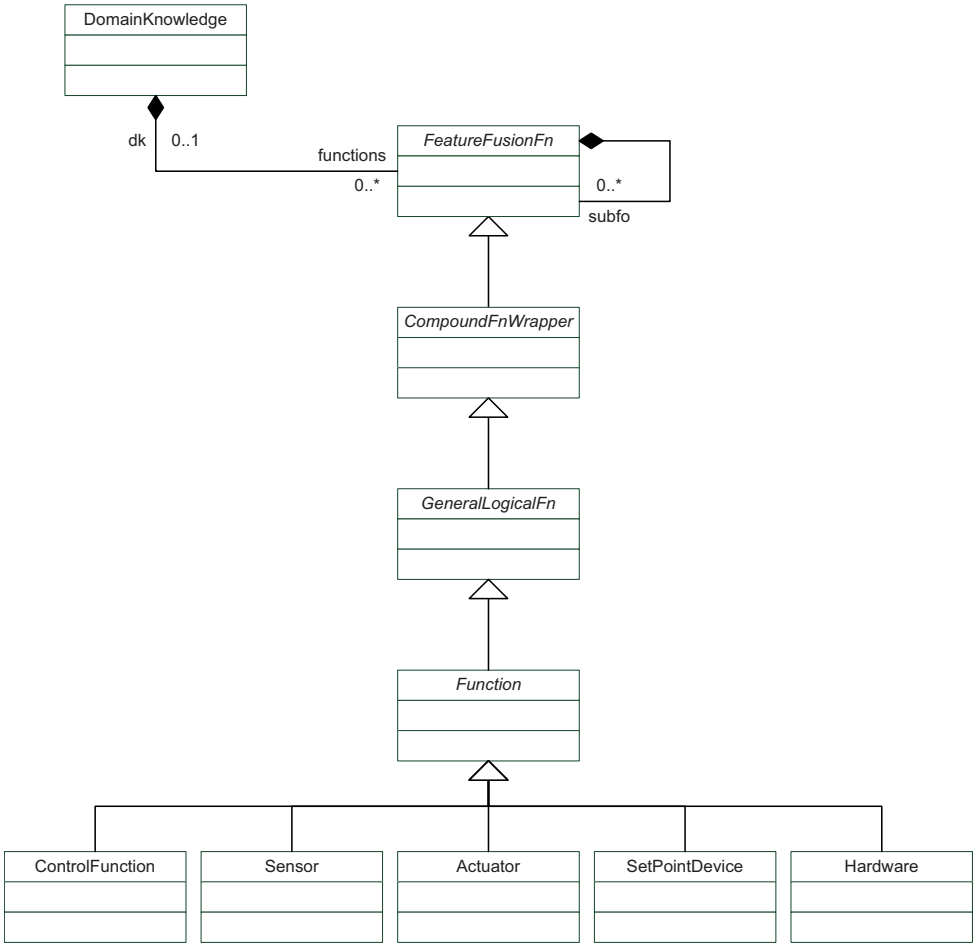


Abbildung 5.25.: Das Domänenmetamodell für Funktionen (Quelle: [Poj11])

Verriegelungs- oder Entriegelungswunsch besteht. Die Masterfunktion Komf_Master des Komfortzugangs initiiert dann die Authentifikation durch die eingebauten Antennen (Antenne_X), sodass ein Anfrage an die Funkfernbedienung gesendet werden kann. Der weitere Ablauf ist dann identisch zu der Zentralverriegelung, d.h. die Gültigkeit des Schlüssels wird verifiziert, von der Funktion ZV_Aussenbedienung ausgewertet und an die Masterfunktion ZV_Master übertragen, die wiederum die Aktuatoren steuert.

Eine Funktionsvariante ist hierbei ein Variabilitätsmechanismus, der in der Lage ist den variablen Anteil zu kapseln. Abbildung 5.27 veranschaulicht dies für das obige Beispiel. Die Funktionsvariante ist hellgrau dargestellt. Da der Inhalt dieser Funktion variabel ist, ist demnach auch ihre Schnittstelle variabel. Visuell wird dies durch die dunkelgraue Notation für Ausgabeports, hellgraue Notation für Eingabeports sowie gestrichelte Linien für Verbindungen dargestellt.

Durch den Variabilitätsmechanismus sind sofort sämtliche Variationspunkte ersichtlich. Sie sind zudem kompakt zusammengefasst, sodass die Komplexität signifikant reduziert wird. Spezifiziert wird der Variationspunkt im Variabilitätsmodell. Dieser wird im folgenden Abschnitt genauer erläutert. Außerdem werden hier auch die Erweiterungen am Metamodell verdeutlicht.

5.4.2. Variabilitätsmodell

Variationspunkte können durch den vorgestellten Variabilitätsmechanismus der Funktionsvarianten erfasst werden. Die Definition der variablen Merkmale werden aber dabei nicht berücksichtigt. Diese werden im Variabilitätsmodell spezifiziert und mit dem Variabilitätsmechanismus assoziiert. Um dies zu erreichen, wird das bisherige Metamodell für Funktionsnetze insofern erweitert, dass ein weiteres Sprachkonzept eingeführt (Funktionsvarianten) und die Definition der Variabilität dieses neuen Konzepts im Variabilitätsmodell ermöglicht wird.

Abbildung 5.28 illustriert diese Erweiterungen. Der Variabilitätsmechanismus zur Realisierung von Variationspunkten wird durch die Klasse FunctionVariationPoint beschrieben. Sie wird aus den Klassen AbstractFunction und VariationPoint spezialisiert. Somit wird sichergestellt, dass eine konsistente Assoziation zwischen dem Variabilitätsmechanismus im Funktionsnetz als auch im Variabilitätsmodell vorliegt. Ein Variationspunkt besteht wiederum aus mehreren Varianten. Die Varianten sind nichts anderes als Funktionen und ihr Zusammenspiel mit weiteren Funktionen. Sie sind daher aus den beiden Klassen AbstractFunction und Variant erweitert. Auf gleiche Weise werden Ports und Verbindungen dieser variablen Funktionen definiert. Demnach werden Ports (PortVariant) aus den beiden Klassen AbstractPort und Variant abgeleitet. Verbindungen (ConnectionVariant) sind wiederum Spezialisierungen der Klassen AbstractConnection und Variant.

Durch diese Erweiterungen ist es nun möglich, mehrere Varianten für einen Variationspunkt zu spezifizieren ohne die Komplexität im Funktionsnetz zu erhöhen. Bei Konfiguration einer Variante durch das Konfigurationsmodell wird somit stets die gewünschte Variante in das Funktionsnetz gebunden. Daher ist es auch an die-

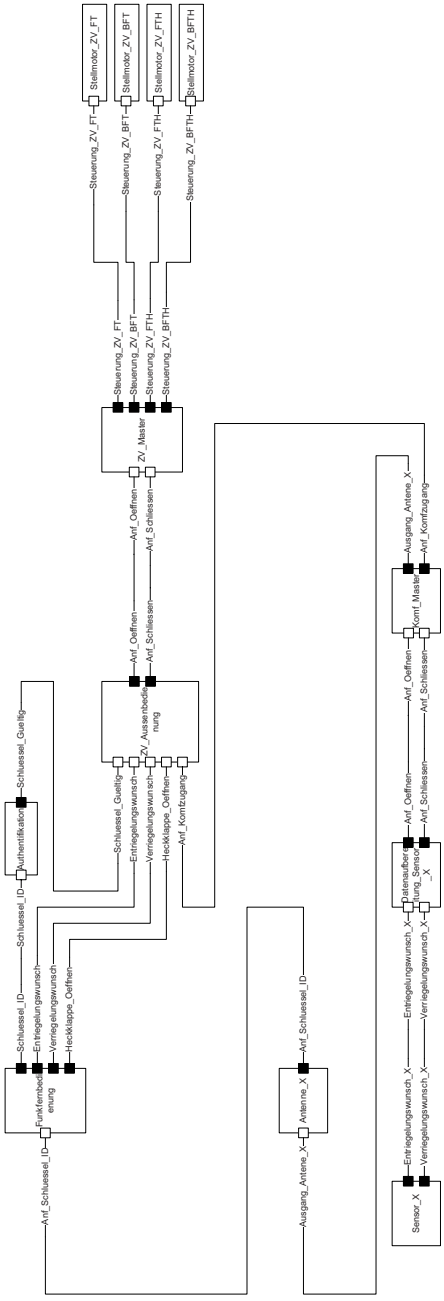


Abbildung 5.26.: Ein Teilausschnitt des Funktionsnetzes für das Fahrzeugzugangssystem mit den beiden Varianten Zentralverriegelung und Komfortzugang

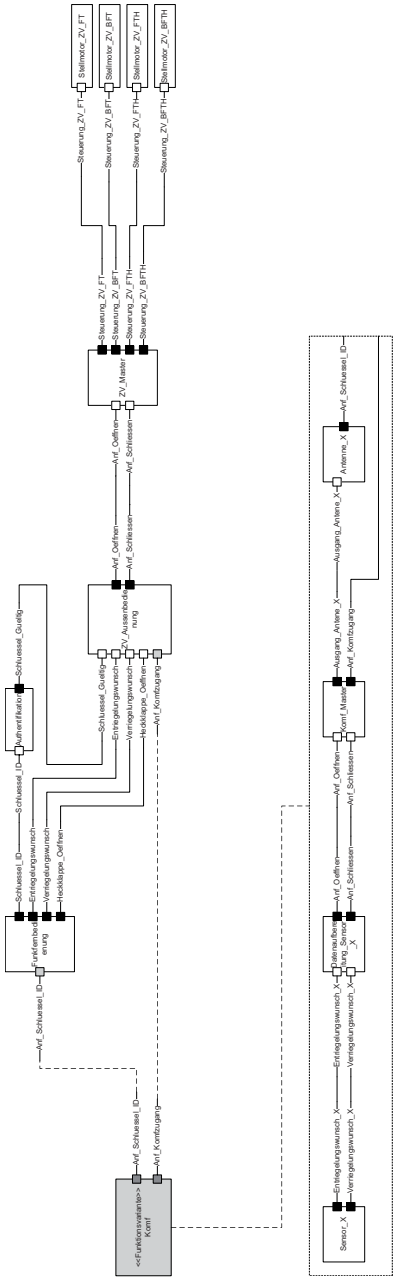


Abbildung 5.27.: Die Anwendung einer Funktionsvariante als Variabilitätsmechanismus

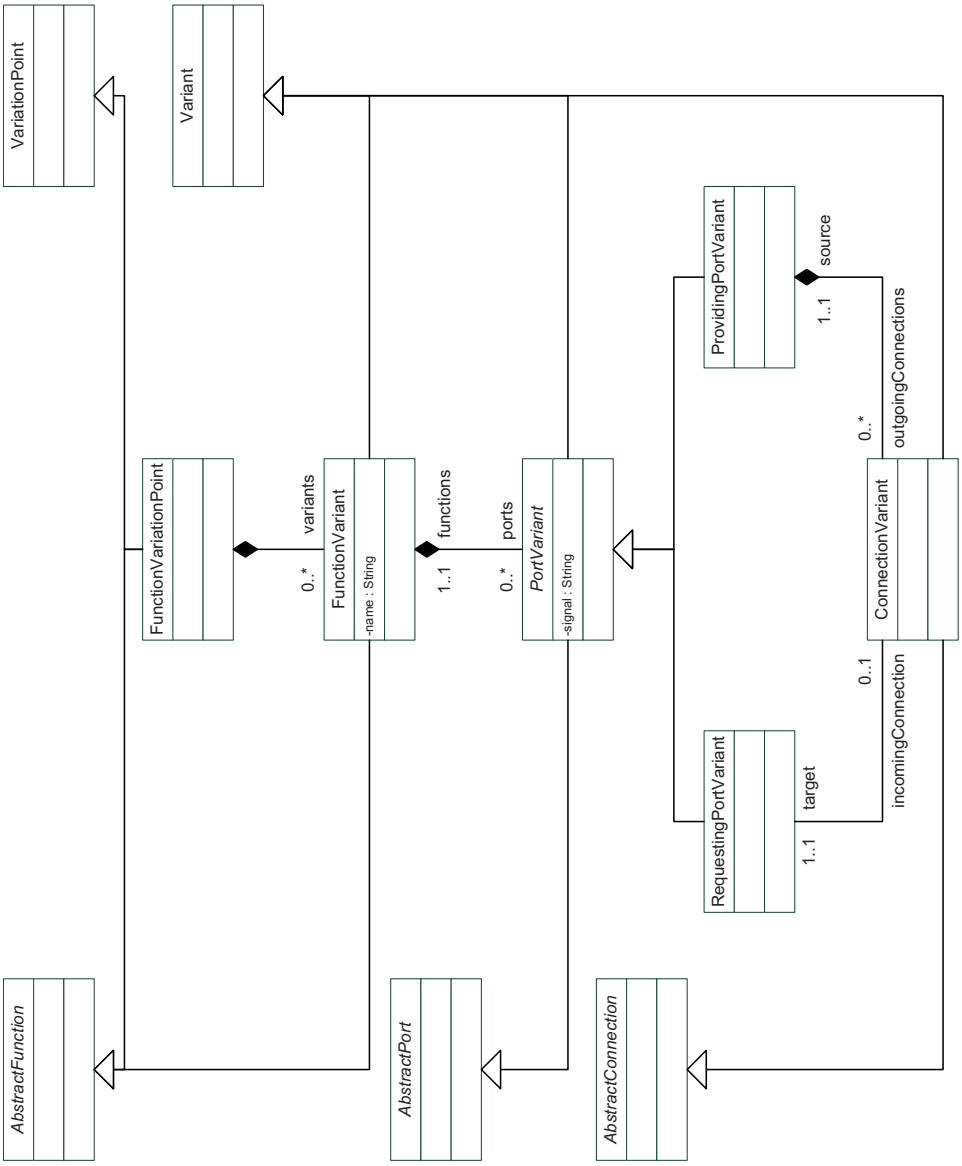


Abbildung 5.28.: Die Erweiterungen des Metamodells zur Beschreibung des Variabilitätsmechanismus der Funktionsvarianten

ser Stelle erforderlich alle Varianten als Funktionsnetz im Variabilitätsmodell zu erfassen, um die Transformierung dieser in das Funktionsnetz zu gewährleisten. Die Transformierung ist dabei bidirektional. Es können sowohl ausgehend vom Funktionsnetz aus Varianten definiert werden und im Anschluss in das Variabilitätsmodell überführt werden als auch zuerst im Variabilitätsmodell definiert und dann in das Funktionsnetz generiert werden. In [MnBRB10] wird dieser Vorgang eingehend beschrieben.

Abbildung 5.29 zeigt für das Beispiel aus Abbildung 5.27 das entsprechende Variabilitätsmodell. Es besteht aus einem Variationspunkt `Fahrzeugzugangssystem`, das genau eine Variante enthält, die optional ist. Die zugehörigen Funktionen, Schnittstellen und Verbindungen sind entsprechend enthalten. Um der aufgrund der großen Menge an Daten entsehender Unübersichtlichkeit entgegenzuwirken, besteht die Möglichkeit Teile aus dem Variabilitätsmodell auszublenden. So kann beispielsweise das Funktionsnetz der Variante `Komfortzugang` vollständig ausgeblendet werden.

Für die Generierung der Varianten in das Funktionsnetz wird dabei das aus Kapitel 4 vorgestellte Konzept mittels der Inferenzmaschine smodels eingesetzt. Weitere Informationen kann der Leser darüber hinaus auch aus der Masterarbeit von *Önder Babur* erhalten [nB10].

5.5. Realisierung

Im Folgenden werden die wichtigsten Aspekte zur Realisierung der bisher erläuterten Konzepte vorgestellt. Sie basieren auf den Arbeiten von *Antonio Navarro Perez* [Per09] und *Jan Pojer* [Poj11]. Dabei wird zunächst das Domänenmodell herangezogen und im Anschluss das Funktionsnetz. Die Realisierung des Variabilitätsmodells wurde bereits in Kapitel 4 beschrieben und daher hier vernachlässigt. Insbesondere wird noch abschließend auf die Integration der entstandenen Werkzeuge in ein umfassendes Werkzeug eingegangen.

5.5.1. Domänenmodell

Die Implementierung der Abstraktionsebenen basiert auf EMF. Die Metamodelle wurden in Abbildung 5.24 und Abbildung 5.25 bereits erläutert. Beide repräsentieren das funktionale Domänenwissen. Der Entwurf wird dabei von Abstraktionsebene 0 gestartet. Demnach können fünf Funktionstypen modelliert werden: Regelungsfunktionen, Sensoren, Aktuatoren, Sollwertgeber und Hardware. Zusätzlich kann zu jeder dieser Funktionen zugehörige Ports und weitere Subfunktionen definiert werden. Alle Elemente verfügen über weitere Attribute, die definiert werden müssen, damit die Abstraktionsregeln angewendet werden können. Weiterhin werden eindeutige IDs für alle modellierten Elemente generiert. Auf diese Weise können Elemente einfacher ermittelt werden. Außerdem wurde die eingestellte Eigenschaftsansicht durch eine erweiterte Form mittels Extended Editing Framework (EEF) ersetzt. Schließlich wurde das Werkzeug durch Verwendung eigener Icons verschönert.

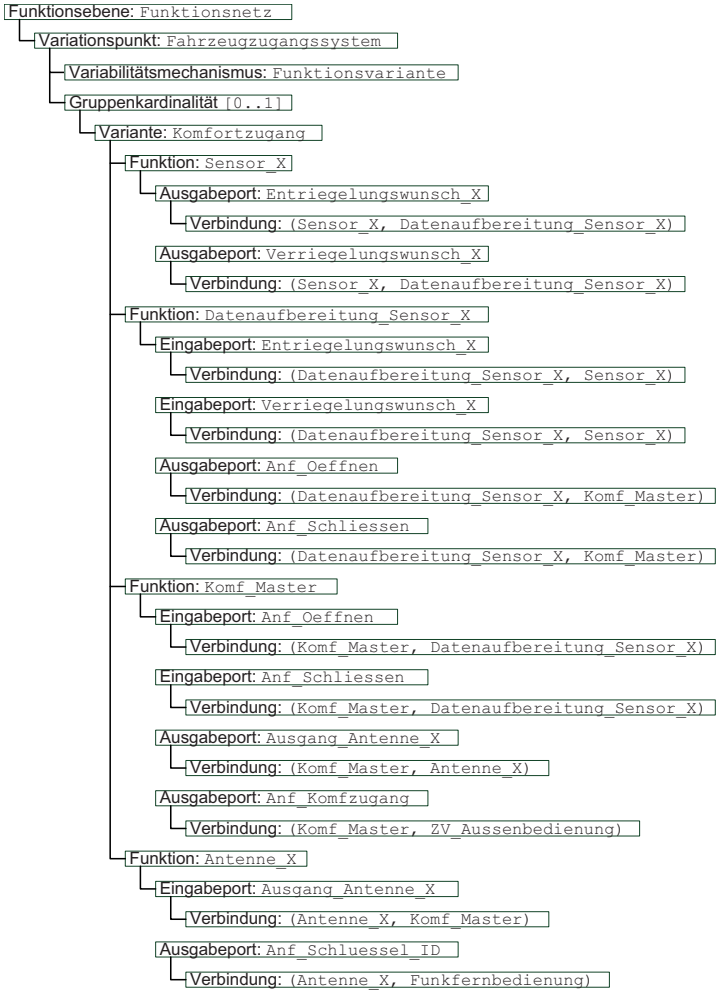


Abbildung 5.29.: Das Variabilitätsmodell in erweiterter Form zur Erfassung von variablen Funktionen und Kommunikationen

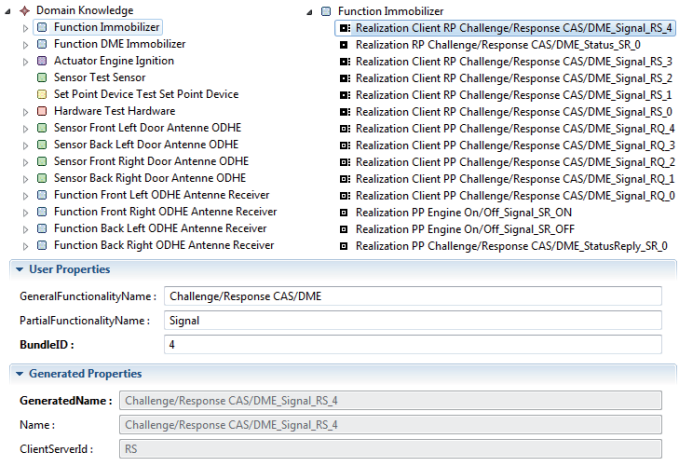


Abbildung 5.30.: Ein Screeshot des Werkzeugs zur Modellierung des Domänenwissens in den Abstraktionsebenen (Quelle: [Poj11])

Abbildung 5.30 zeigt ein Screenshot des entwickelten Editors. Im linken Bereich sind alle Funktionen modelliert. Wenn eine Funktion selektiert wird, kann im rechten Bereich die selektierte Funktion detailliert spezifiziert werden. Der untere Bereich des Screenshots beinhaltet die erweiterte Eigenschaftsansicht, in der Funktions- und Portnamen vergeben werden können.

5.5.2. Funktionsnetz

Die Realisierung des Funktionsnetzes basiert ebenfalls auf den Rahmenwerken EMF und GMF. Das Metamodell wurde bereits in Abbildung 5.8 beschrieben. Zwei wichtige Funktionalitäten, die hierbei realisiert wurden, werden im Folgenden detaillierter erläutert. Diese sind (1) die dynamische Einbindung der Werkzeugpalette auf Basis des Domänenmodells und (2) die Verwaltung der verschiedenen Abstraktionsebenen.

Die Werkzeugpalette für Funktionsnetze wird abhängig von Anpassungen und Änderungen am Domänenmodell immer wieder neu geladen. Zu diesem Zweck wird die aus GMF heraus generierte Fabrikklasse `FnPaletteFactory.java` modifiziert. Insbesondere wurde hierbei die Methode `fillPalette()` geändert. Zudem wurden eigene Palettenlader (`PaletteDrawer.java`) implementiert. Um sicherzustellen, dass die Werkzeugpalette immer dann neu geladen wird, wenn das Domänenmodell Änderungen unterzogen wird, wurde die Methode `doSave()` der Klasse `DKEditor.java` erweitert. Diesbezüglich wird ein Flag eingesetzt, der dann gesetzt wird, wenn das Modell geändert wurde. Dieses Flag wird dabei regelmäßig vom Diagrammeditor überprüft. Listing 5.1 umfasst den Codeausschnitt zur Detektierung des Änderungszustands und den Aufruf zur Aktualisierung der Werkzeugpalette.

Die Verwaltung der Abstraktionsebenen umfasst zum einen die Realisierung der Abstraktionsregeln und zum anderen die Transformationen zwischen den Ebenen.

```
1  public void selectionChanged(IWorkbenchPart part, ISelection selection) {
2          if (ResourceLoaderCustom.state == 1) {
3                  PaletteRoot pr = getEditDomain().getPaletteViewer()
4                                  .getPaletteRoot();
5                  createPaletteRoot(clearPalette(pr));
6                  ResourceLoaderCustom.state = 0;
7          }
8          super.selectionChanged(part, selection);
9  }
10
11 private PaletteRoot clearPalette(PaletteRoot pr) {
12         fnPaletteFactory.clearPalette(pr);
13         return pr;
14 }
15
16 protected PaletteRoot createPaletteRoot(PaletteRoot existingPaletteRoot) {
17         PaletteRoot root = super.createPaletteRoot(existingPaletteRoot);
18         fnPaletteFactory = new FnPaletteFactory();
19         fnPaletteFactory.fillPalette(root);
20         return root;
21 }
```

Listing 5.1: Die modifizierte Klasse `DynamicPaletteListener.java` zur Aktualisierung der Werkzeugpalette

Im Wesentlichen besteht der Prozess für jede Abstraktionsregel aus folgenden Grobschritten:

1. Gruppierung von Funktionen oder Ports entsprechend einer Abstraktionsregel.
2. Entfernung aller aus der Abstraktionsregel heraus resultierenden unnötigen Elemente aus der Werkzeugpalette.
3. Erzeugung einer neuen Menge an Funktionen und Ports auf Basis der gruppierten Daten.
4. Verbindung aller Ports miteinander.

Weiterhin wird das Wechseln zwischen verschiedenen Abstraktionsebenen durch Schalterknöpfe in der Benutzerschnittstelle der Werkzeugpalette realisiert. Abhängig davon, welche Taste gedrückt wurde, wird die entsprechende Transformierung gestartet. Die beiden Klassen `getFunctionsLayer` oder `getPortsLayer` ermitteln dabei zunächst die aktuelle Abstraktionsebene und speichern im Anschluss die Informationen zur gewünschten Abstraktionsebene. Zu jeder Abstraktionsebene werden im Wesentlichen zwei Aufgaben ausgeführt:

1. Der Kopiervorgang des Funktionsnetzes der aktuellen Abstraktionsebene in die nächsthöhere Ebene.
2. Die Modifikation der neuen Abstraktionsebene durch eine entsprechende Abstraktionsregel.

```
1  private static void getPortsLayer(LayeredResourcePorts lRes) throws IOException {
2          String currentRelPath = DKLayerAdjuster.getDiagramPart().getTitle();
3          HierarchyLayersUtil.layerStateSave(currentRelPath);
4          HierarchyLayersUtil.setActiveLayerPorts(lRes);
5          List<ILayeredResource> portLayers = HierarchyLayersUtil.
               getPortLayersToIterate();
6          for (ILayeredResource portLayer: portLayers) {
7                  File f = new File(getLayeredResourceFileName(portLayer));
8                  if (!f.exists()) {
9                          //copy the file and apply the reduction
10                         copy(new File(getLayeredResourceFileName(currentRelPath)), f
                               );
11                         //open the diagram
12                         openLayeredDiagram(portLayer);
13                         //display appropriate items related to the new DK layer
14                         DKLayerAdjuster.run((LayeredResourcePorts) portLayer);
15                 } else {
16                         //open the diagram
17                         openLayeredDiagram(portLayer);
18                 }
19         }
20 }
```

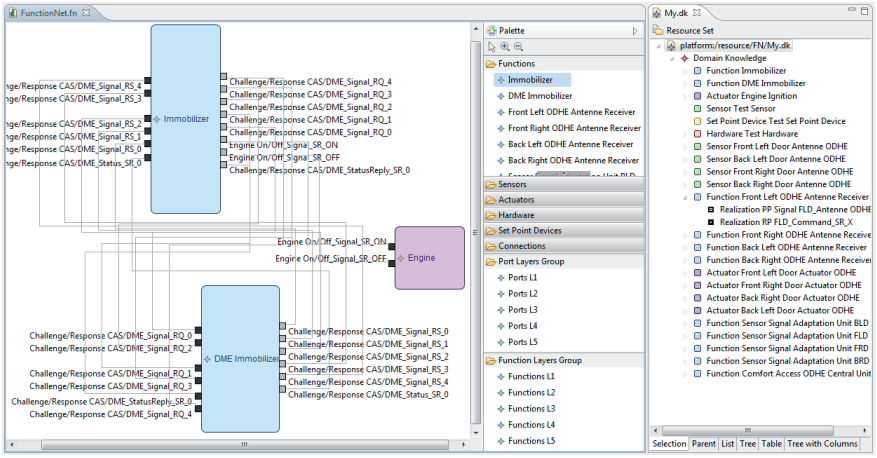
Listing 5.2: Die Klasse `getPortsLayer.java` zur Realisierung der Transformation von Ports

Listing 5.2 zeigt exemplarisch den Vorgang bei der Transformierung von Ports.

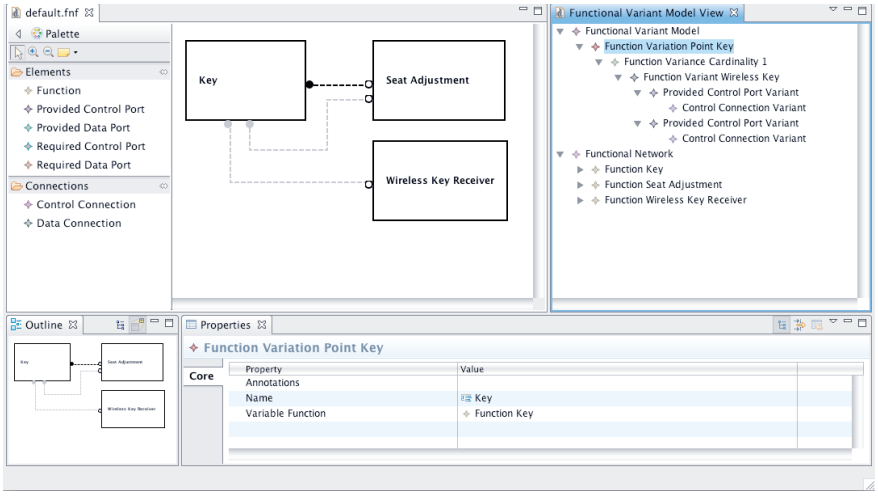
5.5.3. Integration aller Modelle

Die Generatoren für EMF und GMF erzeugen separate Editoren für das Domänenmodell, Funktionsnetz und Variabilitätsmodell. Zur einfachen und konsistenten Handhabung der Editoren werden diese in ein übergreifendes Werkzeug integriert. Damit die Integration der Editoren durchgeführt werden kann, ist es erforderlich, dass alle auf der gleichen Instanz der `EditingDomain` arbeiten. Die `EditingDomain` ist dabei ein Datenmodell, in der die persistenten Dokumente abgelegt werden. Zudem besteht jeder Editor aus weiteren Hilfssichten (Views). Diese Hilfssichten sind Subklassen von `ViewPart`. Damit nun jeder Editor parallel in der Benutzerschnittstelle zu sehen und zu bearbeiten ist, werden diese Hilfssichten herangezogen. Die Idee dabei ist, das Funktionsnetz als zentrales Dokument als Editor zu belassen und die weiteren Modelle, wie etwa das Domänenmodell und das Variabilitätsmodell als Sichten zu integrieren. Daher werden beide Modelle als `ViewPart` implementiert.

Abbildung 5.31 zeigt das Resultat der Integration aller Modelle. Das Hauptfenster wird dabei zur Modellierung von Funktionsnetzen eingesetzt. Die Sichten werden zur Modellierung der Domäne (Abbildung 5.31(a)) und Variabilität (Abbildung 5.31(b)) verwendet. Die Umschaltung zwischen den Abstraktionsebenen wird durch Schalterknöpfe in der Werkzeugpalette durchgeführt (vgl. Abbildung 5.31(a)).



(a) Funktionsnetze und Domänenmodell (Quelle: [Poj11])



(b) Funktionsnetze und Variabilitätsmodell (Quelle: [Per09])

Abbildung 5.31.: Screenshots der Werkzeuge auf Funktionsebene

5.6. Verwandte Arbeiten

Funktionsnetze sind im Softwareentwicklungsprozess ein wichtiges Dokument, um die Lücke zwischen der Anforderungsspezifikation und der E/E-Architektur zu schließen. Sie konkretisieren funktionale Anforderungen ohne Realisierungsdetails, wie etwa Datentypen oder Echtzeitanforderungen, zu spezifizieren. In diesem Kapitel wurde festgestellt, dass es für eine übergreifende Anwendung von Funktionsnetzen oftmals an verschiedenen Aspekten mangelt. So ist die Notationsheterogenität ein wesentliches Hindernis. Fehlende Abstraktionsebenen sowie Defizite in der Variabilitätshandhabung sind weitere Barrieren.

In der Literatur sind Arbeiten vorhanden, die sich diesen Problemstellungen widmen. Diese werden nachfolgend genauer beschrieben. Weiterhin werden sie anhand von Bewertungskriterien evaluiert und im Anschluss mit den Konzepten dieser Arbeit verglichen. Zu diesem Zweck werden folgende Kriterien herangezogen:

1. Funktionsnetzmodellierung
    - *Formale Definition:* Hierbei wird untersucht, ob es eine formale Definition des vorgeschlagenen Funktionsnetzes existiert.
    - *Notation:* Überprüfung, ob eine textuelle, grafische oder hybride Notation eingesetzt wird.
    - *Werkzeugunterstützung:* Existiert eine Werkzeugunterstützung zur Modellierung von Funktionsnetzen?
2. Domänenmodellierung
    - *Abstraktionsregeln:* Sind Abstraktionsregeln vorhanden, um einen nahtlosen Übergang zwischen der Anforderungsspezifikation und der E/E-Architektur zu gewährleisten?
    - *Abstraktionsebenen:* Sind ausreichend viele Abstraktionsebenen vorhanden?
    - *Werkzeugunterstützung:* Gibt es eine Werkzeugunterstützung zur Modellierung der Domäne?
3. Variabilitätsmodellierung
    - *Variabilitätsmechanismus:* Existieren Variabilitätsmechanismen zur Realisierung von Variationspunkten?
    - *Variabilitätsmodell:* Werden Varianten in einem Variabilitätsmodell geeignet dokumentiert?
    - *Werkzeugunterstützung:* Existiert zur Modellierung der Variabilität eine Werkzeugunterstützung?

Im Folgenden werden vier verwandte Arbeiten herangezogen: (1) Funktionsnetze mit UML-RT, (2) MOSES, (3) AutoMoDe und (4) VEIA. Diese werden beschrieben, bewertet und im Anschluss verglichen.

5.6.1. Funktionsnetze mit UML-RT

Michael von der Beeck hat in seinem Beitrag aus [vdB05, vdB07] einen Ansatz zur Architekturmodellierung von Fahrzeugfunktionen mitsamt der Funktionsnetzmodellierung, Softwarearchitekturmodellierung, Hardwarearchitekturmodellierung sowie der Partitionierung von Funktionsnetzen in Software und Hardware als auch das Deployment von Software auf Hardware vorgestellt. Zudem wurde der Ansatz in den bereits vorhandenen Entwicklungsprozess integriert und entsprechende Werkzeugunterstützung realisiert. Für die Architekturmodellierung wurde UML-RT eingesetzt. Motiviert wurde dieser Ansatz durch die stetig steigende Komplexität im Entwicklungsprozess für Fahrzeugsoftware. So sind die hohe Anzahl an Funktionen, der hohe Komplexitätsgrad einer einzelnen Funktion, die schwer nachvollziehbare Menge an Funktionsinteraktionen Schlüsselfaktoren für die Umsetzung eines derartigen Ansatzes.

Zunächst wird im Ansatz von *Michael von der Beeck* zwischen einer logischen und technischen Architektur unterschieden. In der logischen Architektur kommen Funktionsnetze zum Einsatz. Hier werden Funktionen, abstrahiert von Realisierungsentscheidungen, in Beziehung gesetzt. Als Modellierungssprache werden Strukturdiagramme verwendet. Funktionsnetze werden dann in einem weiteren Schritt in Software- und Hardwareanteile partitioniert und auf diese Weise konkretisiert. Das Resultat der Partitionierung ist die technische Architektur bestehend aus der Software- und Hardwarearchitektur. Während Softwarearchitekturen durch Komponentendiagramme modelliert werden, werden Hardwarearchitekturen durch Deploymentdiagramme entworfen. Das Deployment beschreibt, wie Elemente der Softwarearchitektur auf Elemente der Hardwarearchitektur abgebildet werden. Der Modellierungsansatz wird im Entwicklungsprozess als eine Aktivität nach dem Requirements Engineering ausgeführt. Eine Bindung beider Aktivitäten ist in dem Ansatz gewährleistet, sodass die Verfolgbarkeit von Entitäten aus beiden Aktivitäten ermöglicht wird. Die Phase nach der Architekturmodellierung ist die Verhaltensmodellierung und Codegenerierung. Um den Übergang möglichst nahtlos zu realisieren, wurden Transformationsregeln festgelegt, die die Modelle aus der Architekturmodellierungsphase in Modelle für die Verhaltensmodellierung transformieren. Für alle drei Phasen ist die Versionierungs- und Konfigurationsverwaltung realisiert. Die folgenden Erläuterungen beziehen sich größtenteils auf die Beschreibung von Funktionsnetzen. Für Informationen zu den anderen Modellen sei auf [vdB05, vdB07] verwiesen.

Als formale Grundlage dient das Metamodell für UML-RT Strukturdiagramme. Das Metamodell wurde in den Arbeiten [vdB05, vdB07] genauer beschrieben. Nachfolgend werden die wesentlichen Konzepte zusammengefasst dargestellt. Das Metamodell spezifiziert dabei die Entitäten, die zur Modellierung von Funktionsnetzen erforderlich sind. Eine Funktion im Funktionsnetz wird durch die Metamodellklasse `capsule` repräsentiert. Jede Funktion kann aus weiteren Teilfunktionen bestehen, sodass Hierarchien gebildet werden können. Jede Funktion hat zudem eine Schnittstelle. Sie wird durch die Metamodellklassen `port` und `protocol` abgebildet. Ein `port` ist somit der Kommunikationsanschluss für eine Funktion. Ein `protocol` ist

mit einem `port` assoziiert und verwaltet zwei Signalmengen: (1) Exportsignale und (2) Importsignale. Auf diese Weise ist die Schnittstelle einer Funktion in zwei Typen eingeteilt. Ein `connector` verbindet die Ports zweier Funktionen miteinander (ein exportierender Port kann nur mit einem importierenden Port verbunden werden).

Die beschriebenen Konzepte wurden schließlich in die bereits vorhandene Werkzeugumgebung Rational Rose Real-Time integriert. Im Folgenden werden die Kriterien herangezogen, um die Arbeit von *Michael von der Beeck* detaillierter zu bewerten.

1. Funktionsnetzmodellierung
   - *Formale Definition:* Funktionsnetze werden durch das Metamodell für UML-RT Strukturdiagramme definiert.
   - *Notation:* Verwendung einer grafischen Notation.
   - *Werkzeugunterstützung:* Integriert in Rational Rose Real-Time.
2. Domänenmodellierung
   - *Abstraktionsregeln:* Keine Unterstützung.
   - *Abstraktionsebenen:* Es wurden zwei Ebenen eingeführt, eine logische Architekturebene und eine technische Architekturebene.
   - *Werkzeugunterstützung:* Abstraktionsebenen werden in Rational Rose Real-Time modelliert.
3. Variabilitätsmodellierung
   - *Variabilitätsmechanismus:* Keine Unterstützung.
   - *Variabilitätsmodell:* Keine Unterstützung.
   - *Werkzeugunterstützung:* Keine Unterstützung.

5.6.2. MOSES

Das Fraunhofer Institut Software- und Systemtechnik (ISST) hat im Rahmen des Projekts Modellbasierte Systementwicklung (MOSES) zusammen mit BMW einen Ansatz entwickelt, der die E/E-Entwicklung bei BMW durch neue Vorgehensweisen und Modelle definiert. Die Ergebnisse wurden im ISST-Bericht [Kle06] festgehalten. Im Folgenden wird basierend auf diesem Bericht das Projekt genauer beschrieben und insbesondere auf die konzeptionelle Umsetzung von Funktionsnetzen genauer eingegangen.

MOSES ist vom Ansatz her sehr ähnlich zu dem Ansatz von *Michael von der Beeck* aus [vdB07] (vgl. Abschnitt 5.6.1). Auch hier wird die Architekturmodellierung zwischen der logischen und technischen Architektur unterschieden. Auf logischer Ebene kommen Funktionsnetze zum Einsatz, auf technischer Ebene wurde in einem weiteren Projekt der Übergang zu AUTOSAR-Architekturen spezifiziert. Der Grund für die Ähnlichkeit beider Ansätze rührt daher, dass sie sehr nah an den Entwicklungsprozess von BMW basieren und daher ähnliche Konzepte aufweisen. Im Folgenden

wird daher nur kurz auf das Funktionsnetzkonzept in MOSES eingegangen. Für weiterführende Information sei auf [Kle06] verwiesen.

Kernelemente in MOSES zur Beschreibung von Funktionsnetzen sind Funktionen und Verbindungen zwischen Funktionen. Eine Funktion in MOSES ist dabei eine abgeschlossene Entität, die entweder atomar, also keine innere Struktur besitzt, oder hierarchisiert ist. Jede Funktion besitzt eine Schnittstelle, um mit weiteren Funktionen kommunizieren zu können. Die Schnittstelle setzt sich aus Ports, die jeweils ein Interface besitzen, zusammen. Ein Port unterscheidet sich in Required Ports und Provided Ports. Ein Required Port empfängt Signale oder Operationen aus seiner Umwelt, während Provided Ports ihrer Umwelt Signale oder Operationen versenden. Die Menge der Signale und Operationen werden im Interface eines einzelnen Ports spezifiziert. Es gibt dabei zwei Arten von Signalen: (1) Datensignale und (2) Steuerungssignale. Zusätzlich wird in MOSES für jede Funktion ein Verhalten spezifiziert. Dieses Verhalten ist nicht zu verwechseln mit einer implementierungsnahen Verhaltensmodellierung. Sie beschreiben im Wesentlichen das prinzipielle Verhalten, um beispielsweise zeitliche Informationen modellieren zu können. Hierfür werden Statecharts eingesetzt. Zudem wird auch das Verhalten einer Funktion im Kontext seiner Umgebung beschrieben. Zu diesem Zweck werden in MOSES Protokollstatecharts eingesetzt.

Im Folgenden werden die Kriterien herangezogen, um das Projekt MOSES detaillierter zu bewerten.

1. Funktionsnetzmodellierung
    - *Formale Definition:* Eine formale Definition wurde im MOSES-Projekt nicht angegeben.
    - *Notation:* Verwendung einer grafischen Notation.
    - *Werkzeugunterstützung:* Keine Unterstützung.

2. Domänenmodellierung
    - *Abstraktionsregeln:* Keine Unterstützung.
    - *Abstraktionsebenen:* Es wurden zwei Ebenen eingeführt, eine logische Architekturebene und eine technische Architekturebene.
    - *Werkzeugunterstützung:* Keine Unterstützung.

3. Variabilitätsmodellierung
    - *Variabilitätsmechanismus:* Keine Unterstützung.
    - *Variabilitätsmodell:* Verwendung von Featuremodellen zur Dokumentation von Variabilität.
    - *Werkzeugunterstützung:* Keine Unterstützung.

5.6.3. AutoMoDe

Automotive Model-based Development (AutoMoDe) ist ein modellbasierter Ansatz zur Unterstützung der Funktionsentwicklung von Fahrzeugsoftware. Insbesondere wird dabei die frühe Phase im Entwicklungsprozess betrachtet und diese in mehrere Abstraktionsebenen aufgeteilt. Im Folgenden werden die im AutoMoDe-Projekt erzielten Ergebnisse auf Basis des Artikels aus [BBR+07] beschrieben.

AutoMoDe führt die strukturelle und verhaltensorientierte Modellierung von Software ein. Dabei werden für verschiedene Abstraktionsebenen mehrere Beschreibungssprachen realisiert. Damit diese Abstraktionsebenen nahtlos ineinandergreifen und zudem spätere Entwicklungsphasen, wie zum Beispiel die Codegenerierung und Testfallgenerierung unterstützt werden, sind entsprechende Tranformationsschritte definiert. Die definierten Abstraktionsebenen in AutoMoDe umfassen

- Functional Analysis Architecture (FAA)-Ebene
- Functional Design Architecture (FDA)-Ebene
- Logical Architecture (LA)-Ebene
- Technical Architecture (TA)-Ebene
- Operational Architecture (OA)-Ebene

FAA dient zur strukturellen Beschreibung benutzersichtbarer Funktionen eines Fahrzeugs. Dabei wird auf einem Abstraktionsniveau modelliert, ohne Realisierungsdetails heranzuziehen. Zu diesem Zweck wird im AutoMoDe-Ansatz das sogenannte Systemstrukturdiagramm eingesetzt. Konzeptionell betrachtet sind Systemstrukturdiagramme sehr ähnlich zu Funktionsnetzen und werden daher im weiteren Verlauf erneut aufgegriffen und detaillierter behandelt. Auf FAA-Ebene ist es somit möglich frühzeitig Abhängigkeiten und Konflikte zwischen Funktionen zu identifizieren. FDA erweitert FAA um Verhaltensbeschreibungen, die durch (1) Datenflussdiagramme, (2) Mode-Transition-Diagramme und (3) State-Transition-Diagramme spezifiziert werden. Datenflussdiagramme beschreiben den Datenfluss von Funktionen, die untereinander durch sogenannte Kanäle verbunden sind. Jede Funktionskomponente in einem Datenflussdiagramm kann entweder weiter hierarchisiert werden oder atomar sein. Jede hierarchisierte Funktion wird erneut durch ein Datenflussdiagramm beschrieben. Eine atomare Funktion hingegen kann durch ein Mode-Transition-Diagramm oder durch ein State-Transition-Diagramm weiter verfeinert werden. Beide korrespondieren zu endlichen Automaten, haben allerdings unterschiedliche Zwecke. Mode-Transition-Diagramme beschreiben den Übergang zwischen verschiedenen Betriebsmodi zur Laufzeit einer Funktion. State-Transition-Diagramme dienen zur Spezifikation zustandsbasierter Funktionalitäten. Das Ziel auf FDA-Ebene ist primär die frühzeitige Verifikation des Verhaltens interagierender Funktionen sowie die Identifikation wiederverwendbarer Funktionen. Die LA-Ebene fasst die in den höheren Abstraktionsebenen spezifizierten Funktionen in sogenannte Cluster zusammen, um möglichst kleinste Gruppierungen zu identifizieren, die nicht weiter

über Steuergeräte hinweg verteilt werden können. Darüber hinaus werden zeitliche Aspekte als auch Implementierungsdetails spezifiziert. Als Beschreibungsmittel werden auf LA-Ebene sogenannte Cluster-Communication-Diagramme eingesetzt. Mit dieser Beschreibung kann also die Verteilbarkeit von Clustern definiert werden. Cluster-Communication-Diagramme gleichen sehr an Datenflussdiagramme, jedoch mit der Unterscheidung, dass Cluster nicht weiter verfeinert werden können. Durch die TA-Ebene werden schließlich alle spezifizierten Cluster auf Hardwareressourcen abgebildet. Hierfür kommt auch die OA als wesentliches Hilfsmittel in Betracht.

Wie bereits weiter oben erwähnt sind Systemstrukturdiagramme vergleichbar mit Funktionsnetzen. Daher wird im Folgenden das zugrunde liegende Konzept genauer behandelt. Systemstrukutrdiagramme bestehen aus interagierenden Funktionen. Jede Funktion hat eine Schnittstelle, die über Ports definiert werden. Die Interaktion zwischen Funktionen erfolgt über gerichtete Kanäle, die an entsprechende Ports verbunden werden und somit Nachrichten und Signale austauschen. Weiterhin setzt sich eine Funktion aus weiteren Subfunktionen zusammen oder sie ist atomar, d.h., sie kann nicht weiter dekomponiert werden. Jede dekomponierbare Funktion wird in einem separaten Systemstrukturdiagramm modelliert. Schließlich definieren alle Ports, die zu keiner Funktion zugeordnet sind, die Schnittstelle des Systems zum Restsystem.

Die Ergebnisse des AutoMoDe-Ansatzes sind in das sogenannte AutoFOCUS-Werkzeug eingeflossen. Das Werkzeug beinhaltet alle oben beschriebenen Beschreibungssprachen für die verschiedenen Abstraktionsebenen. Realisiert wurden diese durch ein integriertes Metamodell. Zudem ist AutoFOCUS an kommerzielle Werkzeuge angebunden, um den nahtlosen Übergang zu den weiteren Entwicklungsphasen zu gewährleisten.

Im Folgenden werden die Kriterien herangezogen, um das Projekt AutoMoDe detaillierter zu bewerten.

1. Funktionsnetzmodellierung
    - *Formale Definition:* Es wird ein domänenspezifisches Metamodell definiert.
    - *Notation:* Verwendung einer grafischen Notation.
    - *Werkzeugunterstützung:* Unterstützung durch das AutoFOCUS-Werkzeug.
2. Domänenmodellierung
    - *Abstraktionsregeln:* Keine Unterstützung.
    - *Abstraktionsebenen:* Es wurden fünf Abstraktionsebenen eingeführt.
    - *Werkzeugunterstützung:* Unterstützung durch das AutoFOCUS-Werkzeug.
3. Variabilitätsmodellierung
    - *Variabilitätsmechanismus:* Keine Unterstützung.
    - *Variabilitätsmodell:* Keine Unterstützung.
    - *Werkzeugunterstützung:* Keine Unterstützung.

5.6.4. VEIA

Verteilte Entwicklung und Integration von Automotive-Produktlinien (VEIA) ist das Nachfolgerprojekt von MOSES, dass am Fraunhofer ISST zusammen mit Industriepartnern und Universitäten durchgeführt wurde. Die Ergebnisse des Projekts wurden in einer Reihe von Aufsätzen berichtet, unter anderem [GREKM07, GR08, MR09]. Die folgenden Erläuterungen basieren primär auf diesen Publikationen. In VEIA werden im Wesentlichen die aus dem MOSES-Projekt gewonnenen Erfahrungen erweitert und durch neue Problemstellungen ergänzt. So wurde im Referenzprozess Variabilität berücksichtigt und entsprechend integriert. Zudem wurde die Architekturbeschreibungssprache für alle Abstraktionsebenen zur Erfassung variabler Features ergänzt. Schließlich wurden Richtlinien zur Modellierung als auch Metriken zur Bewertung der Modelle als wesentliche Beiträge zum Projekt erstellt. Die Ergebnisse des Projekts sind in ein Werkzeug zur Durchführung des Prozesses eingeflossen. Nachdem im Folgenden der Refenzprozess vorgestellt wird, fokussieren sich die weiteren Erläuterungen auf Konzepte zur Modellierung von Funktionsnetzen in VEIA.

Der Referenzprozess in VEIA unterscheidet zwischen drei Abstraktionsebenen: (1) die featureorientierte Produktebene, (2) die funktionale Ebene und (3) die Infrastrukturebene. Zur Beschreibung aller Produkte wird in VEIA die Automotive-Domäne anhand von Fahrzeugfeatures beschrieben. Hierfür werden klassische Featuremodelle eingesetzt. Auf diese Weise ist es möglich variable Anteile geeignet zu spezifizieren. Das Featuremodell ist auch Startpunkt des Konfigurierungsprozesses. Durch eine Konfiguration werden sämtliche Variationspunkte der nachfolgenden Softwaredokumente gebunden, sodass eine konkrete Produktbeschreibung entsteht. Die funktionale Ebene unterteilt sich in zwei Subebenen: (1) die logische Architektur und (2) die Softwarearchitektur. Die logische Architektur wird durch Funktionsnetze beschrieben. Variabilität kann hier durch weitere zur Verfügung stehender Beschreibungselemente modelliert werden. Die Softwarearchitektur konkretisiert das Funktionsnetz, indem es eine spezifische softwaretechnische Lösung der Funktionen beschreibt. Als Beschreibungssprache wird hier AUTOSAR vorgesehen, sodass in VEIA sowie in MOSES die Notation nicht weiter ausspezifiziert wurde. Auch die Infrastrukturebene, dargestellt durch die technische Architektur, wurde in VEIA nicht im Detail betrachtet und daher auch keine technologische Realisierung entwickelt. Ein weiterer Aspekt, der in VEIA erfasst wurde, aber nicht im Detail spezifiziert wurde, ist die Verbindung zwischen der funktionalen Ebene und der Infrastrukturebene durch sogenannte Verteilungsmodelle.

Auch in VEIA werden Funktionsnetze als hierarchisch strukturierte und kollaborierende Komponenten spezifiziert. Jede Komponente stellt eine Funktion dar. Eine Komponente besteht aus Ports, die in Eingangs- und Ausgangsports unterschieden werden. Jeder Port hat eine Schnittstellenspezifikation, die Signale und Operationen für den entsprechenden Port festlegt. Komponenten werden durch Konnektoren von einem Ausgangsport mit einem Eingangsport verbunden (und umgekehrt). Komponenten werden mit ihren Subkomponenten über Delegationskonnektoren verbunden. Hier gilt allerdings, dass der Ausgangsport der Subkomponente

mit dem Ausgangsport der Komponente verbunden wird (analog bei Eingangsport). Des Weiteren werden in VEIA Mechanismen zur Modellierung von Variabilität in Funktionsnetzen spezifiziert. Diesbezüglich sind Variabilitätskonzepte für Komponenten und Ports entwickelt. Es wird zunächst zwischen konventionellen Komponenten und Variationspunktkomponenten unterschieden. Erstere werden durch Rechtecke mit Spitzen Ecken dargestellt, während Letztere durch abgerundete Ecken visualisiert werden. Die Semantik einer Variationspunktkomponente ist dabei zweierlei. Ist die Komponente atomar, besitzt sie also keine innere Struktur, so besagt die Variationspunktkomponente, dass sie optional ist. Ist die Komponente hingegen hierarchisch strukturiert, also mit einer inneren Struktur versehen, so dient die Variationspunktkomponente als eine Gruppierung alternativer Subkomponenten.

Des Weiteren können variable Eigenschaften für Ports definiert werden. So kann ein Port entweder verbindlich oder optional sein. Zudem kann die Schnittstelle eines Ports, also die Menge der Signale und Operationen, fix oder variabel sein. Schließlich kann ein Port aufgrund der Delegationsmöglichkeit entweder unabhängig oder abhängig sein. Durch abhängige Ports wird ersichtlich, dass Variabilität delegiert wurde, also ihren Ursprung anderorts hat. Unabhängige Ports stellen immer den Ursprung einer Variabilität dar. Aus diesen drei Eigenschaften ergeben sich insgesamt acht mögliche Portarten.

Außerdem können variable Eigenschaften für Ports abhängig von der Komponentenart ausgedrückt werden. Hierbei wird zwischen drei Komponentenarten unterschieden: (1) atomare Komponenten, (2) hierarchisch strukturierbare Komponenten und (3) Variationspunktkomponenten.

Im Folgenden werden die Kriterien herangezogen, um das Projekt VEIA detaillierter zu bewerten.

1. Funktionsnetzmodellierung
    - *Formale Definition:* Eine formale Definition wurde im VEIA-Projekt nicht angegeben.
    - *Notation:* Verwendung einer grafischen Notation.
    - *Werkzeugunterstützung:* Unterstützung durch den VEIA-Demonstrator.
2. Domänenmodellierung
    - *Abstraktionsregeln:* Keine Unterstützung.
    - *Abstraktionsebenen:* Es wurden drei Abstraktionsebenen eingeführt.
    - *Werkzeugunterstützung:* Unterstützung durch den VEIA-Demonstrator.
3. Variabilitätsmodellierung
    - *Variabilitätsmechanismus:* Variabilität wird durch eine Variationspunktkomponente realisiert.
    - *Variabilitätsmodell:* Variabilität wird durch die Verwendung von Featuremodellen dokumentiert.
    - *Werkzeugunterstützung:* Unterstützung durch den VEIA-Demonstrator.

5.6.5. Vergleich

Abbildung 5.32 setzt die beschriebenen verwandten Arbeiten mit dieser Arbeit anhand der Bewertungskriterien in Bezug. Ein formale Definition stellt die Basis zur Modellierung von Funktionsnetzen dar. Aus der Literatur konnte lediglich bei MOSES und VEIA eine derartige Definition nicht ermittelt werden, was nicht bedeutet, dass keine existiert. Bei UML-RT wurde das Metamodell für Strukturdiagramme herangezogen. In AutoMoDe als auch in dieser Arbeit wurden hingegen domänenspezifische Metamodelle definiert. Diese haben den Vorteil, dass sie für die Bedürfnisse der betrachteten Domäne genau angepasst werden können. In allen Arbeiten wurde eine grafische Notation bevorzugt. Eine Werkzeugunterstützung wurde in MOSES nicht realisiert. Diese wurde dann im Nachfolgeprojekt VEIA eingeführt.

Um die systematische Wiederverwendung stärker zu fördern, wurde die Domänenmodellierung als wesentliche Aktivität erkannt. Ein wichtiges Ergebnis dieser Untersuchungen sind die Anforderungen an hinreichend vielen Abstraktionsebenen. Zur Ermittlung dieser Ebenen wurden geeignete Abstraktionsregeln definiert. Diese Arbeit ist die Einzige, die für den Übergang von einer Abstraktionsebene in die nächste entsprechende Regeln definiert. Dabei sind zwei Regeln für Verbindungen, sechs Regeln für Ports und fünf Regeln für Funktionen entstanden.

Abstraktionsebenen sind dennoch in jeder Arbeit eingeführt. Während UML-RT und MOSES zwei Ebenen einführt (logische und technische Architekturebene), werden in AutoMoDe fünf (FAA, FDA, LA, TA und OA) und in VEIA drei Ebenen (Produktebene, funktionale Ebene und Infrastrukturebene) definiert. In dieser Arbeit sind die Abstraktionsebenen im Vergleich zu den anderen Ansätzen feingranularer definiert. So werden zwei Ebenen für Verbindungen, sieben Ebenen für Ports und fünf Ebenen für Funktionen definiert. Auf diese Weise kann das Konzept dieser Arbeit die Lücke zwischen Anforderungsspezifikation und E/E-Architektur vollständig schließen. Die Modellierung der verschiedenen Ebenen werden bis auf im Projekt MOSES werkzeugtechnisch unterstützt.

Schließlich wurde der Aspekt der Variabilitätsmodellierung genauer betrachtet. Ein geeigneter Variabilitätsmechanismus wurde lediglich in dieser Arbeit und in VEIA vorgeschlagen. Während VEIA die Variationspunktkomponente eingeführt hat, wird in dieser Arbeit das Konzept der Funktionsvariante eingesetzt. Letzteres hat den wesentlichen Vorteil, dass sie auf das Konzept der maximalen Schnittstelle basiert, sodass die Einführung verschiedener Porttypen nicht erforderlich wird und somit einen Wust von Porttypen vermeidet.

Zur Dokumentation der Variabilität wurde in MOSES und VEIA ein Featuremodell vorgeschlagen. In dieser Arbeit wurde das aus Kapitel 4 definierte Variabilitätsmodell herangezogen. Die wesentlichen Vorteile dieses Modells wurden bereits dort beschrieben. Eine Werkzeugunterstützung wurde dabei nur in VEIA und in diese Arbeit integriert.

	Mengi	UML-RT	MOSES	AutoMoDe	VEIA
Funktionsnetzmodellierung					
Formale Definition	Domänenspezifisches Metamodell	Metamodell für UML-RT Strukturdiagramme	×	Domänenspezifisches Metamodell	×
Notation	grafische Notation	grafische Notation	grafische Naotation	grafische Notation	grafische Notation
Werkzeugunterstützung	✓	✓	×	✓	✓
Domänenmodellierung					
Abstraktionsregeln	2 Regeln für Verbdg. 6 Regeln für Ports 5 Regeln für Fkt.	×	×	×	×
Abstraktionsebenen	2 Ebenen für Verbdg. 7 Ebenen für Ports 5 Ebenen für Fkt.	Logische Architektur-ebene und technische Architekturebene	Logische Architektur-ebene und technische Architekturebene	5 Abstraktionsebenen FAA, FDA, LA, TA, OA	3 Abstraktionsebenen Produktebene, funktionale Ebene, Infrastrukturebene
Werkzeugunterstützung	✓	✓	×	✓	✓
Variabilitätsmodellierung					
Variabilitäts-mechanismus	Funktionsvarianten	×	×	×	Variationspunkt-komponente
Variabilitätsmodell	Variabilitätsmodell	×	Featuremodell	×	Featuremodell
Werkzeugunterstützung	✓	×	×	×	✓

Abbildung 5.32.: Vergleichsaufstellung mit den Konzepten dieser Arbeit und verwandter Arbeiten

5.7. Zusammenfassung

In diesem Kapitel wurde das Konzept der Funktionsnetze vorgestellt. Es ist im Referenzprozess auf der Funktionsebene angesiedelt und dient als Brücke zwischen der Anforderungsspezifikation und der E/E-Architektur. Aufgrund verschiedener Ansichten von Automobilherstellern und Zulieferern existieren derzeit verschiedene Formalismen und Notationen für Funktionsnetze. Außerdem wurde festgestellt, dass diese Heterogenität nur einen Teil der Lücke zwischen Anforderungsspezifikation und E/E-Architektur schließt. Im Rahmen dieser Arbeit wurden daher zur Modellierung von Funktionsnetzen ein möglichst umfassendes Basiskonzept formalisiert und vertraute Notationen eingeführt. Zudem wurde das Konzept über mehrere Abstraktionsebenen realisiert, sodass die Lücke vollständig geschlossen werden konnte.

Die Wiederverwendung wurde durch Einführung von Domänenmodellen unterstützt. Ein Domänenmodell ist auch gleichzeitig die Zentrale zur Steuerung des Übergangs zwischen den identifizierten Abstraktionsebenen. Hierfür wurden Abstraktionsregeln integriert.

Schließlich wurde zur Handhabung der Variabilität das Konzept der Funktionsvariante eingeführt. Sie ist ein Variabilitätsmechanismus zur Realisierung von Variationspunkten in Funktionsnetzen. Weiterhin wurde der Mechanismus zu Dokumentations, Repräsentations- und Konfigurierungszwecken durch das Variabilitätmodell dieser Arbeit ergänzt.

Kapitel 6.

Architekturebene

6.1. Einleitung und Motivation

Auf Funktionsebene wurden Funktionen und ihre Beziehungen untereinander in mehreren Abstraktionsebenen beschrieben. Variationspunkte wurden durch den Mechanismus der Funktionsvarianten realisiert. Modelliert und repräsentiert wurde diese Variabilität durch das Variabilitätsmodell aus Kapitel 4. Die *Architekturebene* konkretisiert die Funktionsebene durch Modellierung des Funktionsverhaltens. Modellbasierte Beschreibungsmittel werden hierfür zunehmend eingesetzt [Bro04]. Ein prominentes Beispiel, welches auch die Grundlage für dieses Kapitel darstellt, ist die Simulink-Toolbox der Matlab-Software [Webc]. Mit ihr können Funktionen und Funktionsinteraktionen durch eine grafische Beschreibungssprache in einem Detailgrad beschrieben werden, der mit Funktionsnetzen nicht möglich wäre. Insbesondere können diese funktionalen Beschreibungen zu Simulationen am PC oder am Prototypen sowie zur Codegenerierung für Zielplattformen verwendet werden.

Variabilität muss auch auf Architekturebene erfasst werden. Diese Aussage umfasst zum einen Variationspunkte, die bereits auf Funktionsebene erfasst wurden und nun auch auf Architekturebene realisiert und modelliert werden. Beispielsweise muss der Variationspunkt `Fahrzeugzugangssystem` mit den Varianten `Zentralverriegelung` und `Komfortzugang`, welches bereits auf Funktionsebene erfasst wurde, auch in den Simulink-Modellen berücksichtigt werden. Zum anderen werden auch Variationspunkte einbezogen, die auf Architekturebene neu entstehen [PBKW09]. Derartige Variationspunkte sind typischerweise (aber nicht zwingend notwendig) das Resultat einer Verfeinerung bereits vorhandener Variationspunkte. Zum Beispiel kann die Variante `Zentralverriegelung` in Simulink durch verschiedene Verhaltensmodelle realisiert werden, die zu unterschiedlichen Zwecken dienen, wie etwa zur Simulation oder zur Codegenerierung. Dies eröffnet allerdings aus Sicht der Architekturebene eine *neue Dimension der Variabilität: Realisierungsvarianten*. Sie müssen auf Architekturebene ebenfalls geeignet erfasst werden.

In diesem Zusammenhang ist die *Vorgehensweise* bei der Verhaltensrealisierung ein wichtiger Aspekt, denn nur durch dieses Wissen können passende Maßnahmen in Hinblick auf die Handhabung der Variabilität ergriffen werden. In der Automobilindustrie entstehen Varianten in der Regel *inkrementell* und *evolutionär*. Ein Beispiel soll dies verdeutlichen: Zunächst wird eine Realisierungsvariante für einen bestimmten Zweck umgesetzt, wie zum Beispiel die Simulationsvariante der

`Zentralverriegelung`. Auf Basis dieser Variante wird eine weitere Realisierungsvariante modelliert, wie etwa die Steuergerätvariante der `Zentralverriegelung`. Durch erneute Wiederverwendung und Modifikation der Simulationsvariante bzw. Steuergerätvariante für die `Zentralverriegelung` werden die entsprechenden Varianten des `Komfortzugangs` realisiert.

Es ist offensichtlich, dass alle oben genannten Varianten sehr viel gemeinsam haben, aber auch an bestimmten Stellen differieren werden. Diese Gemeinsamkeiten und Variationen sind allerdings noch nicht erfasst. Dies kann zum einen auf die inkrementelle und evolutionäre Vorgehensweise zurückgeführt werden. Zum anderen ist dies aber auch durch die Simulink-Modellierungssprache begründet, die zur Verhaltensrealisierung der Varianten, über ein mächtigeres Sprachvokabular verfügt und somit feingranularere Variabilitätsdimensionen abdeckt. Zum Beispiel können Signale durch verschiedene Datentypen definiert und Funktionen vielfältig parametrisiert werden, welches auf Funktionsebene nicht möglich wäre.

Die wesentliche Frage, die sich hieraus ergibt und somit auch das Thema dieses Kapitels bildet, ist, wie diese noch nicht erfassten gemeinsamen und variablen Anteile identifiziert und modelliert werden können, um Variabilität Ebenen-übergreifend und vollständig zu erfassen. In der Regel ist hierfür ein *Analysevorgang* erforderlich, der möglichst frühzeitig durchgeführt werden sollte. Hierbei werden typischerweise zunächst zwei Simulink-Modelle betrachtet. Ausgehend von diesen beiden Modellen wird versucht, ihre Differenzen zu ermitteln. Der Analysevorgang wird in diesem Zusammenhang auch *Differenzierung* oder *Differenzierungsvorgang/-prozess* genannt. Sind die Modelle relativ klein, ist eine manuelle Differenzierung noch ohne großen Aufwand durchführbar. Werden die Modelle jedoch größer, ist es nicht mehr ohne Weiteres möglich, Gemeinsamkeiten, Variationspunkte und Varianten zu erörtern [PMB$^+$12, MPBK11, BPK10]. Für die Differenzierung ist daher eine weitestgehend *automatisierte Werkzeugunterstützung* erforderlich, um Fehler, die durch die manuelle Analyse entstehen könnten, zu vermeiden und um den Analyseprozess zu beschleunigen.

Ist die Differenzierung einmal abgeschlossen, so können die gewonnenen Erkenntnisse zur Variabilitätsmodellierung herangezogen werden. Hierfür ist zunächst oftmals eine *Restrukturierung* erforderlich [Mar05, Mos09], um die verglichenen Modelle zusammenzuführen und die identifizierten Variationspunkte durch geeignete *Variabilitätsmechanismen* zu realisieren [BPK09]. Um weiterhin die Variabilität adäquat zu dokumentieren und zu repräsentieren, besteht der Bedarf nach einem *Variabilitätsmodell*, mit dem die Variabilitätsinformationen explizit erfasst werden können, sodass dadurch auch die Zusammenhänge über die verschiedenen Abstraktionsebenen hergestellt werden können.

In diesem Kapitel wird daher ein Ansatz vorgestellt, der drei grundlegende Beiträge liefert. Zum einen werden Konzepte vorgestellt, um den *Differenzierungsprozess automatisiert durchführen* zu können. Die Ergebnisse hieraus können dann zum anderen für eine anschließende *Restrukturierung* der Simulink-Modelle und zur *Variabilitätsmodellierung* verwendet werden. Auf diese Weise wird der Wiederverwendungsgrad der Modelle deutlich erhöht.

Die festgelegte Zielsetzung bringt eine Reihe weiterer Fragen mit sich, die in diesem Zusammenhang ebenfalls beantwortet werden müssen. So stellen sich die Fragen, was alles in einem Simulink-Modell variieren kann, in welcher Werkzeugumgebung die Differenzierung durchgeführt werden soll, wie die Ergebnisse repräsentiert werden etc. Diese und weitere Fragen werden im folgenden Abschnitt behandelt.

6.1.1. Herausforderungen und Anforderungen

Die Ideen zur automatischen Differenzierung und Restrukturierung der Simulink-Modelle durch Variabilitätsmechanismen sowie die anschließende Variabilitätsmodellierung führen zu einer Reihe von Herausforderungen, die zum Teil nahe liegend, zum Teil aber auch nicht direkt erkennbar sind. In diesem Abschnitt werden daher angefangen von der Kernidee der Differenzierung die unmittelbaren Herausforderungen und die daraus resultierenden Anfoderungen beschrieben.

6.1.1.1. Differenzierung von Simulink-Modellen

Eine Differenzierung zwischen zwei Simulink-Modellen ist oftmals sehr schwierig, da die Modelle enorm groß werden und somit für einen Menschen nur mühevoll zu untersuchen sind. Auf diese Weise wird die Analyse sehr langwierig und extrem fehleranfällig. Daher stellt sich an dieser Stelle die Frage, wie dieser Differenzierungsprozess beschleunigt werden und zudem stabilere Ergebnisse liefern kann.

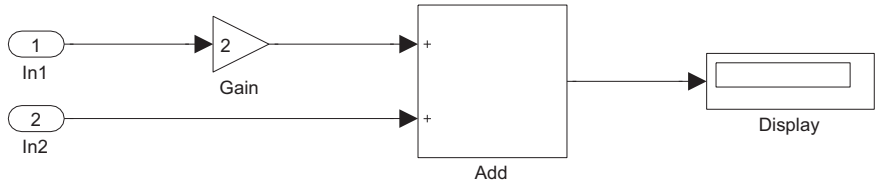
Als mögliche Lösung für dieses Problem bietet es sich an, die Gemeinsamkeiten und Unterschiede *durch algorithmische Verfahren automatisch zu detektieren*. Auf diese Weise kann die Analyse deutlich schneller durchgeführt und gefestigte Resultate ermittelt werden [Wes91].

Als Anwendungsfall sei das folgende Beispiel betrachtet: In Abbildung 6.1(a) und in Abbildung 6.1(b) sind zwei Simulink-Modelle, `model1.mdl` und `model2.mdl`, zu sehen. Das zweite Simulink-Modell `model2` ist durch inkrementelle Realisierung aus dem ersten Modell heraus entstanden. Hierbei wurde der `Add`-Block aus `model1` durch einen `Product`-Block in `model2` ersetzt. Zudem wurde der `Display`-Block aus `model1` entfernt und ein Ausgabeport `Out1` hinzugefügt, der mit dem Ausgabeport des `Product`-Blocks verbunden wurde.

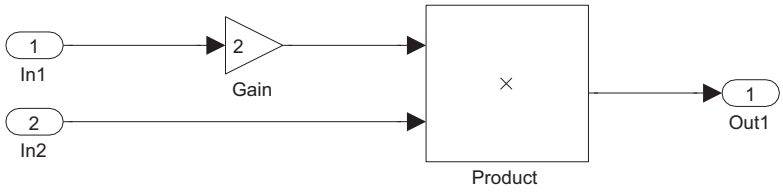
Durch eine automatische Differenzierung werden folgende Erkenntnisse gewonnen: Zum einen hat `model1` keine Ausgabeschnittstelle, wie es in `model2` der Fall ist. Der `Display`-Block ist ebenfalls nicht in `model2` vorhanden. Außerdem werden zwei verschiedene Blocktypen, `Add` und `Product`, verwendet, die wiederum ihre Ergebnissignale an verschiedene Ziele senden. Alle weiteren Teile der Modelle sind gleich.

6.1.1.2. Simulink-Metamodell

Der gewünschte Differenzierungsalgorithmus soll für weitere graphbasierte und datenflussorientierte Beschreibungssprachen eingesetzt werden können, da die



(a) Das Simulink-Modell `model1.mdl`



(b) Das Simulink-Modell `model2.mdl`

Abbildung 6.1.: Zwei Simulink-Modelle, die durch inkrementelle Realisierung entstanden sind

Identifizierung von Gemeinsamkeiten und Variabilität kein auf Simulink-Modelle isoliertes Problem ist, sondern weitere domänenspezifische Sprachen betrifft. Zum Beispiel könnten die Konzepte auch auf Funktionsnetze angewendet werden. Zudem sollen die Ergebnisse aus der Differenzierung durch einfache Anbindung an weitere externe Werkzeuge, wie etwa an das in Kapitel 4 beschriebene Variabilitätsmodellierungswerkzeug, weiterverwendet und -verarbeitet werden können. Daher ist es nicht geeignet, die Differenzierung innerhalb der Matlab-Umgebung durchzuführen, da hierdurch eine feste Anbindung an Matlab geschaffen wird.

Damit der Differenzierungsalgorithmus dennoch Simulink-Eingaben akzeptieren kann, Bedarf es also eines Abbilds der ursprünglichen Simulink-Modelle. Zu diesem Zweck ist es angemessen, ein *Simulink-Metamodell* zu entwickeln, das die Konzepte von Simulink beschreibt und auf dessen Instanzen der Differenzierungsalgorithmus operieren kann. Somit wird die gewünschte Unabhängigkeit geschaffen, die aber trotzdem die Bearbeitung von Simulink-Modellen unterstützt. Des Weiteren können die Ergebnisse der Differenzierung einfacher in weiteren Werkzeugen verwendet werden.

6.1.1.3. Import von Simulink-Modellen

Aus den vorherigen Anforderungen hat sich ergeben, dass der Differenzierungsalgorithmus nicht direkt auf den Simulink-Modellen ausgeführt werden soll. Variationspunkte können demnach auch nicht in diesen Modellen identifiziert werden. Daher ist die *Überführung der Simulink-Modelle in das Abbildformat* ein wesentlicher Schritt. Hierfür ist ein *Importvorgang* erforderlich, der die Entitäten im Simulink-Modell auf Objektinstanzen des Simulink-Metamodells überführt.

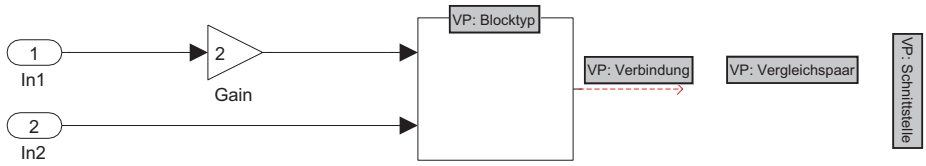


Abbildung 6.2.: Das Kommunalitätsmodell für `model1.mdl` und `model2.mdl`

6.1.1.4. Vergleichspaare in Simulink-Modellen

Es ist nicht die Aufgabe des Differenzierungsalgorithmus, zu entscheiden, welche Elemente eines Simulink-Modells mit den Elementen eines anderen Simulink-Modells verglichen werden sollen. Es fehlt also ein Konzept, das diese *Vergleichspaare aus den Simulink-Modellen ermittelt* und als zusätzliche Eingabe dem Differenzierungsalgorithmus übergibt. Ohne diese zusätzliche Eingabe ist die Differenzierung nicht ausführbar.

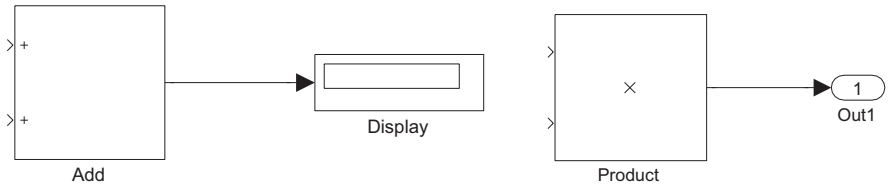
Es gibt mehrere Strategien zur Ermittlung von Vergleichspaaren. Zum Beispiel können diese interaktiv mit einem Benutzer, anhand von Metriken oder durch semantische Analysen bestimmt werden. Durch eine geeignete Strategie wird es möglich, den Differenzierungsalgorithmus mittels Vergleichspaaren korrekt anzuwenden.

6.1.1.5. Markierungen von Gemeinsamkeiten und Variabilität: Kommunalitätsmetamodell und Differenzmetamodell

Die durch den Differenzierungsalgorithmus detektierten Gemeinsamkeiten und Variationen müssen zu Dokumentations- und Visualisierungszwecken markiert werden. In Simulink sind hierfür allerdings keine Konzepte vorhanden. Ohne diese Konzepte können aber keine Markierungen von Variationspunkten und Varianten vorgenommen und daher auch keine aussagekräftigen Ausgaben erzeugt werden. Es ist daher erforderlich, das Simulink-Metamodell um Konzepte zur Markierung gemeinsamer und variabler Anteile zu erweitern. Die Erweiterungen können somit zur Beschreibung der Ergebnisse aus der Differenzierung herangezogen werden. Für die Beschreibung der Gemeinsamkeiten und der Variationspunkte wird ein *Kommunalitätsmetamodell* entwickelt. Varianten, die Variationspunkte binden, werden durch die Einführung eines *Differenzmetamodells* erfasst.

Als Anwendungsfall sei das folgende Beispiel betrachtet: In Abbildung 6.1 sind zwei Simulink-Modelle zu sehen. Gemeinsamkeiten und Variationen wurden bereits in Abschnitt 6.1.1.1 erläutert. Abbildung 6.2 veranschaulicht das Kommunalitätsmodell, welches über Konzepte zur Markierung von Variationspunkten verfügt und somit Variationen explizit erfassen kann. Die Variationspunkte sind in der Abbildung durch grau hinterlegte Rechtecke dargestellt, die jeweils an den Stellen im Modell platziert sind, an denen Variationen identifiziert wurden.

Nachdem nun Gemeinsamkeiten und variierende Punkte aus beiden Modellen anhand des Kommunalitätsmodells illustriert wurden, besteht der nächste Schritt in der Darstellung der Unterschiede bzw. Differenzen aus beiden Modellen. Abbildung 6.3



(a) Das Differenzmodell für `model1.mdl`

(b) Das Differenzmodell für `model2.mdl`

Abbildung 6.3.: Die Differenzmodelle für die beiden Simulink-Modelle

visualisiert diese Unterschiede in Form von zwei Differenzmodellen. Sie sind die Ausprägungen der im Kommunalitätsmodell definierten Variationspunkte. Abbildung 6.3(a) enthält das Differenzmodell für `model1.mdl` und Abbildung 6.3(b) das Differenzmodell für `model2.mdl`. Durch die Differenzmodelle werden nun Variationspunkte vollständig beschrieben und können zudem durch sie gebunden werden. Das erste Differenzmodell für `model1.mdl` bindet demnach folgende Variationspunkte:

- Variationspunkt am Blocktyp: Bindung durch den `Add`-Block.
- Variationspunkt am Vergleichspaar: Bindung durch den `Display`-Block.
- Variationspunkt an der Verbindung: Bindung durch die Verbindung zwischen dem Ausgabeport des `Add`-Blocks zum Eingabeport des `Display`-Blocks.

Das zweite Differenzmodell für `model2.mdl` bindet hingegen folgende Variationspunkte:

- Variationspunkt am Blocktyp: Bindung durch den `Product`-Block.
- Variationspunkt an der Modellschnittstelle: Bindung durch den Ausgabeport `Out1`.
- Variationspunkt an der Verbindung: Bindung durch die Verbindung zwischen dem Ausgabeport des `Product`-Blocks zum Ausgabeport `Out1` der Modellschnittstelle.

6.1.1.6. Export von Kommunalitätsmodellen

Weder für das Kommunalitätsmodell noch für das Differenzmodell existieren grafische Repräsentationen. Ohne eine angemessene Visualisierung ist die Analyse für einen Domänenexperten deutlich schwieriger. Daher stellt sich an dieser Stelle die Frage, wie die Ergebnisse aus der Differenzierung visualisiert werden können.

Eine geeignete Lösung für dieses Problem ist der *Export des Kommunalitätsmodells* in die Matlab-Umgebung, sodass die grafische Visualisierung von Simulink verwendet werden kann. Der wesentliche Vorteil hierbei ist, dass ein Benutzer somit in seiner bekannten und vertrauten Umgebung die Resultate untersuchen und somit deutlich schnellere Erkenntnisse erlangen kann.

Hierbei ist es nicht erforderlich, die Differenzmodelle ebenfalls zu exportieren, da alleine aus dem Kommunalitätsmodell die wichtigsten Informationen zur Restrukturierung der Modelle ermittelt werden können.

6.1.1.7. Repräsentation von Gemeinsamkeiten und Variabilität

Es gibt bisher noch keine Festlegung, wie Variationspunkte nach dem Exportieren dargestellt werden sollen. Ohne diese Darstellung können Variationspunkte nicht erkannt werden. Daher werden zu diesem Zweck alle Variationspunkte durch *farbige Markierungen und textuelle Beschreibungen* in Simulink dargestellt. Somit werden Variationspunkte hervorgehoben, sodass sie einfacher und schneller erkennbar sind.

6.1.1.8. Variationspunkte und Variabilitätsmechanismen

Variationspunkte und Varianten werden zwar durch den Differenzierungsalgorithmus identifiziert, es gibt allerdings noch keine Realisierung dieser durch Variabilitätsmechanismen. Ohne diese Mechanismen können Varianten nur schwer verwaltet werden. Zudem ist die systematische Wiederverwendung nicht möglich. Daher ist die *Untersuchung und Bewertung möglicher Variabilitätsmechanismen* in Simulink eine wichtige Voraussetzung, um sie geeignet einzusetzen.

6.1.1.9. Restrukturierung

Sind geeignete Variabilitätsmechanismen für die Realisierung von Variationspunkten festgelegt, besteht der nächste Schritt aus der entsprechenden Anpassung der Simulink-Modelle. Oftmals ist dies allerdings aufgrund der hohen Zahl und Verschiedenheit der Variationspunkte sehr schwierig. Daher wird dieser *Modernisierungsprozess durch Restrukturierungsregeln unterstützt*, die einem Benutzer den Prozess vereinfachen.

6.1.1.10. Variabilitätsmodellierung

Die Anforderungen zur Identifikation von Variationspunkten und ihre Realisierung durch Variabilitätsmechanismen sind bereits gestellt. Es fehlt allerdings noch die Durchführung der Dokumentation und Repräsentation der Variabilitätsinformationen. Ohne diese Informationen wird keine explizite Erfassung für den Entwicklungsprozess gewährleistet. Zudem gehen die Zusammenhänge aus den verschiedenen Abstraktionsebenen des Entwicklungsprozesses verloren. Daher wird zur Lösung dieses Problems das in Kapitel 4 vorgeschlagene *Variabilitätsmodell zur Erfassung der Variabilitätsinformationen* herangezogen.

6.1.2. Lösungsskizze

Nachdem die grundlegenden Anforderungen im vorangegangenen Abschnitt erläutert wurden, wird in diesem Abschnitt eine Lösung skizziert, die dann im weiteren

Verlauf dieses Kapitels detailliert wird. Die Lösungsideen basieren primär auf den Arbeiten [MN12] und [Men11] sowie den Masterarbeiten von *Youssef Arbach* [Arb11] und *Özgür Akcasoy* [zA11].

Abbildung 6.4 zeigt einen Grobentwurf bestehend aus allen Entitäten, die zur Differenzierung erforderlich sind. Links ist die Matlab-Werkzeugumgebung dargestellt. Im Speziellen ist die Simulink-Toolbox von Interesse. In Abschnitt 6.1.1.2 wurde der Bedarf ermittelt, Simulink-Modelle nicht innerhalb der Matlab-Umgebung zu differenzieren, sondern zu separieren, um den Algorithmus so generisch wie möglich zu halten. Auf diese Weise ist er auch für andere domänenspezifische Sprachen verwendbar. Dies erfordert demnach eine *Import-Funktionalität*, um Simulink-Modelle zu lesen und als Eingabe für die Differenzierung zu übergeben als auch eine *Export-Funktionalität*, um die Ergebnisse der Differenzierung in der Matlab-Umgebung grafisch zu visualisieren. In der Abbildung ist dies anhand der beiden beschrifteten Pfeile dargestellt.

Die Infrastruktur für die Eingaben und Ausgaben der Differenzierung setzt sich aus den drei *Metamodellen für Simulink-, Kommunalitäts- und Differenzmodelle* zusammen. Sie ist in der Abbildung unten rechts dargestellt. Die Metamodelle beschreiben die Konzepte aus den verschiedenen Modellen. Das Simulink-Metamodell ist eine Beschreibung für die importierten Simulink-Modelle. Das Kommunalitätsmetamodell enthält Erweiterungen, um Variationspunkte zu markieren. Das Differenzmetamodell beinhaltet schließlich die Variantendefinitionen. Sowohl das Kommunalitätsmetamodell als auch das Differenzmetamodell werden durch Erweiterung des Simulink-Metamodells entworfen. Daher besitzen sie eine Abhängigkeit zum Simulink-Metamodell. Des Weiteren muss das Differenzmetamodell stets seine Bezugselemente im Kommunalitätsmetamodell kennen, um ein gültiges Produkt erstellen zu können. Daher besteht auch eine Abhängigkeit vom Differenzmetamodell zum Kommunalitätsmetamodell.

Anhand dieser Basis kann die eigentliche Hauptaufgabe ausgeführt werden: die *Differenzierung*. Sie ist oben rechts in der Abbildung dargestellt. Dabei werden zwei Simulink-Modelle, die Instanzen des Simulink-Metamodells sind, differenziert und drei Ausgabemodelle erstellt. Ein Kommunalitätsmodell (Instanz des Kommunalitätsmetamodells) und zwei Differenzmodelle (Instanzen des Differenzmetamodells). Das Kommunalitätsmodell enthält die Gemeinsamkeiten aus beiden Simulink-Modellen und die Markierungen der Variationspunkte. Die Differenzmodelle hingegen beinhalten die variantenspezifischen Teile der Simulink-Modelle. Sie setzen dabei an die Variationspunkte im Kommunalitätsmodell an und spezialisieren diese um die spezifischen Eigenschaften.

Zusätzlich zu diesem Grobentwurf wird in Abbildung 6.5 der *Differenzierungsprozess* als eine weitere Sicht dargestellt. Hierdurch sind die einzelnen Vorgänge einfacher nachzuvollziehen. Das Importieren erzeugt bei der Eingabe von zwei Simulink-Modellen auch zwei interne Simulink-Modelle, die Instanzen des Simulink-Metamodells sind. Darauffolgend werden die Vergleichspaare in beiden Modellen festgelegt. Mit diesen Vergleichspaaren verfügt der Algorithmus über die Kenntnis, welche Elemente aus einem Modell mit den Elementen des anderen Modells

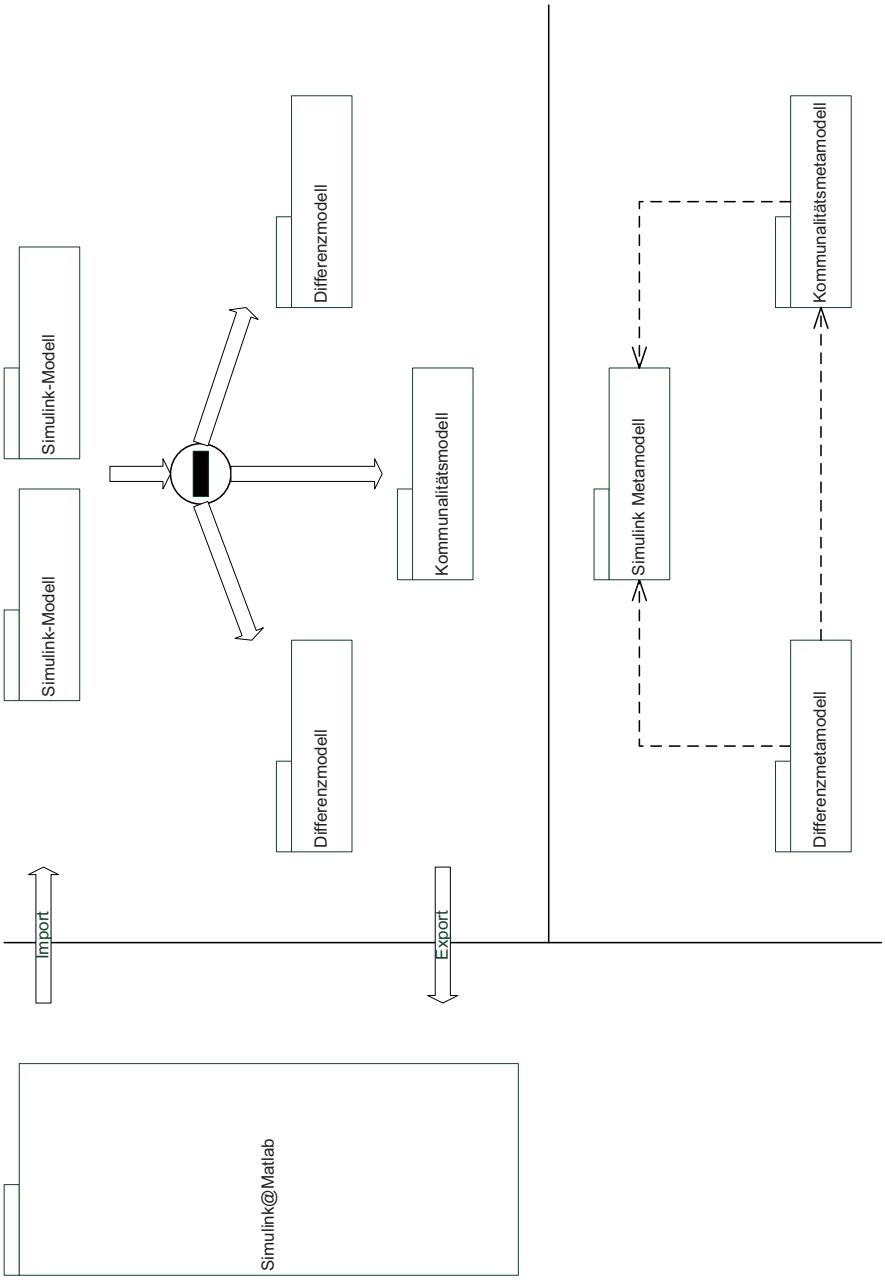


Abbildung 6.4.: Der Grobentwurf zur Differenzierung von Simulink-Modellen

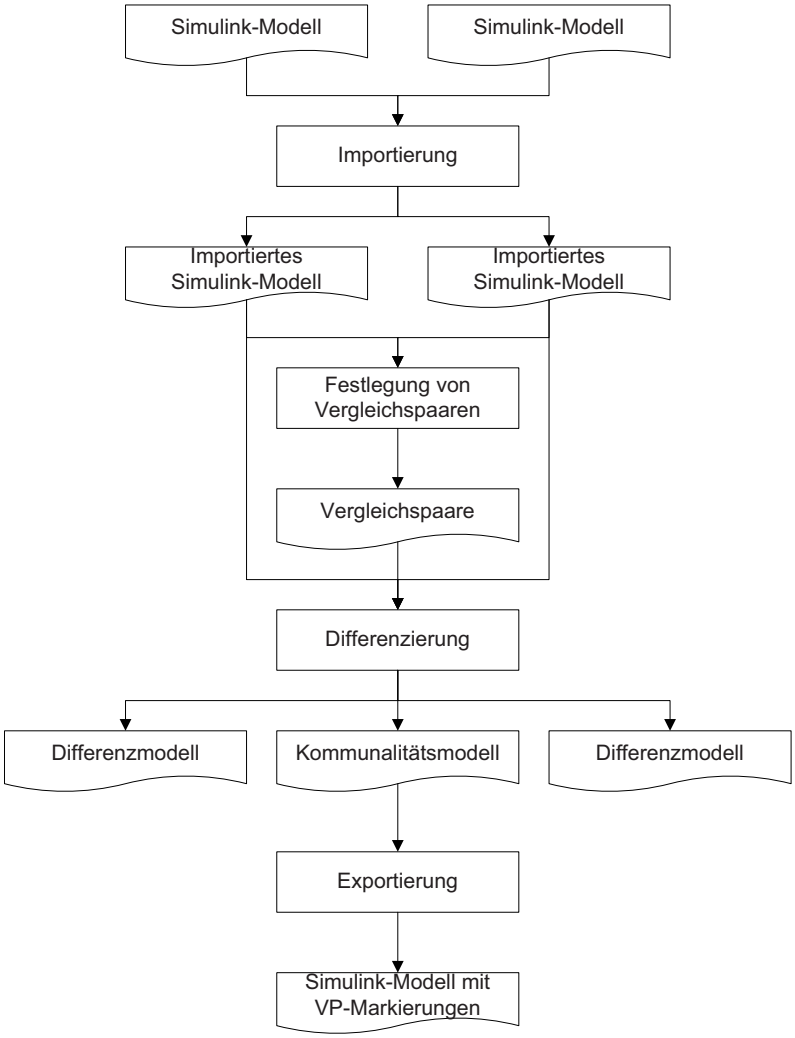


Abbildung 6.5.: Der Prozess zur Differenzierung von Simulink-Modellen

verglichen werden sollen. Je nachdem welche Strategie zur Ermittlung der Vergleichspaare eingesetzt werden, können verschiedene Datenformate erforderlich sein. Anhand dieser drei Eingaben, zwei Simulink-Modelle und den Vergleichspaaren, kann die Differenzierung stattfinden. Diese liefert das Kommunalitätsmodell sowie zwei Differenzmodelle als Ausgabe. Damit die identifizierten Gemeinsamkeiten und Variationspunkte visuell dargestellt werden können, wird in einem letzten Schritt das Kommunalitätsmodell zurück in die Matlab-Umgebung exportiert.

Die Visualisierung unterstützt einen Benutzer dabei, die Analyse einfacher und effizienter durchzuführen. Anhand dessen können die ursprünglichen Simulink-Modelle

manuell restrukturiert werden, um die identifizierten Variationspunkte durch Variabilitätsmechanismen zu kapseln. Die Eignung eines Variabilitätsmechanismus muss dabei anhand verschiedener Kriterien untersucht und bewertet werden. Neben Kriterien wie Benutzerfreundlichkeit oder Verständlichkeit spielen auch weitere Kriterien, wie etwa Bindezeiten und Codegenerierung, eine weitere wichtige Rolle. Sind Variabilitätsmechanismen eingesetzt, können Variationspunkte nun vollständig durch das Variabilitätsmodell aus Kapitel 4 beschrieben werden.

6.1.3. Struktur des Kapitels

Dieses Kapitel ist wie folgt strukturiert: In Abschnitt 6.2 werden zunächst die zugrunde liegenden Metamodelle beschrieben, um die Basis für die funktionalen Aspekte herzustellen. Hierbei werden das Simulink-Metamodell, das Kommunalitätsmetamodell und das Differenzmetamodell erläutert. Die funktionalen Anteile werden dann im Anschluss in Abschnitt 6.3 erläutert. Zu diesen gehören die Importierung der Simulink-Modelle, die Festlegung der Vergleichspaare, der Differenzierungsalgorithmus und schließlich die Exportierung des Kommunalitätsmodells. In Abschnitt 6.4 werden die erforderlichen Variabilitätsmechanismen in Simulink erörtert, bewertet und ein geeignetes ausgewählt, um die Modelle entsprechend umzustrukturieren. Zudem werden die Restrukturierungsmaßnahmen anhand der erzielten Differenzierungsergebnisse erläutert. Hierbei werden die grundlegenden Regeln definiert. Ein weiterer wichtiger Aspekt ist dabei die Dokumentation im Variabilitätsmodell. Abschnitt 6.5 illustriert dann ein Gesamtszenario am Beispiel des Fahrzeugzugangssystems. In Abschnitt 6.6 werden die wichtigsten Implementierungsaspekte der zuvor erläuterten Konzepte geschildert. Im Anschluss werden in Abschnitt 6.7 verwandte Arbeiten beschrieben, bewertet und mit dem Ansatz dieser Arbeit verglichen. Schließlich wird das Kapitel in Abschnitt 6.8 mit einer Zusammenfassung beendet.

6.2. Metamodellierung

Metamodelle stellen die Grundlage für die eigentliche Differenzierung dar. Sie werden im Folgenden genauer vorgestellt. Beginnend vom Simulink-Metamodell folgen das Kommunalitätsmetamodell und das Differenzmetamodell.

6.2.1. Simulink-Metamodell

Abbildung 6.6 illustriert das Metamodell für Simulink in Form eines Klassendiagramms. Die Klasse `RootModel` stellt die Wurzel der Kompositionshierarchie dar. Die Klasse `InnerModel` ist eine Entwurfsentscheidung, die die innere Struktur von Teilsystemen, wie etwa `Subsystem`-Blöcke (repräsentiert durch die abstrakte Klasse `Subsystems`), abbildet. Die abstrakte Klasse `Block` repräsentiert Blöcke aus der Simulink-Blockbibliothek, wie zum Beispiel `Add`, `BusCreator`, `Subsystem` oder `Constant`. In der Abbildung ist lediglich die Spezialisierung zu der Klasse `Subsystems` zu sehen. Im weiteren Verlauf dieses Abschnitts wird ein Konzept zur

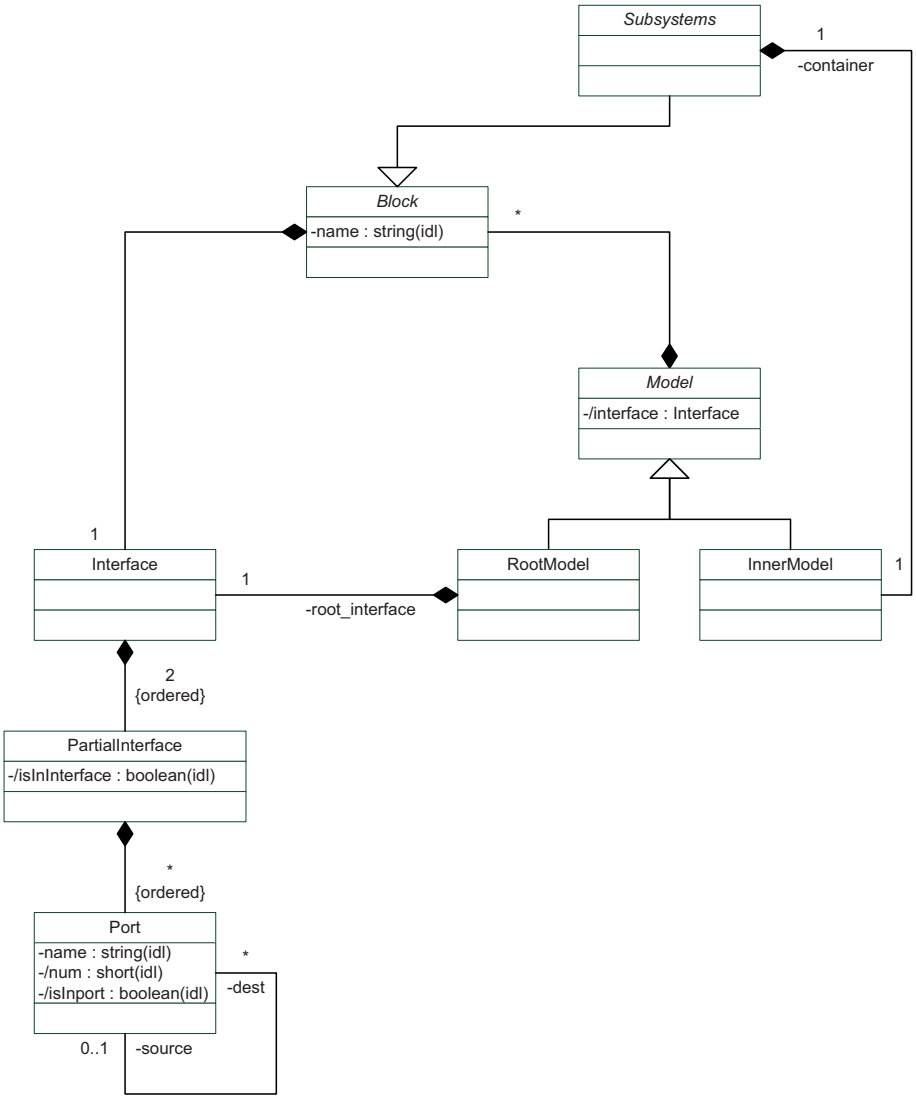


Abbildung 6.6.: Ein Metamodell für Simulink (Quelle: [Arb11])

Erweiterung des Metamodells um Sprachmittel für weitere Blockarten vorgestellt. Jeder Block hat mindestens einen Typ, einen Namen und eine Schnittstelle (`Interface`). Die Schnittstelle besteht dabei aus zwei Teilschnittstellen (`PartialInterface`): eine Ein- und eine Ausgabeschnittstelle. Der Index definiert die Rolle der Schnittstelle. Der Wert 0 stellt die Eingabeschnittstelle dar, der Wert 1 repräsentiert die Ausgabeschnittstelle. Weiterhin besteht eine Schnittstelle aus beliebig vielen Ports (`Port`). Die Unterscheidung zwischen Ein- und Ausgabeports wird an dieser Stelle nicht vorgenommen, da sie im Wesentlichen die gleichen Eigenschaften aufweisen. Eine Unterscheidung kann aber aus den Teilschnittstellen abgeleitet werden. Verbindungen zwischen Blöcken werden im Metamodell durch eine bidirektionale reflexive Assoziation an der Klasse `Port` abgebildet. Eine Verbindung besteht also aus einem Quell- (`source`) und beliebig vielen Zielports (`dest`). Des Weiteren sei noch anzumerken, dass die Schnittstelle einer inneren Struktur (`InnerModel`) durch den übergeordneten bzw. definierenden Block festgelegt wird. Daher ist das Attribut `interface` in der abstrakten Klasse `Model` eine abgeleitete Eigenschaft. Die Schnittstelle der Klasse `RootModel` wird hingegen explizit durch das Attribut `root_interface` festgelegt.

Schließlich sei zu beachten, dass die Sicherstellung der Korrektheit des Modells, wie zum Beispiel die Vermeidung einer Verbindung eines Ports mit sich selbst, weitestgehend nicht berücksichtigt wurde. Dies ist auch für die Zwecke der Einführung des Metamodells nicht erforderlich, da die Instanzen dieses Metamodells nicht manuell konstruiert werden, sondern aus den Simulink-Modellen importiert werden. Da bereits Simulink für die Korrektheit der Modelle Sorge trägt, ist eine redundante Realisierung an dieser Stelle nicht notwendig.

6.2.1.1. Erweiterungen für das Simulink-Metamodell

Das bisher vorgestellte Metamodell erfasst die wesentlichen Sprachkonzepte von Simulink. Es ist allerdings ersichtlich, dass dieses Metamodell alleine nicht ausreichen wird, um Simulink abzubilden. Simulink verfügt über eine umfangreiche Blockbibliothek, für die eine korrespondierende Beschreibung auf Metaebene existieren muss. Zu diesem Zweck wird die Vererbung als elementares Konzept für die Metamodellierung eingesetzt. Abbildung 6.7 illustriert dies für einen kleinen Ausschnitt der Simulink-Blockbibliothek. Die zentrale Andockstelle zum Metamodell ist die Klasse `Block`. Von hier aus wird über die Vererbungsbeziehung die Blockbibliothek abgebildet. In der Abbildung sind beispielhaft drei Teilbereiche in Form von abstrakten Klassen dargestellt: `LogicBitOperations`, `MathOperations` und `Subsystems`. Die konkreten Blöcke werden über einen weiteren Vererbungsschritt modelliert. Zum Beispiel repräsentieren die Klassen `LogicalOperator`, `Add` oder `Subsystem` konkrete Blöcke aus der Simulink-Blockbibliothek. In dieser Arbeit wurden nicht alle Blöcke im Metamodell repräsentiert. Sie ist allerdings soweit vollständig, dass der Differenzierungsalgorithmus geeignet verifiziert und evaluiert werden kann. Durch das hier vorgestellte Konzept der Vererbung ist es allerdings möglich, das Metamodell einfach zu erweitern.

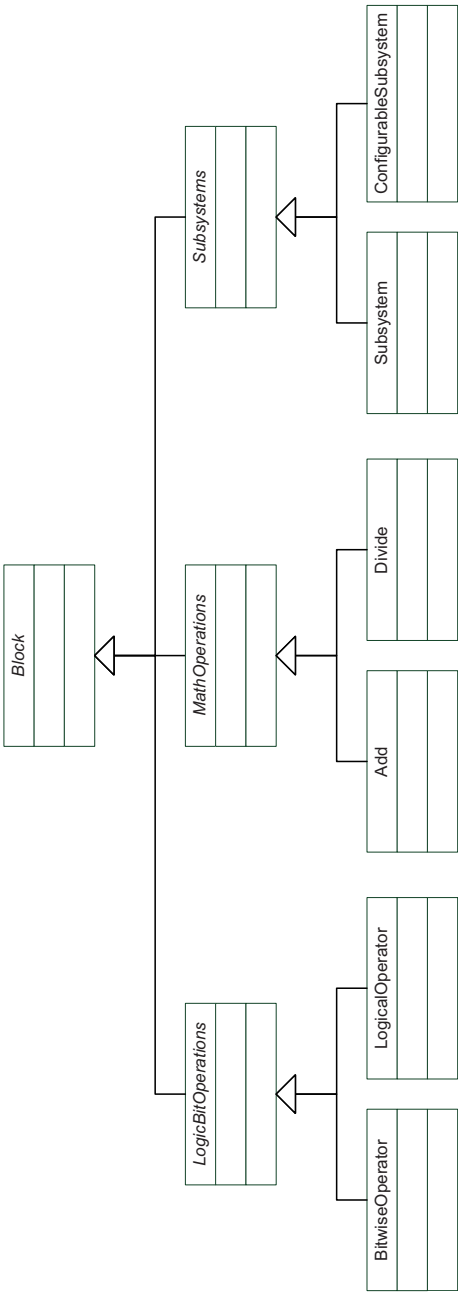


Abbildung 6.7.: Erweiterungen am Metamodell zur Erfassung sämtlicher Blockbibliotheken

6.2.1.2. Beispiel

Abbildung 6.8 und Abbildung 6.9 illustrieren die importierten Simulink-Modelle aus Abbildung 6.1(a) und Abbildung 6.1(b) in Form von Objektdiagrammen. Für jedes Modell wird eine Instanz vom Typ `SimulinkMetaModel :: RootModel` erzeugt. `model1` enthält eine Eingabeschnittstelle mit zwei Ports auf höchster Hierarchieebene (`In1` und `In2`). Demnach enthält das Objekt `model1` eine Schnittstelle `SimulinkMetaModel :: Interface` mit zwei Objekten vom Typ `Port`. Darüber hinaus besteht das Modell aus drei weiteren Blöcken: `Gain`, `Add` und `Display`. Dementsprechend werden drei Objekte vom Typ `SimulinkMetaModel :: Gain`, `SimulinkMetaModel :: Add` und `SimulinkMetaModel :: Display` erzeugt. Auch diese Blöcke verfügen über Schnittstellen. Der `Gain`-Block hat einen Ein- und Ausgabeport. Der `Add`-Block hat zwei Eingabeports und einen Ausgabeport. Schließlich hat der `Display`-Block einen Eingabeport. Entsprechend dieser Feststellung werden die Schnittstellen an den Objekten erzeugt. Die Verbindungen im Modell werden im Objektdiagramm durch die bidirektionale Assoziation an den Ports dargestellt.

Das zweite Modell, `model2`, besitzt eine Ein- und Ausgabeschnittstelle auf höchster Hierarchieebene. Dementsprechend wird für das Objekt `model2` eine Schnittstelle mit zwei Eingabeports und einen Ausgabeport erzeugt. Weiterhin besteht das Modell aus zwei Blöcken: `Gain` und `Product`. Hierfür werden ebenfalls Objekte der Typen `SimulinkMetaModel :: Gain` und `SimulinkMetaModel :: Product` erzeugt. Analog zu `model1` werden die Schnittstellen dieser Blöcke erzeugt und ihre Verbindungen hergestellt.

6.2.2. Kommunalitätsmetamodell

Das Kommunalitätsmetamodell dient zur Beschreibung gemeinsamer und variabler Simulink-Entitäten. Aus den Anforderungen aus Abschnitt 6.1.1 hat sich ergeben, dass die Markierung von Variationspunkten in diesem Zusammenhang den Analysevorgang erheblich unterstützt. An dieser Stelle stellt sich also die Frage, was alles in einem Simulink-Modell variieren kann, damit es gegebenfalls markiert wird. Werden erneut die Beobachtungen aus Abbildung 6.1 aufgegriffen, so können drei Ebenen von Variationspunkten in Simulink ausgemacht werden:

1. **Modellebene**: Sie betrifft die Eingabe- und Ausgabeschnittstelle der Modelle als auch die Anzahl der Blöcke in den Modellen.

2. **Blockebene**: Auf dieser Ebene können Blocktypen als auch die Blockschnittstellen variieren.

3. **Portebene**: Hier variieren die Verbindungen von Ports.

Folgendes Beispiel soll dies verdeutlichen: In Abschnitt 6.1.1.1 wurden bereits Variationspunkte für das Beispiel aus Abbildung 6.1 erläutert. Die Zuordnung dieser Variationspunkte auf die ermittelten Ebenen ist wie folgt:

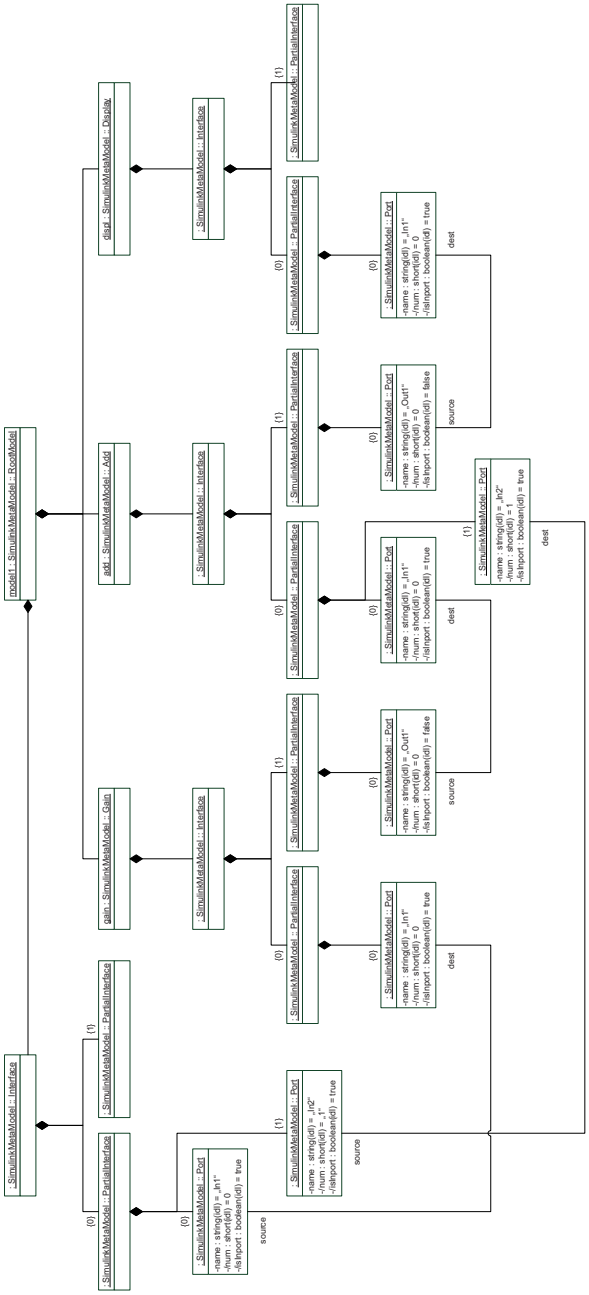


Abbildung 6.8.: Das Objektdiagramm des importierten Simulink-Modells `model1.mdl`

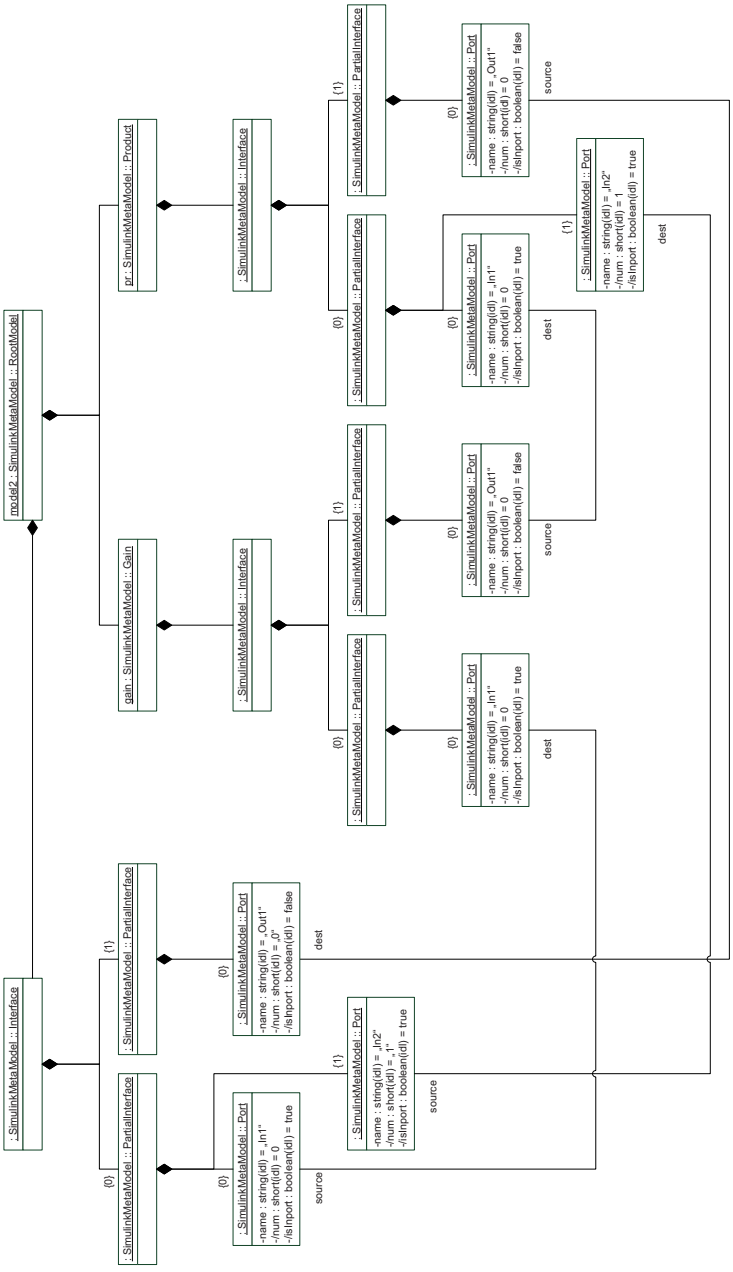


Abbildung 6.9.: Das Objektdiagramm des importierten Simulink-Modells `model2.mdl`

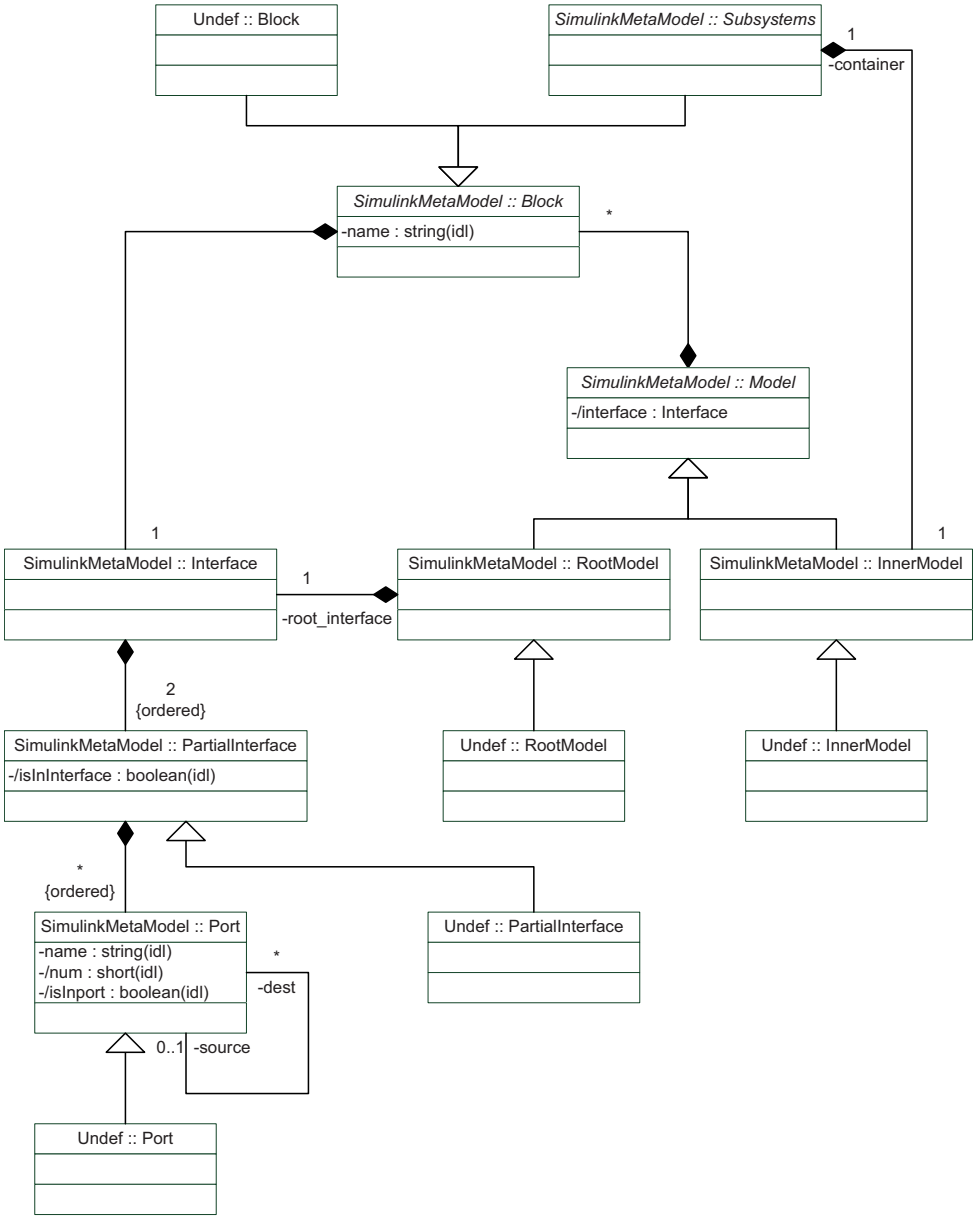


Abbildung 6.10.: Das Kommunalitätsmetamodell (Quelle: [Arb11])

1. Modellebene:
   - Variationspunkt an der Ausgabeschnittstelle, denn `model1` besitzt keine, aber `model2` hingegen verfügt über eine Ausgabeschnittstelle mit einem Ausgabeport `Out1`.
   - Variationspunkt an der Blockanzahl, denn der `Display`-Block ist in `model1` vorhanden, aber nicht in `model2`.
2. Blockebene:
   - Variationspunkt am Blocktyp, denn in `model1` wird ein `Add`-Block und in `model2` ein `Product`-Block eingesetzt.
3. Portebene:
   - Variationspunkt an der Verbindung, denn die Ausgabeports der korrespondierenden Blöcke `Add` und `Product` werden mit zwei verschiedenen Elementen verbunden. In `model1` wird sie mit dem Eingabeport eines `Display`-Blocks verbunden. In `model2` hingegen wird sie mit dem Ausgabeport der Modellschnittstelle verbunden.

Mit diesem Wissen ist es nun möglich, Variationspunkte im Kommunalitätsmetamodell an den geeigneten Stellen zu beschreiben. Abbildung 6.10 illustriert das Kommunalitätsmetamodell. Es beinhaltet die Konzepte zur Festlegung von Gemeinsamkeiten und Variationspunkten zwischen Simulink-Modellen. Da identifizierte Gemeinsamkeiten vollständig durch die Sprachmittel des Simulink-Metamodells ausgedrückt werden können, ist es an dieser Stelle auch von Vorteil, das Metamodell aus Abbildung 6.6 für diesen Zweck zu verwenden. Der Großteil des Kommunalitätsmetamodells aus Abbildung 6.10 besteht daher aus den bereits erläuterten Konzepten aus Abschnitt 6.2.1. Sie werden hier nicht noch einmal aufgegriffen.

Den wesentlichen Unterschied machen die Konzepte zur Identifizierung von Variationspunkten aus. Wie bereits erläutert, gibt es drei Ebenen, in denen Variationen auftreten können: (1) auf Modellebene, (2) auf Blockebene und (3) auf Portebene. Die Grundidee ist nun, Variationspunkte auf allen Ebenen markieren zu können. Hierfür wird die Vererbungsbeziehung an allen Stellen eingesetzt, an denen Variationspunkte auftreten können. In Abbildung 6.10 sind dies folgende Klassen:

- `SimulinkMetaModel :: RootModel`,
- `SimulinkMetaModel :: InnerModel`,
- `SimulinkMetaModel :: Block`,
- `SimulinkMetaModel :: PartialInterface` und
- `SimulinkMetaModel :: Port`.

Jede dieser Klassen wird spezialisiert und als Marke für Variationspunkte verwendet. Entsprechend sind dies die folgenden Klassen:

- `Undef :: RootModel`,
- `Undef :: InnerModel`,
- `Undef :: Block`,
- `Undef :: PartialInterface` und
- `Undef :: Port`.

Es sei darauf hingewiesen, dass dieses Konzept zur Markierung von Variationspunkten ein flaches bzw. lokales Konzept ist. Damit ist gemeint, dass wenn zum Beispiel ein Block nicht als Variationspunkt markiert wird, dennoch seine Ports variabel sein können.

6.2.2.1. Parallele Vererbungshierarchien

In dem bisher dargestellten Ansatz wird auf Blockebene lediglich die Variation am Blocktyp auf höchster Abstraktionsebene markiert. Auf diese Weise können allerdings sämtliche Informationen verloren gehen, die den Typ und die Eigenschaften der variierenden Blöcke betreffen. Zur Illustration sei erneut Abbildung 6.7 betrachtet. Angenommen es wurde ein Variationspunkt an einem Blocktyp erkannt. Die Blocktypvarianten sind beispielsweise `Add` und `Divide`. Mit dem bisherigen Ansatz wird ein Variationspunkt durch eine Undef-Spezialisierung der Klasse `Block` ausgedrückt (vgl. Abbildung 6.10). Um jedoch Maximalität in Bezug auf gemeinsamen Typ und gemeinsame Eigenschaften zu erreichen, wäre die Markierung eines Variationspunktes durch eine Undef-Spezialisierung der Klasse `MathOperations` an dieser Stelle angemessener. Es sind also Markierungen von Variationspunkten auf feineren Abstraktionsebenen erforderlich.

Durch diese Anforderung stellt sich aber sofort die Frage, für welche Klassen ein entsprechendes Undef-Gegenstück erforderlich ist. Im Prinzip muss nicht jede Klasse ein Gegenstück haben. Zum Beispiel gilt dies für die Blätter der Vererbungshierarchie aus Abbildung 6.7. Wenn zwei Blöcke vom gleichen Typ miteinander verglichen werden (zum Beispiel zwei `Add`-Blöcke), ist sofort ersichtlich, dass es keine Variation am Blocktyp geben kann. Daher ist auch keine korrespondierende Undef-Klasse erforderlich. Weiterhin benötigen Klassen, die nur eine Spezialisierung haben, ebenfalls kein Gegenstück. In dieser Arbeit wird aber aus Gründen der Symmetrie, Erweiterbarkeit und Verständlichkeit für jede Klasse aus der Vererbungshierarchie für Blocktypen eine korrespondierende Undef-Klasse modelliert. Diese Undef-Klasse wird stets eine konkrete Klasse sein, unabhängig davon, ob die ursprüngliche Klasse abstrakt oder konkret ist. Zum Beispiel würde die abstrakte Klasse `SimulinkMetaModel :: MathOperations` durch eine konkrete Klasse `Undef :: MathOperations` spezialisiert werden. Diese ist das entsprechende Gegenstück.

Eine weitere Fragestellung, die obige Anforderung mit sich bringt, ist, inwiefern die Vererbungshierarchie der Blocktypen für die Undef-Gegenstücke beibehalten werden muss. Grundsätzlich wäre eine flache Hierarchie vollkommen ausreichend.

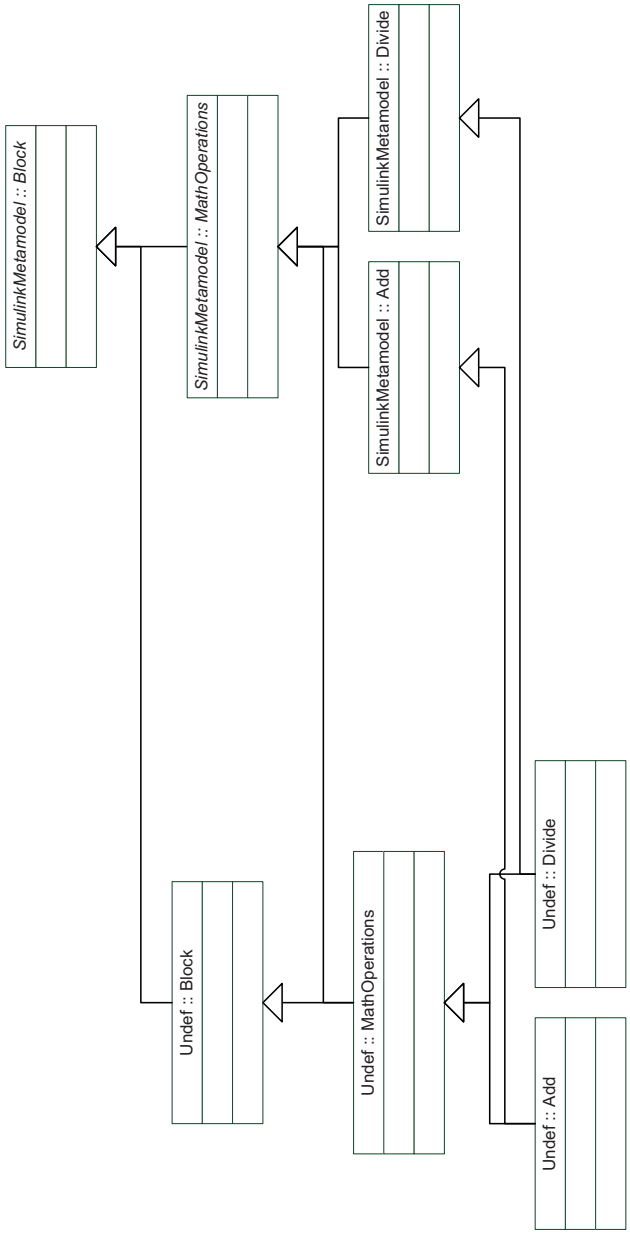


Abbildung 6.11.: Aufbau einer parallelen Vererbungshierarchie für Undef-Klassen

Allerdings wird in dieser Arbeit die Hierarchie der Blocktypen aus Gründen der Symmetrie, Erweiterbarkeit und Verständlichkeit für die Undef-Vererbungshierarchie parallel aufgebaut.

Abbildung 6.11 illustriert das Ergebnis aus den obigen Erläuterungen am Beispiel der Klasse `MathOperations`. Für jede Klasse `SimulinkMetaModel :: XYZ` gibt es ein entsprechendes Gegenstück `Undef :: XYZ` als Spezialisierung. Weiterhin ist ersichtlich, dass die Vererbungshierarchie beibehalten wurde. Wenn nun zwei Blöcke verschiedener Typen miteinander verglichen werden, ist der resultierende Blocktyp das Undef-Gegenstück des ersten gemeinsamen Vorahnens beider Blocktypen. Für die zwei Blocktypen `Add` und `Divide` ist der gemeinsame Vorahne die Klasse `SimulinkMetamodel :: MathOperations`. Dessen Gegenstück ist die Klasse `Undef :: MathOperations`. Bei einem möglichen Variationspunkt wäre also diese Klasse die entsprechende Marke.

Schließlich sei zu beachten, dass die hier vorgestellte Erweiterung auf Mehrfachvererbung basiert (zum Beispiel erbt die Klasse `Undef :: MathOperations` von den Klassen `Undef :: Block` und `SimulinkMetaModel :: MathOperations`). Das Problem der Mehrfachvererbung wird in dieser Arbeit auf Implementierungsebene auf die Einfachvererbung reduziert, indem für jede Klasse eine Interface- und eine Implementierungsklasse eingeführt wird. Für die Beschreibung des Metamodells führt die Mehrfachvererbung demnach zu keinerlei Nachteilen.

6.2.2.2. Beispiel

In Abbildung 6.2 ist das Kommunalitätsmodell für die beiden Simulink-Modelle `model1.mdl` und `model2.mdl` angedeutet. Es ist zu sehen, dass Variation an der Ausgabeschnittstelle, am Vergleichspaar, am Blocktyp und an den Verbindungen herrscht. Abbildung 6.12 zeigt das entsprechende Kommunalitätsmodell in Form eines Objektdiagramms. In `model1` hat der `Display`-Block kein Gegenstück in `model2`. Daher ist das Kommunalitätsmodell vom Typ `Undef :: RootModel`. Des Weiteren wurde festgestellt, dass es für den Ausgabeport `Out1` aus `model2` kein Gegenstück in `model1` existiert. Daher wird im Kommunalitätsmodell für die Ausgabeschnittstelle ein Objekt vom Typ `Undef :: PartialInterface` erzeugt. Eine weitere Variation wurde beim Vergleich des `Add`- und `Product`-Blocks erkannt. Da beide Blöcke verschiedene Typen besitzen, wird im Kommunalitätsmodell ein Undef-Objekt erzeugt, der den ersten gemeinsamen Vorahnen darstellt. In diesem Fall wird also ein Objekt vom Typ `Undef :: MathOperations` erzeugt. Schließlich wurde erfasst, dass ein Variationspunkt an der Verbindung zwischen den beiden Vergleichspaaren `Add` und `Product` vorliegt. In `model1` wird an der Ausgabeschnittstelle von `Add` eine Verbindung zum `Display`-Block hergestellt. In `model2` hingegen wird an der Ausgabeschnittstelle von `Product` eine Verbindung zur Ausgabeschnittstelle des Modells (`Out1`) erzeugt. Daher wird im Kommunalitätsmodell ein Objekt vom Typ `Undef :: Port` erzeugt, der mit dem Ausgabeportobjekt des gemeinsamen Blockobjekts verbunden wird.

Die restlichen Elemente stellen Gemeinsamkeiten aus beiden Modellen dar. Daher werden die Typen dieser Objekte aus dem Simulink-Metamodell erzeugt. Zum Beispiel ist der `Gain`-Block eine Gemeinsamkeit. Daher wird im Kommunalitätsmodell ein Objekt vom Typ `SimulinkMetaModel :: Gain` erzeugt. Analog gilt dies für alle weiteren Blöcke, die Gemeinsamkeiten besitzen.

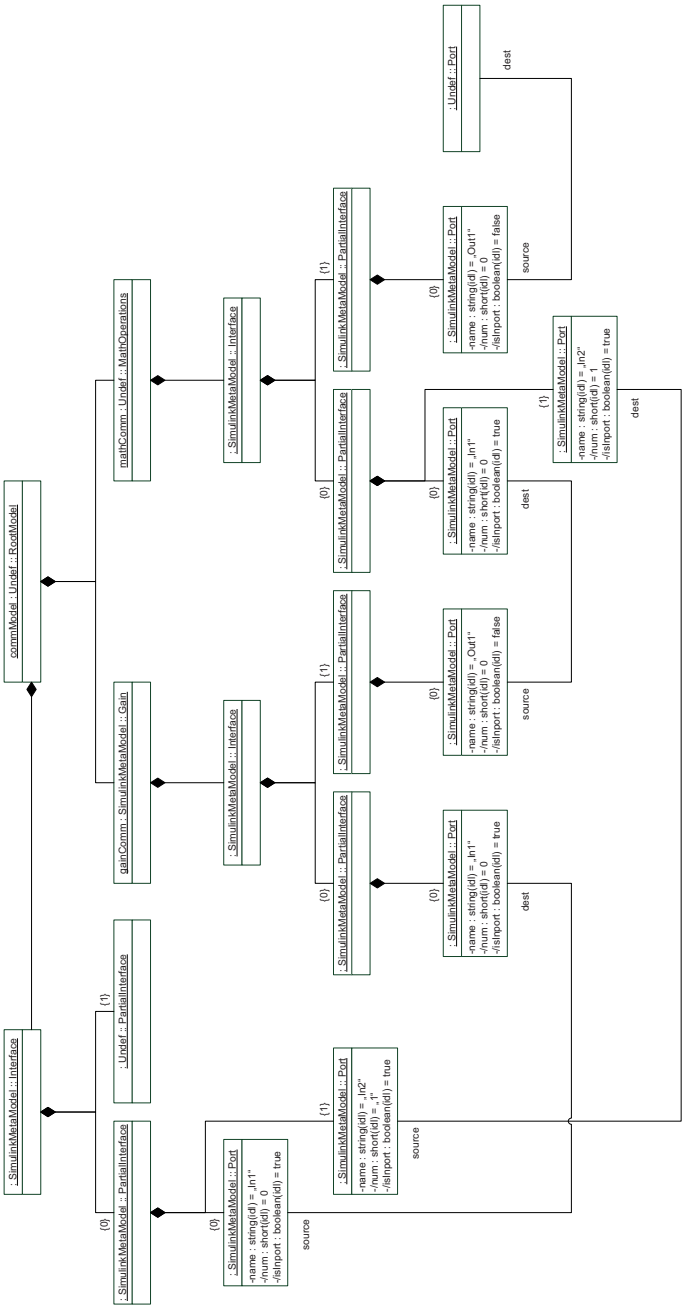


Abbildung 6.12.: Das Objektdiagramm des Kommunalitätsmodells für `model1.mdl` und `model2.mdl`

6.2.3. Differenzmetamodell

Abbildung 6.13 illustriert das Differenzmetamodell. Es ist eine Beschreibung der Varianten, die sich aus der Differenzierung zweier Simulink-Modelle ergeben. Wie bereits in Abschnitt 6.2.2 erläutert, werden Variationspunkte durch Instanziierung der entsprechenden Undef-Klassen markiert. Diese werden dann durch das Differenzmetamodell vollständig definiert. Zusätzlich werden im Differenzmetamodell jegliche Objekte definiert, die nicht durch das Kommunalitätsmetamodell erfasst werden können, weil sie keine Gemeinsamkeit darstellen, sondern modellspezifisch sind. Demnach gibt es zwei Arten der Variantendefinition im Differenzmetamodell:

1. Substitutionelle Variantendefinition

2. Inkrementelle Variantendefinition

Durch die *substitutionelle Variantendefinition* werden alle undefinierten Eigenschaften der Variationspunkte durch die spezifischen Eigenschaften ersetzt bzw. vervollständigt. Objekte des Kommunalitätsmodells exisitieren also auch im Differenzmodell und überschreiben bzw. erweitern Attribute. Daher scheint es angemessen, Klassen aus dem Simulink-Metamodell zu erben und als substitutionelle Variantendefinition zu verwenden. An dieser Stelle stellt sich die Frage, welche Konzepte hiervon betroffen sind. Wie bereits erwähnt, entstehen Variationspunkte auf Modell-, Block- und Portebene. Die Modellebene und die Blockebene beinhalten Variationspunkte, die Blöcke betreffen. Die Portebene bezieht Variationspunkte ein, die Ports und ihre Verbindungen einschließen. Aufgrund dessen ist es vollkommen ausreichend, das Konzept der substitutionellen Variantendefinition durch Spezialisierung der Klassen `SimulinkMetaModel :: Block` und `SimulinkMetaModel :: Port` abzudecken. In Abbildung 6.13 sind die entsprechenden Spezialisierungen die Klassen `Diff :: SubsBlock` und `Diff :: SubsPort`.

Inkrementelle Variantendefinitionen umfassen sämtliche spezifischen Objekte, die nicht im Kommunalitätsmodell markiert sind. Im Gegensatz zu substitutionellen Variantendefinitionen sind inkrementelle Variantendefinitionen vollständig definiert, sodass keine Ersetzung oder Erweiterung erforderlich ist. Daher werden im Differenzmetamodell keine neuen Konzepte zur Beschreibung von inkrementellen Variantendefinitionen eingeführt. Diese werden vollständig durch die Konzepte des Simulink-Metamodells abgebildet.

Das Differenzmodell ist für sich betrachtet in der Regel kein valides Simulink-Modell. Nur zusammen mit dem Kommunalitätsmodell ergibt es ein valides Simulink-Modell. Daher ist es für die Beschreibung des Differenzmetamodells essenziell, die Assoziation zum Kommunalitätsmodell herzustellen. Das Differenzmodell muss also das korrespondierende Kommunalitätsmodell kennen. Weiterhin müssen substitutionelle Variantendefinitionen die zugeordneten Blöcke, Ports und Verbindungen im Kommunalitätsmodell kennen. Inkrementelle Variantendefinitionen sind dagegen vollständig durch das Simulink-Metamodell beschrieben. Sie sind somit eindeutig. In Abbildung 6.13 werden die entsprechenden Verknüpfungen (1) durch die Klasse `Diff :: RootModel`, welches die Klasse `SimulinkMetaModel :: RootModel` spezialisiert

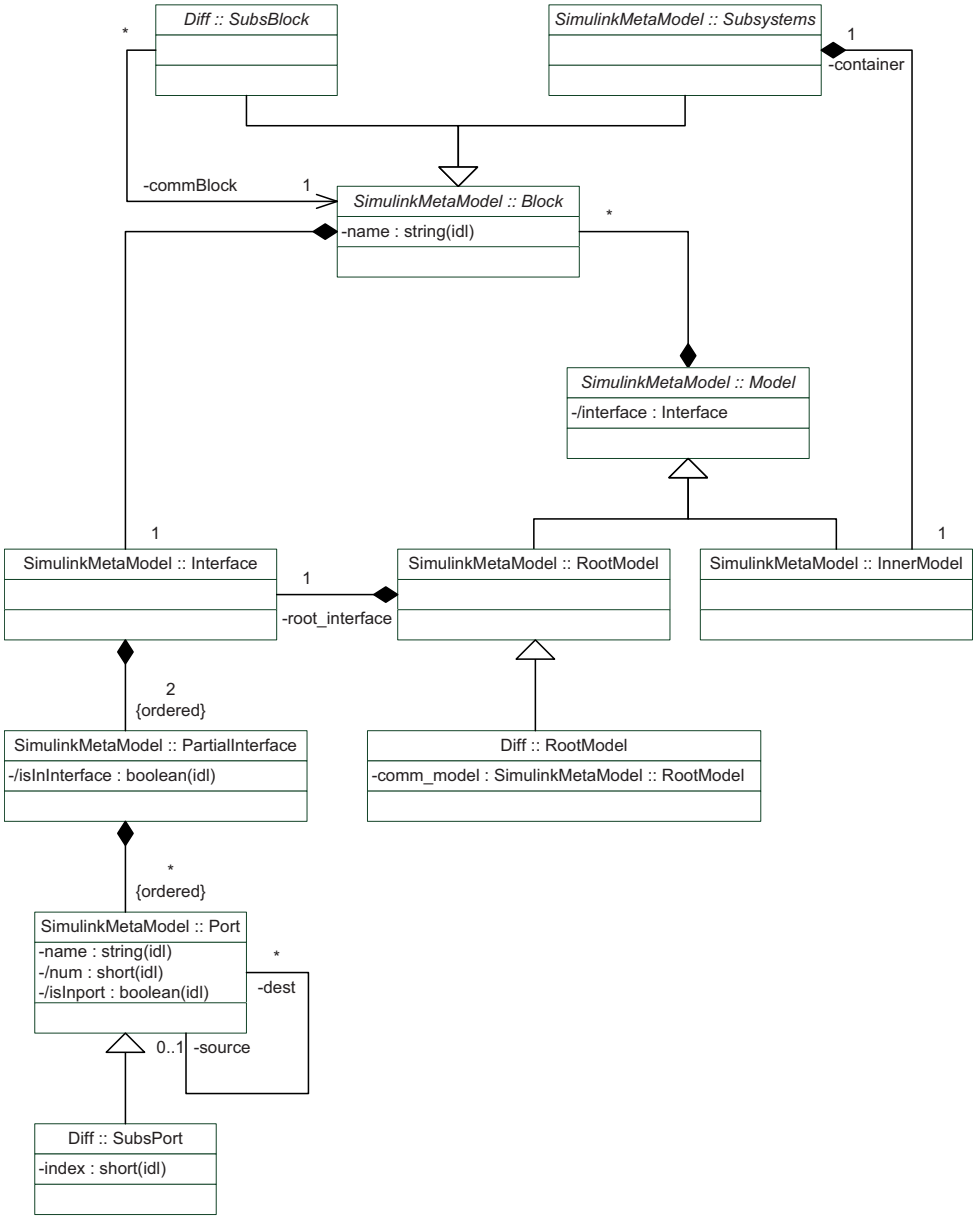


Abbildung 6.13.: Das Differenzmetamodell (Quelle: [Arb11])

und um eine Referenz zum Kommunalitätsmodell erweitert, (2) durch die Referenz `commBlock` der Klasse `Diff :: SubsBlock` und (3) durch das Attribut `index` der Klasse `Diff :: SubsPort` hergestellt.

6.2.3.1. Parallele Vererbungshierarchien

Im Differenzmetamodell sind Verknüpfungen zu Blöcken im Kommunalitätsmetamodell erforderlich, um Korrespondenzen herzustellen. Aufgrund der vorgeschlagenen Vererbungshierarchie stellt sich an dieser Stelle die Frage, ob eine weitere parallele Vererbungshierarchie aufgebaut werden sollte. Obwohl es, ähnlich wie bei den Undef-Klassen, nicht zwingend erforderlich ist, wird aus den gleichen Gründen dennoch eine derartige parallele Struktur vorgeschlagen. Auch muss nicht für jede Klasse ein entsprechendes Diff-Gegenstück modelliert werden, wie zum Beispiel für abstrakte Klassen. Aber auch in diesem Fall wird ein entsprechendes Diff-Gegenstück zu modelliert. Abbildung 6.14 illustriert die dadurch entstehende parallele Vererbungshierarchie für Diff-Klassen.

6.2.3.2. Beispiel

In Abbildung 6.3(a) und Abbildung 6.3(b) sind die beiden Differenzmodelle illustriert, die sich aus der Differenz von `model1` und `model2` ergeben. Die entsprechenden Instanzen aus dem Differenzmetamodell sind in Abbildung 6.15 und Abbildung 6.16 dargestellt. Die Modelle sind vom Typ `Diff :: RootModel`. Die Objekte beinhalten jeweils eine Referenz zum zugehörigen Kommunalitätsmodell (`comm_model`). Der Variationspunkt am Blocktyp wird im Differenzmodel für `model1` durch eine substitutionelle Variantendefinition erweitert. Hierbei wird ein Objekt vom Typ `Diff :: Add` erzeugt, das den Namen des Blocks sowie die Referenz zum Variationspunkt im Kommunalitätsmodell (`commBlock`) beinhaltet. Weiterhin wird der Variationspunkt am Vergleichspaar im Differenzmodell für `model1` durch die inkrementelle Variantendefinition spezifiziert. Zu diesem Zweck wird ein Objekt vom Typ `SimulinkMetaModel :: Display` erzeugt. Schließlich wird der Variationspunkt, der die Verbindung von Ports betrifft, durch die Erzeugung eines Ausgabeportobjekts vom Typ `Diff :: SubsPort` ersetzt. Dieser beinhaltet das Zielportobjekt, mit dem das Ausgabeportobjekt verbunden wird.

Das Differenzmodell für `model2` besitzt ebenfalls eine substitutionelle Variantendefinition. Diese betrifft erneut den Variationspunkt am Blocktyp. Hierfür wird ein Blockobjekt vom Typ `Diff :: Product` erzeugt, der genauso wie das `Diff :: Add`-Objekt in `model1` den Variationspunkt um den Namen sowie um die Referenz (`commBlock`) zum entsprechenden Block im Kommunalitätsmodell erweitert. Ebenfalls ist im Differenzmodell für `model2` eine inkrementelle Variantendefinition vorhanden, die den Variationspunkt an der Ausgabeschnittstelle des Modells betrifft. Hierfür wird ein Ausgabeportobjekt vom Typ `SimulinkMetaModel :: Port` erzeugt. Schließlich wird der Variationspunkt, der die Verbindung von Ports betrifft, durch die Erzeugung eines Ausgabeportobjekts vom Typ `Diff :: SubsPort` spezifiziert.

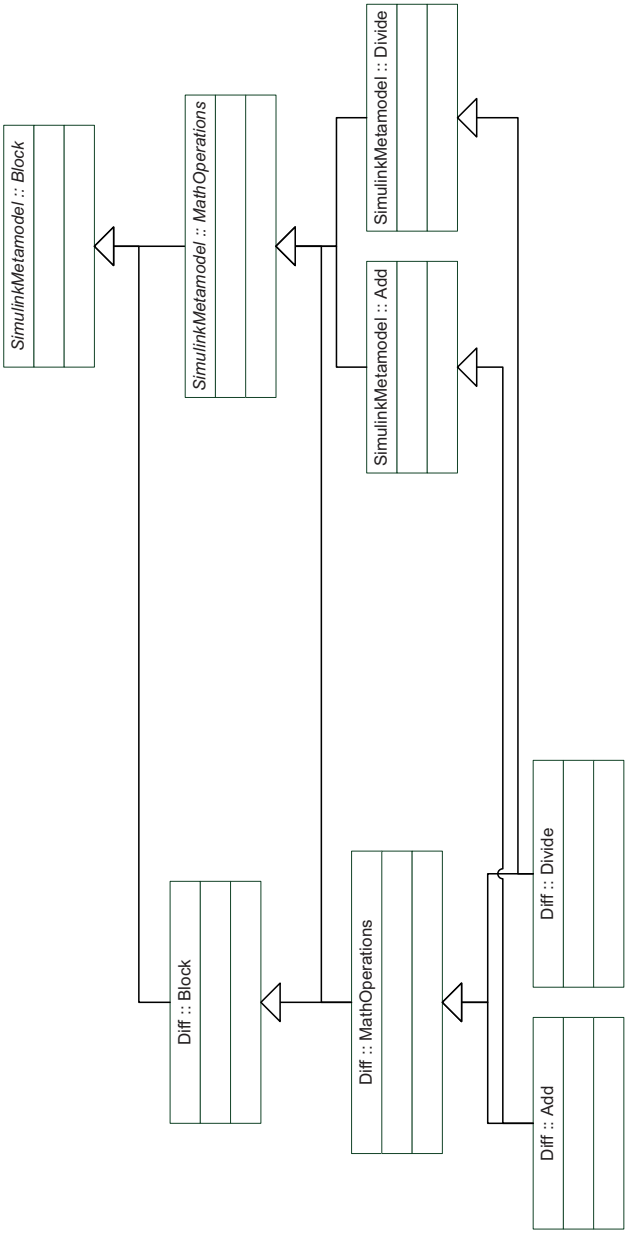


Abbildung 6.14.: Aufbau einer parallelen Vererbungshierarchie für Diff-Klassen

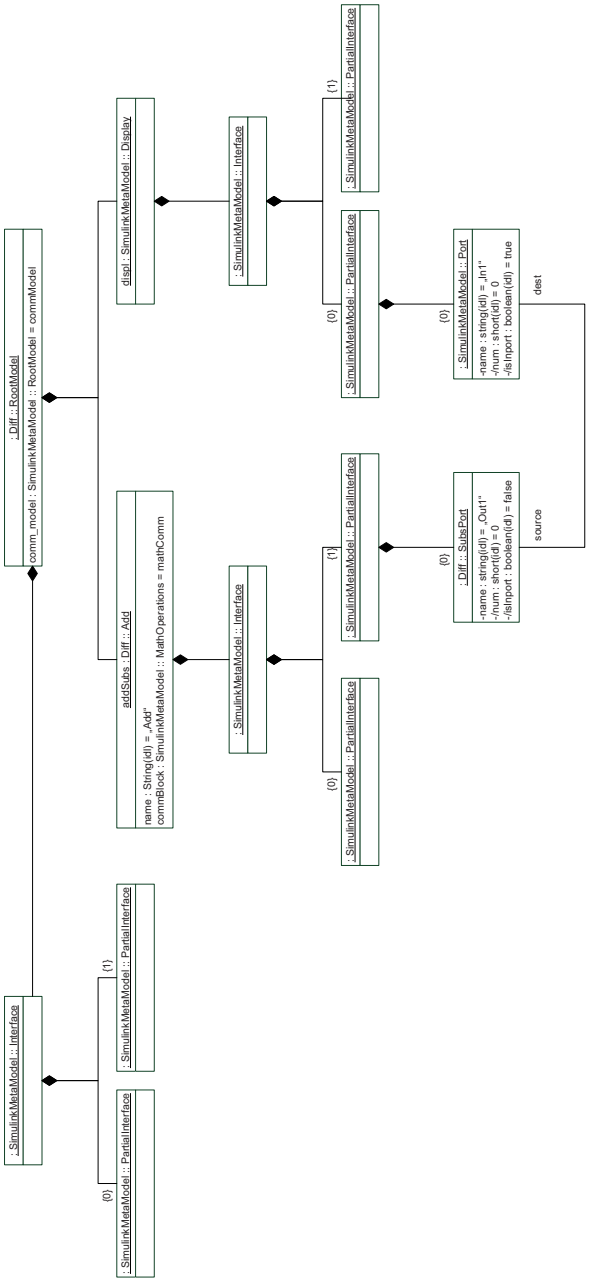


Abbildung 6.15.: Das Objektdiagramm des Differenzmodells für `model1.mdl`

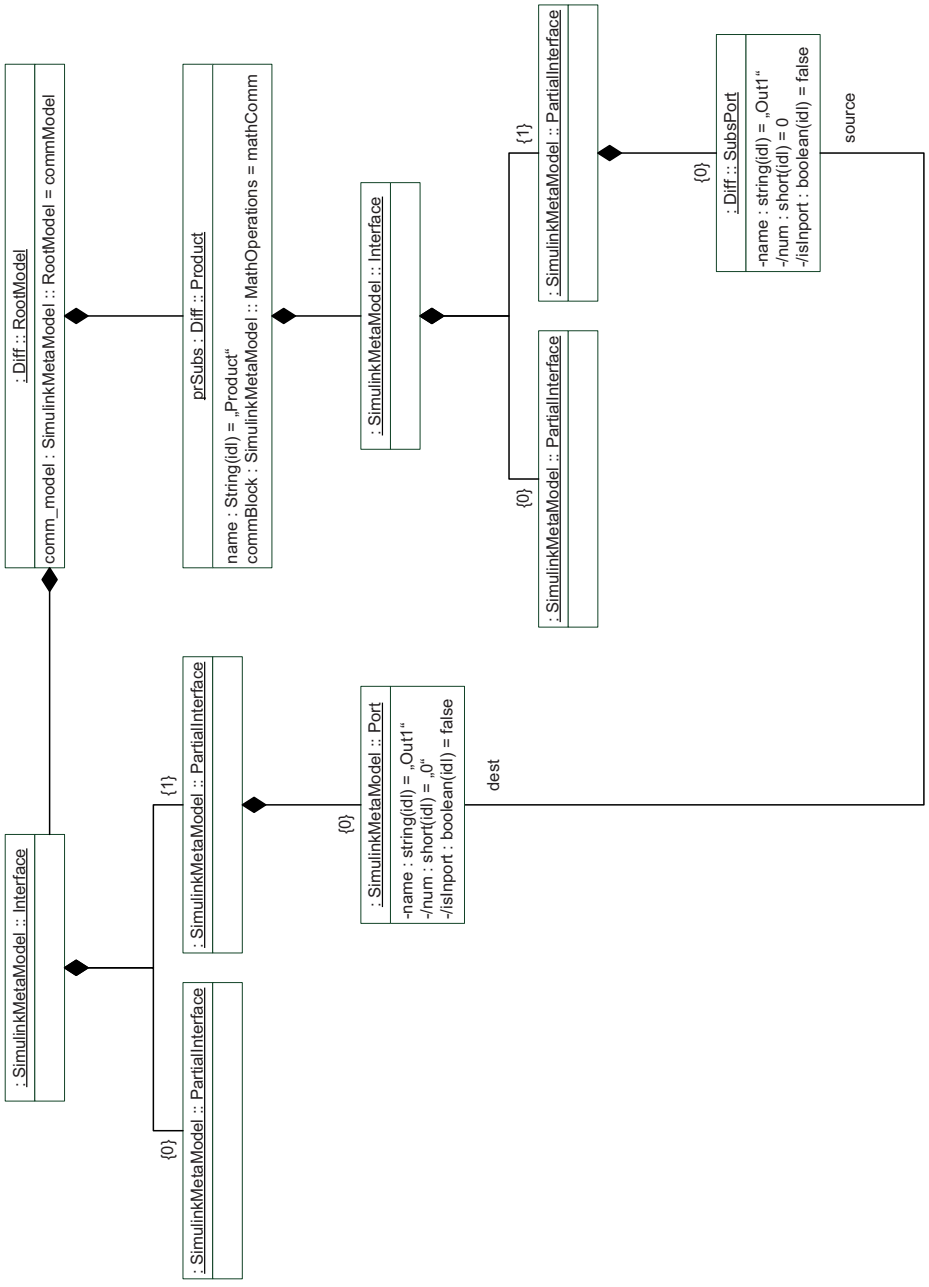


Abbildung 6.16.: Das Objektdiagramm des Differenzmodells für `model2.mdl`

Genau wie im Differenzmodel für `model1` enthält dieses Objekt das Zielportobjekt, mit dem es verbunden ist.

6.3. Differenzierung

Nachdem nun die Basis durch die Beschreibung der Metamodelle geschaffen wurde, wird im Folgenden der funktionale Aspekt des Konzepts näher erläutert. Dabei gibt es im Wesentlichen vier Kernfunktionalitäten: (1) die Importierung, (2) die Festlegung von Vergleichspaaren, (3) der Differenzierungsalgorithmus und (4) die Exportierung. Alle Aspekte werden in den folgenden Abschnitten genauer ausgeführt.

6.3.1. Import

Die Simulink-Modelle werden durch Importierung in die in Abschnitt 6.2.1 vorgestellte Form überführt. Sämtliche Operationen werden an dieser Repräsentationsform durchgeführt. Die Vorteile dieses Ansatzes sind vielfältig. Zunächst wird eine klare Trennung zwischen der Entwicklungsdomäne und ihrer Analyse hergestellt. Die Modelle bleiben somit unberührt und sind vor Veränderungen geschützt. Die Repräsentationsform ist zudem eine temporäre Darstellung. Sie wird nur zur Differenzierung verwendet und nie aufbewahrt. Daher kann es auch nicht zu Inkonsistenzen zwischen den Modellen kommen. Weiterhin sind funktionale Erweiterungen flexibler gestaltbar, da sie nicht von der Matlab-Umgebung beschränkt werden. Zum Beispiel ist eine Anbindung an das Variabilitätsmodell durch diesen Ansatz deutlich einfacher zu realisieren. Schließlich sei noch zu erwähnen, dass der Differenzierungsprozess somit nicht nur auf die Simulink-Modellierungssprache beschränkt ist, sondern durchaus auf andere domänenspezifische Sprachen angewendet werden kann.

Der Importvorgang erzeugt eine Instanz des Simulink-Metamodells. Für jeden Simulink-Block gibt es im Metamodell eine Klasse, die mit diesem Block korrespondiert. Beispielsweise wird für jede `.mdl`-Datei eine Instanz vom Typ `SimulinkMetaModel :: RootModel` erzeugt. Gleiches gilt auch für Blöcke, wie etwa `Port`, `Add` oder `BusCreator`. Für jeden dieser Blocktypen gibt es ein Abbild im Metamodell. Die Kompositionshierarchie wird über `Subsystem`-Blöcke aufgebaut, indem Instanzen vom Typ `SimulinkMetaModel :: InnerModel` erzeugt werden.

6.3.2. Festlegung von Vergleichspaaren

Durch diese Aktivität wird für die beiden importierten Simulink-Modelle eine Liste von Block- bzw. Portpaaren erstellt, die neben den Simulink-Modellen als weitere Eingabe für die Differenzierung dient. Abbildung 6.17 illustriert die wesentlichen Konzepte, die hierfür erforderlich sind.

Vergleichspaare für Blöcke werden durch die Schnittstellenklasse `Diff :: BlockMatcher` erstellt. Die Liste von Blockpaaren wird über die Klasse `Diff :: BlockMatch` verwaltet. Vergleichspaare für Ports werden hingegen durch die Schnittstellenklasse `Diff :: PortMatcher` erzeugt. Die Liste von Portpaaren werden wie-

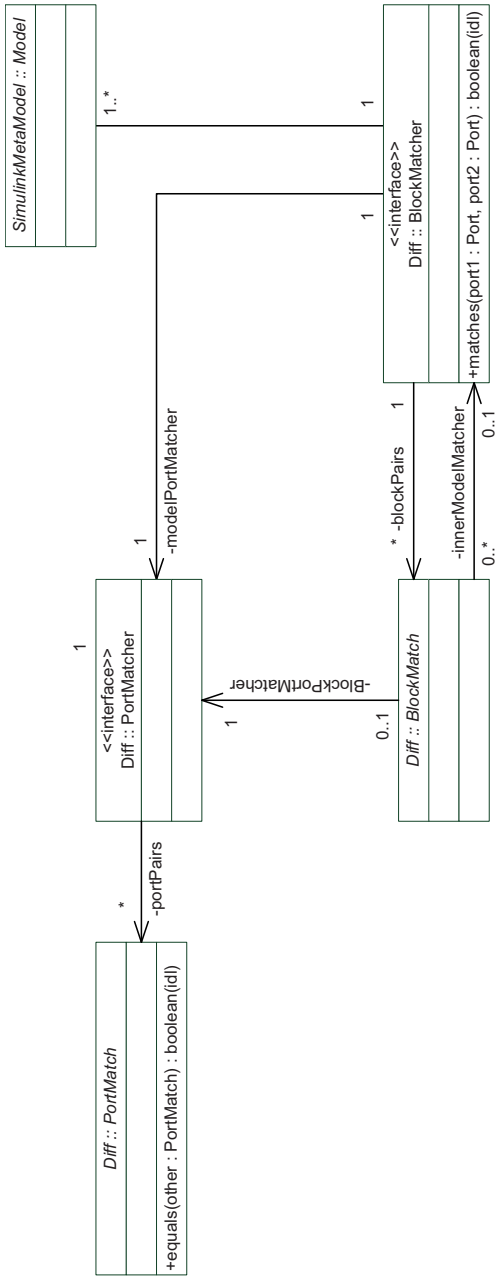


Abbildung 6.17.: Das Klassendiagramm zur Festlegung von Vergleichspaaren

derum anhand der Klasse `Diff :: PortMatch` administriert. Die Klassen `Diff :: BlockMatcher` und `Diff :: BlockMatch` kennen jeweils die Klasse `Diff :: PortMatcher`, um Portpaare auf höchster Hierarchieebene bzw. innerhalb von Blöcken zu erkennen. Für den Fall, dass die Blockpaare Subsysteme sind, kennt die Klasse `Diff :: BlockMatch` die Klasse `Diff :: BlockMatcher` für die innere Struktur des Blocks.

Eine wesentliche Frage hierbei ist, welche Strategie zur Ermittlung von Vergleichspaaren herangezogen werden kann. Hierbei gibt es sehr unterschiedliche Möglichkeiten. Zum einen kann ein interaktiver Ansatz mit einem Benutzer als mögliche Strategie eingesetzt werden. Dabei entscheidet der Benutzer, welche Elemente ein Vergleichspaar bilden. Eine andere Möglichkeit ist die automatische Erkennung von Paaren anhand verschiedener Metriken. Beispiele für derartige Metriken sind unter anderem:

- Name
- Name + Typ
- Name + Typ + Portindex
- Vordefinierte Muster
- Layoutinformationen
- ...

Darüber hinaus ist eine semantische Analyse eine weitere Möglichkeit, um Vergleichspaare zu identifizieren. Welche Strategie realisiert wird, hängt von mehreren Faktoren ab, die es zu bewerten gilt. Das Konzept ist allerdings so entworfen, dass eine Erweiterung oder Änderung einfach durchgeführt werden kann. Hierfür muss lediglich die Schnittstelle `Diff :: BlockMatcher` mit der gewünschten Strategie implementiert werden. Zudem muss die abstrakte Klasse `Diff :: BlockMatch` für die entsprechende Strategie erweitert werden, um die gewünschte Datenstruktur zu erhalten.

In dieser Arbeit wurde der interaktive Ansatz mit einem Benutzer sowie die automatische Erkennung von Vergleichspaaren anhand der Metriken Name, Typ und Portindex realisiert. Der wesentliche Vorteil des interaktiven Ansatzes ist die effiziente Ermittlung der Vergleichspaare durch einen Benutzer. Damit wird eine in der Regel komplexe Suche vermieden. Der Mehraufwand, den ein Benutzer hierdurch hat, ist vernachlässigbar klein, da der Benutzer typischerweise auch der Entwickler der Simulink-Modelle ist und die Gemeinsamkeiten der Modelle implizit kennt. Besonders effizient ist auch die automatische Erkennung anhand des Namens und des Typs, da in der Regel die Modelle durch Kopiervorgänge erweitert werden. Die Namen der einzelnen Elemente bleiben also im Allgemeinen unberührt.

6.3.3. Differenzierungsalgorithmus

Die Differenzierung ist in diesem Prozess die Kernaktivität, welche aus den zwei importierten Simulink-Modellen und einer Liste von Vergleichspaaren das Kommunalitätsmodell und zwei korrespondierende Differenzmodelle als Ergebnis liefert. Im Folgenden wird ein Algorithmus vorgestellt, der anhand der beschriebenen Eingaben die gewünschten Ausgaben automatisch erstellt.

6.3.3.1. Der Algorithmus

Der im Folgenden vorgestellte Algorithmus ist rekursiv gestaltet. Das heißt, dass der Algorithmus von der höchsten Hierarchieebene bis zur untersten Ebene in Form einer Tiefensuche die Differenzierung der Simulink-Modellelemente durchführt. Die Eingabe für den Algorithmus sind zwei Simulink-Modelle `model1` und `model2` sowie eine Liste von Vergleichspaaren. Die Ausgabe nach der Differenzierung sind ein Kommunalitätsmodell und zwei Differenzmodelle (vgl. Abbildung 6.5). Der Algoritmus durchläuft nach Eingabe der Modelle folgende Schritte:

1. Erzeuge ein Objekt `commModel` für das Kommunalitätsmodell. Das Objekt ist vom Typ `SimulinkMetaModel :: RootModel` bei Modellen auf höchster Hierarchieebene bzw. `SimulinkMetaModel :: InnerModel` in allen anderen Fällen genau dann, wenn für alle Blöcke in `model1` ein Korrelat in `model2` existiert. Dies wird anhand der Liste von Vergleichspaaren verifiziert. Falls nicht, dann ist der Typ des Objekts `Undef :: RootModel` bzw. `Undef :: InnerModel`.

2. Erzeuge zwei Objekte `diffModel1` und `diffModel2` für die jeweiligen Differenzmodelle. Die Objekte sind vom Typ `Diff :: RootModel`, wenn `model1` und `model2` Modelle der höchsten Hierarchieebene sind. Falls nicht, dann sind die Objekte vom Typ `SimulinkMetaModel :: InnerModel`.

3. Falls `model1` und `model2` Modelle der höchsten Hierarchieebene sind, dann:

   3.1 Erzeuge jeweils ein Schnittstellenobjekt für das Kommunalitätsmodell und für die zwei Differenzmodelle. Die Objekte sind vom Typ `SimulinkMetaModel :: Interface`.

   3.2 Erzeuge für die beiden Differenzmodelle `diffModel1` und `diffModel2` jeweils zwei partielle Schnittstellenobjekte vom Typ `SimulinkMetaModel :: PartialInterface`.

   3.3 Erzeuge für das Kommunalitätsmodell `commModel` zwei partielle Schnittstellenobjekte. Der Typ eines partiellen Schnittstellenobjekts ist `SimulinkMetaModel :: PartialInterface`, wenn für alle Ports in der partiellen Schnittstelle von `model1` ein Korrelat in `model2` existiert. Dies wird anhand der Liste von Vergleichspaaren verifiziert. Falls nicht, dann ist das partielle Schnittstellenobjekt vom Typ `Undef :: PartialInterface`.

   3.4 Differenziere in `model1` und `model2` zwischen allen Ports und seinen Verbindungen (vgl. Abschnitt 6.3.3.2).

4. Falls nicht, dann sind `model1` und `model2` Modelle aus tieferen Hierarchieebenen. In diesem Fall reicht es aus, nur die Konnektivität der Ports zu differenzieren, da sie bereits im vorherigen Schritt erzeugt und differenziert wurden.

5. Für alle Blockvergleichspaare `block1` aus `model1` und `block2` aus `model2`:

   5.1 Erzeuge ein Objekt `commBlock` für das Kommunalitätsmodell. Wenn `block1` und `block2` den gleichen Typ besitzen, dann hat auch `commBlock` diesen Typ. Falls nicht, dann hat `commBlock` den Typ des Undef-Gegenstücks des ersten gemeinsamen Vorahnens beider Typen.

   5.2 Erzeuge zwei Objekte `diffBlock1` und `diffBlock2` für die beiden Differenzmodelle. Der Typ des Objekts `diffBlock1` ist das Diff-Gegenstück des Typs von `block1`. Der Typ des Objekts `diffBlock2` ist das Diff-Gegenstück des Typs von `block2`.

   5.3 Differenziere zwischen den Namen von `block1` und `block2` (andere Eigenschaften sind auch möglich, falls vorhanden).

   5.4 Differenziere in `block1` und `block2` zwischen allen Ports und seinen Verbindungen. Hierbei werden nur die Portverbindungen nach außen betrachtet, also auf der höheren Hierarchieebene. Portverbindungen innerhalb der Blöcke (falls vorhanden) werden beim Differenzieren der inneren Struktur betrachtet, also im nächsten Rekursionsschritt.

   5.5 Wenn `block1` und `block2` Subsysteme sind (also eine tiefere Hierarchieebene besitzen), dann differenziere zwischen den inneren Hierarchieebenen dieser Blöcke.

   5.6 Wenn nur eins der Blöcke ein Subsystem ist, dann kopiere die innere Hierarchieebene des entsprechenden Blocks und füge es im Differenzmodell an den zugehörigen `diffBlock` hinzu.

6. Kopiere alle Blöcke, die kein Pendant im jeweils anderen Modell haben, zu den entsprechenden Differenzmodellen. Diese sind die inkrementellen Variantendefinitionen.

7. Verbinde alle Ports in den drei erzeugten Modellen `commModel`, `diffModel1` und `diffModel2`.

6.3.3.2. Die Subroutine zur Differenzierung von Ports und Verbindungen

Bei der Differenzierung von Ports und Verbindungen betrifft ein wesentlicher Aspekt die Frage, wann die differenzierten Verbindungen in den entsprechenden Modellen erzeugt werden sollen, da die Quell- bzw. Zielports in einem bestimmten Zustand des Algorithmus noch fehlen könnten. Um diese Situationen zu vermeiden, werden in diesem Ansatz sämtliche Verbindungen am Ende des Algorithmus erzeugt. Somit ist sichergestellt, dass der Quell- bzw. Zielport eines zu verbindenden Ports stets vorhanden ist. Diese Entscheidung führt allerdings zu der Situation, sämtliche

Verbindungsinformationen geeignet zu erfassen und bei Bedarf zur Verfügung zu stellen. Hierfür können zwei verschiedene Strategien eingesetzt werden:

1. Die Verbindungsinformationen werden am Ende des Algorithmus aus den ursprünglichen Eingabemodellen gelesen und im Anschluss werden jegliche Verbindungen aller Ausgabemodelle erzeugt.

2. Die Verbindungsinformationen werden während der Ausführung des Algorithmus bei jedem Überprüfungsschritt der Ports in eine temporäre Warteliste gespeichert. Am Ende des Algorithmus werden anhand dieser Liste sämtliche Verbindungen aller Ausgabemodelle erzeugt.

Die erste Alternative scheint zunächst angemessen zu sein, da es durchaus sinnvoll ist, die Verbindungsinformationen erst dann zu sammeln, wenn die Verbindungen erzeugt werden. Sie bewirkt allerdings eine unnötige redundante Überprüfung. Da die Verbindungsinformation eine Eigenschaft der Ports ist, ist es daher effizienter, diese immer dann zu speichern, wenn ein Port während der Differenzierung überprüft wird. Somit wird am Ende eine erneute Überprüfung vermieden. Der Algorithmus führt daher die zweite Strategie aus.

Ein weiterer wichtiger Aspekt bei der Verbindung von Ports ist das genaue Verständnis der Zusammenhänge zwischen Ports derselben Hierarchieebene sowie zwischen Ports aus zwei Hierarchieebenen. Werden Ports einer Hierarchieebene verbunden, so wird ein Eingabeport mit einem Ausgabeport verbunden und umgekehrt. Wird hingegen ein Port einer bestimmten Hierarchieebene mit einem Port der Subhierarchieebene verbunden, zum Beispiel mit `Subsystem`-Blöcken, so wird ein Eingabeport mit dem Eingabeport der Subhierarchie verbunden. Der Ausgabeport der Subhierarchie wird wiederum mit dem Ausgabeport der nächsthöheren Hierarchieebene verbunden. Bei der Erfassung der Verbindungsinformationen ist es daher wichtig, die Art der Verbindung zu kennen. Mit dieser zusätzlichen Information wird es möglich, den Algorithmus bei der Erzeugung von Verbindungen geeignet zu steuern.

Aus den oben genannten Aspekten ist es daher erforderlich, die Warteliste mit einer Reihe weiterer Informationen auszustatten, damit die Verbindungen in den Modellen ausschließlich durch die Warteliste hergestellt werden können. Ein Port wird demnach in eine Warteliste mit folgenden Informationen hinzugefügt:

- Der Port selbst.
- Die Information, ob der hinzugefügte Port mit einem Quell- oder Zielport verbunden werden soll.
- Der Portindex des Quell-/Zielports.
- Die Information, ob der Quell-/Zielport ein Eingabe- bzw. Ausgabeport ist.
- Der zugehörige Blockname des Quell-/Zielports. Diese Information kann auch leer sein, wenn es sich bei dem Quell-/Zielport um die Schnittstelle des Modells handelt und nicht um die eines Blocks.

- Das Modell, das den Quell-/Zielport enthält.

Nachdem nun unklare Aspekte verdeutlicht und entsprechende Strategien vorgeschlagen wurden, gilt es jetzt zu klären, welche Eingabe zur Ausführung dieser Subroutine erforderlich ist. Demnach sind folgende Elemente relevant:

- die zwei Modelle `model1` und `model2`,
- die zwei Schnittstellen `interface1` und `interface2` der zwei Modelle,
- die Schnittstelle `commInterface` des Kommunalitätsmodells,
- die Schnittstellen `diffInterface1` und `diffInterface2` der beiden Differenzmodelle,
- die Liste der Vergleichspaare und
- eine Warteliste `portsToConnect`.

Die Schritte zur Differenzierung der Ports und den Verbindungen wird daraufhin durch den folgenden Algorithmus bestimmt:

1. Für alle Vergleichspaare `port1` und `port2` der Modellschnittstellen `interface1` und `interface2`:
   - 1.1 Setze `handleInputSource := false`, wenn die Schnittstellen der Modelle betrachtet werden. Falls Blockschnittstellen betrachtet werden, dann setze `handleInputSource := true`.
   - 1.2 Deklariere eine Boolsche Variable `handleSource := handleInputSource XAND port1.isInport();`. Durch diese Variable wird ermittelt, ob die Vergleichspaare mit einem Quell- bzw. Zielport verbunden werden.
   - 1.3 Deklariere zwei Variablen `target1` und `target2` vom Typ Port mit `target1 := handleSource ? port1.source : port1.dest;` und `target2 := handleSource ? port2.source : port2.dest;`. Hierdurch wird festgelegt, ob es sich beim Quell- bzw. Zielport um einen Ein- bzw. Ausgabeport handelt.
   - 1.4 Erzeuge ein Portobjekt `commPort` für das Kommunalitätsmodell. Das Objekt ist vom Typ `SimulinkMetaModel :: Port`.
   - 1.5 Differenziere die Verbindungen der beiden Ports `port1` und `port2` durch Vergleich von `target1` und `target2` anhand der Liste von Vergleichspaaren.
     - 1.5.1 Falls es keine Übereinstimmung durch die Liste der Vergleichspunkte ergibt, dann:
       - 1.5.1.1 Erzeuge ein Portobjekt vom Typ `Undef :: Port`. Wenn `handleSource == false`, dann weise das erzeugte Objekt der Variable `dest` von `commPort` zu. Wenn `handleSource == true`, dann weise das erzeugte Objekt der Variable `source` von `commPort` zu.

1.5.1.2 Erzeuge zwei Objekte `subsPort1` und `subsPort2` vom Typ `Diff :: SubsPort` für die beiden Differenzmodellschnittstellen `diffInterface1` und `diffInterface2`.

1.5.1.3 Ermittle und setze den Index der zu vergleichenden Ports durch `subsPort1.index := port1.num;` und `subsPort2.index := port2.num;`.

1.5.1.4 Füge `subsPort1` und `subsPort2` in die Warteliste `portsToConnect` mit folgenden Informationen hinzu:

- `subsPort1`,
- `handleSource`,
- `target1.num`,
- `target1.isInport()`,
- `null`, wenn `target1` zur Schnittstelle von `model1` gehört. Falls nicht, dann `target1.name`,
- diffModel1.

Analog für `subsPort2`.

1.5.2 Falls es eine Übereinstimmung gibt, dann:

1.5.2.1 Falls `target1 == null` und `target2 == null` und wenn `handleSource == false`, dann deklariere in `commPort` die Variable `dest := null;`. Wenn `handleSource == true`, dann deklariere in `commPort` die Variable `source := null;`.

1.5.2.2 Falls nicht, dann füge `commPort` in die Warteliste `portsToConnect` mit den folgenden Informationen hinzu:

- `commPort`,
- `handleSource`,
- `target1.num`,
- `target1.isInport()`,
- `null`, wenn `target1` zur Schnittstelle von `model1` gehört. Falls nicht, dann `target1.name`,
- commModel.

2. Kopiere alle Ports, die kein Korrelat in der jeweils anderen Schnittstelle besitzen, zu den entsprechenden Differenzmodellen. Diese sind die inkrementellen Variantendefinitionen. Füge diese Ports in die Warteliste `portsToConnect` hinzu.

6.3.4. Export

Nach der Differenzierung entstehen drei Modelle: Zwei Differenzmodelle und ein Kommunalitätsmodell. Bei der Exportierung wird lediglich das Kommunalitätsmodell

Variationspunkt	Notation
Modellschnittstelle	Ports werden in der Farbe Grau markiert
Blöcke im Modell	Der Modellhintergrund wird in der Farbe Hellgrau markiert und eine Beschreibung hinzugefügt
Blocktyp	Der Name des Blocks wird im Italic-Font dargestellt und eine Beschreibung hinzugefügt
Blockschnittstelle	Die Umrandung des Blocks wird in der Farbe Blau markiert und eine Beschreibung hinzugefügt
Verbindungen von Ports	Die Ports werden mit einem neu erzeugten `Subsystem`-Block verbunden, dessen Name durch eine Zufallszahl erzeugt wird

Tabelle 6.1.: Die Notationsfestlegung der Variationspunkte in Simulink

zurück in das Simulink-Format exportiert. Die Differenzmodelle, also die Varianten, werden hierbei nicht exportiert. Der Grund hierfür ist, dass das Kommunalitätsmodell alleine vollständig ausreicht, den Analysevorgang zu unterstützen. Hier sind sowohl die Gemeinsamkeiten als auch Variationspunkte enthalten. Für eine anschließende Restrukturierung reicht dieses Modell also aus.

Um die aus der Differenzierung identifizierten Variationspunkte in Simulink zu kennzeichnen, werden spezielle grafische Notationen eingesetzt. Primär sind dies farbliche Markierung von Blöcken und Verbindungen. Zusätzlich hierzu werden auch textuelle Beschreibungen herangezogen, wenn nicht die gewünschte Detailinformation alleine durch die grafische Notation beschrieben werden kann. Zu diesem Zweck wird die `description`-Eigenschaft eines Blocks bzw. eines Modells in Simulink verwendet. Tabelle 6.1 fasst diesbezüglich die festgelegte Notation zusammen.

Es sei ausdrücklich darauf hingewiesen, dass auch das exportierte Kommunalitätsmodell einen repräsentativen Charakter besitzt. Es soll nicht zur Weiterverarbeitung verwendet werden, sondern die Domänenanalyse durch eine weitere Sicht unterstützen.

6.4. Variabilitätsmodellierung

Nach der Differenzierung ist die nächste Aufgabe der Analyse oftmals die Restrukturierung der Simulink-Modelle. Dadurch wird sichergestellt, den Wiederverwendungsgrad zu erhöhen. Zu diesem Zweck bedarf es geeigneter Variabilitätsmechanismen (Abschnitt 6.4.1). Ist die Eignung eines oder mehrerer dieser Mechanismen festgelegt (Abschnitt 6.4.2), so können die Modelle entsprechend restrukturiert werden. Diesen Prozess erleichtert insbesondere die Definition von Restrukturierungsregeln

(Abschnitt 6.4.3). Schließlich sind die geeignete Dokumentation und Repräsentation der Variabilität durch ein Variabilitätsmodell unabdingbar (Abschnitt 6.4.4).

6.4.1. Variabilitätsmechanismen

Nachfolgend werden die wichtigsten Variabilitätsmechanismen, die durch die Simulink-Blockbibliothek konstruiert werden können, vorgestellt. In der Arbeit von *Weiland* [Wei08] wurden einige dieser Mechanismen bereits erläutert. Unter anderem sind es die Mechanismen (1) `If Action Subsystem`, (2) `Enabled Subsystem` und (3) `Switch`. Diese werden in den folgenden Abschnitten erneut aufgegriffen. Aufgrund der Weiterentwicklung von Matlab (insbesondere Simulink) sind inzwischen weitere Variabilitätsmechanismen hinzugekommen. Im Rahmen der Arbeiten aus [Men11] und [zA11] wurden weitere Untersuchungen durchgeführt, um die neuen Variabilitätsmechanismen hinsichtlich ihrer Anwendung und Effektivität zu evaluieren. Die neuen Mechanismen umfassen (1) `Model Variants` und (2) `Variant Subsystems`. Diese werden ebenfalls in den folgenden Abschnitten beschrieben.

6.4.1.1. If Action Subsystem

Beschreibung Der Variabilitätsmechanismus realisiert Optionalität oder Alternativität. Varianten werden in `If Action Subsystem`-Blöcke gekapselt. Ein `If`-Block steuert die Ausführung der `If Action Subsystem`-Blöcke. Hierfür werden entsprechende Bedingungen im `If`-Block formuliert. Die Eingaben des `If`-Blocks bestimmen die Werte der Bedingungen. Diese können anhand von `Constant`-Blöcken festgelegt werden. Besteht der Bedarf, die Ausgaben der verschiedenen `If Action Subsystem`-Blöcke zusammenzuführen, können hierfür `Merge`-Blöcke eingesetzt werden. Demnach besteht der Variabilitätsmechanismus aus folgenden Blöcken:

- `Constant`
- `If`
- `If Action Subsystem`
- `Merge`

Die folgende Tabelle zeigt die mit diesem Variabilitätsmechanismus unterstützten Bindezeiten und die zugehörigen Bindungsmechanismen:

Unterstützte Bindezeiten		Bindungsmechanismus
Modellkonstruktionszeit	☒	-
Codegenerierungszeit	☑	Aktivierung → Codegenerator
Compilezeit	☑	Aktivierung → Präprozessor/Compiler
Laufzeit	☑	Aktivierung → Ausführungsmaschine

Es ist nicht möglich, eine Variante zur Modellkonstruktionszeit zu binden. Es sind also stets alle Varianten sichtbar. Alle anderen Bindezeiten werden unterstützt.

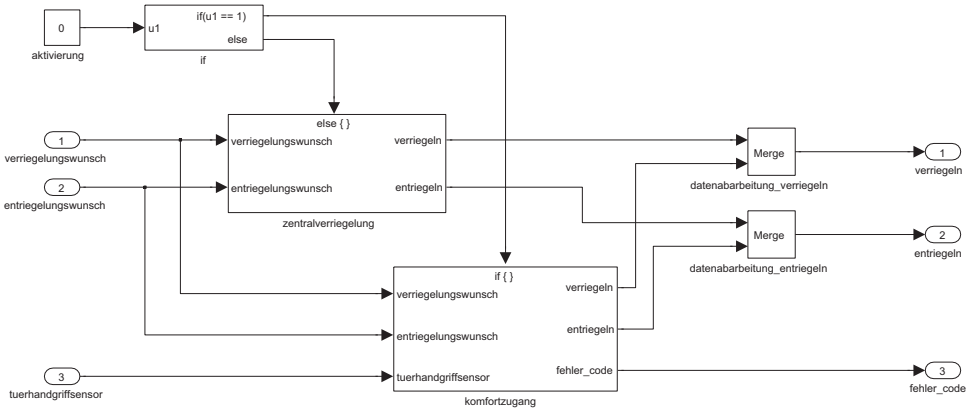


Abbildung 6.18.: Der Variabilitätsmechanismus mittels `If Action Subsystem`

Anhand der Auswertung der Bedingung im `If`-Block ist es möglich, den aktiven `If Action Subsystem`-Block zu ermitteln. Der Codegenerator erzeugt in diesem Fall nur den Code für diesen aktiven `If Action Subsystem`-Block (also nur den Code einer Variante). Bei der Bindung zur Compilezeit erzeugt der Codegenerator den Code für alle Varianten. Variationspunkte werden durch Präprozessordirektiven realisiert. Beim Kompilierungsvorgang durchläuft der Präprozessor den Code und ermittelt die relevanten Fragmente, die dann übersetzt werden. Bei der Bindung zur Laufzeit erzeugt der Codegenerator ebenfalls den Code für alle Varianten. Variationspunkte werden aber in diesem Fall mit Auswahlanweisungen realisiert, wie zum Beispiel `if`- oder `switch`-Anweisungen. Während der Ausführung wird dann ermittelt, welche Verzweigung bearbeitet werden soll.

Beispiel Abbildung 6.18 illustriert ein Beispiel für den Einsatz von `If Action Subsystem`-Blöcken. Der hiermit realisierte Variationspunkt ist das Fahrzeugzugangssystem. Die Varianten sind die Zentralverriegelung und der Komfortzugang. Beide sind in den jeweiligen `If Action Subsystem`-Blöcken `zentralverriegelung` und `komfortzugang` gekapselt. Der `Constant`-Block `aktivierung` bestimmt den Wert für die Bedingung. In der Abbildung ist der Wert auf 0 gesetzt. In diesem Fall wird also `zentralverriegelung` weiter bearbeitet.

6.4.1.2. Enabled Subsystem

Beschreibung Der Variabilitätsmechanismus realisiert Optionalität oder Alternativität (typischerweise zwischen zwei Alternativen). Varianten werden in `Enabled Subsystem`-Blöcke gekapselt. Die Steuerung dieser Blöcke wird in der Regel über die Komposition von logischen und relationalen Operatoren bestimmt. Die Werte für die Operatoren werden durch `Constant`-Blöcke festgelegt. Besteht der Bedarf, die Ausgaben der verschiedenen `Enabled Subsystem`-Blöcke zusammenzuführen,

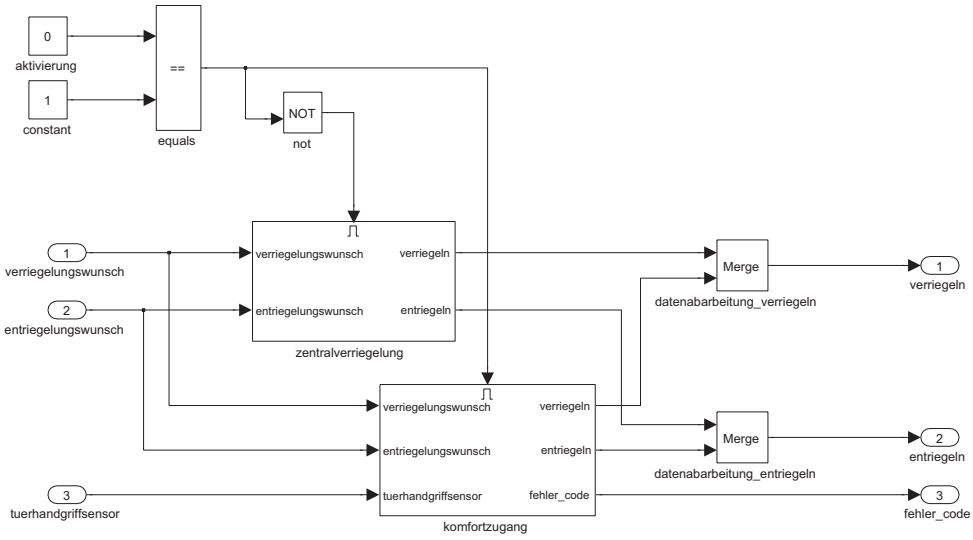


Abbildung 6.19.: Der Variabilitätsmechanismus mittels `Enabled Subsystem`

können hierfür `Merge`-Blöcke eingesetzt werden. Demnach besteht der Variabilitätsmechanismus aus folgenden Blöcken:

- `Constant`
- logische und relationale Operatoren, wie zum Beispiel `AND`, `OR`, `NOT` etc. sowie ==, <=, > etc.
- `Enabled Subsystem`
- `Merge`

Die folgende Tabelle zeigt die mit diesem Variabilitätsmechanismus unterstützten Bindezeiten und die zugehörigen Bindungsmechanismen:

Unterstützte Bindezeiten		Bindungsmechanismus
Modellkonstruktionszeit	☒	-
Codegenerierungszeit	☑	Aktivierung → Codegenerator
Compilezeit	☑	Aktivierung → Präprozessor/Compiler
Laufzeit	☑	Aktivierung → Ausführungsmaschine

Für `Enabled Subsystem`-Blöcke gelten in Bezug auf Bindezeiten und Bindungsmechanismen dieselben Erläuterungen wie für `If Action Subsystem`-Blöcke. Sie werden daher hier nicht erneut aufgeführt.

Beispiel Abbildung 6.19 zeigt ein Beispiel für den Einsatz von `Enabled Subsystem`-Blöcken. Ähnlich wie bei `If Action Subsystem`-Blöcken sind die Varianten des Fahrzeugzugangssystems in den `Enabled Subsystem`-Blöcken `zentralverriegelung` und `komfortzugang` integriert. Die Aktivierung einer Variante wird über den `Constant`-Block `aktiviereung` gesteuert. Dieser wird mit einem weiteren Wert, der ebenfalls durch ein `Constant`-Block realisiert wurde, anhand eines relationalen Operatorblocks == verglichen. Das Ergebnis wird dann an die beiden `Enabled Subsystem`-Blöcke weitergeleitet. Durch den `NOT`-Block wird der Ergebniswert negiert an den Block `zentralverriegelung` gesendet. Dadurch wird sichergestellt, dass stets eine Alternative ausgeführt wird.

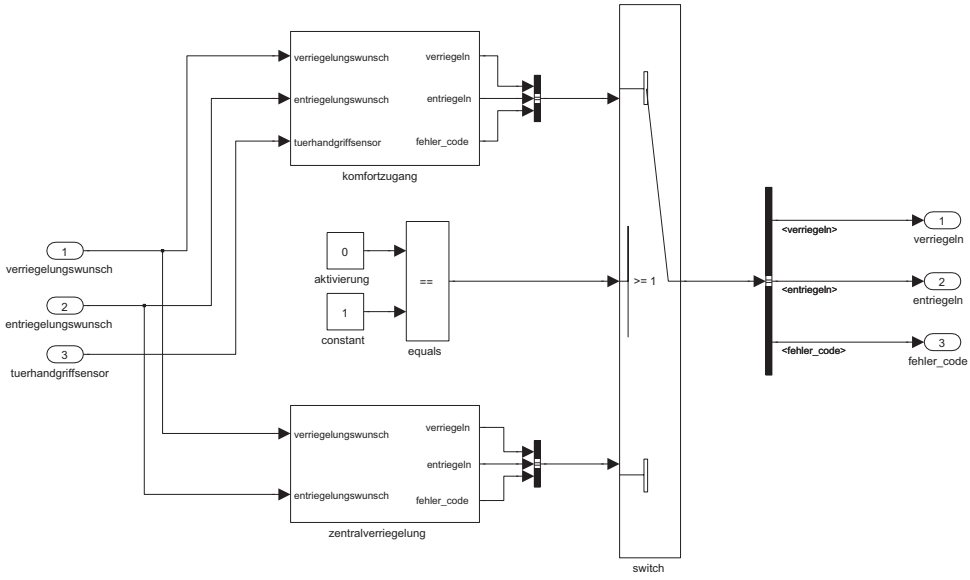
6.4.1.3. Switch

Beschreibung Der Variabilitätsmechanismus realisiert Optionalität oder Alternativität. Varianten werden in `Subsystem`-Blöcke gekapselt. Die Aktivierung der Blöcke erfolgt im `Switch`-Block. Hier wird ein Eingabekriterium für die Schaltung zwischen den `Subsystem`-Blöcken festgelegt. Die Eingabe wird in der Regel über die Komposition von `Constant`-Blöcken sowie logischen als auch relationalen Operatorblöcken modelliert. Da der `Switch`-Block in Simulink vom visuellen als auch datenflussorientierten Aspekt nach den beiden `Subsystem`-Blöcken modelliert wird und für die jeweiligen Varianten nur ein Eingangsport existiert, müssen die Ausgangssignale der `Subsystem`-Blöcke komponiert werden. Hierzu werden `Bus Creator`-Blöcke eingesetzt. Um die Dekomposition nach dem `Switch`-Block gewährleisten zu können, werden `Bus Selector`-Blöcke verwendet. Demnach besteht der Variabilitätsmechanismus aus folgenden Blöcken:

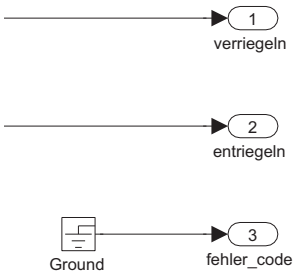
- `Constant`
- logische und relationale Operatoren, wie zum Beispiel `AND`, `OR`, `NOT` etc. sowie ==, <=, > etc.
- `Subsystem`
- `Switch`
- `Bus Creator`
- `Bus Selector`

Die folgende Tabelle zeigt die mit diesem Variabilitätsmechanismus unterstützten Bindezeiten und die zugehörigen Bindungsmechanismen:

Unterstützte Bindezeiten		Bindungsmechanismus
Modellkonstruktionszeit	☒	-
Codegenerierungszeit	☑	Aktivierung → Codegenerator
Compilezeit	☒	-
Laufzeit	☑	Aktivierung → Ausführungsmaschine



(a) Der Variabilitätsmechanismus mittels `Switch`



(b) Die Modellierung des zusätzlichen Ausgangsports `fehler_code`

Abbildung 6.20.: Der Variabilitätsmechanismus mittels `Switch` und eine Umgehungslösung für den `Bus Creator`-/`Bus Selector`-Block

Anders als die zuvor beschriebenen zwei Variabilitätsmechanismen ist die Bindung sowohl zur Modellkonstruktionszeit als auch zur Compilezeit nicht möglich. Lediglich die Codegenerierungszeit und Laufzeit werden unterstützt.

Beispiel Abbildung 6.20(a) illustriert ein Beispiel für den Einsatz von `Switch`-Blöcken. Die Varianten des Fahrzeugzugangssystems sind in den beiden `Subsystem`-Blöcken `zentralverriegelung` und `komfortzugang` gekapselt. Anders als bei den vorherigen Variabilitätsmechanismen ist in diesem Fall allerdings eine Modifikation erforderlich. Der `Switch`-Block akzeptiert stets ein Eingangssignal pro Fall. Daher

werden auch die Ausgangssignale der beiden Subsystem-Blöcke durch Bus Creator-Blöcke komponiert. Um die Dekomposition zu ermöglichen, ist nach dem Switch-Block ein Bus Selector-Block erforderlich. Der Bus Selector-Block kann nur aus den Signalen selektieren, wenn die Anzahl und der Typ der Signale identisch sind. Daher ist für die Variante zentralverriegelung das Signal fehler_code zusätzlich zu modellieren, obwohl es für die eigentliche Funktionslogik nicht relevant ist. Damit das Verhalten der Funktion hierdurch nicht beeinflusst wird, wird innerhalb des Subsystem-Block zentralverriegelung der entsprechende Port mit einem Ground-Block verbunden. Abbildung 6.20(b) illustriert diese Situation.

6.4.1.4. Model Variants

Beschreibung Der Variabilitätsmechanismus realisiert Optionalität oder Alternativität. Varianten werden als eigenständige Simulink-Modelle realisiert (Modellvarianten). Über Referenzierung im Model Variants-Block werden aktivierte Varianten gebunden. Hierfür werden im Base Workspace von Matlab sogenannte Variant Objects vom Typ Simulink Variant erzeugt, die wiederum mit den Modellvarianten assoziiert werden. Jedes Variant Object verfügt über eine Bedingung, die seine Aktivierung bzw. Deaktivierung steuert. Zur Festlegung der Bedingung sind typischerweise Variablen notwendig, die ebenfalls im Base Workspace erzeugt und deklariert werden. In der Regel werden hierfür Variablen vom Typ Matlab Variable oder Simulink Parameter definiert. Demnach besteht der Variabilitätsmechanismus aus folgenden Blöcken:

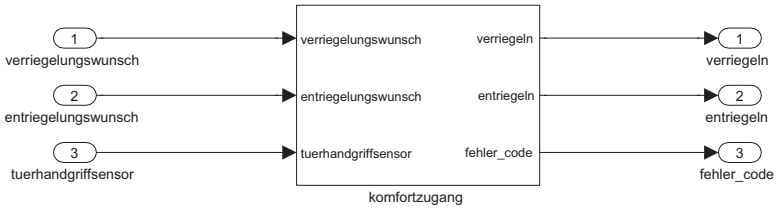
- Model Variants
- Modellvarianten (.mdl-Dateien)
- Variant Objects vom Typ Simulink Variant
- Variablen vom Typ Matlab Variable oder Simulink Parameter

Die folgende Tabelle zeigt die mit diesem Variabilitätsmechanismus unterstützten Bindezeiten und die zugehörigen Bindungsmechanismen:

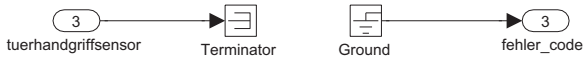
Unterstützte Bindezeiten		Bindungsmechanismus
Modellkonstruktionszeit	☑	Aktivierung → Updatemaschine
Codegenerierungszeit	☑	Aktivierung → Codegenerator
Compilezeit	☑	Aktivierung → Präprozessor/Compiler
Laufzeit	☑	Aktivierung → Ausführungsmaschine

Der Variabilitätsmechanismus mit Model Variants-Blöcken ist der Erste, der die Bindung auch zur Modellkonstruktionszeit unterstützt. Hierzu werden die Bedingungen in den Variant Objects ausgewertet. Die Modellvariante, die mit dem als gültig ausgewerteten Variant Object assoziiert ist, wird durch einen Updateprozess in das Modell geladen, sodass zur Modellkonstruktionszeit nur diese Modellvariante sichtbar ist. Die Mechanismen der weiteren Bindezeiten sind analog zu den Beschreibungen aus den vorangegangenen Variabilitätsmechanismen.

(a) Die Zentralverrieglung als separates Simulink-Modell



(b) Der Komfortzugang als separates Simulink-Modell

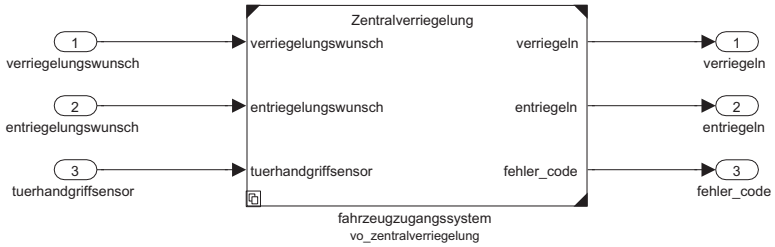


(c) Die Erweiterungen für die Zentralveriegelung durch den Einsatz von `Terminator`- und `Ground`-Blöcken

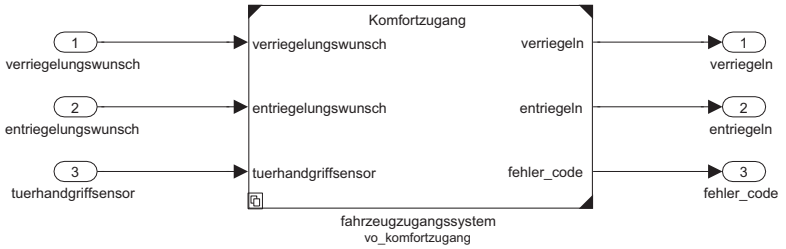
Abbildung 6.21.: Die Modellvarianten Zentralverriegelung und Komfortzugang

Beispiel Wie bereits erwähnt, werden Varianten in separaten Simulink-Modellen realisiert. Abbildung 6.21(a) und Abbildung 6.21(b) zeigen die beiden Varianten des Fahrzeugzugangssystems. Sowohl die Zentralverriegelung als auch der Komfortzugang werden in `Subsystem`-Blöcken gekapselt. Eine Auffälligkeit hierbei ist, dass der `Subsystem`-Block `zentralverriegelung` über die gleiche Schnittstelle wie der `Subsystem`-Block `komfortzugang` verfügt. Dies ist erforderlich, da die fehlerfreie Referenzierung von `Model Variants`-Blöcken nur somit gewährleistet werden kann. Dies hat zur Folge, dass der Block `zentralverriegelung` entsprechenden Modifikationen unterzogen werden muss. Abbildung 6.21(c) zeigt die notwendigen Maßnahmen. Um die gleiche Schnittstelle zu erhalten, muss der Block `zentralverriegelung` um einen Eingabeport `tuerhandgriffsensor` erweitert werden. Da dieser aber für die Funktionslogik nicht erforderlich ist, wird er mit einem `Terminator`-Block verbunden, um den Signalfluss an dieser Stelle zu beenden. Weiterhin ist zur Gewährleistung einer identischen Schnittstelle ein weiterer Ausgabeport `fehler_code` erforderlich. Genau wie bei `Switch`-Blöcken wird der Ausgabeport mit einem `Ground`-Block verbunden. Auf diese Weise wird die Funktionslogik der Zentralverriegelung nicht beeinträchtigt und zudem wird eine Schnittstellengleichheit zwischen den Varianten erreicht.

Werden nun im Base Workspace die `Variant Objects` definiert und die Bedingungen zur Aktivierung bzw. Deaktivierung festgelegt, kann über einen `Model Variants`-Block die Zentralverriegelung und der Komfortzugang referenziert werden. Abbil-



(a) Der `Model Variants`-Block mit Referenzierung der Zentralverrieglung



(b) Der `Model Variants`-Block mit Referenzierung des Komfortzugangs

Abbildung 6.22.: Der `Model Variants`-Block und die Bindung seiner Varianten

dung 6.22 zeigt diese Situation. Wenn das `Variant Object` der Zentralverriegelung `vo_zentralverriegelung` aktiviert wird, bindet der `Model Variants`-Block dessen Modell in den Block ein. Wird hingegen `vo_komfortzugang` aktiv, wird entsprechend der Komfortzugang eingebunden. Insbesondere kann die Bindung der Modellvarianten aufgrund der Schnittstellengleichheit zu keiner Inkonsistenz bezüglich der Schnittstelle führen.

6.4.1.5. Variant Subsystem

Beschreibung Der Variabilitätsmechanismus realisiert Optionalität oder Alternativität. Varianten werden in `Subsystem`-Blöcke gekapselt (Subsystemvarianten). Es ist eine auf Subsystemebene erweiterte Form der Konzepte aus dem Variabilitätsmechanismus für `Model Variants`-Blöcke. Die Methodik zur Modellierung des Variabilitätsmechanismus mit `Variant Subsystem`-Blöcken ist daher vergleichbar mit `Model Variants`-Blöcken. Es werden also hier ebenfalls `Variant Objects` und Variablen erzeugt als auch Bedingungen zur Aktivierung/Deaktivierung festgelegt. Demnach besteht der Variabilitätsmechanismus aus folgenden Blöcken:

- `Variant Subsystem`
- Subsystemvarianten (`Subsystem`-Blöcke)
- `Variant Objects` vom Typ `Simulink Variant`
- Variablen vom Typ `Matlab Variable` oder `Simulink Parameter`

Die folgende Tabelle zeigt die mit diesem Variabilitätsmechanismus unterstützten Bindezeiten und die zugehörigen Bindungsmechanismen:

Unterstützte Bindezeiten		Bindungsmechanismus
Modellkonstruktionszeit	☑	Aktivierung → Updatemaschine
Codegenerierungszeit	☑	Aktivierung → Codegenerator
Compilezeit	☑	Aktivierung → Präprozessor/Compiler
Laufzeit	☑	Aktivierung → Ausführungsmaschine

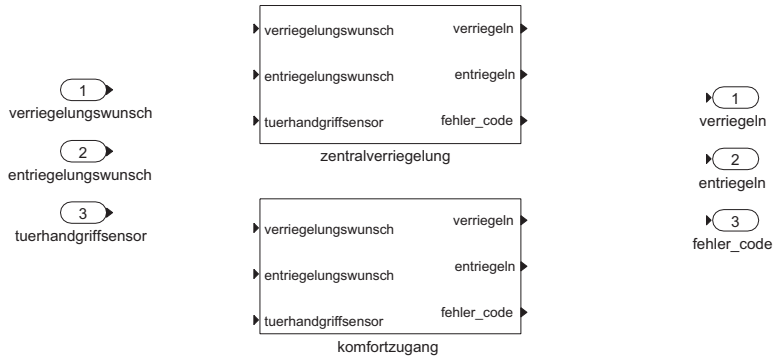
Entsprechend den Beschreibungen bei `Model Variants`-Blöcken unterstützen `Variant Subsystem`-Blöcke ebenfalls alle Bindezeiten.

Beispiel Die Modellierung der Variabilität im Fahrzeugzugangssystem mit `Variant Subsystem`-Blöcken erfordert den Einsatz von `Subsystem`-Blöcken, mit denen die Varianten gekapselt werden. Abbildung 6.23(a) illustriert diesen Fall. Die Funktionslogik beider Varianten sind in den `Subsystem`-Blöcken `zentralverriegelung` und `komfortzugang` integriert. Auch bei `Variant Subsystem`-Blöcken ist eine Schnittstellengleichheit erforderlich. Daher wird der Block `zentralverriegelung` um den Eingabeport `tuerhandgriffsensor` und den Ausgabeport `fehler_code` erweitert. Damit die Funktionslogik unberührt bleibt, werden der ergänzte Eingabeport mit einem `Terminator`-Block und der zusätzliche Ausgabeport mit einem `Ground`-Block verbunden (vgl. Abbildung 6.23(b)). Weiterhin ist auffällig, dass die Ports nicht mit den `Subsystem`-Blöcken verbunden werden. Der Grund hierfür ist die Vermeidung von zu vielen Verbindungen, die die Übersicht beeinträchtigen können. Stattdessen wird die Zuordnung der Ports durch Namensgleichheit sichergestellt.

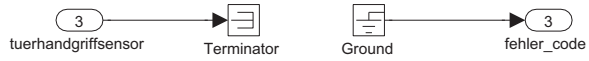
Wie bei `Model Variants`-Blöcken werden nun im Base Workspace `Variant Objects` definiert und die Bedingungen zur Aktivierung bzw. Deaktivierung festgelegt. In Abbildung 6.24 wird das Ergebnis gezeigt, wenn die Zentralverriegelung aktiviert wird. Der weitere Block `komfortzugang` wird ausgeblendet, um zu verdeutlichen, welche Variante aktuell aktiv ist (vgl. Abbildung 6.24(a)). In der übergeordneten Hierarchieebene ist dies ebenfalls anhand des Variantennamens im `Variant Subsystem`-Block ersichtlich (vgl. Abbildung 6.24(b)).

6.4.2. Bewertung der Variabilitätsmechanismen

Nachdem nun die Variabilitätsmechanismen von Simulink vorgestellt wurden, werden sie in diesem Abschnitt anhand verschiedener Kriterien bewertet. Dies dient primär zur Bestimmung geeigneter Mechanismen für die Restrukturierung. Es gibt sehr viele Kriterien, mit denen Variabilitätsmechanismen evaluiert werden können. Unter anderem sind dies Übersichtlichkeit, Verständlichkeit und Anpassbarkeit. In [Men11] wurden sie bereits unter diesen Gesichtspunkten untersucht. Dabei hat sich herausgestellt, dass drei wesentliche Kriterien die Unterschiede in den Variabilitätsmechanismen beleuchten:



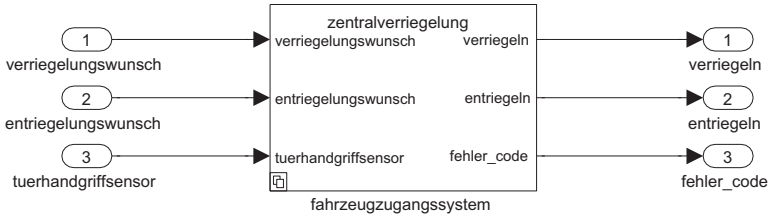
(a) Die Zentralverrieglung und der Komfortzugang innerhalb eines `Variant Subsystem`-Blocks



(b) Die Erweiterungen für die Zentralveriegelung durch den Einsatz von `Terminator`- und `Ground`-Blöcken

Abbildung 6.23.: Die Subsystemvarianten Zentralverriegelung und Komfortzugang

(a) Die Aktivierung des Blocks `zentralverrieglung`



(b) Der `Variant Subsystem`-Block bei Bindung der Zentralverriegelung

Abbildung 6.24.: Die Einbindung der Zentralverriegelung durch Aktivierung des `Variant Objects`

1. Die Sichtbarkeit der Bindung von Varianten auf Modellebene
2. Die Möglichkeit zur Variantenverwaltung
3. Die Unterstützung von Bindezeiten

Anhand der Sichtbarkeit von Variantenbindungen auf Modellebene wird überprüft, ob Simulink durch den Variabilitätsmechanismus seine Visualisierung entsprechend der gebundenen Varianten anpasst. Dies ist nur bei `Model Variants` und `Variant Subsystems` der Fall. Bei allen anderen Variabilitätsmechanismen sind stets alle Varianten gleichermaßen sichtbar. Sowohl Übersichtlichkeit als auch Verständlichkeit werden somit bei `Model Variants` und `Variant Subsystems` besser unterstützt.

Die Verwaltung der Varianten werden bei `If Action Subsystem`, `Enabled Subsystem` und `Switch` über eine Kontrollvariable geregelt, die entsprechende Varianten aktiviert bzw. deaktiviert. Bei `Model Variants` und `Variant Subsystems` geschieht dies über `Variant Objects`. Der wesentliche Vorteil Letzteres gegenüber Kontrollvariablen ist die Möglichkeit, Bedingungen anzugeben, die eine automatische Aktivierung bzw. Deaktivierung der Varianten erlauben. Bei Kontrollvariablen ist stets ein Benutzereingreifen erforderlich, welches mühselig und fehleranfällig ist.

Schließlich werden nur bei `Model Variants` und `Variant Subsystems` alle Bindezeiten unterstützt. `If Action Subsystem`, `Enabled Subsystem` und `Switch` geben keine Unterstützung zur Bindung bei Modellkonstruktionszeit. `Switch` unterstützt zudem auch nicht die Compilezeit.

Tabelle 6.2 fasst die Ergebnisse dieser Bewertung zusammen. Angesichts dieser Feststellungen ist es offensichtlich, dass der Einsatz von `Model Variants` oder `Variant Subsystems` als Variabilitätsmechanismus im Vergleich zu den anderen Mechanismen angemessener ist. Welcher von beiden eingesetzt werden soll, hängt nun von einer wesentlichen Frage ab: Wie groß (z.B. in KB oder MB) werden die Modelle? Sind die Modelle eher klein, sollte immer `Variant Subsystems` eingesetzt werden. Erreichen die Modelle eine Größe, die zum Beispiel nicht in den Arbeitsspeicher eines handelsüblichen PCs geladen werden können, sollten `Model Variants` angewendet werden, da sie aufgrund des Referenzierungsmechanismus Ressourcen schonen.

Wenn die Differenzierung also abgeschlossen ist und die Analyse des Kommunalitätsmodells weiter durchgeführt wird, sollte bei Bedarf eine Restrukturierung vorgenommen werden. Zu diesem Zweck sollten identifizierte Variationspunkte durch die beiden Variabilitätsmechanismen `Model Variants` oder `Variant Subsystems` gekapselt werden.

6.4.3. Restrukturierung mit Model Variants und Variant Subsystem

Wurde nach der Differenzierung von zwei Simulink-Modellen der Bedarf einer Restrukturierung erkannt, so müssen die Modelle zusammengeführt und die Variationspunkte durch die beiden präferierten Variabilitätsmechanismen realisiert werden. Im

	Sichtbarkeit	Variantenverwaltung	Bindezeiten			
			MKZ	CGZ	CZ	LZ
If Action Subsystem	✗	Kontrollvariable	✗	✓	✓	✓
Enabled Subsystem	✗	Kontrollvariable	✗	✓	✓	✓
Switch	✗	Kontrollvariable	✗	✓	✗	✓
Model Variants	✓	Variant Object	✓	✓	✓	✓
Variant Subsystems	✓	Variant Object	✓	✓	✓	✓

Tabelle 6.2.: Zusammenfassung der Bewertungsergebnisse für die vorgestellten Variabilitätsmechanismen

Folgenden werden zu diesem Zweck Regeln vorgestellt, die den Restrukturierungsprozess unterstützen. Die Regeln umfassen Variationspunkte an der Schnittstelle (sowohl Modell- als auch Blockschnittstelle), Variationspunkte am Vergleichspaar, Variationspunkte am Blocktyp und Variationspunkte an Verbindungen. Anhand dieser Regeln wird es zudem möglich auch komplexe Variationen zu erfassen. Ein Beispiel wird dies am Ende des Abschnitts illustrieren.

6.4.3.1. Restrukturierungsregeln

Restrukturierungsregel 1 Ein Variationspunkt, der an der Modellschnittstelle identifiziert wurde, wird durch Kapselung der variierenden Ports mit dem entsprechenden Variabilitätsmechanismus realisiert. Um die Adaptivität des restrukturierten Modells sicherzustellen, wird stets eine maximale Modellschnittstelle realisiert. Sie setzt sich aus der Vereinigung aller Modellschnittstellen zusammen, die im Rahmen der Differenzierung herangezogen werden. Ports, die in einer Variante nicht erforderlich sind, werden mit `Terminator`- und `Ground`-Blöcken verbunden. So wird die Adaptivität sowohl innerhalb des Modells als auch nach außen gewährleistet. Je nach Art der variierenden Modellschnittstelle werden die Blöcke wie folgt modelliert:

1. Variationspunkt an der Eingabeschnittstelle
   a) Erforderliche Eingabeports:
      Ein erforderlicher Eingabeport wird direkt mit einem Ausgabeport verbunden.
   b) Nicht erforderliche Eingabeports:
      Ein nicht erforderlicher Eingabeport wird direkt mit einem `Terminator`-Block verbunden. Weiterhin wird ein `Ground`-Block hinzugefügt, der mit einem Ausgabeport verbunden wird.
2. Variationspunkt an der Ausgabeschnittstelle
   a) Erforderliche Ausgabeports:
      Der erforderliche Ausgabeport wird direkt mit einem Eingabeport verbunden.

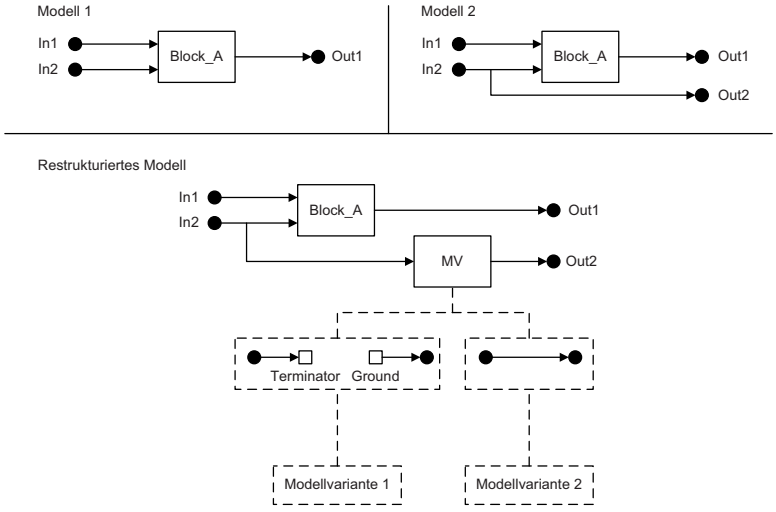
b) Nicht erforderliche Ausgabeports:
Der nicht erforderliche Ausgabeport wird direkt mit einem `Ground`-Block verbunden. Weiterhin wird ein Eingabeport hinzugefügt, der mit einem `Terminator`-Block verbunden wird.

Abbildung 6.25 visualisiert ein Beispiel für beide Variabilitätsmechanismen. Hier sind jeweils zwei Modelle dargestellt: Modell 1 und Modell 2. Der Variationspunkt in beiden Modellen befindet sich in der Ausgabeschnittstelle. Hier ist der Ausgabeport `Out2` in Modell 1 nicht enthalten, während er in Modell 2 präsent ist. In Abbildung 6.25(a) wird der Variationspunkt im restrukturierten Modell durch Modellierung der maximalen Schnittstelle erfasst. Durch einen `Model Variants`-Block wird dann der Datenfluss für beide Varianten gesteuert. In der ersten Modellvariante werden ankommende Datensignale über den Eingabeport empfangen. Am `Terminator`-Block, der mit dem Eingabeport verbunden ist, wird der Datenfluss beendet. Somit wird die Variante für das Modell stets korrekt angepasst. Damit nun das Modell auch nach außen hin anpassbar ist, wird über einen `Ground`-Block ein Füllsignal erzeugt und mit einem Ausgabeport verbunden. Für die zweite Modellvariante wird hingegen der Datenfluss einfach weitergeleitet, da dieser für die Variante erforderlich ist.

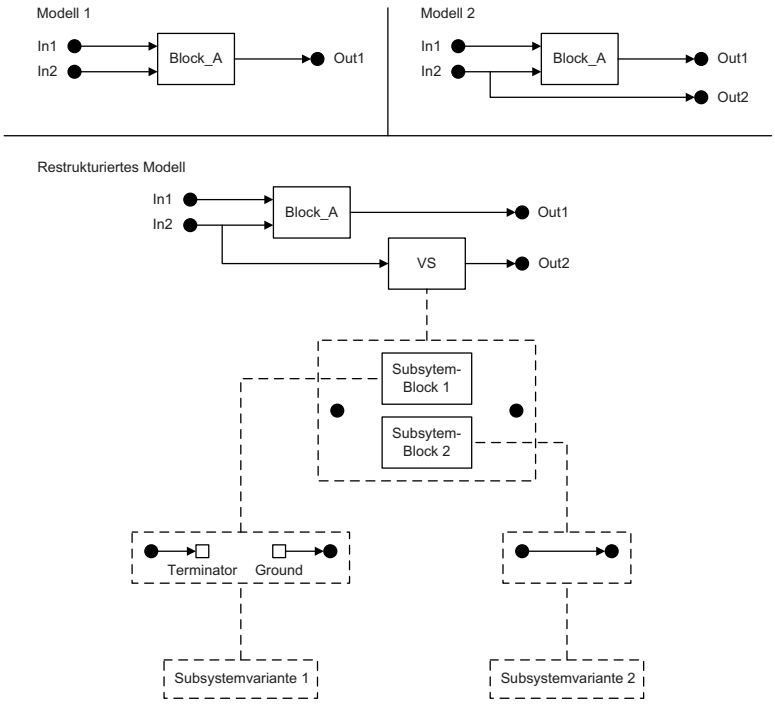
Gleiches gilt für Abbildung 6.25(b). Hier wird allerdings ein `Variant Subsystem`-Block eingesetzt. Der wesentliche Unterschied ist die Einführung einer weiteren Hierarchieebene, die alle Varianten durch Subsystem-Blöcke kapselt. Innerhalb dieser Subsysteme sind dann die variantenspezifischen Modelle enthalten.

Restrukturierungsregel 2 Ein Variationspunkt, der an einem Block aufgrund eines fehlenden Pendants entsteht, wird in den Variabilitätsmechanismus integriert. Je nach Blockart werden folgende Maßnahmen durchgeführt:

1. Variationspunkt an einem Quellblock aufgrund eines fehlenden Pendants
   a) Variante ohne Quellblock:
   Da ein Quellblock nicht Teil der Eingabeschnittstelle ist, beeinflusst er auch nicht die Adaptivität nach außen. Es ist somit kein Eingabeport erforderlich, der mit einem `Terminator`-Block verbunden wird. Da allerdings ein Quellblock die Adaptivität innerhalb des Modells beeinflusst, wird ein `Ground`-Block mit einem Ausgabeport verbunden.
   b) Variante mit Quellblock:
   Der Quellblock wird mit einem Ausgabeport verbunden.
2. Variationspunkt an einem Ein-Ausgabeblock aufgrund eines fehlenden Pendants
   a) Variante ohne Ein-Ausgabeblock:
   Da ein Ein-Ausgabeblock das Modell sowohl an der Ein- als auch an der Ausgabeschnittstelle beeinflusst, wird in der Variante ein Eingabeport modelliert, der mit einem `Terminator`-Block verbunden wird (Abfangen von Datensignalen). Weiterhin wird ein `Ground`-Block hinzugefügt, der mit einem Ausgabeport verbunden wird (Generierung von Füllsignalen).



(a) Anwendung mit `Model Variants`



(b) Anwendung mit `Variant Subsystem`

Abbildung 6.25.: Ein Beispiel für die Anwendung der Restrukturierungsregel 1

b) Variante mit Ein-Ausgabeblock:
Der Ein-Ausgabeblock wird in die Variante integriert. Zur Anbindung an das Modell werden Ein- und Ausgabeports modelliert, die mit dem Block verbunden werden.

3. Variationspunkt an einem Zielblock aufgrund eines fehlenden Pendants

a) Variante ohne Zielblock:
Da ein Zielblock nicht Teil der Ausgabeschnittstelle ist, beeinflusst er auch nicht die Adaptivität nach außen. Es ist somit kein Ausgabeport erforderlich, der mit einem `Ground`-Block verbunden wird. Da allerdings ein Zielblock die Adaptivität innerhalb des Modells beeinflusst, wird ein `Terminator`-Block mit einem Eingabeport verbunden.

b) Variante mit Zielblock:
Der Zielblock wird mit einem Eingabeport verbunden.

In Abbildung 6.26 ist hierfür ein Beispiel illustriert. Der Zielblock im zweiten Modell hat kein Pendant im ersten Modell. Bei der Verwendung von `Model Variants` wird entsprechend der Restrukturierungsregel für die erste Modellvariante das Datensignal über dem Eingabeport und `Terminator`-Block abgefangen. Die zweite Modellvariante hingegen überlässt empfangene Datensignale über den Eingabeport an den Zielblock zu (vgl. Abbildung 6.26(a)). Abbildung 6.26(b) zeigt das Resultat der Restrukturierung für `Variant Subsystem`.

Restrukturierungsregel 3 Ein Variationspunkt, der an der Blockschnittstelle identifiziert wurde, wird durch Kapselung der variierenden Blockschnittstellen mit dem entsprechenden Variabilitätsmechanismus erfasst. Da die Blockschnittstelle nicht vom Block zu trennen ist, wird somit der Block selbst in den Variabilitätsmechanismus integriert. Um die Adaptivität zu gewährleisten, wird die maximale Blockschnittstelle modelliert. Nicht erforderliche Eingabe- bzw. Ausgabeports werden mit `Terminator`- bzw. `Ground`-Blöcken verbunden. Je nach Art der variierenden Blockschnittstelle wird der Variabilitätsmechanismus wie folgt eingesetzt:

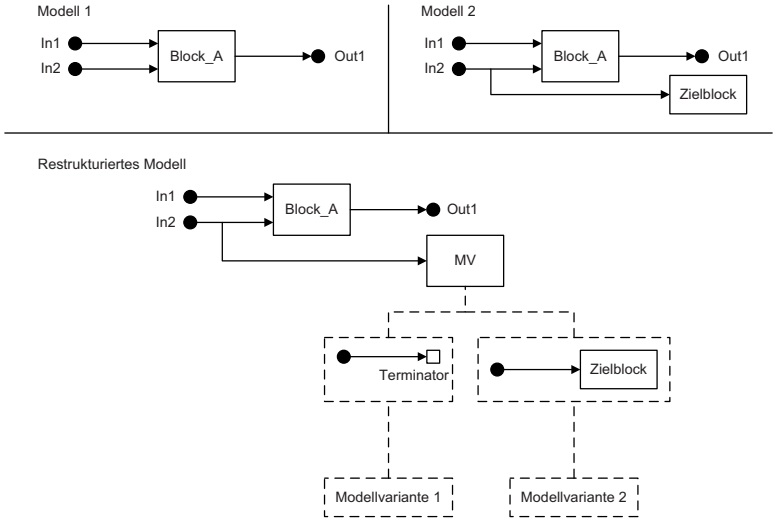
1. Variationspunkt an der Eingabeblockschnittstelle

a) Erforderliche Eingabeports:
Ein erforderlicher Eingabeport an der Blockschnittstelle wird mit einem Eingabeport an der Modellschnittstelle der Variante verbunden.

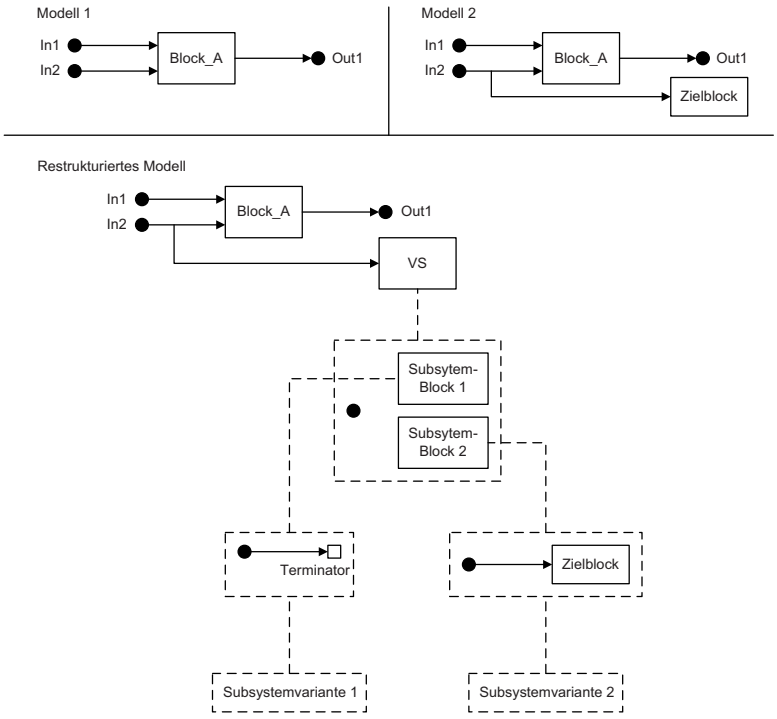
b) Nicht erforderliche Eingabeports:
Ein nicht erforderlicher Eingabeport an der Blockschnittstelle wird in der Variante durch ein Eingabeport modelliert, der mit einem `Terminator`-Block verbunden wird.

2. Variationspunkt an der Ausgabeblockschnittstelle

a) Erforderliche Ausgabeports:
Ein erforderlicher Ausgabeport an der Blockschnittstelle wird mit einem Ausgabeport an der Modellschnittstelle der Variante verbunden.



(a) Anwendung mit `Model Variants`



(b) Anwendung mit `Variant Subsystem`

Abbildung 6.26.: Ein Beispiel für die Anwendung der Restrukturierungsregel 2

b) Nicht erforderliche Ausgabeports:
Ein nicht erforderlicher Ausgabeport an der Blockschnittstelle wird in der Variante durch ein Ausgabeport modelliert, der mit einem `Ground`-Block verbunden wird.

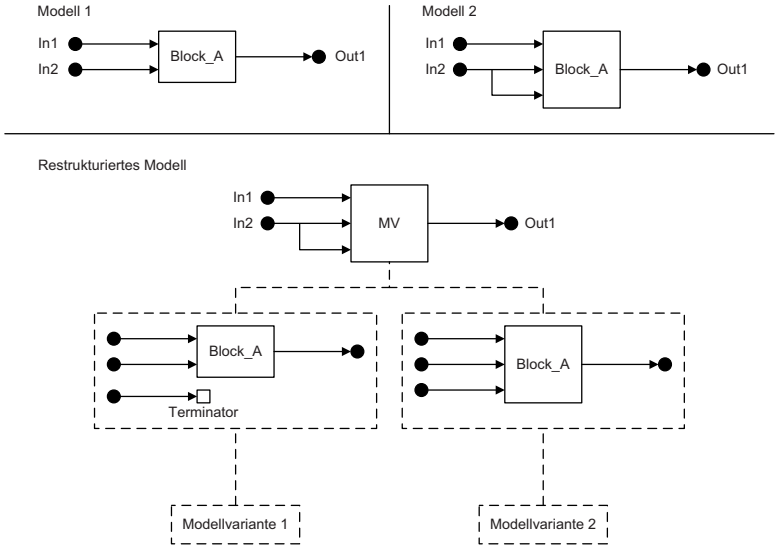
Abbildung 6.27 zeigt ein Beispiel. An der Eingabeschnittstelle des Blocks `Block_A` liegt ein Variationspunkt vor. In Abbildung 6.27(a) wird zu diesem Zweck ein `Model Variants`-Block eingesetzt. Der `Model Variants`-Block setzt sich dabei aus der maximalen Blockschnittstelle zusammen. Da für die erste Modellvariante der dritte Eingabeport nicht erforderlich ist, wird dieser daher mit einem `Terminator`-Block verbunden. In Abbildung 6.27(b) wird der Variationspunkt durch ein `Variant Subsystem`-Block realisiert.

Restrukturierungsregel 4 Ein Variationspunkt, der an einem Blocktyp identifiziert wurde, wird durch Kapselung des Blocktyps im Variabilitätsmechanismus erfasst. Da der Blocktyp nicht vom Block zu trennen ist, wird somit der Block selbst in den Variabilitätsmechanismus integriert. Um die Anbindung der Varianten sicherzustellen, müssen entsprechende Schnittstellen modelliert und mit den Blöcken verbunden werden.

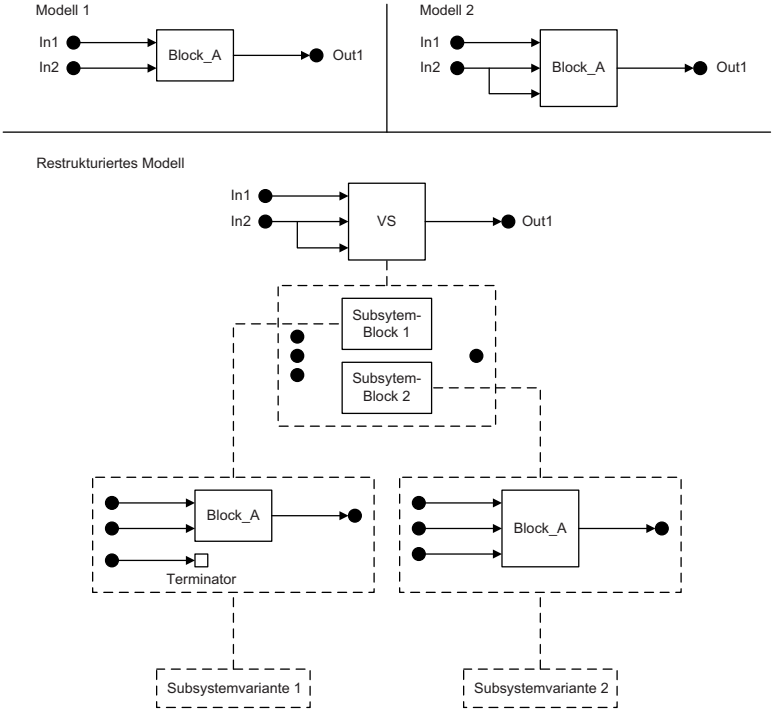
Abbildung 6.28 zeigt diesbezüglich ein Beispiel. Die beiden zu vergleichenden Modelle unterscheiden sich lediglich am Blocktyp. Der entsprechende Block wird in Abbildung 6.28(a) in einen `Model Variants`-Block und in Abbildung 6.28(b) in einen `Variant Subsystem`-Block integriert. Damit eine Anbindung der Blöcke gegeben ist, werden zusätzlich Ein- und Ausgabeschnittstellen modelliert. Somit ist der Datenfluss sichergestellt. Es sei hierbei erwähnt, dass die ursprüngliche Schnittstelle in den übergeordneten restrukturierten Modellen erfasst ist.

Restrukturierungsregel 5 Ein Variationspunkt, der an Verbindungen identifiziert wurde, wird in den Varianten des Variabilitätsmechanismus durch Ein- und Ausgabeports modelliert, die das Signalrouting steuern. Dabei müssen mindestens zwei Verbindungen betroffen sein. Bei einem Variationspunkt an nur einer Verbindung wird kein dedizierter Variabilitätsmechanismus eingesetzt, da in diesem Fall die Variation bereits durch andere Restrukturierungsregeln erfasst wird.

Abbildung 6.29 veranschaulicht ein Beispiel. Die Ausgabeports des Blocks `Block_A` sind in beiden Modellen durch verschiedene Verbindungen modelliert. Während im ersten Modell der erste Ausgabeport des Blocks mit dem ersten Ausgabeport des Modells (`Out1`) und der zweite Ausgabeport des Blocks mit dem zweiten Ausgabeport des Modells (`Out2`) verbunden sind, sind im zweiten Modell der erste Ausgabeport des Blocks mit dem zweiten Ausgabeport des Modells (`Out2`) und der zweite Ausgabeport des Blocks mit dem ersten Ausgabeport des Modells (`Out1`) verbunden. In den restrukturierten Modellen aus Abbildung 6.29(a) und Abbildung 6.29(b) wird der Variationspunkt an der Verbindung durch einen `Model Variants`- bzw. `Variant Subsystem`-Block gekapselt. Die Modell- bzw. Subsystemvarianten enthalten das entsprechende Signalrouting.

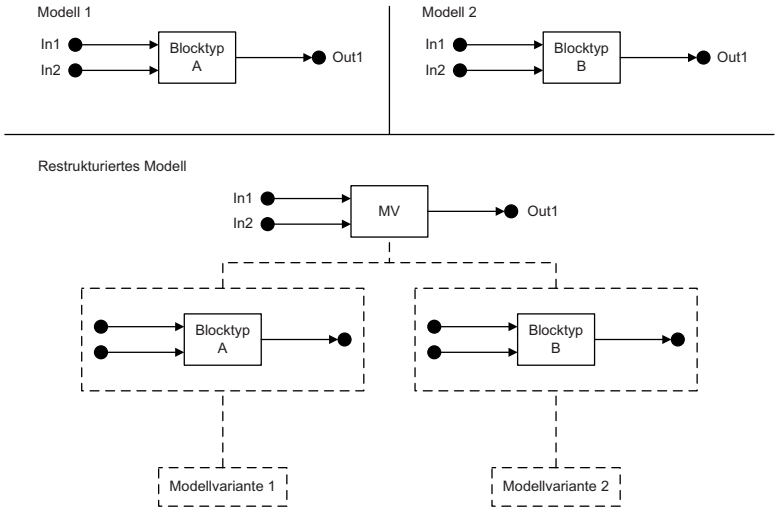


(a) Anwendung mit `Model Variants`

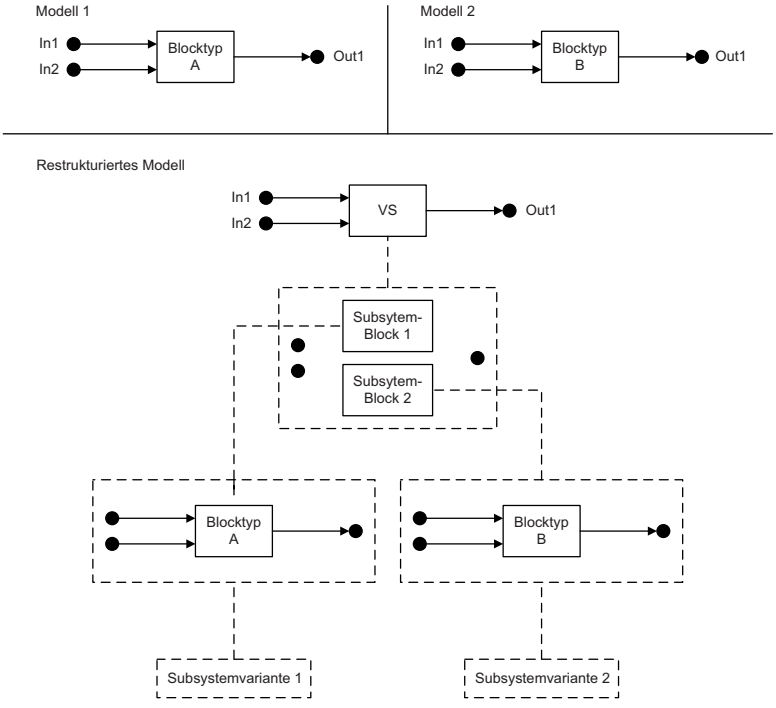


(b) Anwendung mit `Variant Subsystem`

Abbildung 6.27.: Ein Beispiel für die Anwendung der Restrukturierungsregel 3

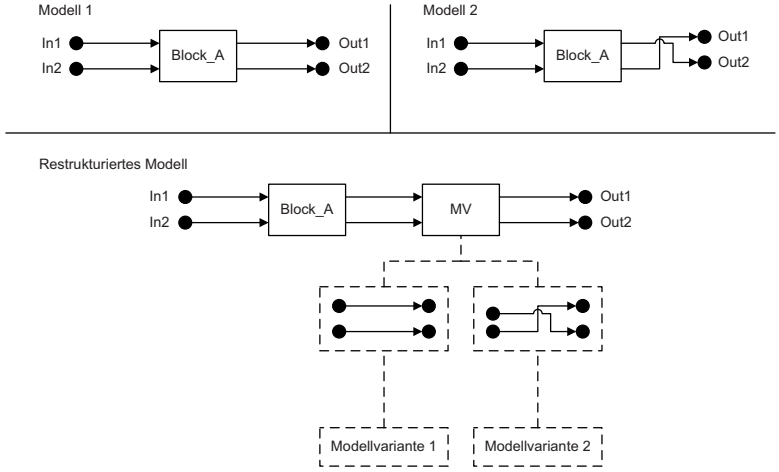


(a) Anwendung mit `Model Variants`

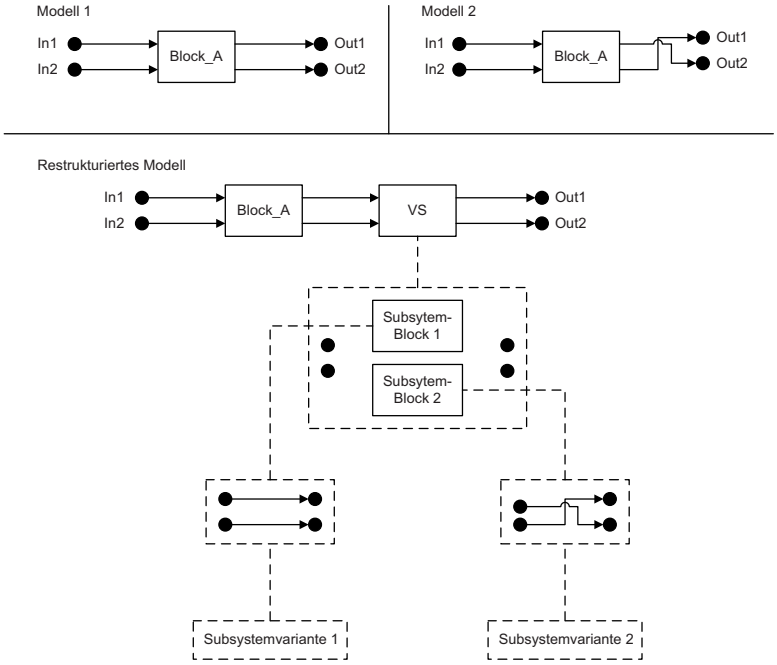


(b) Anwendung mit `Variant Subsystem`

Abbildung 6.28.: Ein Beispiel für die Anwendung der Restrukturierungsregel 4



(a) Anwendung mit `Model Variants`



(b) Anwendung mit `Variant Subsystem`

Abbildung 6.29.: Ein Beispiel für die Anwendung der Restrukturierungsregel 5

6.4.3.2. Beispiel

Nachdem nun grundlegende Restrukturierungsregeln vorgestellt wurden, wird in diesem Abschnitt anhand des abstrakten Beispiels aus Abbildung 6.1, das komplexerer Natur ist, die Anwendung der Regeln illustriert. Eine Differenzierung der beiden Simulink-Modelle ergab insgesamt vier Variationspunkte:

1. Ein Variationspunkt am Blocktyp:
   a) Modell 1: `Add`
   b) Modell 2: `Product`
2. Ein Variationspunkt am Vergleichspaar:
   a) Modell 1: `Display`
   b) Modell 2: –
3. Ein Variationspunkt an der Ausgabeschnittstelle:
   a) Modell 1: –
   b) Modell 2: Ein Ausgabeport `Out1`
4. Ein Variationspunkt an der Verbindung:
   a) Modell 1:
      i. Quelle: Ausgabeport des `Add`-Blocks
      ii. Ziel: Eingabeport des `Display`-Blocks
   b) Modell 2:
      i. Quelle: Ausgabeport des `Product`-Blocks
      ii. Ziel: Ausgabeport `Out1` des Modells

Werden die Regeln aus Abschnitt 6.4.3.1 angewendet, ergeben sich die restrukturierten Modelle aus Abbildung 6.30. Abbildung 6.30(a) zeigt dabei die Anwendung der Regeln mit `Model Variants`. Abbildung 6.30(b) illustriert hingegen den Einsatz von `Variant Subsystem`. Der Variationspunkt am Blocktyp wird entsprechend der Restrukturierungsregel 4 in einen Variabilitätsmechanismus integriert. Bei dem Variationspunkt am Vergleichspaar handelt es sich um einen Zielblock. Daher wird dieser Block wie ein Ausgabeport einer Schnittstelle behandelt. Da zudem ein Variationspunkt an der Ausgabeschnittstelle vorliegt, werden beide Variationspunkte zusammengefasst und gemeinsam behandelt. Sie werden also ebenfalls in einen Variabilitätsmechanismus gekapselt. Um die Konsistenz nach außen sicherzustellen, wird für die erste Variante zusätzlich ein Ausgabeport modelliert, der mit einem `Ground`-Block verbunden ist. Damit ist die maximale Schnittstelle hergestellt. Schließlich wird der Variationspunkt an der Verbindung nicht gesondert behandelt, da nach Restrukturierungsregel 5 mindestens zwei Variationspunkte an Verbindungen vorliegen müssen, um diese in einen Variabilitätsmechanismus zu kapseln.

Bei nur einem Variationspunkt an der Verbindung, wird dieser bereits durch die Variabilitätsmechanismen erfasst, die den Quell- bzw. Zielblock umfassen.

Insgesamt werden also zwei `Model Variants`- bzw. `Variant Subsystem`-Blöcke eingesetzt. Durch die Definition von `Variant Objects`, wie etwa `Modellvariante1` und `Modellvariante2` bzw. `Subsystemvariante1` und `Subsystemvariante2`, werden die komplexen Variabilitätsmechanismen in eine Einheit gebracht, sodass bei Aktivierung die korrekten Varianten konstruiert werden.

6.4.4. Variabilitätsmodell

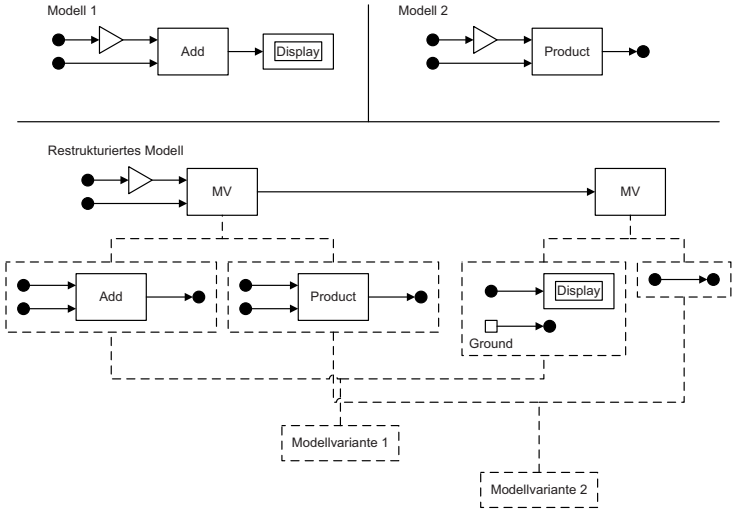
Um Gemeinsamkeiten und Variationspunkte zwischen zwei Simulink-Modellen zu identifizieren, wurde in Abschnitt 6.3 der Differenzierungsprozess erläutert. Weiterhin wurden Regeln beschrieben, die nach der Differenzierung zur Restrukturierung angewendet werden können. Variationspunkte wurden dabei durch `Model Variants` oder `Variant Subsystem` realisiert. Eine explizite Modellierung, Dokumentation und Repräsentation der erfassten Variationspunkte fehlt allerdings bislang. Hierfür wird in diesem Abschnitt das Variabilitätsmodell aus Kapitel 4 herangezogen.

Im Gegensatz zu der feingranularen Modellierung der Variationspunkte auf Simulink-Ebene, wird im Variabilitätsmodell dieses nicht angestrebt. Das bedeutet, dass ein Variationspunkt in einem Simulink-Modell, der durch einen Variabilitätsmechanismus realisiert wurde, nicht notwendigerweise auch als Variationspunkt im Variabilitätsmodell erscheinen wird. Stattdessen werden hierfür `Variant Objects` herangezogen, die zusammenhängende aber verteilte Variationspunkte in einem Simulink-Modell zusammenfassen.

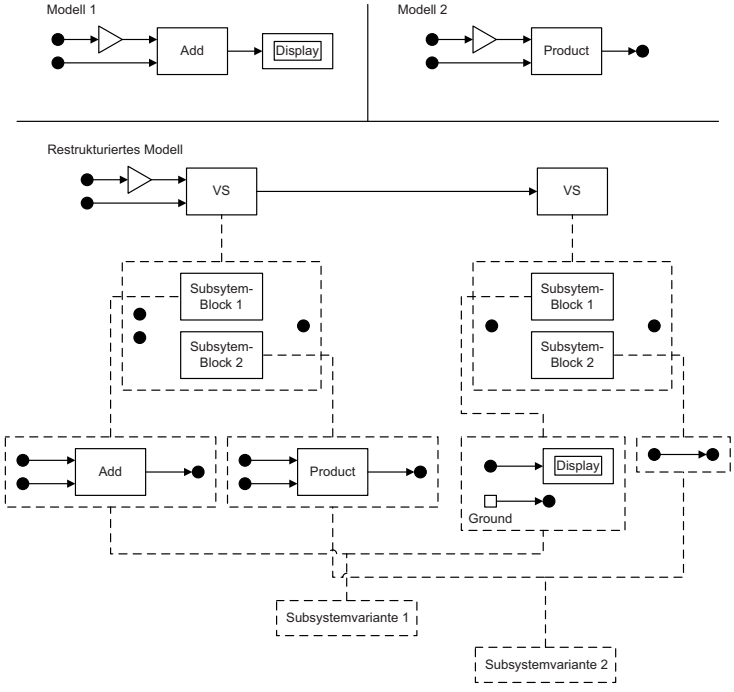
Abbildung 6.31 zeigt hierfür ein Beispiel. Das Simulink-Modell auf der linken Seite der Abbildung besteht aus zwei Variationspunkten, die jeweils durch einen `Model Variants`-Block realisiert wurden. Die Varianten beider Variationspunkte sind insofern zusammenhängend, dass jeweils die beiden linken bzw. rechten Varianten zusammengehören. Sie werden über die `Variant Objects` mit dem Namen `Modellvariante 1` und `Modellvariante 2` gebündelt. Es existieren also zwei Varianten, die im Variabilitätsmodell erfasst werden müssen. Für diese muss also ein Variationspunkt definiert werden, der beide Varianten umfasst. Auf der rechten Seite der Abbildung wird dies veranschaulicht. Es wurde ein Variationspunkt bestehend aus zwei Varianten definiert: (1) `Add_Simulation` und (2) `Product_Codegen`. Diese beiden Varianten werden mit den `Variant Objects` des Variabilitätsmechanismus assoziiert, sodass die Korrespondenz sichergestellt wird.

6.5. Anwendungsbeispiel: Fahrzeugzugangssystem

Die wesentlichen Konzepte für dieses Kapitel wurden in den vorangegangenen Abschnitten erläutert. In diesem Abschnitt werden diese anhand eines Anwendungsbeispiels illustriert. Das Beispiel basiert auf dem Fahrzeugzugangssystem. Es umfasst dabei die Sensorik, die erforderlich ist, wenn die Zentralverriegelung durch einen Sollwertgeber, wie beispielsweise dem mechanischen Schlüssel oder der



(a) Anwendung mit `Model Variants`



(b) Anwendung mit `Variant Subsystem`

Abbildung 6.30.: Anwendung der Restrukturierungsregeln an einem komplexen Beispiel

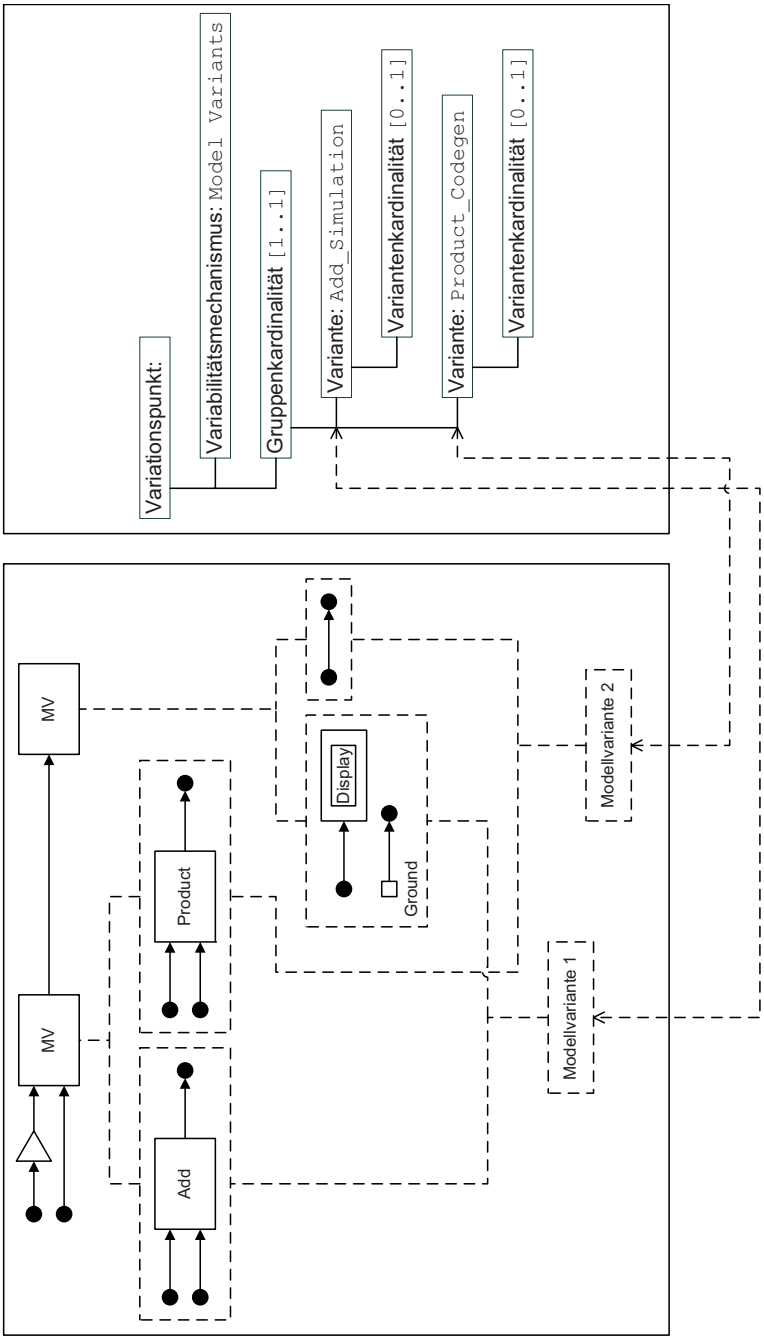


Abbildung 6.31.: Das Variabilitätsmodell und die Anbindung an das restrukturierte Simulink-Modell

sensoren_zentralverriegelung.mdl	sensoren_komfortzugang.mdl
schluessel_linksdrehung :Inport(1)	schluessel_linksdrehung :Inport(1)
schluessel_rechtsdrehung :Inport(2)	schluessel_rechtsdrehung :Inport(2)
fernbedienung_taste_verriegeln :Inport(3)	fernbedienung_taste_verriegeln :Inport(3)
fernbedienung_taste_entriegeln :Inport(4)	fernbedienung_taste_entriegeln :Inport(4)
verriegelungswunsch :Outport(1)	verriegelungswunsch :Outport(1)
entriegelungswunsch :Outport(2)	entriegelungswunsch :Outport(2)
hallsensor_schluesselbewegung :Subsystem	hallsensor_schluesselbewegung :Subsystem
aussenantenne_fernbedienung :Subsystem	aussenantenne_fernbedienung :Subsystem
datenaufbereitung_verriegeln :BusCreator	datenaufbereitung_verriegeln :BusCreator
datenaufbereitung_entriegeln :BusCreator	datenaufbereitung_entriegeln :BusCreator

Tabelle 6.3.: Eine Liste von Vergleichspaaren für die zu differenzierenden Simulink-Modelle

Funkfernbedienung, initiiert wird. Eine Erweiterung hiervon ist der Komfortzugang, der die Initiierung ohne aktive Verwendung eines Sollwertgebers ermöglicht. Dabei werden weitere Sensoren an die Türgriffe sowie Sendeantennen für die automatische Kommunikation mit der Funkfernbedienung angebracht.

In Abbildung 6.32 sind zwei Simulink-Modelle dargestellt, die sowohl die Sensorik für die Grundausstattung (`sensoren_zentralverriegelung.mdl`) als auch für die Sonderausstattung (`sensoren_komfortzugang.mdl`) modellieren. Die Initiierung kann dabei über den mechanischen Schlüssel vollzogen werden. Über einen Hallsensor, der am Türschloss installiert ist, wird erkannt, wenn mit dem Schlüssel eine Links- bzw. eine Rechtsdrehung stattfindet. Entsprechend wird der Verriegelungs- bzw. Entriegelungswunsch an die Kernfunktion der Zentralverriegelung weitergeleitet. Weiterhin kann die Verriegelung/Entriegelung über die Tasten auf der Funkfernbedienung gesteuert werden. Die entsprechenden Eingaben werden über die Außenantenne erkannt und an die Kernfunktion weitergeleitet.

Der Komfortzugang wird durch Beibehaltung der bisher beschriebenen Funktionen und die Erweiterung für den passiven Zugang ins Fahrzeug modelliert. Für die Erweiterung werden zwei kapazitive Sensoren eingesetzt, die an der Griffmulde und an einer sensitiven Fläche am Türgriff angebracht sind. Bei Berührung dieser beiden Flächen wird ebenfalls der Verriegelungs- bzw. Entriegelungswunsch erkannt.

Bevor nun die Differenzierung angewendet werden kann, müssen zunächst die Vergleichspaare festgelegt werden, damit der Algorithmus entscheiden kann, welche Simulink-Blöcke miteinander verglichen werden. Durch den interaktiven Ansatz wurden die Vergleichspaare aus Tabelle 6.3 festgelegt. Der Differenzierungsalgorithmus durchläuft im Anschluss folgende Schritte:

1. Es wird ein Kommunalitätsmodell erzeugt. Da in `sensoren_komfortzugang.mdl` Blöcke existieren, die kein Korrelat im Simulink-Modell `sensoren_zentralverriegelung.mdl` besitzen (zum Beispiel `kapazitiver_drucksensor1_komfort` und `kapazitiver_drucksensor2_komfort`), wird das Kommunalitätsmodell als Objekt vom Typ `Undef :: RootModel` erzeugt. Als Repräsentation wird in diesem Fall die Hintergrundfarbe des Kommunalitätsmodell hellgrau dargestellt.

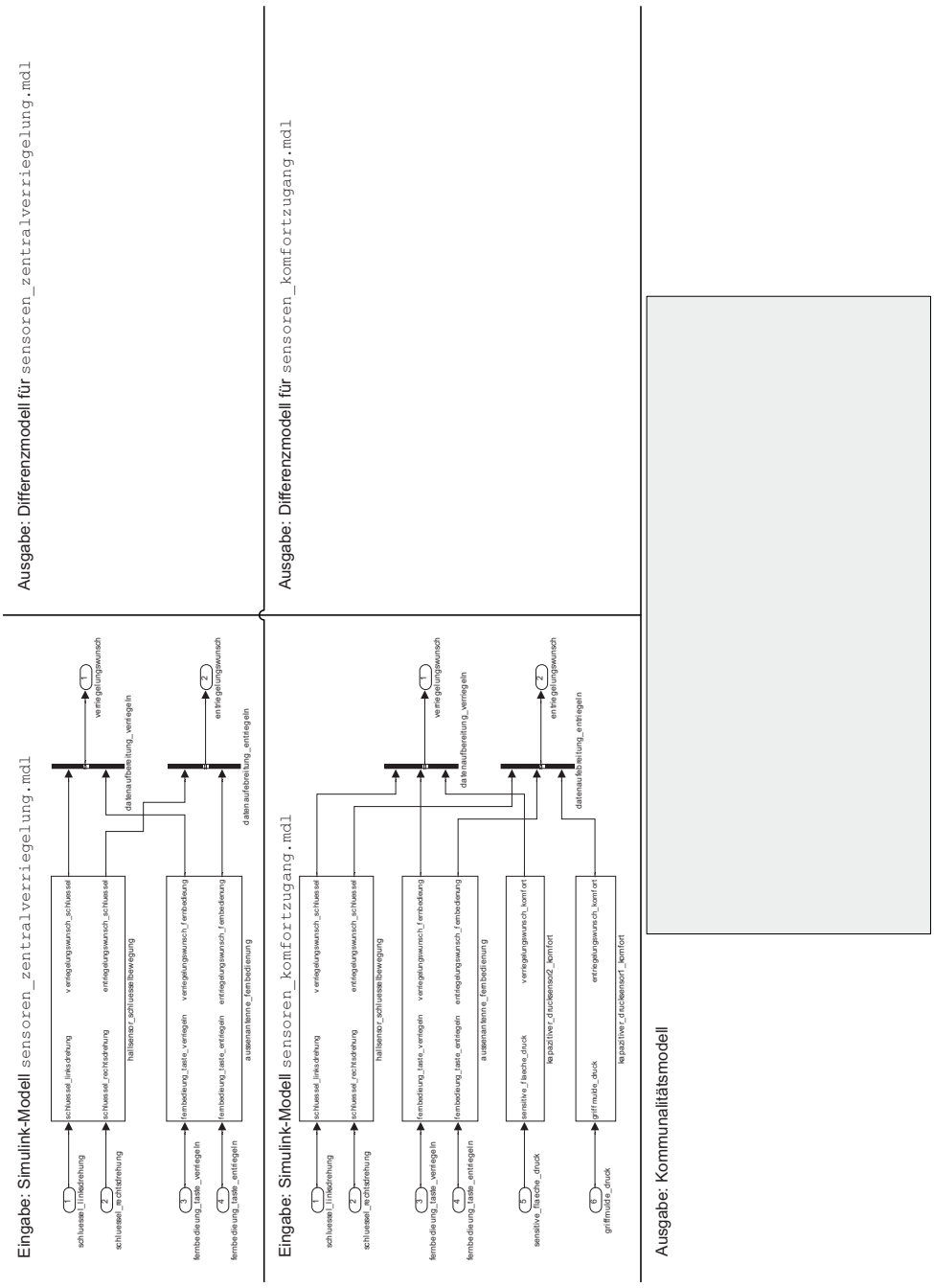


Abbildung 6.32.: Die Situation nach Ausführung der ersten zwei Schritte

2. Es werden zwei Differenzmodelle erzeugt. Sie sind beide vom Typ `Diff :: RootModel`.

Abbildung 6.32 zeigt das Resultat nach diesen beiden Schritten. Zu sehen sind die zwei Eingabemodelle (links oben), beide Differenzmodelle (rechts oben) und das Kommunalitätsmodell mit der hellgrauen Hintergrundfarbe (unten).

3. Der Vergleich beider Modelle findet auf höchster Hierarchieebene statt:
   3.1 Für das Kommunalitätsmodell und für beide Differenzmodelle werden jeweils Schnittstellenobjekte vom Typ `SimulinkMetaModel :: Interface` erzeugt.
   3.2 Für beide Differenzmodelle werden weiterhin zwei partielle Schnittstellenobjekte vom Typ `SimulinkMetaModel :: PartialInterface` erzeugt.
   3.3 Da im Modell `sensoren_komfortzugang.mdl` Eingabeports existieren, die kein Gegenstück in `sensoren_zentralverriegelung.mdl` besitzen (zum Beispiel `sensitive_flaeche_druck` und `griffmulde_druck`), wird für das Kommunalitätsmodell eine Eingabeschnittstelle vom Typ `Undef :: PartialInterface` erzeugt. Da für alle Ausgabeports jeweils ein Korrelat existiert, wird eine Ausgabeschnittstelle vom Typ `SimulinkMetaModel :: PartialInterface` erzeugt.
   3.4 Die Differenzierung zwischen allen Ports und ihren Verbindungen ergibt folgende Situation:
   - Es werden vier Eingabeports im Kommunalitätsmodell erzeugt und zwei inkrementelle Eingabeports im Differenzmodell für `sensoren_komfortzugang.mdl`. Alle Eingabeports sind dabei vom Typ `SimulinkMetaModel :: Port`. Als visuelle Repräsentation im Kommunalitätsmodell werden die Eingabeports grau markiert. Dies deutet somit auf einen Variationspunkt in der Eingabeschnittstelle.
   - Es werden zwei Ausgabeports im Kommunalitätsmodell erzeugt. Sie sind ebenfalls vom Typ `SimulinkMetaModel :: Port`.
   - Variationen an Verbindungen gibt es keine, sodass hier keine gesonderten Maßnahmen getroffen werden.

Abbildung 6.33 illustriert das Ergebnis nach Ausführung der bisher beschriebenen Schritte. Die Ports der Eingabeschnittstelle im Kommunalitätsmodell sind grau markiert, da hier ein Variationspunkt identifiziert wurde. Die inkrementelle Variantendefinition ist im Differenzmodell für `sensoren_komfortzugang.mdl` modelliert. In der Ausgabeschnittstelle hingegen sind keine Variationspunkte vorhanden, sodass diese wie üblich repräsentiert werden.

4. –

5. Für das Blockvergleichspaar `hallsensor_schluesselbewegung` aus `sensoren_zentralverriegelung.mdl` und `hallsensor_schluesselbewegung` aus `sensoren_komfortzugang.mdl` werden folgende Schritte durchgeführt:

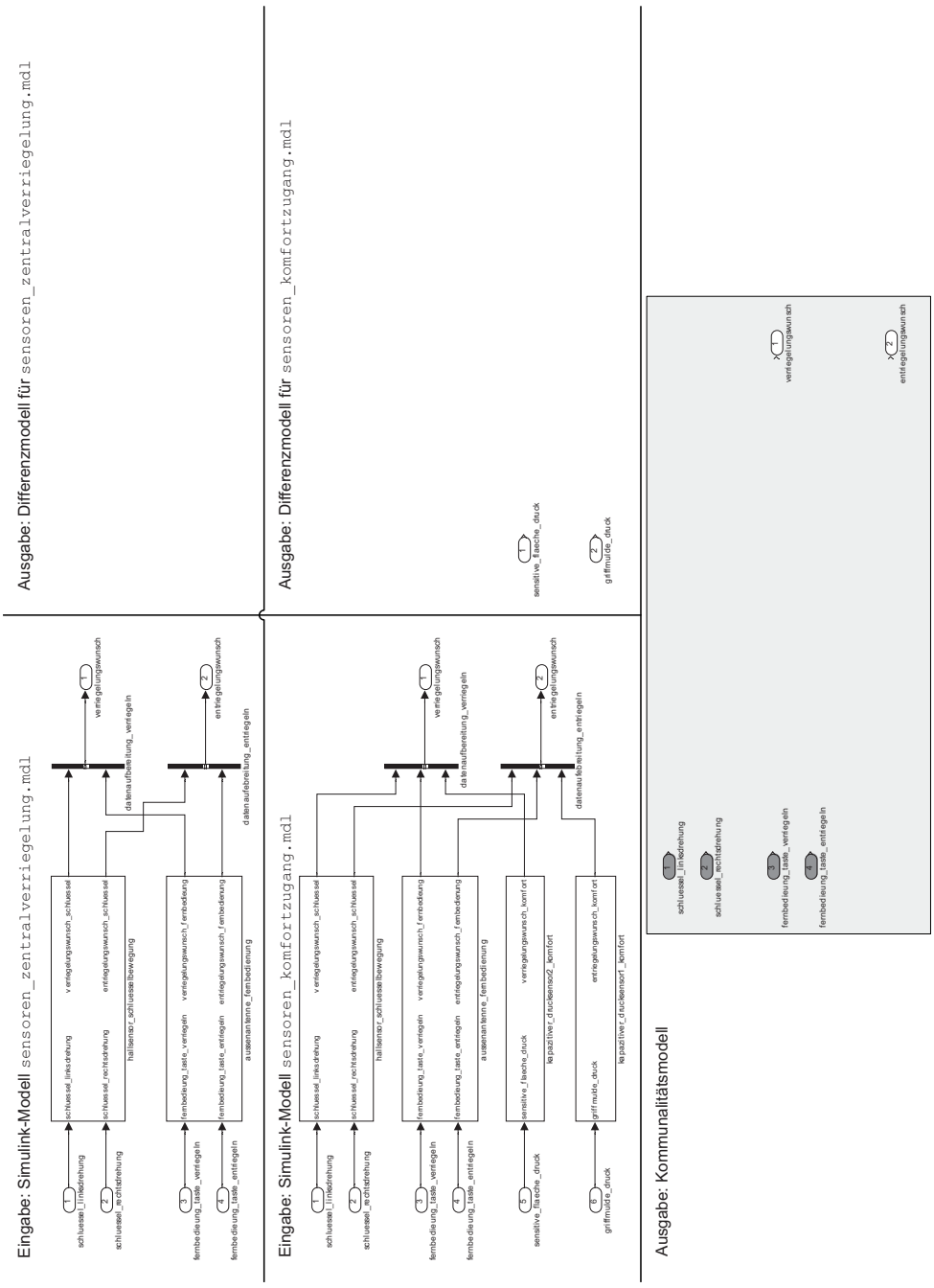


Abbildung 6.33.: Die Situation nach Ausführung der ersten drei Schritte

5.1 Beide Blöcke sind vom gleichen Typ (`Subsystem`). Im Kommunalitätsmodell wird daher ein Objekt vom Typ `SimulinkMetaModel :: Subsystem` erzeugt.

5.2 Weiterhin werden zwei Objekte vom Typ `Diff :: Subsystem` in den jeweiligen Differenzmodellen erzeugt.

5.3 Die `name`-Attribute der Objekte in den Differenzmodellen werden entsprechend gesetzt.

5.4 Die Differenzierung zwischen allen Ports und ihren Verbindungen ergibt keine Differenzen, sodass folgende Situation entsteht:

- Es werden zwei Eingabeports und ein Ausgabeport im Kommunalitätsmodell für das Subsystemobjekt erzeugt. Alle Ports sind dabei vom Typ `SimulinkMetaModel :: Port`.
- Variationen an Verbindungen gibt es keine, sodass hier keine gesonderten Maßnahmen getroffen werden.

5.5 Beide Blöcke sind Subsysteme. Der Algorithmus sieht nun vor, rekursiv in den inneren Hierarchieebenen zu differenzieren. Diese Schritte werden hier zur Vereinfachung übersprungen.

5.6 –

Analog wird das Blockvergleichspaar `aussenantenne_fernbedieung` aus `sensoren_zentralverriegelung.mdl` und `aussenantenne_fernbedieung` aus `sensoren_komfortzugang.mdl` behandelt. Bei der Differenzierung der `Bus Creator`-Blöcke ergibt sich allerdings ein kleiner Unterschied, da die Blockschnittstellen in den jeweiligen Modellen variieren. Exemplarisch wird der Vorgang im Folgenden für ein Vergleichspaar erläutert. Für den Zweiten erfolgt dies analog.

5. Für das Blockvergleichspaar `datenaufbereitung_verriegeln` aus `sensoren_zentralverriegelung.mdl` und `datenaufbereitung_verriegeln` aus `sensoren_komfortzugang.mdl` werden folgende Schritte durchgeführt:

   5.1 Beide Blöcke sind vom gleichen Typ (`Bus Creator`). Im Kommunalitätsmodell wird daher ein Objekt vom Typ `SimulinkMetaModel :: BusCreator` erzeugt.

   5.2 Weiterhin werden zwei Objekte vom Typ `Diff :: BusCreator` in den jeweiligen Differenzmodellen erzeugt.

   5.3 Die `name`-Attribute der Objekte in den Differenzmodellen werden entsprechend gesetzt.

   5.4 Die Differenzierung zwischen allen Ports und ihren Verbindungen ergibt folgende Situation:

   - Es werden zwei Eingabeports für das `BusCreator`-Objekt im Kommunalitätsmodell und ein inkrementeller Eingabeport für das `BusCreator`-Objekt im Differenzmodell für `sensoren_komfortzugang.mdl` erzeugt. Sie sind dabei vom Typ `SimulinkMetaModel :: Port`. Als visuelle Repräsentation wird der Block im Kommunalitätsmodell blau markiert. Dies deutet auf einen Variationspunkt an der Schnittstelle.

- Es wird ein Ausgabeport für das `BusCreator`-Objekt im Kommunalitätsmodell erzeugt.
- Variationen an Verbindungen gibt es keine, sodass hier keine gesonderten Maßnahmen getroffen werden.

5.5 –

5.6 –

6. Die Blöcke `kapazitiver_drucksensor1_komfort` und `kapazitiver_drucksensor2_komfort` aus `sensoren_komfortzugang.mdl` haben keine Gegenstücke in `sensoren_zentralverriegelung.mdl`. Sie sind also inkrementelle Variantedefinitionen, die in das Differenzmodell für `sensoren_komfortzugang.mdl` kopiert werden.

7. Schließlich werden alle Ports in den drei erzeugten Modellen miteinander verbunden.

Abbildung 6.34 zeigt das Resultat nach der Differenzierung. Insgesamt kann festgestellt werden, dass Variationspunkte an Vergleichspaaren, an der Eingabeschnittstelle des Modells sowie an der Eingabeschnittstelle der beiden `Bus Creator`-Blöcke existieren. In den Differenzmodellen sind sowohl die substitutionellen als auch die inkrementellen Variantendefinitionen enthalten. Sie ersetzen bzw. erweitern das Kommunalitätsmodell, sodass die Zusammenführung beider stets eine gültige Variante ergibt.

Anhand dieser durch die Differenzierung gewonnenen Erkenntnisse kann nun eine Restrukturierung durchgeführt werden, die identifizierte Variationspunkte durch die Variabilitätsmechanismen `Model Variants` oder `Variant Subsystem` realisiert. Im Folgenden wird diese Restrukturierung durch die Verwendung von `Variant Subsystem` illustriert. Durch die Anwendung der Restrukturierungsregeln ergibt sich die Situation in Abbildung 6.35. Es sind nun nicht mehr zwei Modelle vorhanden, die zum einen für die Zentralverriegelung und zum anderen für den Komfortzugang existierten. Stattdessen wurden beide Modelle zusammengeführt. Variationen werden dabei durch den Variabilitätsmechanismus geeignet gekapselt. Der Variationspunkt an der Eingabeschnittstelle des Modells wird durch einen `Variant Subsystem` erfasst. Da für die Zentralverriegelung die beiden Eingabeports `sensitive_flaeche_druck` und `griffmulde_druck` nicht erforderlich sind, werden sie in der entsprechenden Variante durch `Terminator`-Blöcke abgefangen. Um die Adaptivität im Modell sicherzustellen, werden aber zwei Füllsignale durch die beiden `Ground`-Blöcke generiert. Die beiden Blöcke `kapazitiver_drucksensor1_komfort` und `kapazitiver_drucksensor2_komfort` besaßen kein Pendant in `sensoren_zentralverriegelung.mdl`. Dieser Variationspunkt wird ebenfalls durch den `Variant Subsystem`-Block realisiert. Ähnlich wie am Variationspunkt der Eingabeschnittstelle werden bei der Variante für die Zentralverriegelung die Signale durch `Terminator`-Blöcke abgefangen und durch `Ground`-Blöcke weitere Füllsignale generiert. Schließlich wird der Variationspunkt an der Eingabeschnittstelle des `Bus Creator`-Blocks gekapselt. Der dritte Eingabeport ist dabei für die Variante der Zentralverriegelung

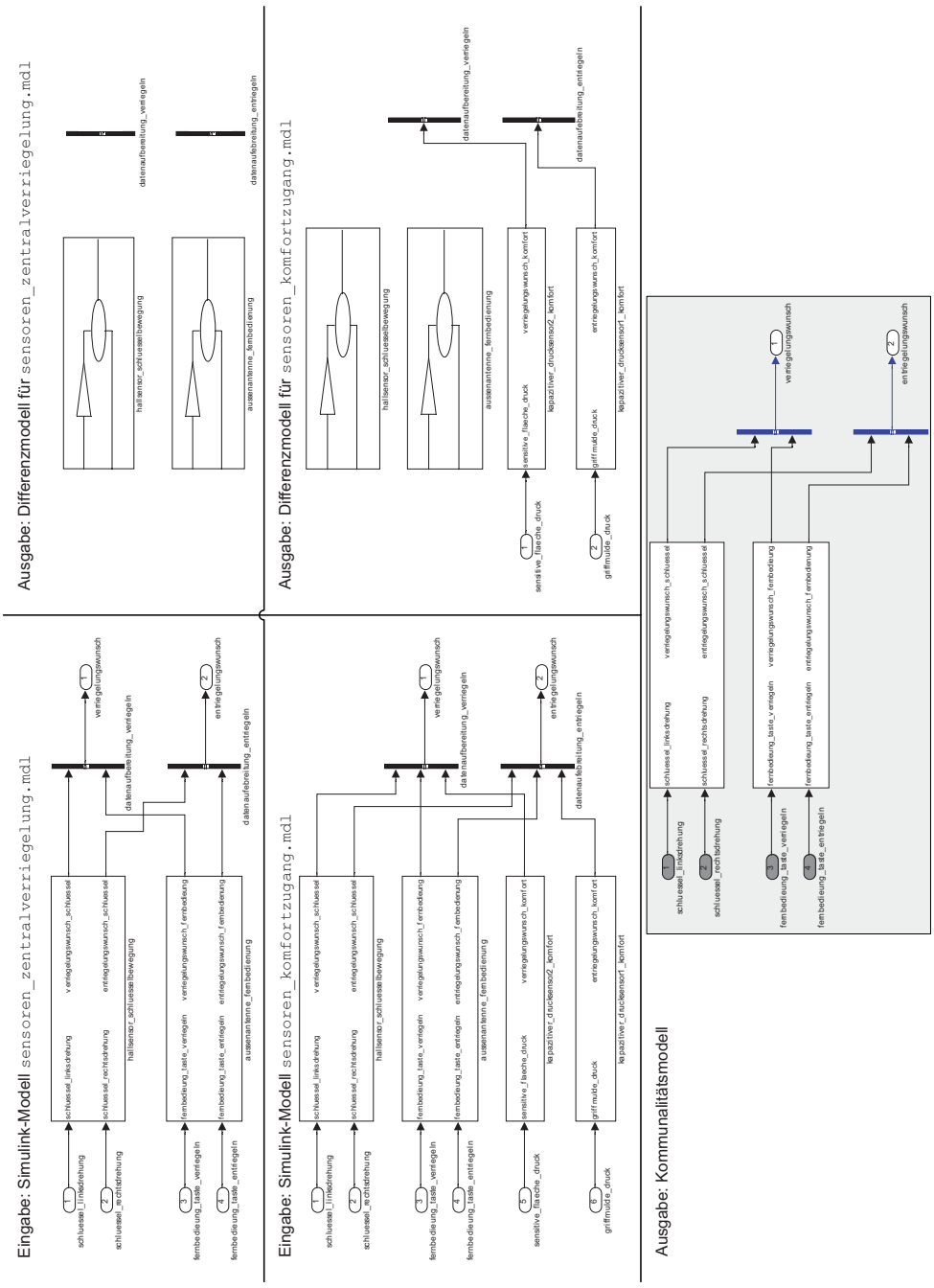


Abbildung 6.34.: Die Situation nach Abschluss der Differenzierung

nicht erforderlich. Dieser wird daher durch einen Terminator-Block abgefangen. Für den weiteren Bus Creator-Block erfolgt dies analog. Durch die Definition von Variant Objects und die korrekte Zuordnung zu den Varianten, wird es immer möglich sein, die zugehörigen Bestandteile einer Variante zu laden. In der Abbildung wird dies über die beiden Variant Objects Zentralverriegelung und Komfortzugang sichergestellt.

Die Variant Objects bilden auch den Zugriffspunkt für das Variabilitätsmodell. Da durch Variant Objects bereits feingranulare Variationspunkte erfasst werden, reduziert sich somit der Modellierungsaufwand im Variabilitätsmodell. Hier können somit Variationspunkte auf grobgranularer Ebene modelliert werden. Es besteht allerdings auch die Möglichkeit das Variabilitätsmodell ebenfalls feingranularer zu modellieren, sodass alle im Simulink-Modell bestehenden Variationspunkte erfasst werden. In Abbildung 6.36 wird die beschriebene Assoziation dargestellt. Der Variationspunkt Fahrzeugzugangssystem wird durch den Variabilitätsmechanismus Variant Subsystem realisiert. Die Varianten Zentralverriegelung und Komfortzugang werden mit den korrespondierenden Variant Objects verknüpft. Bei einer Konfiguration werden die Parameter entsprechend gesetzt, sodass die korrekte Variante im Simulink-Modell aktiviert wird.

6.6. Realisierung

In diesem Abschnitt werden die wesentlichen Realisierungsentscheidungen erläutert. Insbesondere wurde die Differenzierung und die hieraus erforderlichen Aspekte durch ein Werkzeug realisiert. Als Programmiersprache wurde Java eingesetzt. Weiterhin wurde die modellbasierte Technologie EMF angewendet. Auch externe Werkzeuge, die eine Anbindung an Matlab erlauben, wurden verwendet. Im Folgenden werden die wichtigsten Punkte beschrieben.

6.6.1. Metamodelle

Die Basis zur Differenzierung von Simulink-Modellen bilden die erstellten Metamodelle: (1) Simulink-Metamodell, (2) Kommunalitätsmetamodell und (3) Differenzmetamodell. Sie wurden in Abschnitt 6.2 detailliert erläutert. Realisiert wurden diese Modelle durch die Eclipse EMF-Technologie [Gro09, SBPM08]. EMF wurde auch bereits in Abschnitt 4.4 beschrieben. Die Technologie wird daher hier nicht erneut erklärt. Die beschriebenen Metamodelle dienen als Eingabe für EMF. Hieraus wird der entsprechende Code automatisch generiert. Implementierungsdetails werden daher hier ausgelassen.

6.6.2. Interaktionen mit Matlab Simulink

Da das Werkzeug in der Eclipse-Umgebung realisiert ist, aber die Simulink-Modelle in der Matlab-Umgebung entstehen und verwaltet werden, bedarf es an einer Kommunikation beider Umgebungen. Im Speziellen bedeutet dies, dass Simulink-Modelle

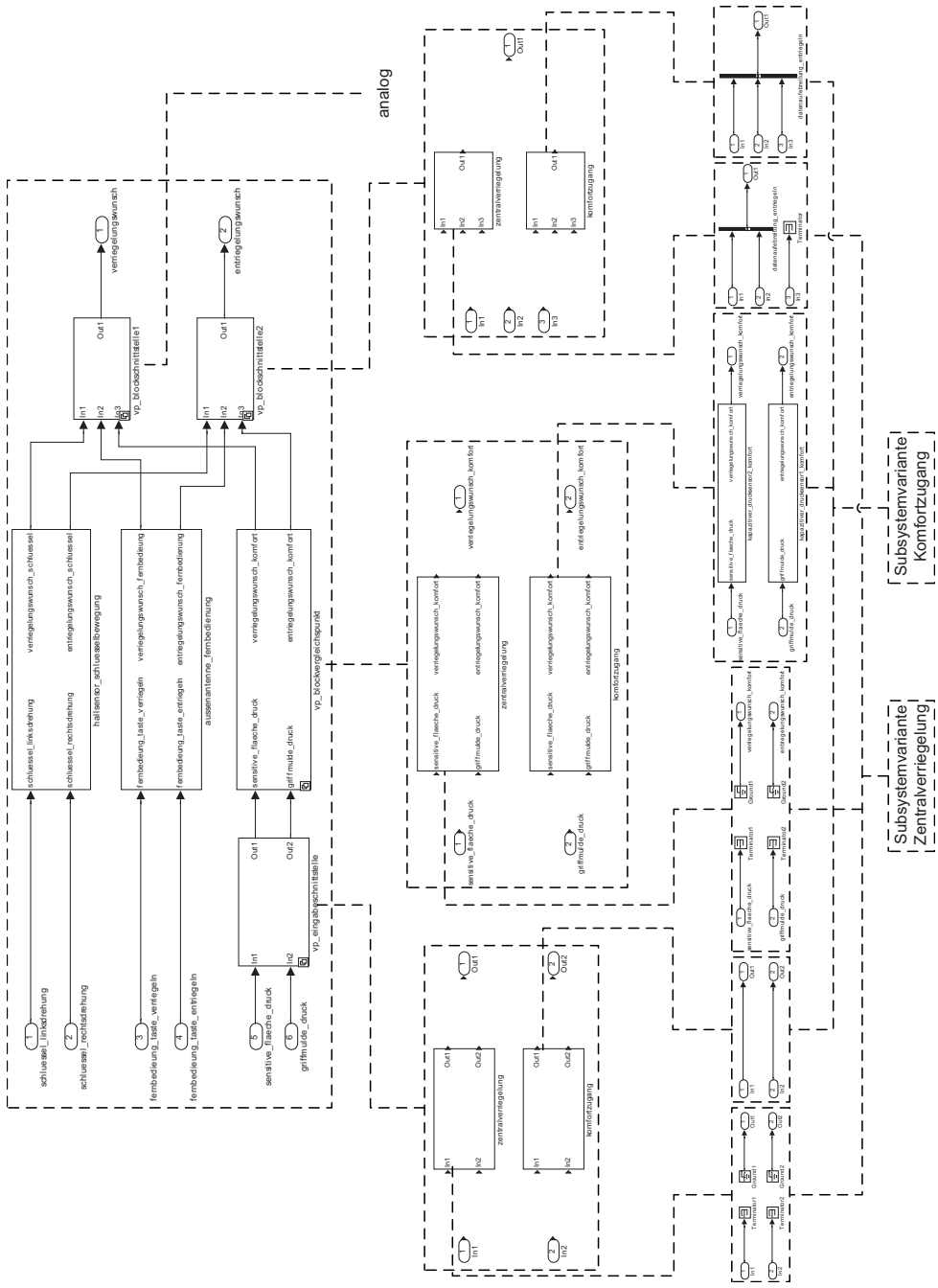


Abbildung 6.35.: Das Simulink-Modell nach der Restrukturierung

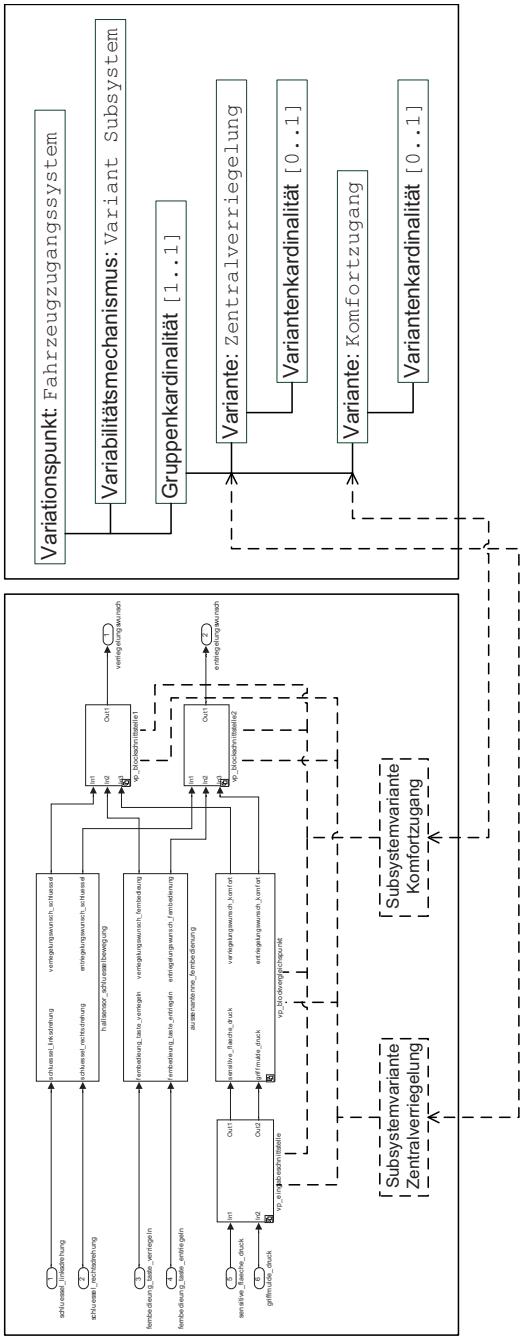


Abbildung 6.36.: Die Verknüpfung des restrukturierten Modells zum Variabilitätsmodell

in die Eclipse-Umgebung importiert werden müssen, damit die Differenzierung hier angewendet werden kann. Nach Beendigung der Differenzierung ist schließlich eine Exportierung aus der Eclipse-Umgebung wieder zurück in die Matlab-Umgebung erforderlich, um die erzielten Ergebnisse repräsentieren zu können.

In dieser Arbeit werden zu diesem Zweck keine Simulink-Dateien gelesen oder beschrieben. Stattdessen wird ein Ansatz gewählt, der eine Verbindung mit Matlab herstellt und Anfragen in Bezug auf Eigenschaften der zu differenzierenden Simulink-Modelle stellt. Beispiele für derartige Anfragen sind

- Öffne ein Modell
- Schließe ein Modell
- Füge einen Block/Linie/Parameter hinzu
- Lösche einen Block/Linie/Parameter
- Finde alle Blöcke/Linien/Ports
- Speichere ein Modell
- ...

Hierfür wird das Werkzeug `matlabcontrol v3.1.0`, entwickelt von Joshua Kaplan, verwendet [Kap]. `matlabcontrol` bietet ein Java Application Programming Interface (API), welches Matlab-Aufrufe aus Java heraus unterstützt. Die API basiert auf Java Matlab Interface (JMI). JMI ist Teil der Matlab-Software.

Im Folgenden werden sowohl der Import- als auch der Exportvorgang genauer erläutert.

6.6.2.1. Import

Die Importierung ist die Aktivität, in der Simulink-Modelle aus der Matlab-Umgebung in die Eclipse-Umgebung abgebildet werden. Für alle gelesenen Simulink-Elemente werden dabei korrespondierende Objekte aus der Klasse `SimulinkMetaModel` instanziiert. Der Großteil dieser Klassen wird sich aus den Beschreibungen der verschiedenen Simulink-Blöcke ergeben. Wird erneut das Simulink-Metamodell aus Abbildung 6.6 betrachtet, ist ersichtlich, dass diese Beschreibungen als Erweiterungen der abstrakten Klasse `Block` manuell implementiert werden müssen. Dies gilt zum Beispiel für `Add`-, `Product`- oder `Bus Creator`-Blöcke. Die Importfunktion muss über diese Blockimplementierungen informiert werden. Zu diesem Zweck wird das Entwurfsmuster Fabrikmethode (engl. Factory Method) angewendet [GHJV10].

Listing 6.1 zeigt die Schnittstelle `BlockFactory`, welche bei der Importierung verwendet wird, um Objekte für Blöcke auf Basis der Simulink-Typnamen zu erzeugen. Hierfür wird eine Zuordnung zwischen Simulink-Typnamen und den manuell implementierten Klassen realisiert.

```
1  package SMD.transformation.importing;
2
3  /**
4   * A factory for creating instances of different blocks according to Block Type
5   * names of Simulink
6   */
7  public interface BlockFactory {
8          /**
9           *
10          * @param typeName
11          *            block-type name according to Simulink
12          * @return an instance of the block, or null if the type is not supported
13          */
14         ImportableBlock createBlockInstance(String typeName);
15 }
```

Listing 6.1: Die Klasse `BlockFactory.java` für die Fabrikmethode

6.6.2.2. Export

Für die Exportierung wird über die Schnittstelle `ExportableBlock` eine Menge von Methoden definiert, die von einer Blockklasse implementiert werden muss, um exportiert werden zu können. Bei der Exportierung ist noch zu beachten, dass jeder Blocktyp eine eigene Methode zur Erzeugung von Ports und Verbindungen besitzt. Die Schnittstelle `ExportableBlock` muss daher zu diesem Zweck eine geeignete Methode zur Verfügung stellen (`createPorts()`). Listing 6.2 illustriert die Methoden dieser Klasse.

6.6.3. Differenzierungsalgorithmus

Der Differenzierungsalgorithmus wurde bereits in Abschnitt 6.3.3 im Detail behandelt. Die Implementierung des Algorithmus ist eine reine Abbildung der Beschreibung in die Programmiersprache Java. Daher werden an dieser Stelle entsprechende Codeausschnitte ausgelassen. Stattdessen werden anhand von Screenshots die Implementierungsergebnisse erläutert.

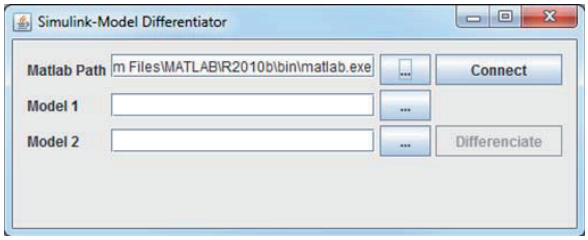
Abbildung 6.37 zeigt die grafische Oberfläche des implementierten Werkzeugs. Es besteht aus einem einfachen Fenster, das drei Eingaben benötigt (vgl. Abbildung 6.37(a)):

1. Den Pfad der Matlab-Ausführungsdatei
2. Das erste zu differenzierende Simulink-Modell
3. Das zweite zu differenzierende Simulink-Modell

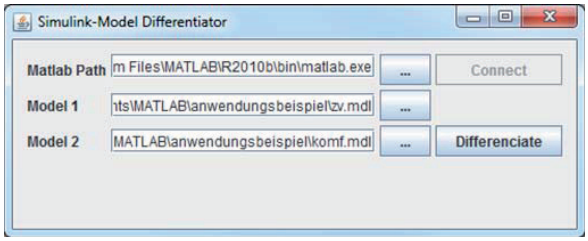
Nach Eingabe des Pfads kann die Verbindung zu Matlab hergestellt werden. Hierfür steht die Connect-Taste zur Verfügung. Erst nachdem die Verbindung hergestellt ist, wird die Differenciate-Taste aktiviert. Nun können auch die Pfade der zu differenzierenden Modelle angegeben werden (vgl. Abbildung 6.37(b)). Das Ergebnis

```
1  package SMD.transformation.exporting;
2
3  import matlabcontrol.MatlabInvocationException;
4  import matlabcontrol.RemoteMatlabProxy;
5  import SMD.SimulinkMetaModel.Block;
6
7  public interface ExportableBlock extends Block {
8          /**
9           * Create the block in the exported Simulink model. It might be done in many
10          * ways: creating new block, cloning from a template ...etc.
11          *
12          * @param blockFQN
13          *             the block fully qualified names name (path in the nested
14          *                         models)
15          * @param proxy
16          *             matlab connection
17          * @throws MatlabInvocationException
18          *              when an error occurs during communication with Matlab
19          */
20         void createBlock(String blockFQN, RemoteMatlabProxy proxy)
21                         throws MatlabInvocationException;
22
23         /**
24          * Creates the ports of the block in the way suitable for the block type. It
25          * can be ignored for subsystems.
26          *
27          * @param blockFQN
28          *             the fully qualified name of the block containing the ports
29          * @param proxy
30          *             the connection to matlab
31          * @throws MatlabInvocationException
32          */
33         void createPorts(String blockFQN, RemoteMatlabProxy proxy)
34                         throws MatlabInvocationException;
35
36         /**
37          * To export the block-specific attributes.
38          *
39          * @param blockFQN
40          * @param proxy
41          * @throws MatlabInvocationException
42          */
43         void exportSpecificAttributes(String blockFQN, RemoteMatlabProxy proxy)
44                         throws MatlabInvocationException;
45
46         /**
47          * Returns the name of this block type used in Simulink
48          * @return
49          */
50         String getSimulinkBlockTypeName();
51 }
```

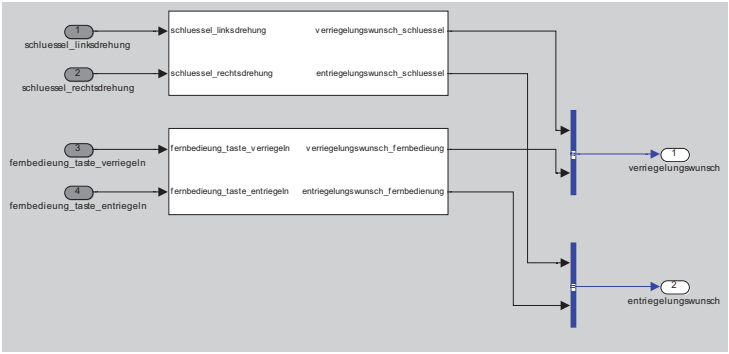
Listing 6.2: Die Schnittstelle `ExportableBlock.java`



(a) Die Verbindung zu Matlab herstellen



(b) Die Simulink-Modelle auswählen und differenzieren



(c) Das Resultat nach der Differenzierung

Abbildung 6.37.: Screenshots der Werkzeugein- und -ausgabe

der Differenzierung wird in einem Simulink-Modell durch farbliche Markierungen dargestellt (vgl. Abbildung 6.37(c)). Dieses Modell kann zur weiteren Analyse in Bezug auf Restrukturierung herangezogen werden.

6.7. Verwandte Arbeiten

Auf Architekturebene werden Variabilitätsdimensionen betrachtet, die auf Funktionsebene nicht existieren. Zudem wird aufgrund der unzweckmäßigen Vorgehensweise durch etwa Copy-Paste-Strategien die systematische Wiederverwendung bedeutend erschwert. Es wird zu der Situation kommen, in der ein Komplexitätsgrad erreicht

wird, der eine Analyse der entworfenen Architekturen erfordern wird. Die Werkzeugunterstützung der Analyse in Form einer Differenzierung von Simulink-Modellen ist daher ein wichtiger Aspekt in diesem Kapitel gewesen. Die Differenzierung ist allerdings nur ein erster Schritt in Richtung Wiederverwendbarkeit. Es Bedarf an Restrukturierungsmaßnahmen, um Variabilität in den Modellen geeignet zu realisieren und durch Variabilitätsmodelle zu dokumentieren und zu repräsentieren. Auch diese Aspekte wurden in diesem Kapitel bearbeitet.

In der Literatur existieren ebenfalls einige Arbeiten, die oben erwähnte Aspekte mehr oder weniger berücksichtigen. Diese werden im Folgenden genauer beschrieben. Zudem werden sie durch Bewertungskriterien, die nachfolgend erläutert werden, evaluiert und im Anschluss miteinander verglichen. Die Kriterien umfassen dabei folgende Punkte:

1. *Vergleichspaare:* Durch dieses Kriterium wird untersucht, auf welche Art Vergleichspaare zwischen Simulink-Modellen ermittelt werden.
2. *Differenzierung:* Hierbei wird geprüft, welche Methode zur Differenzierung herangezogen wird.
3. *Repräsentation:* Die Ergebnisse der Differenzierung werden zur weiteren Verarbeitung benötigt. Daher ist es wichtig, die Repräsentationsform der Ergebnisse zu ermitteln und ihre Eignung zu bewerten.
4. *Strukturerhaltung:* Ein mit dem Repräsentationskriterium eng verbundenes Kriterium ist die Frage nach der Strukturerhaltung der Modelle. Dabei wird analysiert, ob die differenzierten Modelle dieselbe Modellstruktur (wie zum Beispiel den Erhalt aller Hierarchieebenen) aufweisen oder ob aufgrund der Differenzierung Änderungen am Modell vorgenommen wurden.
5. *Variabilitätsmechanismen:* Durch das Kriterium wird ausgewertet, ob Variabilitätsmechanismen zur Realisierung der ermittelten Variationspunkte betrachtet werden.
6. *Restrukturierung:* Wenn Variabilitätsmechanismen eingesetzt werden, ist eine Restrukturierung der Modelle erforderlich. Wie dabei die Restrukturierung vorgenommen wird, wird durch dieses Kriterium beantwortet.
7. *Variabilitätsmodellierung:* Hierbei wird bewertet, ob die identifizierte Variabilität durch Variabilitätsmodelle erfasst und repräsentiert wird.

6.7.1. CloneDetective

Deißenböck et al. entwickelten in ihren Arbeiten ein Rahmenwerk zur Identifizierung von Klonen auf Modell- und Codeebene [DHJ+08, JDH09, JDHW09, DHJ10, JDH10]. Auf Modellebene wurden dabei Konzepte zur Unterstützung von Matlab Simulink entwickelt. Auch in dieser Arbeit liegt der Fokus in der Identifizierung von Gemeinsamkeiten in Simulink-Modellen, sodass im Folgenden dieser Aspekt genauer beschrieben wird.

Motiviert wird der Ansatz aufgrund steigender Komplexität in modellbasierten Softwaresystemen, die zum Teil infolge von redundanten Modellstrukturen innerhalb dieser Systeme entsteht. Diese Art von Redundanz wird in diesem Zusammenhang als Klon bezeichnet. Das Erkennen dieser Klone ist somit ein essenzieller Schritt, die Komplexität von Simulink-Modellen zu reduzieren und gleichzeitig die Wiederverwendbarkeit zu erhöhen.

Der Ansatz besteht im Wesentlichen aus der Umwandlung von Simulink-Modellen in markierte Graphen und der Ermittlung von Klonpaaren aus diesen Graphen. Jedes Simulink-Modell wird dabei zunächst so modifiziert, das es ausschließlich aus atomaren Blöcken besteht. Es werden also sämtliche Subsysteme eliminiert. Ein Block entspricht dabei einem Knoten im Graphen. Eine Verbindung zwischen Blöcken entspricht den Kanten zwischen Knoten. Jedem Knoten und jeder Kante wird im Anschluss eine Marke zugewiesen. Die Marke enthält Attribute, die die wesentlichen Differenzierungsmerkmale beinhalten. Für Blöcke wäre dies zum Beispiel der Typ des Blocks. Für Kanten sind dies die Indizes der Quell- und Zielports. Zwei Blöcke oder zwei Kanten werden dann als äquivalent betrachtet, wenn sie die gleiche Marke besitzen. Die eigentliche Ermittlung von Klonpaaren wird dann durch einen heuristischen Algorithmus durchgeführt.

Die Bewertung dieses Ansatzes anhand der beschriebenen Kriterien liefert folgende Ergebnisse:

1. *Vergleichspaare:* Zuweisung von Marken an Knoten und Kanten. Die Attribute einer Marke enthalten die typischen Merkmale des Simulink-Blocks oder der Verbindung. Zwei Konten oder Kanten sind dann gleich, wenn sie die gleiche Marke enthalten. Dies wird automatisch ermittelt.

2. *Differenzierung:* Es wird ein heuristischer Graphmatchingalgorithmus angewendet.

3. *Repräsentation:* Gemeinsamkeiten werden in Simulink durch farbliche Markierungen der Blöcke gekennzeichnet.

4. *Strukturerhaltung:* Die Simulink-Modelle werden verflacht, sodass sämtliche Subsysteme eliminiert sind. Demnach wird die ursprüngliche Struktur nicht beibehalten.

5. *Variabilitätsmechanismen:* Keine Unterstützung.

6. *Restrukturierung:* Keine Unterstützung.

7. *Variabilitätsmodellierung:* Keine Unterstützung.

6.7.2. Automatische Identifikation von Varianten und Variationspunkten

Ryssel et al. haben in ihren Arbeiten festgestellt, dass für die Migration alter Simulink-Bibliotheken eine Werkzeugunterstützung erforderlich ist, die sämtliche

Modelle auf Gemeinsamkeiten und Variabilität untersucht und hieraus Featuremodelle generiert [RPK10b, RPK10a]. Der Migrationsprozess wird somit zum Teil automatisiert, sodass Kosten und Zeit eingespart werden können.

Der Migrationsprozess besteht dabei im Wesentlichen aus drei Schritten:

1. Ermittlung von Varianten
2. Identifikation von Variationspunkten und Restriktionen
3. Generierung von Featuremodellen

Für eine Menge von Simulink-Modellen werden Varianten anhand einer Ähnlichkeitsberechnung ermittelt. Hierfür wurden gewichtete Kriterien eingeführt, die bei der Berechnung herangezogen werden. Derartige Kriterien sind beispielsweise: (1) Blocknamen, (2) Anzahl identischer Parameterwerte, (3) Verbindungen zu benachbarten Blöcken, (4) Ports und (5) Distanz zu Nachbarblöcken. Die Berechnung erfolgt zudem bei Subsystemen rekursiv. Die Ähnlichkeitsberechnung gibt für jedes Blockpaar anhand der beschriebenen Kriterien einen sogenannten Ähnlichkeitswert aus, der zwichen null und eins liegt. Der Wert null deutet auf keine Ähnlichkeit und der Wert eins besagt eine vollständige Ähnlichkeit. Die gesamte Ähnlichkeitsberechnung zwischen allen Blockpaaren resultiert in einer Ähnlichkeitsmatrix. Im Anschluss werden alle Blöcke in ein sogenanntes Cluster überführt, um die potenziellen Varianten zu ermitteln.

Die Identifikation von Variationspunkten verläuft wiederum auch in mehreren Schritten ab. Zunächst werden Blöcke, Verbindungen und Parameter zwischen den ermittelten Varianten anhand eines Graphen entsprechend zugeordnet. Ein maximaler Clique beinhaltet durch diese Zuordnung die Gemeinsamkeiten zwischen den Varianten. Überschneidungen zwischen Cliques werden auch behandelt, um Fehler, die aufgrund falscher Zuordnung entstehen, zu vermeiden. Anhand einer sogenannten formalen Konzeptanalyse (engl. formal concept analysis) können dann Variationspunkte und Restriktionen identifiziert werden [GW99].

Die identifizierten Variationspunkte und Restriktionen werden im Weiteren in ein Featuremodell überführt, um die Variabilität explizit erfassen zu können. Durch dieses Featuremodell wird es möglich, durch geeignete Konfigurationen, zugeschnittene Simulink-Modelle aus den Bibliotheken zu erhalten.

Die Bewertung dieses Ansatzes anhand der beschriebenen Kriterien liefert folgende Ergebnisse:

1. *Vergleichspaare:* Durch Metriken, wie der Typ oder der Index eines Ports, werden Vergleichspaare automatisch detektiert.
2. *Differenzierung:* Gemeinsamkeiten werden durch Bestimmung des maximalen Cliques mit einer anschließenden formalen Konzeptanalyse bestimmt.
3. *Repräsentation:* Keine Unterstützung.
4. *Strukturerhaltung:* Keine Unterstützung.

5. *Variabilitätsmechanismen:* Keine Unterstützung.

6. *Restrukturierung:* Keine Unterstützung.

7. *Variabilitätsmodellierung:* Gemeinsamkeiten und Variabilität werden in ein Featuremodell überführt.

6.7.3. Modellierung und Konfiguration von Funktionsvarianten

Weiland et al. haben in einer Reihe von Beiträgen ihren Ansatz vorgestellt, um Variabilität in Simulink-Modellen systematisch zu beschreiben und einheitlich zu konfigurieren [WR05, Wei08, DLPW08, DW09, BW09]. Das wichtigste Konzept hierbei ist die Definition des Variationspunktes. Nach *Weiland et al.* kapselt ein Variationspunkt Informationen, wie etwa Variabilitätsmechanismen und Variabilitätsparameter. Variabilitätsmechanismen wurden dabei in den Arbeiten [Wei08, Sch10] ermittelt und für diese ein geeignetes Beschreibungsformat definiert. Der Variabilitätsparameter enthält die Menge der Varianten in Form von Werten sowie die selektierte Variante als Konfiguration.

Zur Modellierung der Variabilität wurde eine Blockbibliothek in Simulink spezifiziert, die sämtliche Blöcke beinhaltet, um Variabilitätsmechanismen zu modellieren. Ein weiterer Bestandteil hierbei ist die Festlegung der Variabilitätsinformationen, mit denen der Variationspunkt einheitlich beschrieben werden kann. Auf diese Weise wird es möglich, die eigentliche Funktionsmodellierung von variantenspezifischen Aspekten zu trennen.

Die Bewertung dieses Ansatzes anhand der beschriebenen Kriterien liefert folgende Ergebnisse:

1. *Vergleichspaare:* Keine Unterstützung.

2. *Differenzierung:* Keine Unterstützung.

3. *Repräsentation:* Keine Unterstützung.

4. *Strukturerhaltung:* Die Modelle werden bei diesem Ansatz nicht verändert.

5. *Variabilitätsmechanismen:* Es können alle definierten Variabilitätsmechanismen verwendet werden.

6. *Restrukturierung:* Keine Unterstützung.

7. *Variabilitätsmodellierung:* Gemeinsamkeiten und Variabilität werden in einem Featuremodell erfasst.

6.7.4. Vergleich

Zur Differenzierung zwischen Simulink-Modellen muss ermittelt werden, welche Blöcke miteinander verglichen werden sollen. Zu diesem Zweck werden Vergleichspaare definiert. Wie bereits in Abschnitt 6.3.2 erwähnt, gibt es sehr vielfältige Möglichkeiten, diese zu bestimmen. In dieser Arbeit wurden zwei Ansätze realisiert: (1) interaktiv durch Benutzereingaben und (2) automatisch durch Überprüfung von bestimmten Metriken. Die hierbei herangezogenen Metriken sind Blocknamen, Blocktypen und Portindizes. Die Arbeiten von *Deißenböck et al.* und *Rysell et al.* verfolgen einen ähnlichen Ansatz. Bei diesen wird allerdings der interaktive Ansatz ausgelassen. Zudem werden Blocknamen als Metrik nicht verwendet. Die Arbeit von *Weiland et al.* hingegen beinhaltet einen derartigen Ansatz nicht, da die Differenzierung nicht angewendet wird.

Für die Differenzierung wurde in Abschnitt 6.3.3 ein rekursiver Algorithmus vorgestellt, der Gemeinsamkeiten und Unterschiede zwischen zwei Simulink-Modellen ermittelt. Auch in den Arbeiten von *Deißenböck et al.* und *Rysell et al.* wird eine Differenzierung angewendet. Während *Deißenböck et al.* einen heuristischen Graphmatchingalgorithmus anwenden, wird in der Arbeit von *Rysell et al.* ein zweistufiges Verfahren eingesetzt, in dem zunächst durch die Bestimmung des maximalen Cliques die Gemeinsamkeiten ermittelt werden und durch die formale Konzeptanalyse im Anschluss die Variationspunkte identifiziert werden. Wie bereits erwähnt, wird in der Arbeit von *Weiland et al.* keine Differenzierung durchgeführt.

Eine weitere wichtige Fragestellung in diesem Zusammenhang war, wie die durch die Differenzierung erhobenen Ergebnisse dem Benutzer dargestellt werden können. In diesem Zusammenhang wurde in dieser Arbeit das Kommunalitätsmodell in die Simulink-Umgebung exportiert und hier wurden durch farbliche Markierungen alle Variationspunkte kenntlich gemacht. In der Arbeit von *Deißenböck et al.* werden ebenfalls Farben eingesetzt. Allerdings werden hier Gemeinsamkeiten markiert. In den weiteren beiden Arbeiten werden keine Maßnahmen zur Repräsentation vorgenommen.

Ein weiterer wichtiger Punkt ist die Frage nach der Strukturänderung von Simulink-Modellen, die aufgrund der Differenzierung notwendig ist. In dieser Arbeit wird die Struktur der Simulink-Modelle nicht verändert. Daher sind die Ergebnisse für einen Benutzer leichter nachzuvollziehen. In der Arbeit von *Deißenböck et al.* werden Simulink-Modelle verflacht, sodass die Ergebnisse nicht ohne Weiteres verwendet werden können. Auch in der Arbeit von *Weiland et al.* wird die Struktur eines Simulink-Modells nicht verändert. Dies liegt allerdings primär an der Tatsache, dass keine Differenzierung angewendet wird.

Variabilitätsmechanismen wurden in dieser Arbeit ebenfalls betrachtet. Zudem wurden sie bewertet und aus dieser Bewertung heraus wurden zwei geeignete Mechanismen als angemessen bestimmt. Diese sind `Model Variants` und `Variant Subsystem`. Auch in der Arbeit von *Weiland et al.* werden Variabilitätsmechanismen bewertet und eingesetzt. Die beiden Variabilitätsmechanismen `Model Variants` und `Variant Subsystem` sind allerdings in dieser Bewertung nicht enthalten, da sie erst mit Matlab-Release R2009b und R2010b eingeführt wurden. Die Arbeit von *Weiland*

	Mengi	Deißenböck et al.	Ryssel et al.	Weiland et al.
Vergleichspaare	interaktiv oder automatisch durch Metriken (Name, Typ, Index)	automatisch durch Metriken (Typ, Index)	automatisch durch Metriken (Typ, Index)	×
Differenzierung	Differenzierungsalgorithmus	heuristischer Graphmatching-algorithmus	Bestimmung des maximalen Cliques mit anschließender formalen Konzeptanalyse	×
Repräsentation	farbliche Markierungen von Variationspunkten in Simulink	farbliche Markierungen von Gemeinsamkeiten in Simulink	×	×
Strukturerhaltung	✓	×	×	✓
Variabilitätsmechanismen	Model Variants und Variant Subsystem	×	×	sämtliche Variabilitätsmechanismen
Restrukturierung	Restrukturierungsregeln	×	×	×
Variabilitätsmodellierung	Variabilitätsmodell	×	Featuremodell	Featuremodell

Abbildung 6.38.: Vergleichsaufstellung mit den Konzepten dieser Arbeit und verwandter Arbeiten

et al. stammt aber aus Zeiten vor diesen Releases.

Diese Arbeit ist zudem die Einzige, die das Thema der Restrukturierung nach einer Differenzierung behandelt. In diesem Zusammenhang wurden Restrukturierungsregeln (vgl. Abschnitt 6.4.3.1) definiert, die Benutzer verwenden können, um ihre Modelle zu restrukturieren.

Schließlich wird in dieser Arbeit das Variabilitätsmodell aus Kapitel 4 zur Dokumentation und Repräsentation der identifizierten Variationspunkte eingesetzt. Besonders geeignet ist das Variabilitätsmodell im Zusammenhang mit den `Variant Objects` der Variabilitätsmechanismen. In den Arbeiten von *Rysell et al.* und *Weiland et al.* werden zu diesem Zweck Featuremodelle angewendet.

6.8. Zusammenfassung

Simulink-Modelle sind durch feingranulare Beschreibungselemente charakterisiert, die auf Funktionsebene nicht existieren. Dadurch wird es möglich, Funktionsverhalten sehr detailliert zu spezifizieren. Dies führt allerdings zu der Situation, dass neue Variabilitätsdimensionen eröffnet werden [PKB09]. In diesem Kapitel wurden sie unter dem Begriff der *Realisierungsvarianten* zusammengefasst.

Eine weitere Besonderheit betrifft die in der Automobilindustrie bevorzugte *Vorgehensweise* bei der Realisierung von Funktionen. Sie ist *inkrementell* und *evolutionär* und primär durch Copy-Paste-Strategien beeinflusst. Der Verlust über das Wissen gemeinsamer und variabler Bestandteile ist das Resultat einer derartigen Vorgehensweise.

Um dennoch Variabilität auf dieser Ebene realisieren zu können, wurde der Bedarf eines *Analysevorgangs* ermittelt. Dieser besteht aus zwei wesentlichen Aktivitäten: (1) die Differenzierung und (2) die Variabilitätsmodellierung. Die *Differenzierung* ist ein interaktiver werkzeuggestützter Prozess mit dem Ziel das verlorengegangene Wissen bezüglich Gemeinsamkeiten und Variabilität in Simulink-Modellen wiederzugewinnen. Zu diesem Zweck wurde eine Softwareinfrastruktur erarbeitet, die alle zur Differenzierung erforderlichen Metamodelle (bzw. abstrakten Datentypen) enthält. Diese sind das *Simulink-Metamodell*, das *Kommunalitätsmetamodell* und das *Differenzmetamodell*. Auf dieser Infrastruktur operieren die zur Differenzierung erforderlichen Funktionalitäten, wie etwa die *Importierung*, die *Festlegung von Vergleichspaaren*, der *Differenzierungsalgorithmus* und die *Exportierung*. Die hieraus gewonnenen Erkenntnisse werden in einem weiteren Schritt zur Variabilitätsmodellierung verwendet.

Die *Variabilitätsmodellierung* umfasst drei Teilaspekte: (1) die Evaluierung möglicher Variabilitätsmechanismen, (2) die Restrukturierung der Simulink-Modelle durch geeignete Variabilitätsmechanismen und (3) die Dokumentation und Repräsentation der Variabilität durch das Variabilitätsmodell aus Kapitel 4. Bei der Evaluierung vorhandener Variabilitätsmechanismen in Simulink hat sich ergeben, dass `Model Variants` und `Variant Subsystems` besonders geeignet sind. Daher wurden diese für die Restrukturierung der Modelle herangezogen. Zur Unterstützung der Benutzer bei der Restrukturierung wurden Restrukturierungsregeln eingeführt, die sowohl

für einfache als auch komplexe Situation anwendbar sind. Die restrukturierten Simulink-Modelle können insbesondere mit einem Variabilitätsmodell einfacher verknüpft werden, da die Varianten nun durch angemessene Variabilitätsmechanismen realisiert sind.

Schließlich wurde ein Anwendungsbeispiel zur Illustration der Konzepte erläutert, auf Realisierungsaspekte der Softwarewerkzeuge eingegangen sowie verwandte Arbeiten beschrieben und mit dem Ansatz dieses Kapitels verglichen.

Kapitel 7.

Codeebene

7.1. Einleitung und Motivation

Die auf Funktions- und Architekturebene modellierten Strukturen werden auf der Codeebene implementiert. Zum Teil wird der Code aus den Modellen der höheren Abstraktionsebenen generiert und diversen Anpassungen unterworfen. Oftmals wird der Code aber auch vollständig manuell implementiert. Variabilität muss aber in beiden Fällen stets berücksichtigt werden.

In der Automobilindustrie ist die Programmiersprache C mit 51% die verbreitetste Sprache, gefolgt von C++ mit 30%, Assembler mit 8% und weiteren Sprachen, die unter 5% liegen [Bar09]. Gemäß diesen Beobachtungen liegt der Fokus in dieser Arbeit auf der Programmiersprache C (und daher auch zum Teil auf C++) [KR88].

Variationspunkte werden durch Variabilitätsmechanismen der Sprache realisiert (vgl. Kapitel 4). In der Programmiersprache C sind dies in der Regel Präprozessordirektiven oder Auswahlanweisungen. Der folgende Codeausschnitt zeigt einen Teil der Implementierung des Fahrzeugzugangssystems mit den zwei Varianten Zentralverriegelung und Komfortzugang. Der Variationspunkt wird hier durch Präprozessordirektiven realisiert.

Der Codeausschnitt von Zeile 1 bis 27 beinhaltet zwei Funktionen, eine zum Verriegeln (`lock()`) und eine zum Entriegeln (`unlock()`) der Fahrzeugtüren. Innerhalb dieser Funktionen werden spezifische Funktionen aufgerufen, die bei Definition der Zentralverriegelung oder des Komfortzugangs ausgeführt werden. Um diese Alternativen zu realisieren, werden **`#if`**-Präprozessordirektiven eingesetzt. Es ist auf einem Blick erkennbar, wie eine einfache Alternative durch den Variabilitätsmechanismus in eine komplexe Struktur überführt wird.

```
 1 #if ZENTRALVERRIEGELUNG || KOMFORTZUGANG
 2    static void lock()
 3    {
 4 #endif
 5 #if ZENTRALVERRIEGELUNG
 6       cl_lock();
 7 #endif
 8 #if KOMFORTZUGANG
 9       komf_lock();
10 #endif
11 #if ZENTRALVERRIEGELUNG || KOMFORTZUGANG
12    }
13 #endif
14 #if ZENTRALVERRIEGELUNG || KOMFORTZUGANG
15
16    static void unlock()
17    {
18 #endif
19 #if ZENTRALVERRIEGELUNG
20       cl_unlock();
21 #endif
22 #if KOMFORTZUGANG
23       komf_unlock();
24 #endif
25 #if ZENTRALVERRIEGELUNG || KOMFORTZUGANG
26    }
27 #endif
```

Ruben Zimmermann hat im Rahmen seiner Diplomarbeit [Zim09] derartige komplexe Strukturen untersucht und komplexitätsreduzierende Maßnahmen entworfen, die schließlich in einem Papier [MFZA09] veröffentlicht wurden. Die folgenden Erläuterungen basieren primär auf diesen zwei Veröffentlichungen.

In den Untersuchungen wurden primär vier Schwachstellen entdeckt, die aufgrund dieser komplexen Strukturen entstehen:

1. Der Programmierer ist stets mit allen Varianten ohne jegliche Form der Unterstützung konfrontiert. Er kann sich also nur mühsam auf eine Variante fokussieren. Im obigen Quelltextausschnitt ist sowohl der Code für die Zentralverriegelung als auch für den Komfortzugang gleichermaßen zu sehen. Das Lesen einer derartigen Struktur bestehend aus zwei Varianten ist bedeutend schwieriger als eine Variante.

2. Der Code einer Variante ist in der Regel in der gesamten Codebasis verstreut. Der Programmierer muss diesen manuell ausfindig machen. Zum Beispiel ist der spezifische Code der Zentralverriegelung in den Zeilen 6 und 20. Bei größeren

Systemen ist diese Verstreuung nicht mehr auf einem Blick sichtbar, da die Zeilenabstände deutlich größer und zudem mehrere Dateien involviert sind.

3. Restriktionen, die zwischen Varianten vorherrschen, sind entweder gar nicht oder nur implizit in den Bedingungen enthalten. Der Programmierer muss diese ebenfalls manuell ermitteln. Beispielsweise erfordert der Komfortzugang das Vorhandensein der Sendeantennen in den Türhandgriffen. Zur besseren Verständlichkeit müsste der Programmierer also auch den Softwareanteil dieser Sendeantennen betrachten. Diese Abhängigkeit kann allerdings nicht aus dem oben dargestellten Code ermittelt werden.

4. Bei sehr vielen Varianten mit komplexen Abhängigkeiten ist es enorm schwierig, eine valide Konfiguration zu erstellen. Typischerweise müssen alle definierten Bezeichner bei einer Softwarekonfiguration geeignet belegt werden. Beispielsweise bewirkt die Existenz einer Variante den Ausschluss einer anderen. Da derartige Abhängigkeiten nicht explizit erfasst sind, wird es sehr schwierig, gültige Konfiguration zu erstellen.

Angesichts dieser vier Punkte wird es immer schwieriger und zeitintensiver, Software zu verstehen, anzupassen und zu warten. Es müssen also geeignete Konzepte erstellt werden, um den Umgang mit Variabilität zu unterstützen. Ein wesentlicher Schritt hierbei ist die explizite Identifizierung aller im Code vorhandenen Varianten. Das Variabilitätsmodell, so wie es in Kapitel 4 beschrieben wurde, kann zu diesem Zweck genutzt werden. So werden zum einen alle Varianten erfasst und zum anderen können verstreute Codefragmente zentralisiert werden (Neutralisierung der Schwachstelle 2). Insbesondere ist es somit auch möglich, Restriktionen, die gar nicht oder nur teilweise im Code beschrieben sind, vollständig im Variabilitätsmodell zu formulieren (Neutralisierung der Schwachstelle 3). Auf diese Weise sind jegliche Restriktionen vom Programmierer in zentraler Form einsehbar. Zudem kann aus dem Variabilitätsmodell das Konfigurationsmodell abgeleitet werden, sodass hieraus eine Konfiguration erstellt werden kann (Neutralisierung der Schwachstelle 4). Herrscht eine Assoziation zwischen dem Quelltextdokument und dem Variabilitätsmodell vor, so kann anhand der Konfiguration eine variantenspezifische Sicht erzeugt werden (Neutralisierung der Schwachstelle 1).

In diesem Zusammenhang ist ein wichtiges Ziel, diese genannten Punkte als integrale Aktivität in die Codeebene zu verankern. Somit wird die Anwendbarkeit der erforderlichen Aspekte gesichert. Demnach sind nicht nur Quelltextdateien benötigte Dokumente, sondern auch das Variabilitätsmodell. Diesbezüglich wird im Folgenden ein entsprechender Prozess definiert, der die ursprüngliche Implementierungsaktivität um Aspekte der Variabilität erweitert. Abbildung 7.1 zeigt den in diesem Zusammenhang definierten Prozess.

Den Ausgangspunkt bilden zwei Dokumente: (1) der C Quellcode und (2) das Variabilitätsmodell. Insbesondere ist die Identifizierung geeigneter Variabilitätsmechanismen in der zugrunde liegenden Sprache Voraussetzung (vgl. Abschnitt 7.2.1), um die Verknüpfung zwischen dem Quellcode und dem Variabilitätsmodell herzustellen. Aus der Prozessdefinition ist zudem erkennbar, dass die Implementierung

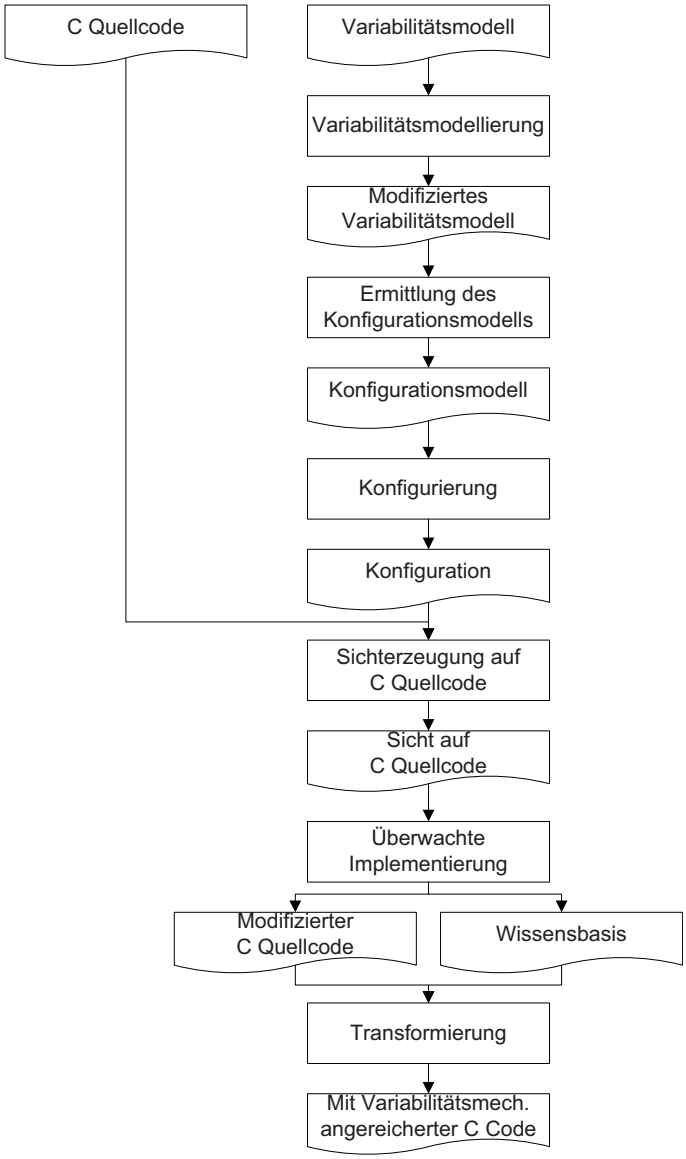


Abbildung 7.1.: Der Prozess zur Modellierung von Variabilität auf Codeebene

weiter nach hinten verschoben wurde und die Variabilitätsmodellierung mit der Konfigurierung vorgeschaltet ist (vgl. Abschnitt 7.2.2 und Abschnitt 7.3.1). Ein weiterer wesentlicher Schritt ist die Erzeugung einer Sicht auf den Quellcode, die basierend auf einer Konfiguration erstellt wird. Sie ist temporär und dient primär zur Komplexitätsreduzierung durch Verbergung irrelevanter Informationen (vgl. Abschnitt 7.3.2). Die Implementierung findet dann auf dieser temporären Sicht statt. Besonders ist hierbei, dass die Implementierung überwacht wird, sodass jede Modifikation mit der entsprechenden Konfiguration assoziiert wird (vgl. Abschnitt 7.3.3). Ist die Implementierung beendet, verfällt die Gültigkeit der Sicht und jegliche Modifikationen müssen durch Variabilitätsmechanismen in den ursprünglichen Quellcode überführt werden. Hierfür werden entsprechende Transformationsregeln definiert (vgl. Abschnitt 7.3.4).

Die Realisierung dieser Aspekte kann somit die identifizierten Schwachstellen auflösen und dem Programmierer in Form eines Werkzeugs auf Codeebene bei den Implementierungsaufgaben unterstützen.

7.2. Variabilitätsmodellierung

Im Folgenden werden, beginnend mit der Beschreibung der Variabilitätsmechanismen in der Programmiersprache C, alle im Prozess (vgl. Abbildung 7.1) definierten Aktivitäten erläutert.

7.2.1. Variabilitätsmechanismen

In der Programmiersprache C eignen sich zwei Anweisungsarten zur Realisierung von Variationspunkten: (1) Präprozessordirektiven und (2) Auswahlanweisungen. Sie sind erforderlich, um die im Prozess durchgeführten Änderungen geeignet zu strukturieren. Beide Anweisungsarten werden im Folgenden genauer erläutert.

7.2.1.1. Präprozessordirektiven

Präprozessordirektiven sind Anweisungen, die vor der Kompilierung von einem sogenannten Präprozessor ausgewertet werden. Es gibt verschiedene Arten derartiger Direktiven. Beispiele sind das bedingte Inkludieren von Codefragmenten, Integration von Quelltextdateien oder auch die Ersetzung von Makros (weitere können aus [KR88] entnommen werden). Sind die Direktiven ausgewertet, kann die eigentliche Kompilierung starten. Zur Realisierung von Variationspunkten eignen sich besonders Präprozessordirektiven durch bedingtes Inkludieren. Sie unterstützen das Binden von Varianten zur Compilezeit und werden im Folgenden näher vorgestellt.

`#ifdef` Präprozessordirektive

```
#ifdef identifier
   ...
#endif
```

Die Präprozessordirektive **#ifdef** wertet den Bezeichner identifier aus. Ergibt die Auswertung den Wert 0 (false), wird die Anweisungssequenz innerhalb der Direktive bis zum **#endif** für die Kompilierung ignoriert. Resultiert die Auswertung hingegen zu 1 (true), wird die Anweisungssequenz innerhalb der Direktive bis zum **#endif** für die Kompilierung verarbeitet.

#ifndef Präprozessordirektive

```
#ifndef identifier
   ...
#endif
```

Die Präprozessordirektive **#ifndef** wertet den Bezeichner identifier aus. Ergibt die Auswertung den Wert 0 (false), wird die Anweisungssequenz innerhalb der Direktive bis zum **#endif** für die Kompilierung verarbeitet. Resultiert die Auswertung hingegen zu 1 (true), wird die Anweisungssequenz innerhalb der Direktive bis zum **#endif** für die Kompilierung ignoriert.

#if Präprozessordirektive

```
#if constant-expression
   ...
#endif
```

Die Präprozessordirektive **#if** wertet den konstanten Ausdruck constant-expression aus. Der konstante Ausdruck setzt sich im Gegensatz zu einem Bezeichner aus komplexen arithmetischen und logischen Ausdrücken zusammen. Ergibt die Auswertung den Wert 0 (false), wird die Anweisungssequenz innerhalb der Direktive bis zum **#endif** für die Kompilierung ignoriert. Resultiert die Auswertung hingegen zu 1 (true), wird die Anweisungssequenz innerhalb der Direktive bis zum **#endif** für die Kompilierung verarbeitet.

#if, #elif und #else Präprozessordirektive

```
#if constant-expression-1
   ...
#elif constant-expression-2
   ⋮
#elif constant-expression-N
   ...
#else
   ...
#endif
```

Die Kontrollstruktur bestehend aus den Präprozessordirektiven **#if**, **#elif**, **#else** und **#endif** ermöglicht, eine Reihe von alternativen Anweisungsblöcken zu strukturieren und abhängig von der Auswertung der konstanten Ausdrücke die entsprechende Anweisungssequenz für die Kompilierung zu ignorieren oder zu verarbeiten.

7.2.1.2. Auswahlanweisungen

Auswahlanweisungen selektieren in Abhängigkeit der Auswertung eines Ausdrucks die auszuführende Anweisungssequenz. In der Programmiersprache C gibt es zwei Arten von Auswahlanweisungen: (1) die **`if`**-Anweisung und (2) die **`switch`**-Anweisung. Beide eignen sich zur Realisierung von Variationspunkten, die optionale oder alternative Varianten zur Laufzeit binden. Sie werden im Folgenden genauer erläutert.

`if`-Anweisung

```
if ( expression )
{
   ...
else
   ...
}
```

Die **`if`**-Anweisung wertet den Ausdruck `expression` aus. Ergibt die Auswertung den Wert 0 (`false`), wird die darauffolgende Anweisungssequenz bis zum **`else`**-Schlüsselwort ignoriert. Stattdessen wird die Anweisungssequenz im **`else`**-Teil ausgeführt. Resultiert die Auswertung hingegen zu 1 (`true`), wird die darauffolgende Anweisungssequenz ausgeführt und der **`else`**-Teil ignoriert. Es sei an dieser Stelle darauf hingewiesen, dass die beschriebene **`if`**-Anweisung nur eine bestimmte Form darstellt (zweiseitige **`if`**-Anweisung). Es gibt zudem noch weitere Formen. Zum Beispiel könnte der **`else`**-Teil auch fehlen (einseitige **`if`**-Anweisung) oder weitere **`else`**-Teile hinzugefügt werden (mehrseitige **`if`**-Anweisung). Diese werden hier nicht weiter vertieft (für weitere Informationen diesbezüglich sei auf [KR88] verwiesen).

`switch`-Anweisung

```
switch ( expression ) {
{
   case constant-expression-1 :
      ...
   case constant-expression-2 :
      ...
   ⋮
   case constant-expression-N :
      ...
   default :
      ...
}
```

Die **`switch`**-Anweisung wertet den Ausdruck `expression` aus und vergleicht diesen mit den konstanten Ausdrücken der alternativen **`case`**-Marken. Bei Gleichheit wird die zugehörige Anweisungssequenz ausgeführt. Wenn keine Übereinstimmung gefunden wird, wird die Anweisungssequenz der **`default`**-Marke ausgeführt.

7.2.2. Variabilitätsmodell

In Kapitel 4 wurde die Modellierung und Bindung von Variabilität ausführlich erläutert. Neben dem Variabilitätsmodell wurde weiterhin das Restriktionsmodell im Detail beschrieben. Beide Modelle sollen als Hilfsmittel vom Programmierer eingesetzt werden. Vor der eigentlichen Implementierung wird in diesem Prozess also zunächst die zu modellierende Variabilität identifiziert. Es müssen also Variationspunkte und Varianten definiert, hierarchische Strukturierungen durch Gruppen gebildet, Kardinalitäten festgelegt sowie Restriktionen formuliert werden. Darüber hinaus ist es essenziell, Variabilitätsmechanismen, Bindungsmechanismen sowie Bindezeiten für die jeweiligen Variationspunkte festzulegen. Nach dieser Aktivität erhält der Programmierer das modifizierte Variabilitätsmodell, das die neu zu implementierenden Varianten beschreibt. Der nächste Schritt besteht aus der Konfigurierung einer Variante.

7.3. Variantengetriebene Implementierung

Dieser Abschnitt umfasst neben der Konfigurierung des Variabilitätsmodells Erläuterungen zur Erzeugung von Sichten und zur anschließenden überwachten Implementierung. Insbesondere werden darauffolgend Transformationsregeln definiert, um die in einer Sicht durchgeführten Modifikationen in den Quellcode zu integrieren.

7.3.1. Konfigurierung

Aus dem modifizierten Variabilitätsmodell wird das Konfigurationsmodell entsprechend den Beschreibungen aus Abschnitt 4.3.1 erstellt. Es werden also alle relevanten Informationen aus dem Variabilitätsmodell extrahiert. Diese sind Variationspunkte, Gruppenkardinalitäten, Varianten und Variantenkardinalitäten. Der Programmierer konfiguriert im Anschluss die zu implementierende Variante und erhält hieraus eine Konfiguration. Die Konfiguration wird im Weiteren für die Erzeugung einer gültigen Sicht benötigt.

7.3.2. Erzeugung von Sichten

In dieser Aktivität wird aus dem Quellcode eine der Konfiguration entsprechende temporäre Sicht erzeugt. Hierbei wird die Menge der selektierten Varianten im Code untersucht und anhand der Bedingungen der Variabilitätsmechanismen überprüft. Der Code aller als gültig identifizierten Variabilitätsmechanismen wird in die Sicht eingebunden. Die Erzeugung einer Sicht spiegelt im Wesentlichen den Präprozessor wieder, mit dem Unterschied, dass die Auswertung dem Programmierer sichtbar gemacht wird.

7.3.3. Überwachte Implementierung

Ist die Sicht erzeugt, besteht der nächste Schritt aus der Implementierung der konfigurierten Variante. Dabei wird jede Modifikation des Programmierers durch ein Werkzeug überwacht. Eine Wissensbasis bildet die Grundlage für das Werkzeug. Hierdurch werden sämtliche Modifikationen automatisch erfasst und dokumentiert. Auf diese Weise wird es möglich, nach der Implementierung die Modifikationen durch geeignete Variabiltätsmechanismen in den ursprünglichen Quellcode zu überführen. Hierfür werden entsprechende Transformationsregeln definiert, um die Transformierung durchzuführen.

7.3.4. Transformierung

Um die Transformierung auszuführen, ist es zunächst wichtig zu wissen, zur welcher Bindezeit die implementierte Variante gebunden werden soll. Hierbei wird zwischen zwei Bindezeiten unterschieden: (1) Compilezeit und (2) Laufzeit. Je nachdem, welche Bindezeit definiert wurde, werden entsprechende Transformationsregeln ausgeführt. Im Folgenden werden daher die Regeln nach ihrer Bindezeit unterschieden und erläutert.

7.3.4.1. Transformationsregeln für die Bindung zur Compilezeit

Varianten und Konfigurationen

Transformationsregel 1 Die Menge aller in einem Variabilitätsmodell modellierten Varianten $V_1, V_2, \ldots, V_n$, mit $n \in N$, die eine Compilezeitbindung definieren, werden zu Präprozessordirektiven der Form

```
#define V_1 0
#define V_2 0
⋮
#define V_n 0
```

transformiert.

Transformationsregel 2 Gegeben sei eine Konfiguration $\{V_k, V_{k+1}, \ldots, V_{k+l}\} \subseteq \{V_1, \ldots, V_n\}$ mit $k, l \in N$ und $1 \leq k \leq k+l \leq n$. Weiterhin sind Gruppen $G_1, \ldots, G_m$ gegeben. Für jedes Element V_i der Konfiguration gilt $V_i \in G_j$ für $i \in \{k, \ldots, k+l\}$ und $j \in \{1, \ldots, m\}$. Jede Variante ist also einer Gruppe zugehörig. Alle Varianten $V_i, \ldots, V_{i+p}$ mit $k \leq i \leq i+p \leq k+l$ werden zu einem Ausdruck der Form

```
(V_i && ...&& V_i+p)
```

transformiert, wenn für alle $V_i, \ldots, V_{i+p}$ gilt, dass $V_q \in G_r$, $V_t \in G_s$ und $r \neq s$ mit $q, t \in \{i, \ldots, i+p\}$ und $r, s \in \{1, \ldots, m\}$. Somit werden also alle Varianten der

Konfiguration, die nicht aus der gleichen Gruppe stammen, durch logische UND-Symbole (&&) miteinander verknüpft.

Alle Varianten $V_i, \ldots, V_{i+p}$ mit $k \leq i \leq i+p \leq k+l$ werden zu einem Ausdruck der Form

```
(V_i || ...|| V_i+p)
```

transformiert, wenn für alle $V_i, \ldots, V_{i+p}$ gilt, dass $V_q \in G_r$, $V_t \in G_s$ und $r = s$ mit $q, t \in \{i, \ldots, i+p\}$ und $r, s \in \{1, \ldots, m\}$. Somit werden also alle Varianten der Konfiguration, die aus der gleichen Gruppe stammen, durch logische ODER-Symbole (||) miteinander verknüpft.

Beide Teiltransformationen werden durch ein logisches UND-Symbol (&&) verbunden. Die Transformation der Konfiguration wird für das bedingte Inkludieren in den Präprozessordirektiven verwendet. Im Folgenden wird zur Vereinfachung dieser Ausdruck durch `config-expression` abgekürzt.

Modifikationen am Quellcode außerhalb existierender Präprozessorblöcke

Transformationsregel 3 Sei eine Konfiguration $V_k, V_{k+1}, \ldots, V_{k+l}$ gegeben. Das Hinzufügen der Codezeilen

```
zeile_1
zeile_2
⋮
zeile_p
```

wird in eine Präprozessordirektive der Form

```
#if config-expression
   zeile_1
   zeile_2
   ⋮
   zeile_p
#endif
```

transformiert.

Transformationsregel 4 Sei eine Konfiguration $V_k, V_{k+1}, \ldots, V_{k+l}$ gegeben. Das Löschen der Codezeilen

```
~~zeile_1~~
~~zeile_2~~
⋮
~~zeile_p~~
```

wird in eine Präprozessordirektive der Form

```
#if ! config-expression
   zeile_1
   zeile_2
   ⋮
   zeile_p
#endif
```

transformiert.

Es sei an dieser Stelle erwähnt, dass bei einer gegebenen Konfiguration das Entfernen von Codezeilen nicht zu einer tatsächlichen Löschung aus der Quellcodedatei führt. Stattdessen impliziert die Entfernung von Codezeilen, dass diese bei der festgelegten Konfiguration ausgeschlossen werden sollen. Genau dies wird durch die Bedingung in der **#if**-Präprozessordirektive erreicht.

Modifikationen am Quellcode innerhalb existierender Präprozessorblöcke

Transformationsregel 5 Sei eine Konfiguration $V_k, V_{k+1}, \ldots, V_{k+l}$ und folgender Präprozessorblock gegeben:

```
#if constant-expression
   zeile_1
   zeile_2
   ⋮
   zeile_p
#endif
```

Das Hinzufügen der Codezeilen `zeile_p-i`, ..., `zeile_p-i+j` in den existierenden Präprozessorblock

```
#if constant-expression
   zeile_1
   zeile_2
   ⋮
   zeile_p-i-1
   zeile_p-i
   ⋮
   zeile_p-i+j
   zeile_p-i+j+1
   ⋮
   zeile_p
#endif
```

wird in eine Präprozessordirektive der Form

```
#if constant-expression
   zeile_1
   zeile_2
   ⋮
   zeile_p-i-1
#endif
#if config-expression
   zeile_p-i
   ⋮
   zeile_p-i+j
#endif
#if constant-expression
   zeile_p-i+j+1
   ⋮
   zeile_p
#endif
```

transformiert. Der ursprüngliche Präprozessorblock wird also gesplittet, um den neu hinzugefügten Code durch eine weitere **#if**-Präprozessordirektive zu umschließen.

Eine zusätzliche Transformationsregel für dei gleiche Situation wäre das Verschachteln des neu hinzugefügten Codes in den ursprünglichen Präprozessorblock. Diese Regel würde demnach folgende Struktur liefern:

```
#if constant-expression
   zeile_1
   zeile_2
   ⋮
   zeile_p-i-1
   #if config-expression
      zeile_p-i
      ⋮
      zeile_p-i+j
  #endif
   zeile_p-i+j+1
   ⋮
   zeile_p
#endif
```

Die Semantik dieses Transformationsergebnisses ist allerdings eine gänzlich andere als die zuvor durch die Aufspaltung beschriebene. Ist der neu hinzugefügte Code in den übergeordneten Präprozessorblock integriert, bedeutet dies, dass dieser Code nur dann kompiliert wird, wenn auch der übergeordnete Block einbezogen wird. Diese alternative Regel würde also einen unerwünschten Seiteneffekt hervorrufen. Durch die Aufspaltung wird dieser Effekt vermieden.

Transformationsregel 6 Sei eine Konfiguration $V_k, V_{k+1}, \ldots, V_{k+l}$ und folgender Präprozessorblock gegeben:

```
#if constant-expression
   zeile_1
   zeile_2
   ⋮
   zeile_p-i-1
   zeile_p-i
   ⋮
   zeile_p-i+j
   zeile_p-i+j+1
   ⋮
   zeile_p
#endif
```

Das Löschen der Codezeilen `zeile_p-i`, ..., `zeile_p-i+j` innerhalb des existierenden Präprozessorblocks

```
#if constant-expression
   zeile_1
   zeile_2
   ⋮
   zeile_p-i-1
   ~~zeile_p-i~~
   ⋮
   ~~zeile_p-i+j~~
   zeile_p-i+j+1
   ⋮
   zeile_p
#endif
```

wird in eine Präprozessordirektive der Form

```
#if constant-expression
   zeile_1
   zeile_2
   ⋮
   zeile_p-i-1
#endif
#if constant-expression && ! config-expression
   zeile_p-i
   ⋮
   zeile_p-i+j
#endif
#if constant-expression
   zeile_p-i+j+1
   ⋮
   zeile_p
#endif
```

transformiert.

Im Gegensatz zum Hinzufügen wird beim Löschen von Codezeilen der Bezug zum übergeordneten Präprozessorblock hergestellt. Dies ist an der Bedingung der **#if**-Präprozessordirektive zu sehen, die den gelöschten Code umfasst. Wenn stattdessen der Bezug zum übergeordneten Block weggelassen werden würde, also die Bedingung nur aus `! config-expression` bestehen würde, hätte dies zur Folge, dass die gelöschten Codezeilen in jeder Konfiguration erscheinen würden, die nicht diese bezugslose Bedingung erfüllt. Daher ist es in dieser Situation erforderlich, einen Bezug herzustellen.

Alternativ wäre in diesem Fall also auch eine Verschachtelung möglich. Die Modifikationen würden dann in eine Präprozessordirektive der Form

```
#if constant-expression
   zeile_1
   zeile_2
   ⋮
   zeile_p-i-1
   #if ! config-expression
     zeile_p-i
     ⋮
     zeile_p-i+j
   #endif
   zeile_p-i+j+1
   ⋮
   zeile_p
#endif
```

transformiert werden.

Modifikationen am Quellcode durch ganze Präprozessorblöcke

Transformationsregel 7 Sei eine Konfiguration $V_k, V_{k+1}, \ldots, V_l$ gegeben. Das Hinzufügen der Präprozessordirektive

```
#if constant-expression
   zeile_1
   zeile_2
   ⋮
   zeile_p
#endif
```

wird in eine Präprozessordirektive der Form

```
#if constant-expression || config-expression
   zeile_1
   zeile_2
   ⋮
   zeile_p
#endif
```

transformiert.

Transformationsregel 8 Sei eine Konfiguration $V_k, V_{k+1}, \ldots, V_l$ gegeben. Das Löschen der Präprozessordirektive

```
#if constant-expression
   zeile_1
   zeile_2
   ⋮
   zeile_p
#endif
```

wird in eine Präprozessordirektive der Form

```
#if constant-expression && ! config-expression
   zeile_1
   zeile_2
   ⋮
   zeile_p
#endif
```

transformiert.

7.3.4.2. Transformationsregeln für die Bindung zur Laufzeit

Varianten und Konfigurationen

Transformationsregel 1 Die Menge aller in einem Variabilitätsmodell modellierten Varianten $V_1, V_2, \ldots, V_n$, die eine Laufzeitbindung definieren, werden zu globalen Variablen vom Typ `Int16` der Form

```
extern Int16 V_1;
extern Int16 V_2;
⋮
extern Int16 V_n;
```

transformiert.

Transformationsregel 2 Gegeben sei eine Konfiguration $\{V_k, V_{k+1}, \ldots, V_{k+l}\} \subseteq \{V_1, \ldots, V_n\}$ mit $k, l \in N$ und $1 \leq k \leq k+l \leq n$. Weiterhin sind Gruppen $G_1, \ldots, G_m$ gegeben. Für jedes Element V_i der Konfiguration gilt $V_i \in G_j$ für $i \in \{k, \ldots, k+l\}$ und $j \in \{1, \ldots, m\}$. Jede Variante ist also einer Gruppe zugehörig. Alle Varianten $V_i, \ldots, V_{i+p}$ mit $k \leq i \leq i+p \leq k+l$ werden zu einem Ausdruck der Form

```
(V_i && ...&& V_i+p)
```

transformiert, wenn für alle $V_i, \ldots, V_{i+p}$ gilt, dass $V_q \in G_r$, $V_t \in G_s$ und $r \neq s$ mit $q, t \in \{i, \ldots, i+p\}$ und $r, s \in \{1, \ldots, m\}$. Somit werden also alle Varianten der Konfiguration, die nicht aus der gleichen Gruppe stammen, durch logische UND-Symbole (&&) miteinander verknüpft.

Alle Varianten $V_i, \ldots, V_{i+p}$ mit $k \leq i \leq i+p \leq k+l$ werden zu einem Ausdruck der Form

```
(V_i || ...|| V_i+p)
```

transformiert, wenn für alle $V_i, \ldots, V_{i+p}$ gilt, dass $V_q \in G_r$, $V_t \in G_s$ und $r = s$ mit $q, t \in \{i, \ldots, i+p\}$ und $r, s \in \{1, \ldots, m\}$. Somit werden also alle Varianten der Konfiguration, die aus der gleichen Gruppe stammen, durch logische ODER-Symbole (||) miteinander verknüpft.

Beide Teiltransformationen werden durch ein logisches UND-Symbol (&&) verbunden. Die Transformation der Konfiguration wird für das bedingte Inkludieren in den Präprozessordirektiven verwendet. Im Folgenden wird zur Vereinfachung dieser Ausdruck durch `config-expression` abgekürzt.

Modifikationen am Quellcode

Transformationsregel 3 Sei eine Konfiguration $V_k, V_{k+1}, \ldots, V_l$ gegeben. Das Hinzufügen der Codezeilen

```
zeile_1
zeile_2
⋮
zeile_p
```

wird in eine **if**-Anweisung der Form

```
if (config-expression)
{
   zeile_1
   zeile_2
   ⋮
   zeile_p
}
```

transformiert.

Transformationsregel 4 Sei eine Konfiguration $V_k, V_{k+1}, \ldots, V_l$ gegeben. Das Löschen der Codezeilen

```
~~zeile_1~~
~~zeile_2~~
⋮
~~zeile_p~~
```

wird in eine **if**-Anweisung der Form

```
if ( ! config-expression )
{
   zeile_1
   zeile_2
   ⋮
   zeile_p
}
```

transformiert.

7.4. Anwendungsbeispiel: Fahrzeugzugangssystem

In diesem Abschnitt werden die beschriebenen Konzepte an einem Beispiel durchgängig erläutert. Das hier verwendete Beispiel basiert auf dem Fahrzeugzugangssystem. Der Programmierer startet zunächst mit der Modellierung der Variabilität. Abbildung 7.2 illustriert das Ergebnis dieser Modellierung. Hierbei wurden der Variationspunkt (`Fahrzeugzugangssystem`) und die zugehörigen Varianten (`Zentralverriegelung` und `Komfortzugang`) festgelegt. Außerdem sind Gruppen- und Variantenkardinalitäten definiert. Ein wichtiger Schritt des Programmierers ist die Entscheidung der Bindezeit. Denn abhängig hiervon sind unterschiedliche Variabilitätsmechanismen einzusetzen. In dem Beispiel wurde die Bindung zur Compilezeit festgelegt. Dementsprechend sind Präprozessordirektiven als Variabilitätsmechanismus einzusetzen (vgl. Abschnitt 7.2.1). Durch Selektion in einem Konfigurierungsvorgang und der Ausführung des Präprozessors werden die Varianten gebunden. Sie stellen somit den Bindungsmechanismus dar.

Ist die Variabilität modelliert, besteht der nächste Schritt des Programmierers aus der Konfigurierung einer Variante. Dazu wird aus dem Variabilitätsmodell das entsprechende Konfigurationsmodell abgeleitet (vgl. Abschnitt 4.3.1). Anhand diesem Konfigurationsmodell wird die Konfigurierung durchgeführt. Der Programmierer konfiguriert zunächst sowohl die `Zentralverriegelung` (im Folgenden durch `ZV` abgekürzt) als auch den `Komfortzugang` (im Folgenden durch `Komf` abgekürzt), um den gemeinsamen Anteil des Codes zu implementieren. Durch diese Konfiguration wird dann eine Sicht auf den vorhandenen C Quellcode ermittelt. Dabei werden jegliche Quellcodezeilen entfernt, die nicht zu der aktuellen Konfiguration gehören. Für die aktuelle Konfiguration gilt also, dass alle Codezeilen entfernt werden, da bisher weder der Code für die `Zentralverriegelung` noch für den `Komfortzugang` geschrieben wurde. Mit dieser Sicht und Konfiguration beginnt der Programmierer, den gemeinsamen Anteil zu implementieren. Alle folgenden Modifikationen werden überwacht und mit dieser Konfiguration assoziiert.

Der Programmierer implementiert zunächst die Funktion zum Verriegeln (`lock()`). Jede implementierte Zeile wird dabei mit der Konfiguration assoziiert. Folgende Darstellung illustriert diese Situation:

Sicht	
`static void lock()`	`(ZV || Komf)`
`{`	`(ZV || Komf)`
`}`	`(ZV || Komf)`

Der linke Teil stellt den C Quellcode dar. Der rechte Teil beinhaltet das Konfigurationswissen zu der jeweiligen Zeile. Aktuell ist jede Zeile mit der Konfiguration durch die `Zentralverriegelung` und dem `Komfortzugang` assoziiert. Ist die Implementierung beendet, so wird auch die Überwachung gestoppt und mit der Transformation begonnen, um den Code durch geeignete Variabilitätsmechanismen zu umfassen. Entsprechend den Transformationsregeln aus Abschnitt 7.3.4 ergibt sich somit folgender Variabilitätsmechanismus für die implementierten Codezeilen:

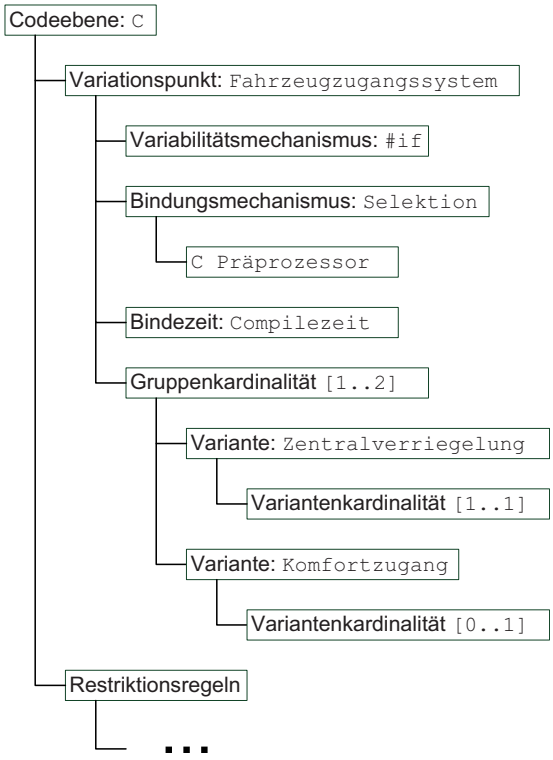


Abbildung 7.2.: Das Ergebniss der Variabilitätsmodellierung

Quellcode

```
#if ZENTRALVERRIEGELUNG || KOMFORTZUGANG
   static void lock()
   {
   }
#endif
```

Nach der Implementierung der Verriegelungsfunktion beginnt der Programmierer erneut mit der gleichen Konfiguration, um die Entriegelungsfunktion (`unlock()`) zu implementieren. Dazu startet er die Konfiguration, aus der dann die Sicht zu der angegebenen Konfiguration erzeugt wird. Diese Sicht besteht aus den Codezeilen, die die Verriegelungsfunktion beinhalten:

Sicht

```
static void lock()
{
}
```

Der dargestellte Code wird also um die Entriegelungsfunktion erweitert. Es entsteht folgende Modifikation:

Sicht	
`static void lock()`	`(ZV || Komf)`
`{`	`(ZV || Komf)`
`}`	`(ZV || Komf)`
	`(ZV || Komf)`
`static void unlock()`	`(ZV || Komf)`
`{`	`(ZV || Komf)`
`}`	`(ZV || Komf)`

Nach Beendigung dieser Änderungen wird erneut die Transformierung angestoßen, um den geeigneten Variabilitätsmechanismus zu erhalten. Dieser umfasst die `unlock()`-Funktion gleichermaßen, wie es bei der `lock()`-Funktion der Fall war. Es entsteht also folgender Quellcode:

Quellcode

```
#if ZENTRALVERRIEGELUNG || KOMFORTZUGANG
   static void lock()
   {
   }
#endif
#if ZENTRALVERRIEGELUNG || KOMFORTZUGANG

   static void unlock()
   {
   }
#endif
```

Daraufhin ändert der Programmierer die Konfiguration, indem er den `Komfortzugang` abwählt und nur die `Zentralverriegelung` selektiert. Die Absicht hierbei ist den Codeanteil zu schreiben, der für die `Zentralverriegelung` spezifisch ist. Aus der neuen Konfiguration wird die zugehörige Sicht abgeleitet. Dieser besteht aus genau dem oben dargestelltem Codeblock, da dieser der Einzige ist, der die aktuelle Konfiguration erfüllt:

Sicht

```
static void lock()
{
}

static void unlock()
{
}
```

Innerhalb der Funktion `lock()` wird ein Aufruf zur speziellen `lock()`-Funktion der Zentralverriegelung definiert. Auf gleiche Weise wird dies für die `unlock()`-Funktion durchgeführt. Das Ergebnis dieser Modifikation mit dem zugehörigen Konfigurationswissen ist wie folgt:

Sicht	
`static void lock()`	`(ZV || Komf)`
`{`	`(ZV || Komf)`
`  cl_lock();`	`ZV`
`}`	`(ZV || Komf)`
	`(ZV || Komf)`
`static void unlock()`	`(ZV || Komf)`
`{`	`(ZV || Komf)`
`  cl_unlock();`	`ZV`
`}`	`(ZV || Komf)`

Die neu hinzugefügten Zeilen sind ausschließlich der `Zentralverriegelung` zugeordnet, sodass sie auch nur in Sichten einbezogen werden, wenn in einer Konfiguration die `Zentralverriegelung` enthalten ist. Werden erneut die Transformationsregeln angewendet, ergibt sich folgende Codestruktur:

Quellcode

```
#if ZENTRALVERRIEGELUNG || KOMFORTZUGANG
   static void lock()
   {
#endif
#if ZENTRALVERRIEGELUNG
     cl_lock();
#endif
#if ZENTRALVERRIEGELUNG || KOMFORTZUGANG
   }
#endif
#if ZENTRALVERRIEGELUNG || KOMFORTZUGANG

   static void unlock()
   {
#endif
#if ZENTRALVERRIEGELUNG
     cl_unlock();
#endif
#if ZENTRALVERRIEGELUNG || KOMFORTZUGANG
   }
#endif
```

Die letzte Konfiguration, die der Programmierer durchführt, bezieht den `Komfortzugang` ein und deselektiert die `Zentralverriegelung`. Die Sicht, die hieraus generiert wird, ist wie folgt:

Sicht

`static void lock()`
`{`
`}`
 	
`static void unlock()`
`{`
`}`

Ähnlich wie bei der `Zentralverriegelung` wird für den `Komfortzugang` ebenfalls sepzifische Verriegelungs- und Entriegelungsfunktionen aufgerufen. Es entstehen folgende Modifikationen:

Sicht	
`static void lock()`	`(ZV || Komf)`
`{`	`(ZV || Komf)`
`  komf_lock();`	`Komf`
`}`	`(ZV || Komf)`
	`(ZV || Komf)`
`static void unlock()`	`(ZV || Komf)`
`{`	`(ZV || Komf)`
`  komf_unlock();`	`Komf`
`}`	`(ZV || Komf)`

Auch hier ist zu erkennen, dass die spezifischen Funktionen des `Komfortzugangs` ausschließlich diesem zugeordnet sind. Die Anwendung der Transformationsregeln passt den Quellcode durch die Variabilitätsmechanismen wie folgt an:

Quellcode

```
#if ZENTRALVERRIEGELUNG || KOMFORTZUGANG
   static void lock()
   {
#endif
#if ZENTRALVERRIEGELUNG
     cl_lock();
#endif
#if KOMFORTZUGANG
     komf_lock();
#endif
#if ZENTRALVERRIEGELUNG || KOMFORTZUGANG
   }
#endif
#if ZENTRALVERRIEGELUNG || KOMFORTZUGANG

   static void unlock()
   {
#endif
#if ZENTRALVERRIEGELUNG
     cl_unlock();
#endif
#if KOMFORTZUGANG
     komf_unlock();
#endif
#if ZENTRALVERRIEGELUNG || KOMFORTZUGANG
   }
#endif
```

7.5. Realisierung

Die folgenden Erläuterungen wurden bereits in der Diplomarbeit von *Ruben Zimmermann* [Zim09] ausführlich beschrieben. Hier werden die wesentlichen Aspekte zusammenfassend dargestellt. Die beschriebenen Konzepte wurden dabei prototypisch als Erweiterung der Eclipse C/C++ Development Tooling (CDT)-Entwicklungsumgebung realisiert. Abbildung 7.3 zeigt diesbezüglich den Grobentwurf in Form eines Komponentendiagramms.

Die Komponenten `ContextProvider` und `ModelProvider` sind die zwei wichtigsten Anbindungspunkte an die CDT-Entwicklungsumgebung. Über sie werden Ereignisinformationen aus der Entwicklungsumgebung gesammelt und allen weiteren Komponenten bereitgestellt. Während die Komponente `ContextProvider` den aktuellen Kontext, wie etwa geöffnete Projekte, Editoren etc., zur Verfügung stellt, bietet die Komponente `ModelProvider` das Variabilitätsmodell an.

Die Komponenten `ModelProvider`, `View` und `ModelController` bilden das Ent-

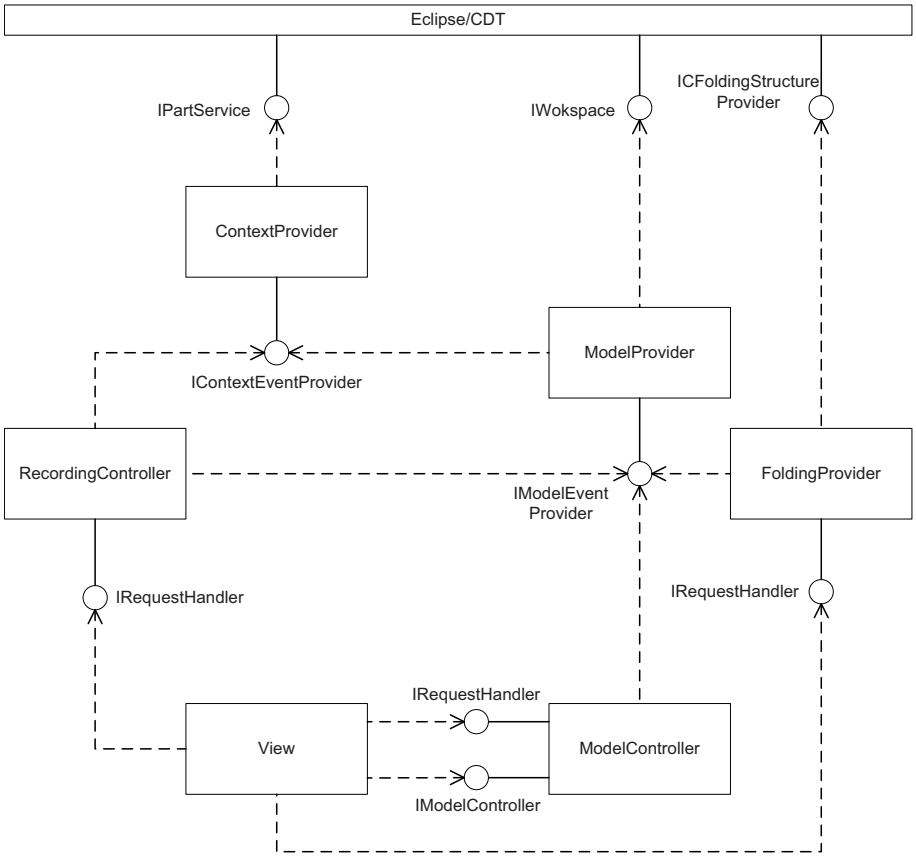


Abbildung 7.3.: Komponenten des Plugins in einer Übersicht (Quelle: [Zim09])

wurfsmuster Model-View-Controller nach. Somit besteht die Hauptaufgabe der Komponente `View` aus der Repräsentation des Variabilitätsmodells, welches von der Komponente `ModelProvider` zur Verfügung gestellt wird. Die Komponente `ModelController` steuert nach diesem Prinzip jegliche Modifikationen am Modell.

Die überwachte Implementierung wird von der Komponente `RecordingController` reguliert. Sie realisiert somit die Assoziation von Modifikationen am Quellcode und der zugehörigen Wissensbasis, welche mithilfe der Konfiguration aus dem Variabilitätsmodell heraus erstellt wird.

Schließlich wird die Erzeugung einer Sicht über die Komponente `FoldingProvider` umgesetzt. Sie blendet entsprechend einer gegebenen Konfiguration Codefragmente ein oder aus.

Abbildung 7.4 zeigt ein Screenshot der erweiterten CDT-Entwicklungsumgebung. Auf der rechten Seite ist das Variabilitätsmodell dargestellt, mit der die überwachte Implementierung gesteuert werden kann.

In den folgenden Abschnitten wird der Feinentwurf der oben beschriebenen

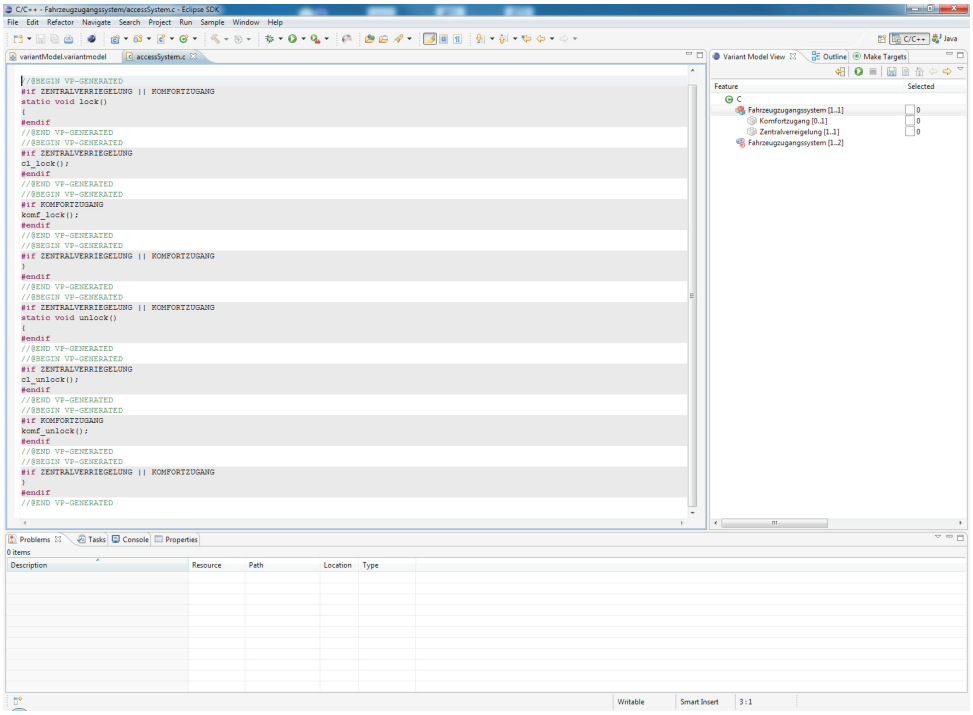


Abbildung 7.4.: Ein Screenshot der erweiterten CDT-Entwicklungsumgebung

Konzepte erläutert und auf die wichtigsten Aspekte eingegangen.

7.5.1. Context Provider

Die Komponente `ContextProvider` stellt den Kontext der Eclipse Entwicklungsumgebung für weitere Komponenten bereit, wie etwa dem `ModelProvider` (vgl. Abschnitt 7.5.2) oder dem `RecordingController` (vgl. Abschnitt 7.5.5). Abbildung 7.5 illustriert die innere Struktur der Komponente anhand eines Klassendiagramms.

Zu diesem Zweck registriert sich die Komponente `ContextProvider` als Listener bei der Eclipse Workbench. Hierfür implementiert die Klasse `VPContextProvider` die Schnittstelle `IPartListener`. Die Komponente `ContextProvider` stellt seinerseits allen registrierten Listenern ermittelte Ereignisse bereit. Hierfür implementiert die Klasse `VPContextProvider` die Schnittstelle `IContextEventProvider`.

7.5.2. Model Provider

Das Variabilitätsmodell wird von der Komponente `ModelProvider` zur Verfügung gestellt. Über die Komponente `ContextProvider` ermittelt die Komponente `ModelProvi-`

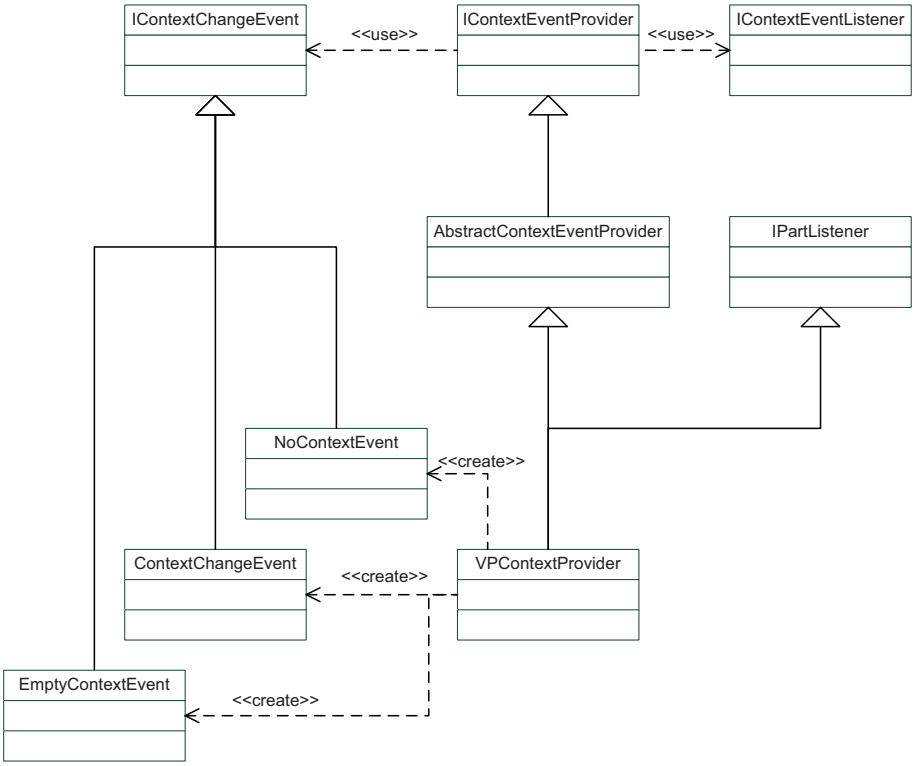


Abbildung 7.5.: Das Klassendiagramm der Komponente `ContextProvider` (Quelle: [Zim09])

der den Pfad zum Variabilitätsmodell. Zu diesem Zweck implementiert die Klasse `VPModelProvider` die Schnittstelle `IContextEventListener` und registriert sich somit als Listener bei der Komponente `ContextProvider` (vgl. Abbildung 7.6).

Aus dem Variabilitätsmodell wird ein entsprechendes `View`-Modell erzeugt und den weiteren Komponenten, wie etwa `RecordingController`, `ModelController` und `FoldingProvider`, über die Schnittstelle `IModelEventProvider` bereitgestellt. Änderungen am Variabilitätsmodell erfährt die Komponente, indem es die Schnittstellen `IRessourceDeltaVisitor` und `IRessourceChangeListener` implementiert. Die Klasse `VPModelProvider` aktualisiert in diesem Fall das `View`-Modell.

7.5.3. ModelController

Die Komponente `ModelController`, welche als Listener bei der Komponente `ModelProvider` registriert ist, stellt das `View`-Modell der Komponente `View` zur Verfügung. Zudem dient sie als `RequestHandler` und realisiert die Funktionen zum Serialisieren und Deserialisieren der `View`-Modelle. Abbildung 7.7 zeigt die innere Struktur der Komponente anhand eines Klassendiagramms.

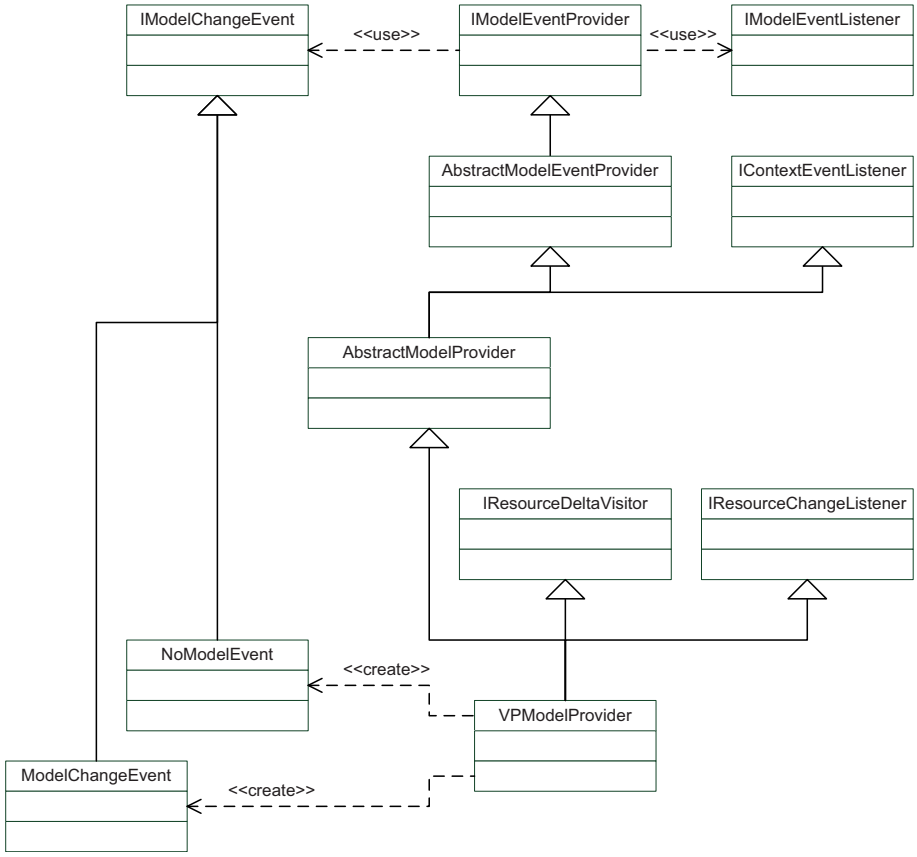


Abbildung 7.6.: Das Klassendiagramm der Komponente `ModelProvider` (Quelle: [Zim09])

Über die Klasse `CommandStack` können Änderungen am `View`-Modell umgesetzt werden. Die Klasse `VPModelController` kann somit die Änderungen erfassen. Über die Schnittstelle `IModelController` werden Informationen bzgl. Modifikationen am `View`-Modell an seine Listener weitergegeben.

7.5.4. View

Die Komponente `View` dient zur Repräsentation des `View`-Modells in der Eclipse Entwicklungsumgebung. Alle Bedienelemente werden hier bereitgestellt. Sie stellt somit die Benutzerschnittstelle dar. Die Anwendung einer Aktion eines Benutzers wird von der Komponente an alle `RequestHandler` in Form eines Requests übermittelt. Wenn Reaktionsbedarf besteht, erzeugt der entsprechende `RequestHandler` ein `Command`-Objekt, welches die Reaktion auf die Benutzeranfrage umsetzt.

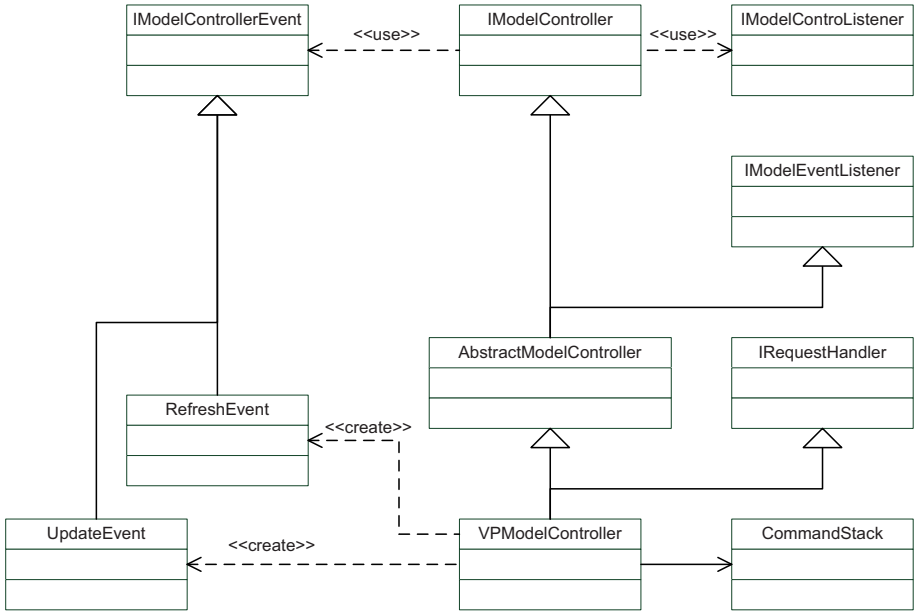


Abbildung 7.7.: Das Klassendiagramm der Komponente `ModelController` (Quelle: [Zim09])

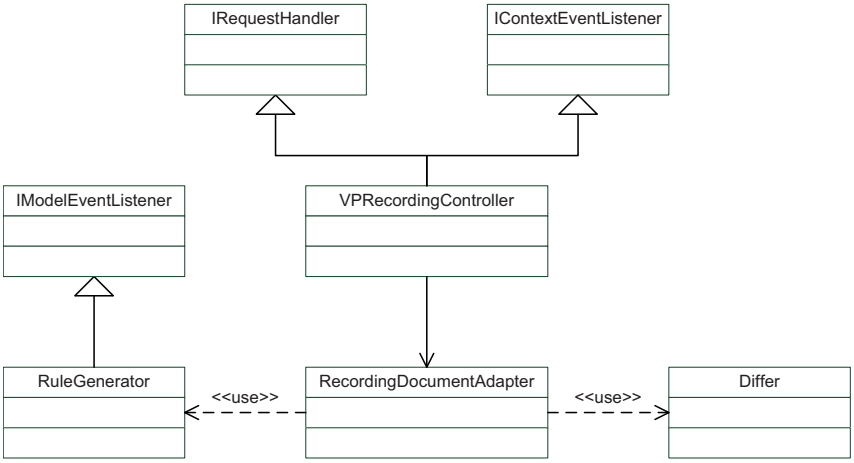


Abbildung 7.8.: Das Klassendiagramm der Komponente `RecordingController` (Quelle: [Zim09])

7.5.5. Recording Controller

Die Komponente `RecordingController` realisiert die überwachte Implementierung. Durch die Komponente `ContextProvider` kennt die Komponente die aktiven Edito-

ren und somit auch die Quelltextdokumente. Sie ist auch Listener der Komponente ModelProvider und verfügt somit über das `View`-Modell. Abbildung 7.8 zeigt die innere Struktur der Komponente anhand eines Klassendiagramms.

Die Klasse `VPRecordingController` erzeugt temporäre Kopien aller sich im Kontext befindenden Quelltextdokumente. Zu diesem Zweck dient die Klasse `RecordingDocumentAdapter`. Nach der überwachten Implementierung wird der neue Zustand des Quellcodes mit dem alten Zustand verglichen. Die Unterschiede werden durch die Klasse `Differ` ermittelt. Alle identifizierten Änderungen werden anhand der aktuell vorliegenden Konfiguration und den definierten Transformationsregeln zurück in die Quelltextdokumente überführt. Hierfür wird die Klasse `RuleGenerator` realisiert.

7.5.6. Folding Provider

Die Komponente `FoldingProvider` realisiert die Sicht auf den Code durch das Ein- bzw. Ausblenden von Codefragmenten. Bei der überwachten Implementierung wird jede Änderung bei einer gegebenen Konfiguration durch spezifische Kommentare bestückt. Durch einen Parser können somit spezifische Codefragmente ausfindig gemacht und entsprechend ein- bzw. ausgeblendet werden.

7.6. Verwandte Arbeiten

Auf Codeebene existieren sehr unterschiedliche Komplexitätstreiber, die das Verständnis, die Entwicklung und die Wartung von Software erschweren. So führen fehlende Abstraktionsschritte, unzureichende Modularisierung sowie Defizite in der Werkzeugunterstützung zu komplexen Softwaresystemen, die nicht ohne Weiteres gehandhabt werden können. Auch mangelnde Unterstützung von Variabilität gehört zu einen der vielen Komplexitätstreibern auf Codeebene.

In der Literatur existieren viele Arbeiten, die komplexitätsreduzierende Maßnahmen vorschlagen. Die meisten dieser Arbeiten betrachten allerdings die Situation nicht aus dem Blickwinkel der Variantenproblematik, wie zum Beispiel in [Wei81, Rob02, CC04, MKRC05]. Es gibt nur sehr wenige Arbeiten, die explizite Vorschläge in Bezug auf die Modellierung und Verwaltung von Variabilität machen, wie etwa in [Käs07]. Nichtsdestotrotz sind viele Konzepte auf die Problemstellungen dieser Arbeit übertragbar. Zum Beispiel wird in allen Ansätzen die Erzeugung einer Sicht als Schlüsselfaktor zur Reduzierung der Komplexität gesehen. Daher werden im Folgenden auch die Arbeiten beschrieben, die nur implizit mit dem Thema dieses Kapitels verwandt sind.

Nachfolgend werden fünf Arbeiten herangezogen: (1) Program Slicing, (2) FEAT, (3) Spotlight, (4) Mylar/Mylin und (5) CIDE. Neben der Beschreibung der Arbeiten werden die enthaltenen Konzepte durch Bewertungskriterien evaluiert und im Anschluss miteinander verglichen. Die hierbei festgelegten Kriterien umfassen primär alle Faktoren, mit denen die anfangs ermittelten Schwachstellen neutralisiert werden:

1. *Varianten:* Dieses Kriterium bewertet, ob Varianten mit dem beschriebenen Ansatz erfasst und repräsentiert werden können.

2. *Abhängigkeiten:* Einerseits können Programmstrukturen explizit ermittelbare Abhängigkeiten aufweisen. Zum Beispiel steht ein Programmmodul A mit einem Programmmodul B in Abhängigkeit, wenn A Funktionen von B benutzt. Andererseits gibt es auch implizite Abhängigkeiten, die nicht ohne Weiteres ermittelbar sind. Zum Beispiel bei zwei Funktionen, die auf unterschiedliche Steuergeräte deployed sind und über ein Bussystem kommunizieren. Durch dieses Kriterium wird untersucht inwiefern jegliche Formen von Abhängikeiten mit dem entsprechenden Ansatz festgestellt werden können.

3. *Sichten:* Dieses Kriterium ist eine wichtige komplexitätsreduzierende Maßnahme. Es wird ermittelt, welche Konzepte zur Erzeugung von Sichten auf den Code existieren.

4. *Werkzeugunterstützung:* Hierbei wird untersucht, ob die Konzepte durch Softwarewerkzeuge realisiert wurden.

5. *Programmiersprache:* Dieses Kriterium evaluiert, welche Programmiersprachen durch die Ansätze unterstützt werden.

7.6.1. Program Slicing

Mark Weiser hat im Jahr 1981 den Begriff *Program Slicing* eingeführt. Er bezeichnet damit eine Methode zur Dekomposition von Programmen durch Daten- und Kontrollflussanalyse [Wei81, Wei84]. Auf diese Weise werden Sichten auf das Programm erzeugt, die nur den Teil des Codes enthalten, der von Interesse ist. Da zur Anwendung dieser Methode das Programm bereits geschrieben sein muss, eignet sie sich besonders für das Testen, für die Wartung und für die Fehlersuche. Es ist also eine weitere Abstraktionsmethode, die klassische Methoden, wie etwa Informationsverbergung, Datenabstraktion oder Teilsystementwurf, erweitert.

Die Sicht auf ein Programm wird in diesem Kontext auch als *Slice* bezeichnet. Ein Slice wird anhand eines sogenannten *Slicekriteriums* bestimmt. Ein Slicekriterium $< i, V >$ besteht aus der Anweisungsnummer i und aus einer Menge V von Variablen, die an der Stelle i von Interesse sind. Der *Slice Algorithmus* berechnet hiermit alle relevanten Codeanweisungen und löscht den restlichen Teil des Programms.

Program Slicing wird im Folgenden durch die bereits eingeführten Bewertungskriterien genauer analysiert.

1. *Varianten:* Keine Unterstützung.

2. *Abhängigkeiten:* Explizit ermittelbare Abhängigkeiten durch Daten- und Kontrollflussanalyse.

3. *Sichten:* Definition von Slicekriterien und Anwendung des Slice-Algorithmus.

4. *Werkzeugunterstützung:* Keine Unterstützung.

5. *Programmiersprache:* Unabhängig.

7.6.2. Feature Exploration and Analysis Tool (FEAT)

Robillard und Murphy entwickelten in ihren Arbeiten ein Werkzeug, um Softwareentwickler bei der Modifizierung und Erweiterung von bestehendem Quellcode zu unterstützen [Rob02, RM02, RM03]. Der Code, der den Entwickler interessiert, ist typischerweise in der gesamten Codebasis verstreut. Daher ist es nicht immer ohne Weiteres möglich, Modifizierungen oder Erweiterungen durchzuführen. Um die Lokalisierung des Codes zu vereinfachen, haben *Robillard und Murphy* einen Lösungsansatz vorgestellt, der die Programmstruktur heranzieht und aus dieser iterativ die Elemente extrahiert, die für den Programmierer von Interesse sind.

Die Grundlage dieses Ansatzes bildet eine gerichtete Graphstruktur, die automatisch aus dem Programm ermittelt wird. Die Knoten des Graphen umfassen Klassen, Methoden und Felder. Kanten repräsentieren semantische Beziehungen zwischen Knoten. Beispielsweise kann eine Klasse Methoden deklarieren. Dies wird durch eine `declares`-Kante zwischen den entsprechenden Knoten ausgedrückt. Weitere Kantenarten können sein: `reads`, `writes` oder `superclass`. Dies wird allerdings hier nicht weiter ausgeführt.

Damit nun aus diesem Graphen nur der Teil extrahiert werden kann, der den Softwareentwickler interessiert, werden entsprechende Anfragen an die Programmstruktur gestellt. Die Ergebnisse dieser Anfragen sind erneut Elemente und Beziehungen, die einen Teilgraphen der Programmstruktur darstellen. Auf diese Weise kann ein Entwickler diesen Graphen konstruieren.

Das Konzept von *Robillard und Murphy* ist in das Werkzeug Feature Exploration and Analysis Tool (FEAT) eingeflossen [RM03]. Es ermöglicht Entwicklern, das Programm auf einem höheren Abstraktionsniveau zu betrachten. Modifikationen oder Erweiterungen müssen nichtsdestotrotz im Code selbst durchgeführt werden. Hierfür bietet das Werkzeug bei Selektion von Elementen im Graphen eine komfortable Navigation in den Quellcode.

Zur genaueren Bewertung von FEAT werden im Folgenden anhand der eingeführten Kriterien die Konzepte evaluiert.

1. *Varianten:* Keine Unterstützung.

2. *Abhängigkeiten:* Explizit ermittelbare Abhängigkeiten durch Aufstellung einer Graphrepräsentation des Programms.

3. *Sichten:* Iterative Erstellung durch Anfragen an die Graphstruktur.

4. *Werkzeugunterstützung:* FEAT.

5. *Programmiersprache:* Java.

7.6.3. Spotlight

Coppit et al. haben in ihren Arbeiten festgestellt, dass Änderungen in großen Softwaresystemen nur schwer durchgeführt werden können, da die Komplexität des Systems dies nicht ohne Weiteres zulässt. Softwareentwickler müssen mit sehr vielen Codefragmenten umgehen, auch wenn der Großteil dieser Fragmente für die aktuelle Implementierungsaufgabe nicht von Belangen ist. Ein Belang (engl. concern) ist dabei alles, was den Programmierer interessiert.

In diesem Zusammenhang haben *Coppit et al.* das Werkzeug Spotlight entwickelt, das die Separation von Belangen (engl. seperation of concerns) ermöglicht, um die Komplexität des Softwaresystems angemessen zu reduzieren [CC04, PC05a, PC05b, CPR07]. Hierfür wurden sogenannte Softwarepläne eingeführt. Ein Softwareplan ist eine editierbare Sicht auf eine Software, die aus einer Sequenz von Codeblöcken besteht, die wiederum mit verschiedenen Belangen assoziiert sind. Für das Verschmelzen der Pläne bedarf es an einem weiteren Softwareplan, der alle Pläne zusammen mit dem entsprechenden Code für die Zusammenführung beinhaltet.

Der Vorteil von derartigen Softwareplänen ist, dass durch die Separation von Belangen der Programmierer mehr Klarheit über die Funktionalität bekommt. Die Erzeugung dieser Sichten wird durch sogenannte Graphdokumente gewährleistet. Sie stellen die zugrunde liegende Datenstruktur dar. Ein Graphdokument ist ein gerichteter Graph. Die Knoten des Graphen sind Pläne oder Codeblöcke. Die Kanten sind mit Plannamen beschriftet und verbinden die Knoten miteinander. Pläne werden dabei kreisförmig repräsentiert, Codeblöcke hingegen quadratisch. Soll nun eine Sicht für einen Plan erstellt werden, wird von diesem Knoten angefangen entlang der gerichteten Kanten mit dessen Namen traversiert und alle besuchten Codeblöcke hintereinander konkateniert. Wird ein Softwareplan editiert, bedeutet dies für die zugrunde liegende Datenstruktur, dass Transformationen zum Splitten, Löschen und Erzeugen von Blöcken durchgeführt werden müssen.

Schließlich werden erneut die Kriterien zur Bewertung der Konzepte von Spotlight herangezogen.

1. *Varianten:* Liste von Belangen.
2. *Abhängigkeiten:* Keine Unterstützung.
3. *Sichten:* Graphdokumente mit Softwareplänen.
4. *Werkzeugunterstützung:* Spotlight.
5. *Programmiersprache:* Java.

7.6.4. Mylar und Mylyn

Kersten et al. haben in ihren Arbeiten die Mängel integrierter Entwicklungsumgebungen in Bezug auf Handhabung großer Programmsysteme untersucht. Sie haben dabei festgestellt, dass Programmierer im Rahmen einer bestimmten Aufgabe oftmals sehr viele Stellen im Programmsystem aufsuchen müssen. Zur Unterstützung

des Programmierers bieten integrierte Entwicklungsumgebungen, wie zum Beispiel Eclipse, sogenannte Sichten auf das System, um Codefragmente einfacher zu finden. Bei größeren Systemen reichen allerdings die Sichten alleine nicht mehr aus. Daher haben *Kersten et al.* ein Werkzeug entwickelt, welches das Konzept der Sichten ergänzt. In diesem Zusammenhang ist Mylar entstanden [MKRC05, KM05, KM06]. Mittlerweile ist es in erweiterter Form unter dem Namen Mylyn als Eclipse Plugin erhältlich [STH11, Pro11]. Im Folgenden wird basierend auf Mylar das Konzept genauer erläutert.

Zur Illustration der Einschränkungen integrierter Entwicklungsumgebungen bei der Abarbeitung einer Aufgabe wurden zwei Eclipse Plugins herangezogen: (1) Java Development Tooling (JDT) und (2) AspectJ Development Tooling (AJDT). Folgende Defizite wurden bei den Untersuchungen identifiziert:

1. JDT Package Explorer: Bei sehr vielen Knoten ist es schwierig, den Explorer zu benutzen. Die hierarchischen Beziehungen können nur durch das Hoch- und Runterscrollen ermittelt werden.

2. JDT Search: Die Suche liefert oftmals eine zu hohe Anzahl an Ergebnissen. Die Ergebnisliste muss vom Programmierer manuell gefiltert werden.

3. JDT Outline: Auch diese Sicht erfordert das Hoch- und Runterscrollen, um relevante Informationen zu finden.

4. JDT Hierarchy: Die Typhierarchie besteht aus enorm vielen Klassen, die die gesuchte Klasse erweitern.

5. AJDT Outline: Die Suche nach Advices führt zu einer rasanten Zunahme der Anzahl an Elementen im Package Explorer, da diese über mehrere Klassen verstreut sind.

6. AJDT Visualiser: Diese Sicht beinhaltet sehr oft viele Dateien, sodass die Namen nicht zu lesen sind. Der Programmierer muss alle für die Aufgabe relevanten Informationen manuell filtern.

Zur Lösung der oben genannten Defizite haben *Kersten et al.* Mylar entwickelt. Es überwacht die Aktivitäten eines Programmierers und erfasst die Relevanz von Codefragmenten in einem sogenannten Degree-of-Interest (DoI)-Modell. Das DoI-Modell weist jedem Programmelement einen Wert zu. Wenn eines dieser Elemente selektiert oder editiert wird, erhöht sich dieser Wert. Falls ein Element über eine bestimmte Zeit nicht betrachtet wird, nimmt der Wert automatisch ab. Anhand dieser Werte werden die Sichten entsprechend angepasst. Die Elemente werden im Werkzeug durch farbliche Hervorhebung visualisiert. Je dunkler die Schattierung ist, desto höher ist sein Wert im DoI-Modell. Es ergeben sich folgende Erweiterungen im Vergleich zu den obigen Betrachtungen:

1. Mylar Package Explorer: Eine Filterung durch Auswertung des DoI-Modells reduziert die Anzahl der Knoten. Es sind nur relevante Dateien und Bibliotheken sichtbar.

2. Mylar Problems: Es werden nur die Problemmeldungen ausgegeben, die entsprechend dem DoI-Modell als relevant betrachtet werden.

3. Mylar Outline: In dieser Sicht werden ebenfalls nur die relevanten Elemente dargestellt. Zudem ist diese Sicht mit dem Editor verknüpft, sodass nur die Codefragmente aufgefalten werden, die in der Outline dargestellt sind.

4. Mylar Pointcut Navigator: Hier werden ebenfalls nur Advices angezeigt, die als relevant bewertet werden.

Anhand dieser erläuterten Maßnahmen unterstützt Mylar einen Programmierer dabei, sich nur auf die Teile des Programmsystems zu fokussieren, die für die Abarbeitung seiner Aufgabe relevant sind. Somit wird die Zeit zur Suche nach Informationen deutlich reduziert.

Abschließend werden die beschriebenen Konzepte von Mylar anhand der Bewertungskriterien analysiert.

1. *Varianten:* Keine Unterstützung.

2. *Abhängigkeiten:* Keine Unterstützung.

3. *Sichten:* DoI-Modell.

4. *Werkzeugunterstützung:* Eclipse Mylin Plugin.

5. *Programmiersprache:* Unabhängig.

7.6.5. Colored Integrated Development Environment (CIDE)

Kästner et al. haben in einer Reihe von Beiträgen einen werkzeuggestützten Ansatz vorgestellt, der die Refaktorierung von Altsystemen in eine Menge von Features unterstützt [Käs07, KAK08, KTA08, FKF+10]. In diesem Zusammenhang wurde ein Eclipse Plugin für Java-Entwicklungsumgebungen realisiert. Benutzer können dabei das vorhandene Softwaresystem in das Werkzeug laden, Codefragmente markieren und diese mit Features assoziieren. Jedem Feature ist eine Farbe zugewiesen. Im Editor werden dementsprechend alle markierten Codefragmente mit diesen zugewiesenen Farben repräsentiert. Aufgrund dieser farblichen Hervorhebung wurde das Werkzeug von den Autoren als Colored Integrated Development Environment (CIDE) bezeichnet. Im Quellcode können nicht beliebige Fragmente markiert werden, sondern nur die Teile, die als Element im abstrakten Syntaxbaum repräsentiert sind. Diese sind zum Beispiel Klassen, Methoden, Anweisungen, Parameter etc.

Ist das Altsystem refaktorisiert, können über einen Konfigurierungsvorgang Features selektiert werden. Das Werkzeug entfernt daraufhin jeglichen Code, die mit Features assoziiert sind, aber nicht Teil der Konfiguration sind.

Auch für CIDE wird im Folgenden eine Bewertung entsprechend den vorgestellten Kriterien durchgeführt.

1. *Varianten:* Farbliche Markierung von Varianten ohne Repräsentation.

2. *Abhängigkeiten:* Keine Unterstützung.
3. *Sichten:* Selektion mit Konfigurationsmaschine.
4. *Werkzeugunterstützung:* Eclipse CIDE Plugin.
5. *Programmiersprache:* Java.

7.6.6. Vergleich

Nachdem nun verwandte Arbeiten in den vorherigen Abschnitten erläutert und bewertet wurden, dient dieser Abschnitt zum Vergleichen der Ansätze mit dem Ansatz dieser Arbeit. Tabelle 7.1 fasst das Resultat zusammen.

Zusätzlich zu dieser Arbeit werden Varianten noch explizit durch CIDE unterstützt. In CIDE werden Varianten durch farbliche Markierungen gekennzeichnet. Eine Repräsentation der Variabilität ist dabei nicht gegeben. Der primäre Zweck hierbei liegt in der Refaktorierung des Quellcodes. In dieser Arbeit wird hingegen der Fokus auf einen vollständigen Ansatz gelegt, d.h. von der Identifikation der Variabilität (Bottom-Up), seiner Modellierung (Top-Down) und Umsetzung (Hybrid) gelegt. Dabei werden insbesondere die Variabilitätsmechanismen der Programmiersprache als Anknüpfungspunkt zum Code gesehen. Eine weitere Arbeit, dessen Hauptzweck nicht in der Modellierung von Variabilität liegt, aber dafür implizit verwendet werden könnte, ist Spotlight. Die Liste von Belangen könnte hierbei als eine Liste von Varianten interpretiert werden. Allerdings bietet eine derartige Liste nicht bei Weitem die Möglichkeiten eines Variabilitätsmodells (vgl. hierzu Kapitel 4).

Abhängigkeiten werden in dieser Arbeit durch die Restriktionsmodellierung ausgedrückt. Hierbei können sowohl explizite als auch implizite Abhängigkeiten modelliert werden. Die beiden Ansätze Slicing und FEAT unterstützen ebenfalls das explizite Erfassen von Abhängigkeiten. Während im Slicing-Verfahren dies durch eine Daten- und Kontrollflussanalyse geschieht, wird in FEAT die Programmstruktur in Form eines Graphens repräsentiert. Auf diese Weise werden Abhängigkeiten identifiziert.

Der wohl wichtigste Ansatz zur Komplexitätsreduzierung des Quellcodes ist die Erzeugung von Sichten. Daher wird dies auch von jedem Ansatz unterstützt. Während in dieser Arbeit und in CIDE dies über die Selektion im Konfigurationsmodell mit einer Konfigurationsmaschine im Hintergrund stattfindet, wird im Slicing-Verfahren ein Slicingkriterium definiert und der Slicealgorithmus angewendet. Weiterhin wird in FEAT eine Sicht durch iterative Anfragen an die Graphstruktur erzeugt. In Spotlight erfolgt die Sichtgenerierung über Graphdokumente mit Softwareplänen. Schließlich wird in Mylar/Mylin das DoI-Modell zur Erzeugung der Sicht eingesetzt.

Bis auf das Slicing-Verfahren sind alle Konzepte in ein Werkzeug eingeflossen. Diese Arbeit erweitert das Eclipse CDT-Plugin. Auf Eclipse basieren zudem Mylar/Mylin als auch CIDE. FEAT und Spotlight sind eigenständige Java-Entwicklungen.

Das Slicing-Verfahren und Mylar/Mylin sind Programmiersprachen-unabhängige Ansätze. Alle anderen Ansätze basieren auf einer Programmiersprache. Diese Arbeit unterstützt die Programmiersprache C und damit teilweise auch C++. FEAT, Spotlight und CIDE unterstützen die Programmiersprache Java.

	Mengi	Slicing	FEAT	Spotlight	Mylar/Mylin	CIDE
Varianten	Variabilitäts-modellierung, Code-anbindung durch Variabilitätsmecha-nismen	×	×	Liste von Belangen	×	Farbliche Markierung, ohne Repräsentation
Abhängig-keiten	Explizit und implizit durch Restriktionsmodel-lierung	Explizit durch Daten- und Kontrollflussana-lyse	Explizit durch Graphrepräsenta-tion des Programms	×	×	×
Sichten	Selektion mit Konfigurations-maschine	Slicekriterium mit Slicealgorithmus	Iterativ durch Anfragen an die Graphstruktur	Graphdokumente mit Softwareplänen	DoI-Modell	Selektion mit Konfigurations-maschine
Werkzeug-unterstützung	Erweiterung des Eclipse CDT Plugins	×	FEAT	Spotlight	Eclipse Mylin Plugin	Eclipse CIDE Plugin
Programmier-sprache	C, teils C++	Unabhängig	Java	Java	Unabhängig	Java

Tabelle 7.1.: Vergleichsaufstellung mit den Konzepten dieser Arbeit und verwandter Arbeiten

7.7. Zusammenfassung

In diesem Kapitel wurde ein Ansatz vorgestellt, der die Implementierungsaufgabe um Aspekte der Variabilität erweitert. Der Fokus hierbei lag in der Unterstützung für die Programmiersprache C. Dabei wurden dieser Sprache zugrunde liegenden Variabilitätsmechanismen ermittelt und ein Prozess definiert, der die Variabilitätsmodellierung, Restriktionsmodellierung sowie Konfigurierung als integrale Aktivitäten einbezieht. Insbesondere ist hierbei hervorzuheben, dass durch die Konfigurierung eine entsprechende temporäre Sicht erzeugt wird, die für jegliche Modifikationen herangezogen werden kann. Da diese Modifikationen überwacht werden, kann der Code anhand von Transformationsregeln und den Variabilitätsmechanismen geeignet strukturiert in den ursprünglichen Quellcode überführt werden.

Teil IV.

Epilog

Kapitel 8.

Schlussbemerkungen

8.1. Zusammenfassung

Die Softwareentwicklung im Automobilbau hat in den letzten Jahrzehnten immer mehr an Bedeutung gewonnen. Gleichzeitig ist sie auch deutlich komplexer geworden. Faktoren, wie etwa die hardwaregetriebene Softwareentwicklung, Softwareabhängigkeiten oder die Vielfalt an Softwarevarianten, haben hierzu beigetragen.

Insbesondere führt das Streben nach einer hohen Produktvielfalt zu einer Variabilität im Softwareentwicklungsprozess, die bisher unzureichend beherrscht wird. Die geeignete Modellierung und Bindung dieser Variabilität ist eine wichtige Voraussetzung, um diesen Komplexitätsfaktor in den Modellen des Entwicklungsprozesses zu reduzieren. In diesem Zusammenhang wurde im Rahmen dieser Arbeit ein Variabilitätsmodell entwickelt, mit dem die Variabilität in den Modellen explizit erfasst werden kann. Das Variabilitätsmodell basiert auf das Konzept der Auswahlmodelle, bei dem der Variationspunkt und die Varianten die Kernkomponenten darstellen. Aufgrund fehlender Strukturierungsmöglichkeiten wurde das Modell um ein Gruppierungskonzept erweitert. Zusätzlich beinhaltet das Modell die Möglichkeit, Variabilitätsmechanismen festzuhalten. Auf diese Weise kann eine geeignete Kopplung mit den Softwaredokumenten im Entwicklungsprozess erreicht werden. Durch Gruppen- und Variantenkardinalitäten können entsprechende variable Eigenschaften definiert und somit einfache restriktive Aussagen, wie etwa Optionalität, Alternativität oder Exklusivität, über Varianten und Variantengruppen getroffen werden. Für komplexere Restriktionen wurde das Variabilitätsmodell um ein Restriktionsmodell erweitert. Als Restriktionssprache wurde eine erweiterte Form der Aussagenlogik eingeführt. Zusätzlich wurde WCRL integriert. Neben der Modellierung von Variabilität wurde die Bindung als weiteres wichtiges Merkmal identifiziert. Hierbei wurde ein Konfigurationsmodell entwickelt, das sich aus dem Variabilitätsmodell ableitet und die interaktive Konfigurierung durch eine proaktive und reaktive Validierung unterstützt.

Die beschriebenen Konzepte zur Variabilitätsmodellierung stellen eine wichtige Voraussetzung dar, Variabilität im Softwareentwicklungsprozess zu erfassen. Sie reichen allerdings alleine nicht aus, um die Komplexität zu beherrschen, denn die Modelle im Entwicklungsprozess bleiben von der Komplexität nicht unberührt. Sie sind somit auch wesentliche Komplexitätsträger. Erst eine detaillierte Untersuchung dieser Modelle und die entsprechende Einführung von geeigneten Konzepten er-

möglicht die effiziente Anwendung des Variabilitätsmodells. Diese Arbeit hat sich in diesem Zusammenhang mit Modellen aus der frühen Softwareentwicklungsphase beschäftigt. Auf Funktionsebene wurden Funktionsnetze betrachtet. Auf Architekturebene wurden Simulink-Modelle untersucht. Schließlich wurde auf Codeebene der C-Quellcode behandelt.

Auf Funktionsebene wurde festgestellt, dass die bisherige Anwendung von Funktionsnetzen nicht ausreicht, da sie nicht die Lücke zwischen der Anforderungsspezifikation und der E/E-Architektur schließen kann. Insbesondere erschweren verschiedene Formalismen und Notationen die Integration der Funktionsnetzkonzepte. In dieser Arbeit wurde daher ein Metamodell für Funktionsnetze entwickelt, das die verschiedenen Konzepte umfasst und um weitere Abstraktionsebenen erweitert. Auf diese Weise konnte die Lücke zwischen der Anforderungsspezifikation und der E/E-Architektur vollständig geschlossen werden. Für den Übergang zwischen den verschiedenen Ebenen wurden Abstraktionsregeln eingeführt, mit denen bestimmte Merkmale im Funktionsnetz einer Ebene abstrahiert werden können. Darüber hinaus wurde die Wiederverwendung in Funktionsnetzen methodisch und konzeptionell unterstützt. Durch einen Modellierungsprozess wurde in einer ersten Phase die Domänenbibliothek für Funktionsnetze aufgebaut. Zu diesem Zweck wurde das Domänenmodell eingeführt. In der zweiten Phase werden die Elemente und Strukturen aus der ersten Phase instanziiert und für die Funktionsnetzmodellierung auf allen Abstraktionsebenen verwendet. An dieser Stelle wird entsprechend des Funktionsnetzmetamodells modelliert. Damit das beschriebene Variabilitätsmodell eine Verknüpfung zum Funktionsnetz besitzt, wurde der Variabilitätsmechanismus der Funktionsvariante entwickelt. Hiermit werden die Varianten eines Variationspunktes gekapselt.

Simulink-Modellvarianten wurden durch einen Differenzierungs- und Restrukturierungsprozess in ein Simulink-Familienmodell überführt. Der Differenzierungsprozess besteht dabei aus der Ermittlung der Gemeinsamkeiten und Variabilität. Zu diesem Zweck wurde ein interaktiver Ansatz und ein automatischer etabliert. Letzteres basiert dabei auf Namens- und Typgleichheit. Die Ergebnisse der Differenzierung bestehen aus einem Kommunalitätsmodell und zwei Differenzmodellen. Durch farbliche Markierungen im Kommunalitätsmodell wurden Variationspunkte gekennzeichnet. Die Ergebnisse werden im Restrukturierungsprozess als Grundlage zur Analyse herangezogen. Ziel dabei ist es, die Differenzmodelle in das Kommunalitätsmodell durch Verwendung von geeigneten Variabilitätsmechanismen zu integrieren. Durch eine Analyse und Bewertung existierender Variabilitätsmechanismen hat sich ergeben, dass Model Variants und Variant Subsystem besonders geeignet sind. Hierauf basierend wurden Restrukturierungsregeln definiert, die bei der Zusammenführung unterstützend wirken. Die Regeln umfassen dabei die Schnittstelle von Modellen und Blöcken, Blocktypen und Verbindungen. Durch Kombination dieser Regeln können auch komplexere Situationen behandelt werden. Wie bei Funktionsnetzen stellen die Variabilitätsmechanismen Anknüpfungspunkte für das Variabilitätsmodell dar.

Der Quellcode hat in Bezug auf Variabilität die Nachteile, dass (1) der Programmierer alle Varianten gleichzeitig sieht, (2) der Code einer Variante verstreut ist,

(3) Restriktionen zwischen Varianten nicht ohne Weiteres auszudrücken sind und (4) eine valide Konfiguration nur schwer zu erstellen ist. In dieser Arbeit wurde daher ein variantengetriebener Implementierungsprozess eingeführt, um die identifizierten Probleme zu beherrschen. Der Prozess bezieht dabei zusätzlich zur Implementierung die Variabilitätsmodellierung, Restriktionsmodellierung und Konfigurierung als integrale Aktivitäten mit ein. Durch die Variabilitätsmodellierung werden Varianten zentralisiert. Restriktionen zwischen Varianten können dann durch das Restriktionsmodell definiert werden. Aus einer Konfigurierung wird eine variantenspezifische Sicht erzeugt, die bei der Implementierung überwacht wird. Nach Beendigung der Implementierung wird der Code anhand von Transformationsregeln durch Variabilitätsmechanismen angereichert, sodass die Modifikationen in den ursprünglichen Code überführt werden können.

Aus den beschriebenen Konzepten sind verschiedene prototypische Werkzeuge entstanden: (1) ein Editor zur Variabilitätsmodellierung, (2) ein Editor zur Restriktionsmodellierung, (3) ein Editor zur Konfigurierung, (4) ein Editor zur Domänenmodellierung, (5) ein Editor zur Funktionsnetzmodellierung mit verschiedenen Abstraktionsniveaus, (6) ein Werkzeug zur Differenzierung und (7) ein Editor zur variantengetriebenen Implementierung. Sie zeigen den Nachweis der Machbarkeit der in dieser Arbeit beschriebenen Konzepte.

8.2. Ausblick

Wie bereits anfangs erläutert, ist diese Arbeit ein weiterer Beitrag zur Behebung bestimmter Komplexitätsfaktoren im Softwareentwicklungsprozess eines Automobils. Es gibt noch viel Handlungsbedarf in verschiedenen Themengebieten. Drei derartige Bereiche, die eng mit den Themen dieser Arbeit verknüpft sind, werden im Folgenden vorgestellt.

Variabilität wurde auf verschiedenen Abstraktionsebenen behandelt. Der Übergang zwischen diesen Ebenen hinsichtlich der Variabilitätseigenschaften wurde in diesem Kontext nicht behandelt. So gibt es zwischen Funktionsnetze und Simulink-Modelle viele korrespondierende Konzepte, die den Übergang zwischen diesen beiden Modellen unterstützen. Zum einen handelt es sich in beiden Fällen um datenflussorientierte Sprachen und zum anderen verwenden sie ähnliche Variabilitätsmechanismen zur Realisierung von Variationspunkten. Eine Untersuchung diesbezüglich könnte aufschlussreichere Ergebnisse bei der (teil-)automatisierten Erzeugung und Verfolgung bestimmter Modellstrukturen liefern.

Weiterhin wurde im Rahmen dieser Arbeit die Hardwarearchitektur kaum beachtet. Sie spielt allerdings in Bezug auf Variabilität eine wichtige Rolle. Es gibt Hardwarearchitekturvarianten, die verschiedene Auswirkungen auf das Gesamtfahrzeug haben. Diese Variabilität zu erfassen ist genauso wichtig, wie es in anderen Softwaredokumenten der Fall ist. Darüber hinaus wird es auch Deploymentvarianten geben, die ebenfalls verschiedene Auswirkungen haben. Werden Deploymentvarianten und Hardwarearchitekturvarianten gemeinsam betrachtet, entsteht eine weitere Dimension der Komplexität, die es zu beherrschen gilt.

Schließlich sei noch in diesem Zusammenhang der AUTOSAR-Standard erwähnt. AUTOSAR gewinnt immer mehr Einfluss in der Softwareentwicklung eines Autos. Welche Änderungen der Standard für Geschäftsmodelle und Entwicklungsprozesse mit sich bringt, ist immer noch unklar. Untersuchungen diesbezüglich könnten Klarheit geben. Das Thema Variabilität wird ebenfalls vom AUTOSAR-Standard beeinflusst werden. Wie der Standard zu diesem Punkt effektiv integriert werden kann, ist eine weitere Fragestellung.

Literaturverzeichnis

[AG09] KPMG International AG: *Automotive Product Diversity: Not for Profit?* Studie KPMG Audit Tax Advisory, August 2009. http://www.kpmg.de/.

[Arb11] Youssef Arbach: *Detecting Variability in Simulink Models for Software Product Lines*. Masterarbeit, Rheinisch-Westfälische Technische Hochschule (RWTH) Aachen, Juli 2011. Betreuer: Cem Mengi.

[AUT10a] AUTOSAR: *Feature Specification of the BSW Architecture and the RTE*. Release 4, Version 1.0.0, Oktober 2010. http://www.autosar.org/.

[AUT10b] AUTOSAR: *Glossary*. Release 4, Version 2.3.0, Oktober 2010. http://www.autosar.org/.

[AUT10c] AUTOSAR: *Main Requirements*. Release 4, Version 2.2.0, Oktober 2010. http://www.autosar.org/.

[AUT10d] AUTOSAR: *Virtual Functional Bus*. Release 4, Version 2.1.0, Oktober 2010. http://www.autosar.org/.

[AvW00] Pierre America und Jan van Wijgerden: *Requirements Modeling for Families of Complex Systems*. In: Frank van der Linden (Herausgeber): *Software Architectures for Product Families, International Workshop IW-SAPF-3*, Band 1951 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 199–209, Las Palmas de Gran Canaria, Spanien, März 2000. Springer. ISBN 3-540-41480-0.

[Bal01] Helmut Balzert: *Lehrbuch der Software-Technik*. Spektrum Akademischer Verlag, 2001. ISBN 978-3827403018.

[Bar06] John Barnes: *Programming in Ada 2005*. Addison-Wesley Longman, Mai 2006. ISBN 978-0321340788.

[Bar09] Michael Barr: *Real men program in C*. Embedded Systems Design, 22(7):3, Juli/August 2009.

[BBD+06] Danilo Beuche, Andreas Birk, Heinrich Dreier, Andreas Fleischmann, Heidi Galle, Gerald Heller, Dirk Janzen, Isabel John, Ramin Tavakoli Kolagari, Thomas von der Maßen (Ed.) und Andreas Wolfram: *Report of the GI Work Group "Requirements Management Tools for Product Line Engineering"*. Aachener Informatik Berichte AIB-2006-14, Departement of Computer Science, Rheinisch-Westfälische Technische Hochschule (RWTH) Aachen, Dezember 2006. ISSN 0935-3232.

[BBR+07] Andreas Bauer, Manfred Broy, Jan Romberg, Bernhard Schätz, Peter Braun, Ulrich Freund, Nuria Mata, Robert Sandner, Pierre Mai und Dirk Ziegenbein: *Das AutoMoDe-Projekt*. Informatik - Forschung und Entwicklung, 22(1):45–57, Dezember 2007. Springer Berlin / Heidelberg, ISSN 0178-3564.

[Bec03] Martin Becker: *Towards a General Model of Variability in Product Families*. In: *Proceedings of the 1st Workshop on Software Variability Management*, Februar 2003.

[Beh00] Anita Behle: *Wiederverwendung von Softwarekomponenten im Internet*. Dissertation, Rheinisch-Westfälische Technische Hochschule (RWTH) Aachen, 2000. ISBN 978-3824404964.

[BKPS07] Manfred Broy, Ingolf H. Krüger, Alexander Pretschner und Christian Salzmann: *Engineering Automotive Software*. Proceedings of the IEEE, 95(2):356 –373, Februar 2007. ISSN 0018-9219.

[BLP05] Stan Bühne, Kim Lauenroth und Klaus Pohl: *Modelling Requirements Variability across Product Lines*. In: *13th IEEE International Conference on Requirements Engineering (RE 2005)*, Seiten 41–52, Paris, Frankreich, 2005. IEEE Computer Society. ISBN 0-7695-2425-7.

[BPK09] Goetz Botterweck, Andreas Polzer und Stefan Kowalewski: *Using Higher-Order Transformations to Derive Variability Mechanism for Embedded Systems*. In: Sudipto Ghosh (Herausgeber): *Models in Software Engineering, Workshops and Symposia at MODELS 2009*, Band 6002 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 68–82, Denver, CO, USA, Oktober 2009. Springer. ISBN 978-3-642-12260-6.

[BPK10] Goetz Botterweck, Andreas Polzer und Stefan Kowalewski: *Variability and Evolution in Model-based Engineering of Embedded Systems*. In: Holger Giese, Michaela Huhn, Jan Phillips und Bernhard Schätz (Herausgeber): *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VI*, Seiten 87–96, Dagstuhl, Deutschland, 2010. fortiss GmbH, München.

[BPSP04] Danilo Beuche, Holger Papajewski und Wolfgang Schröder-Preikschat: *Variability management with feature models*. Sci. Comput. Program., 53(3):333–352, Dezember 2004. ISSN 0167-6423.

[Bör94] Jürgen Börstler: *Programmieren-im-Großen: Sprachen, Werkzeuge, Wiederverwendung*. Dissertation, Rheinisch-Westfälische Technische Hochschule (RWTH) Aachen, 1994.

[Bro04] Alan W. Brown: *Model driven architecture: Principles and practice*. Software and System Modeling, 3(4):314–327, 2004.

[Bro06] Manfred Broy: *Challenges in automotive software engineering*. In: Leon J. Osterweil, H. Dieter Rombach und Mary Lou Soffa (Herausgeber): *28th International Conference on Software Engineering (ICSE 2006)*, Seiten 33–42, Shanghai, China, Mai 2006. ACM. ISBN 1-59593-375-1.

[Bur97] Stanley Burris: *Logic for Mathematics and Computer Science*. Prentice Hall, 1997. ISBN 0132859742.

[Bur05] Ed Burnette: *Eclipse IDE Pocket Guide*. O'Reilly Media, August 2005. ISBN 978-0-596-10065-0.

[BW09] Danilo Beuche und Jens Weiland: *Managing Flexibility: Modeling Binding-Times in Simulink*. In: Richard F. Paige, Alan Hartman und Arend Rensink (Herausgeber): *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA '09)*, Band 5562 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 289–300, Enschede, Niederlande, Juni 2009. Springer-Verlag Berlin, Heidelberg. ISBN 978-3-642-02673-7.

[CA04] Krzysztof Czarnecki und Michal Antkiewicz: *FeaturePlugin: Feature Modeling Plug-in for Eclipse*. In: Michael G. Burke (Herausgeber): *Proceedings of the 2004 OOPSLA workshop on Eclipse Technology eXchange (ETX 2004)*, Seiten 67–72, New York, NY, USA, Oktober 2004. ACM.

[CC04] David Coppit und Benjamin Cox: *Software Plans for Seperation of Concerns*. In: Yvonne Coady und David H. Lorenz (Herausgeber): *Proceedings of the Third Aspect-Oriented Software Development Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS 2004)*, Seiten 22–27, Lancaster, UK, März 2004.

[CE00] Krzysztof Czarnecki und Ulrich W. Eisenecker: *Generative programming - methods, tools and applications*. Addison-Wesley, 2000. ISBN 978-0-201-30977-5.

[CHE04] Krzysztof Czarnecki, Simon Helsen und Ulrich Eisenecker: *Staged Configuration Using Feature Models*. In: Robert L. Nord (Herausgeber): *Proceedings of the Third Software Product Line Conference (SPLC 2004)*, Band 3154 der Reihe *Lecture Notes in Computer Science*, Seiten 266–283, Boston, MA, USA, September 2004. Springer.

[CHE05a] Krzysztof Czarnecki, Simon Helsen und Ulrich Eisenecker: *Formalizing cardinality-based feature models and their specialization*. Software Process: Improvement and Practice, 10(1):7–29, 2005.

[CHE05b] Krzysztof Czarnecki, Simon Helsen und Ulrich Eisenecker: *Staged configuration through specialization and multilevel configuration of feature*

models. Software Process: Improvement and Practice, 10(2):143–169, 2005.

[CK05] Krzysztof Czarnecki und Chang Hwan Peter Kim: *Cardinality-Based Feature Modeling and Constraints: A Progress Report*. In: *Proceedings of the International Workshop on Software Factories at OOPSLA 2005*, San Diego, CA, USA, Oktober 2005. ACM.

[CN07] Paul Clements und Linda M. Northrop: *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., 6. Auflage, 2007. ISBN 9780201703320.

[CPR07] David Coppit, Robert R. Painter und Meghan Revelle: *Spotlight: A Prototype Tool for Software Plans*. In: Stephanie Kawada (Herausgeber): *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*, Seiten 754–757, Minneapolis, MN, USA, Mai 2007. IEEE Computer Society. ISBN 978-0-7695-2828-1.

[CR06] Eric Clayberg und Dan Rubel: *Eclipse: Building Commercial-Quality Plug-ins*. Addison-Wesley Professional, 2. Auflage, März 2006. ISBN 978-0-321-42672-7.

[Cub05] Ulrich Cuber: *Einstieg in Eclipse 3 - Einführung, Programmierung, Plugin-Nutzung*. Galileo Computing, 2005. ISBN 978-3-89842-552-0.

[Dau06] Berthold Daum: *Das Eclipse-Codebuch: 182 Tipps, Tricks und Lösungen für Eclipse-spezifische Probleme*. Dpunkt Verlag, Januar 2006. ISBN 978-3898643764.

[DHJ+08] Florian Deissenboeck, Benjamin Hummel, Elmar Jürgens, Bernhard Schätz, Stefan Wagner, Jean Francois Girard und Stefan Teuchert: *Clone Detection in Automotive Model-Based Development*. In: Wilhelm Schäfer, Matthew B. Dwyer und Volker Gruhn (Herausgeber): *Proceedings of the 30th International Conference on Software Engineering (ICSE 2008)*, Seiten 603–612, Leipzig, Deutschland, Mai 2008. ACM. ISBN 978-1-60558-079-1.

[DHJ10] Florian Deissenboeck, Benjamin Hummel und Elmar Jürgens: *Code clone detection in practice*. In: Jeff Kramer, Judith Bishop, Premkumar T. Devanbu und Sebastián Uchitel (Herausgeber): *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2 (ICSE 2010)*, Band 2, Seiten 499–500, Cape Town, South Africa, Mai 2010. ACM. ISBN 978-1-60558-719-6.

[DLPW08] Christian Dziobek, Joachim Loew, Wojciech Przystas und Jens Weiland: *Von Vielfalt und Variabilität - Handhabung von Funktionsvarianten in Simulink-Modellen*. Elektronik automotive, 2:33–37, Februar 2008.

[Dok11] EMFText Dokumentation: *emftext USER GUIDE*, März 2011. `http://www.emftext.org/`.

[DW09] Christian Dziobek und Jens Weiland: *Variantenmodellierung und -konfiguration eingebetteter automotive Software mit Simulink*. In: Holger Giese, Michaela Huhn, Ulrich Nickel und Bernhard Schätz (Herausgeber): *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme V*, Band 2009-01 der Reihe *Informatik-Bericht*, Seiten 36–45, Mühlenpfordstraße 23, 3106 Braunschweig, Deutschland, April 2009. Institut für Software Systems Engineering, Technische Universität Braunschweig,.

[EFT96] Heinz Dieter Ebbinghaus, Jörg Flum und Wolfgang Thomas: *Einführung in die mathematische Logik*. Spektrum-Akademischer Vlg, 4. Auflage, August 1996. ISBN 3827401305.

[EMF] EMFText: *http://www.emftext.org/*. Letzter Aufruf: 23. Juni 2011.

[FKF+10] Janet Feigenspan, Christian Kästner, Mathias Frisch, Raimund Dachselt und Sven Apel: *Visual Support for Understanding Product Lines*. In: *The 18th IEEE International Conference on Program Comprehension (ICPC 2010)*, Seiten 34–35, Braga, Minho, Portugal, Juni 2010. IEEE Computer Society. ISBN 978-0-7695-4113-6.

[FS00] Martin Fowler und Kendall Scott: *UML distilled - a brief guide to the Standard Object Modeling Language*. The Addison-Wesley object technology series. Addison-Wesley-Longman, 2. Auflage, 2000. ISBN 978-0-201-65783-8.

[GA02] Jeff Garland und Richard Anthony: *Large-Scale Software Architecture: A Practical Guide using UML*. Wiley, Dezember 2002. ISBN 978-0470848494.

[GBRW11] Helko Glathe, Margot Bittner, Mark Oliver Reiser und Matthias Weber: *Artefaktübergreifendes Varianten-Management: Werkzeuggestützte Erweiterung der herkömmlichen Produktlinienentwicklung*. Online Elektronik automotive, April 2011. `http://www.elektroniknet.de/`.

[GFd98] Martin L. Griss, John Favaro und Massimo d'Alessandro: *Integrating Feature Modeling with the RSEB*. In: *Proceedings of the 5th International Conference on Software Reuse*, ICSR '98, Seiten 76–85, Washington, DC, USA, 1998. IEEE Computer Society.

[GHJV10] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides: *Entwurfsmuster - Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, 6. Auflage, November 2010. ISBN 978-3827330437.

[GHK+07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel und Bernhard Rumpe: *View-Based Modeling of Function Nets*. In: Matthias Gehrke, Holger Giese und Joachim Stroop (Herausgeber): *Proceedings of the 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER4)*, Band 236, Seiten 40–45, Heinz Nixdorf Institut, Paderborn, Deutschland, Oktober 2007. HNI Verlagsschriftenreihe.

[GHK+08a] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt und Bernhard Rumpe: *Modelling Automotive Function Nets with Views for Features, Variants, and Modes*. In: Societe des Ingenieurs de l'Automobile (SIA) (Herausgeber): *4th International Congress Embedded Real Time Software (ERTS 2008)*, Toulouse, Frankreich, Januar 2008.

[GHK+08b] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt und Bernhard Rumpe: *View-Centric Modeling of Automotive Logical Architectures*. In: Holger Giese, Michaela Huhn, Ulrich Nickel und Bernhard Schätz (Herausgeber): *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV*, Band 2008-2 der Reihe *Informatik-Bericht*, Mühlenpfordstraße 23, 3106 Braunschweig, Deutschland, 2008. Institut für Programmierung und Reaktive Systeme, Technische Universität Braunschweig.

[GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell und Bernhard Rumpe: *Modeling Variants of Automotive Systems using Views*. In: Torsten Klein und Bernhard Rumpe (Herausgeber): *Tagungsband des Modellierungs-Workshops: Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen*, Band 2008-01 der Reihe *Informatik-Bericht*, Seiten 76–89, Carl-Friedrich-Gauß-Fakultät für Mathematik und Informatik, März 2008. Institut für Software Systems Engineering, Technische Universität Braunschweig.

[GR08] Martin Große-Rohde: *Architekturzentriertes Variantenmanagement für eingebettete Systeme: Ergebnisse des Projekts »Verteilte Entwicklung und Integration von Automotive-Produktlinien«*. ISST-Bericht 89/08, Fraunhofer Institut Software- und Systemtechnik (ISST), Mollstraße 1, 10178 Berlin, Deutschland, Oktober 2008. ISSN 0943-1624.

[Grä11] Erich Grädel: *Mathematische Logik*. Vorlesung, Lehr- und Forschungsgebiet Mathematische Grundlagen der Informatik, Informatik 7, RWTH Aachen, Sommersemester 2011. `http://www.logic.rwth-aachen.de/`.

[GREKM07] Martin Große-Rohde, Simon Eiuringer, Ekkart Kleinod und Stefan Mann: *Grobentwurf des VEIA-Referenzprozesses*. ISST-Bericht 80/07, Fraunhofer Institut Software- und Systemtechnik (ISST), Mollstraße 1, 10178 Berlin, Deutschland, Januar 2007. ISSN 0943-1624.

[Gro09] Richard C. Gronback: *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, März 2009. ISBN 978-0-321-53407-1.

[Gro10a] Object Management Group: *Object Constraint Language*. Version 2.2, Februar 2010. `http://www.omg.org/spec/OCL/2.2/`.

[Gro10b] Object Management Group: *Unified Modeling Language*. Version 2.3, Mai 2010. `http://www.omg.org/spec/UML/2.3/`.

[GW99] Bernhard Ganter und Rudolf Wille: *Formal Concept Analysis - Mathematical Foundations*. Springer, 1. Auflage, Januar 1999. ISBN 978-3540627715.

[HKW+06] Lothar Hotz, Thorsten Krebs, Katharina Wolter, Jos Nijhuis, Sybren Deelstra, Marco Sinnema und John MacGregor: *Configuration in Industrial Product Families: The ConIPF Methodology*. IOS Press, Inc., Juli 2006. ISBN 1-58603-641-6.

[Hol04a] Steve Holzner: *Eclipse*. O'Reilly Media, April 2004. ISBN 978-0-596-00641-9.

[Hol04b] Steve Holzner: *Eclipse Cookbook*. O'Reilly Media, Juni 2004. ISBN 978-0-596-00710-2.

[JDH09] Elmar Jürgens, Florian Deissenboeck und Benjamin Hummel: *CloneDetective - A Workbench for Clone Detection Research*. In: *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, Seiten 603–606, Vancouver, Canada, Mai 2009. IEEE. ISBN 978-1-4244-3452-7.

[JDH10] Elmar Jürgens, Florian Deissenboeck und Benjamin Hummel: *Code Similarities Beyond Copy & Paste*. In: Rafael Capilla, Rudolf Ferenc und Juan C. Dueñas (Herausgeber): *14th European Conference on Software Maintenance and Reengineering (CSMR 2010)*, Seiten 78–87, Madrid, Spain, März 2010. IEEE.

[JDHW09] Elmar Jürgens, Florian Deissenboeck, Benjamin Hummel und Stefan Wagner: *Do code clones matter?* In: *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, Seiten 485–495, Vancouver, Canada, Mai 2009. IEEE. ISBN 978-1-4244-3452-7.

[JGJ97] Ivar Jacobson, Martin L. Griss und Patrick Jonsson: *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley Professional, 1. Auflage, Juni 1997. ISBN-10 0201924765, ISBN-13 978-0201924763.

[KAK08] Christian Kästner, Sven Apel und Martin Kuhlemann: *Granularity in software product lines*. In: Wilhelm Schäfer, Matthew B. Dwyer und Volker Gruhn (Herausgeber): *30th International Conference on Software Engineering (ICSE 2008)*, Seiten 311–320, Leipzig, Germany, Mai 2008. ACM. ISBN 978-1-60558-079-1.

[Kap] Joshua Kaplan: *matlabcontrol - A Java API to interact with MATLAB*. Website. `http://code.google.com/p/matlabcontrol/`.

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak und A. Spencer Peterson: *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technischer Bericht CMU/SEI-90-TR-21, Carnegie-Mellon University Software Engineering Institute, Pittsburgh, Pennsylvania 15213, November 1990.

[Kle02] Stephen Cole Kleene: *Mathematical Logic*. Dover Publications, Dezember 2002. ISBN 978-0486425337.

[Kle06] Ekkart Kleinod: *Modellbasierte Systementwicklung in der Automobilindustrie: Das Moses Projekt*. ISST-Bericht 77/06, Fraunhofer Institut Software- und Systemtechnik (ISST), Mollstraße 1, 10178 Berlin, Deutschland, April 2006. ISSN 0943-1624.

[KM05] Mik Kersten und Gail C. Murphy: *Mylar: a degree-of-interest model for IDEs*. In: Mira Mezini und Peri L. Tarr (Herausgeber): *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD 2005)*, Seiten 159–168, Chicago, Illinois, USA, März 2005. ACM. ISBN 1-59593-042-6.

[KM06] Mik Kersten und Gail C. Murphy: *Using Task Context to Improve Programmer Productivity*. In: Michal Young und Premkumar T. Devanbu (Herausgeber): *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2006)*, Seiten 1–11, Portland, Oregon, USA, November 2006. ACM. ISBN 1-59593-468-5.

[KR88] Brian W. Kernighan und Dennis Ritchie: *The C Programming Language*. Prentice-Hall, 2. Auflage, 1988. ISBN 0-13-110370-9.

[Käs07] Christian Kästner: *CIDE: Decomposing Legacy Applications into Features*. In: *Proceedings of the 11th International Conference on Software Product Lines (SPLC)*, Seiten 149–150, Tokyo, Japan, September 2007. Kindai Kagaku Sha Co. Ltd. ISBN 978-4-7649-0342-5.

[KTA08] Christian Kästner, Salvador Trujillo und Sven Apel: *Visualizing Software Product Line Variabilities in Source Code*. In: Steffen Thiel und Klaus Pohl (Herausgeber): *Proceedings of the 12th International Conference*

on Software Product Lines (SPLC), Seiten 303–312, Limerick, Ireland, September 2008. Lero Int. Science Centre. ISBN 978-1-905952-06-9.

[MA09a] Cem Mengi und Ibrahim Armac: *Ein Klassifikationsansatz zur Variabilitätsmodellierung in E/E-Entwicklungsprozessen*. In: Jürgen Münch und Peter Liggesmeyer (Herausgeber): *Software Engineering 2009 - Workshopband, Fachtagung des GI-Fachbereichs Softwaretechnik*, Band 150 der Reihe *Lecture Notes in Informatics (LNI)*, Seiten 125–130, Kaiserslautern, Deutschland, März 2009. Gesellschaft für Informatik e.V. (GI). ISBN 978-3-88579-244-4.

[MA09b] Cem Mengi und Ibrahim Armac: *Functional Variant Modeling for Adaptable Functional Networks*. In: David Benavides, Andreas Metzger und Ulrich Eisenecker (Herausgeber): *VaMoS 2009: Third International Workshop on Variability Modelling of Software-Intensive Systems*, Band 29 der Reihe *ICB Research Report*, Seiten 83–92, Sevilla, Spain, Januar 2009. Universität Duisburg-Essen. ISSN 1860-2770.

[Mar05] André Marburger: *Reverse Engineering of Complex Legacy Telecommunication Systems*. Dissertation, Rheinisch-Westfälische Technische Hochschule (RWTH) Aachen, Juli 2005. ISBN 978-3-8322-4154-4.

[Men08] Cem Mengi: *Variant Configuration of Automotive System Architectures*. In: *Proceedings des gemeinsamen Workshops der Graduiertenkollegs 2008*, Seiten 16–17, Dagstuhl, Deutschland, Mai 2008. GITO-Verlag. ISBN 978-3-940019-39-4.

[Men09] Cem Mengi: *Integrated Variability Modeling for Automotive Software Systems*. In: Artin Avanes, Dirk Fahland, Joanna Geibig, Siamak Haschemi, Sebastian Heglmeier, Daniel A. Sadile, Falko Theisselmann, Guido Wachsmuth und Stephan Weißleder (Herausgeber): *Proceedings des gemeinsamen Workshops der Graduiertenkollegs 2009*, Seiten 9–10, Dagstuhl, Deutschland, Juni 2009. GITO-Verlag. ISBN 978-3-940019-73-8.

[Men10] Cem Mengi: *Integrated Design and Configuration of Versatile Software Documents in Automotive Software Engineering*. In: Kai Bollue, Dominique Gückel, Ulrich Loup, Jacob Spönemann und Melanie Winkler (Herausgeber): *Proceedings of the Joint Workshop of the German Research Training Groups in Computer Science*, Seite 21, Dagstuhl, Deutschland, Mai 2010. Verlagshaus Mainz. ISBN 3-86130-146-6.

[Men11] Cem Mengi: *Konzeption variantenreicher Funktionsnetze für Simulink-Modelle*. Interner Bericht SE-26-FuNets-10-07-14, Lehrstuhl für Informatik 3 (Software Engineering), RWTH Aachen, Februar 2011.

[MFZA09] Cem Mengi, Christian Fuß, Ruben Zimmermann und Ismet Aktas: *Model-driven Support for Source Code Variability in Automotive Software Engineering*. In: John D. McGregor und Jaejoon Lee (Herausgeber): *Proceedings of the 13th International Software Product Line Conference (SPLC 2009): First International Workshop on Model-Driven Approaches in Software Product Line Engineering (MAPLE 2009)*, Band 2, Seiten 68–74, San Francisco, CA, USA, August 2009. Carnegie Mellon University.

[MKRC05] Gail C. Murphy, Mik Kersten, Martin P. Robillard und Davor Cubranic: *The Emergent Structure of Development Tasks*. In: Andrew P. Black (Herausgeber): *In Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP 2005)*, Band 3586 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 33–48, Glasgow, UK, Juli 2005. Springer. ISBN 3-540-27992-X.

[MN12] Cem Mengi und Manfred Nagl: *Refactoring of Automotive Models to Handle the Variant Problem*. Softwaretechnik-Trends, 32(2):2, Mai 2012. ISSN 0720-8928.

[MnBRB10] Cem Mengi, Önder Babur, Holger Rendel und Christian Berger: *Model-driven Configuration of Function Net Families in Automotive Software Engineering*. In: Goetz Botterweck, Patrick Heymans, Itay Maman, Andreas Pleuss und Julia Rubin (Herausgeber): *Proceedings of the 2nd International Workshop on Model-driven Product Line Engineering (MDPLE 2010)*, Band 625, Seiten 49–60, Paris, Frankreich, Juni 2010. CEUR Workshop Proceedings. ISSN 1613-0073.

[Mos09] Christof Mosler: *Graphbasiertes Reengineering von Telekommunikationssystemen*. Dissertation, Rheinisch-Westfälische Technische Hochschule (RWTH) Aachen, Juni 2009. ISBN 978-3-8322-8240-0.

[MPBK11] Daniel Merschen, Andreas Polzer, Goetz Botterweck und Stefan Kowalewski: *Experiences of applying model-based analysis to support the development of automotive software product lines*. In: Patrick Heymans, Krzysztof Czarnecki und Ulrich W. Eisenecker (Herausgeber): *Fifth International Workshop on Variability Modelling of Software-Intensive Systems (VaMos 2011)*, ACM International Conference Proceedings Series, Seiten 141–150, Namur, Belgien, Januar 2011. ACM. ISBN 978-1-4503-0570-9.

[MPF09] Cem Mengi, Antonio Navarro Perez und Christian Fuß: *Modellierung variantenreicher Funktionsnetze im Automotive Software Engineering*. In: Stefan Fischer, Erik Maehle und Rüdiger Reischuk (Herausgeber): *Informatik 2009: Im Focus das Leben, Beiträge der 39. Jahrestagung der Gesellschaft für Informatik e.V. (GI)*, Band 154 der Reihe *Lecture Notes in Informatics (LNI)*, Seiten 2689–2701, Lübeck, Deutschland,

September 2009. Gesellschaft für Informatik e.V. (GI). ISBN 978-3-88579-248-2.

[MR09] Stefan Mann und Georg Rock: *Dealing with Variability in Architecture Descriptions to Support Automotive Product Lines*. In: David Benavides, Andreas Metzger und Ulrich W. Eisenecker (Herausgeber): *Third International Workshop on Variability Modelling of Software-Intensive Systems 2009 (VaMoS 2009)*, Band 29 der Reihe *ICB Research Report*, Seiten 111–120, Seville, Spain, Januar 2009. Institut für Informatik und Wirtschaftsinformatik (ICB), Universität Duisburg-Essen. ISSN 1860-2770.

[Nag90] Manfred Nagl: *Softwaretechnik: Methodisches Programmieren im Großen*. Springer-Verlag Berlin Heidelberg, 1990. ISBN 3-540-52705-2.

[Nag03] Manfred Nagl: *Softwaretechnik mit Ada 95: Entwicklung großer Systeme*. Vieweg+Teubner, 2. Auflage, Mai 2003. ISBN 978-3528155834.

[nB10] Önder Babur: *Model-driven Configuration of Function Net Families in Automotive Software Engineering*. Masterarbeit, RWTH Aachen, November 2010. Betreuer: Cem Mengi.

[PBKS07] Alexander Pretschner, Manfred Broy, Ingolf H. Krüger und Thomas Stauner: *Software Engineering for Automotive Systems: A Roadmap*. In: Lionel C. Briand und Alexander L. Wolf (Herausgeber): *Workshop on the Future of Software Engineering (FOSE 2007)*, Seiten 55–71, Minneapolis, MN, USA, Mai 2007.

[PBKW09] Andreas Polzer, Goetz Botterweck, Stefan Kowalewski und Iris Wangerin: *Variabilität im modelbasierten Engineering von eingebetteten Systemen*. In: Stefan Fischer, Erik Maehle und Rüdiger Reischuk (Herausgeber): *Informatik 2009: Im Focus das Leben, Beiträge der 39. Jahrestagung der Gesellschaft für Informatik e.V. (GI)*, Band 154 der Reihe *Lecture Notes in Informatics (LNI)*, Seiten 2702–2710, Lübeck, Deutschland, Oktober 2009. Gesellschaft für Informatik (GI). ISBN 978-3-88579-248-2.

[PBvdL05] Klaus Pohl, Günter Böckle und Frank J. van der Linden: *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag Berlin Heidelberg, Secaucus, NJ, USA, 2005. ISBN 3-540-24372-0.

[PC05a] Robert R. Painter und David Coppit: *A Model for Software Plans*. ACM SIGSOFT Software Engineering Notes, 30(4):1–5, Mai 2005. ISSN 0163-5948.

[PC05b] Robert R. Painter und David Coppit: *A Model for Software Plans*. In: *Proceedings of the 2005 Workshop on Modeling and Analysis of Concerns in Software (MACS 05)*, Seiten 1–5, New York, NY, USA, 2005. ACM. ISBN 1-59593-119-8.

[Per09] Antonio Navarro Perez: *Ein Prototyp zur Modellierung von Funktionsnetz-Familien im Automotive Software Engineering*. Diplomarbeit, RWTH Aachen, September 2009. Betreuer: Cem Mengi.

[PKB09] Andreas Polzer, Stefan Kowalewski und Goetz Botterweck: *Applying software product line techniques in model-based embedded systems engineering*. In: *ICSE 2009 Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES 2009)*, Seiten 2–10, Vancouver, Canada, Mai 2009. IEEE Computer Society. ISBN 978-1-4244-3721-4.

[Plu03] Joe Pluta: *Eclipse: Step by Step - A Practical Guide to Becoming Proficient in Eclipse*. MC Press, Juni 2003. ISBN 978-1-58347-044-2.

[PMB+12] Andreas Polzer, Daniel Merschen, Goetz Botterweck, Andreas Pleuss, Jacques Thomas, Bernd Hedenetz und Stefan Kowalewski: *Managing complexity and variability of a model-based embedded software product line*. Innovations in Systems and Software Engineering (ISSE), 8(1):35–49, März 2012.

[Pog10] Maxim Pogrebinski: *Konzeption, Implementierung und Integration von Constraints in ein Werkzeug für variantenreiche Funktionsnetze*. Bachelorarbeit, RWTH Aachen, Oktober 2010. Betreuer: Cem Mengi.

[Poj11] Jan Pojer: *Design and Application of Object-Oriented Domain Knowledge for Function Nets*. Masterarbeit, RWTH Aachen, September 2011. Betreuer: Cem Mengi.

[Por04a] OSEK VDX Portal: *OSEK/VDX Communication*. Version 3.0.3, Juli 2004. `http://portal.osek-vdx.org/`.

[Por04b] OSEK VDX Portal: *OSEK/VDX Network Management - Concept and Application Programming Interface*. Version 2.5.3, Juli 2004. `http://portal.osek-vdx.org/`.

[Por05] OSEK VDX Portal: *OSEK/VDX Operating System*. Version 2.2.3, Februar 2005. `http://portal.osek-vdx.org/`.

[Pro11] Eclipse Project: *Mylyn*. Website, August 2011. `http://www.eclipse.org/mylyn/`.

[psG03] pure-systems GmbH: *Variant Management with pure::variants*. Technical White Paper, pure-systems GmbH, Agnetenstr. 14, 39106 Magdeburg, Deutschland, 2003. `http://www.pure-systems.com`.

[RM02] Martin P. Robillard und Gail C. Murphy: *Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies*. In: *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2002)*, Seiten 406–416, Orlando, Florida, USA, Mai 2002. ACM.

[RM03] Martin P. Robillard und Gail C. Murphy: *FEAT. A Tool for Locating, Describing, and Analyzing Concerns in Source Code*. In: *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, Seiten 822–823, Portland, Oregon, USA, Mai 2003. IEEE Computer Society.

[RN03] Stuart Russell und Peter Norvig: *Artificial Intelligence - A Modern Approach*. Prentice Hall, 2. Auflage, 2003. ISBN 0-13-080302-2.

[Rob02] Martin P. Robillard: *A Representation for Describing and Analyzing Concerns in Source Code*. In: *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2002)*, Seiten 721–722, Orlando, Florida, USA, Mai 2002. ACM.

[RPK10a] Uwe Ryssel, Joern Ploennigs und Klaus Kabitzsch: *Automatic library migration for the generation of hardware-in-the-loop models*. Science of Computer Programming, Juni 2010. ISSN 0167-6423.

[RPK10b] Uwe Ryssel, Joern Ploennigs und Klaus Kabitzsch: *Automatic Variation-Point Identification in Function-Block-Based Models*. In: Eelco Visser und Jaakko Järvi (Herausgeber): *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering (GPCE 2010)*, Seiten 23–32, Eindhoven, Niederlande, Oktober 2010. ACM. ISBN 978-1-4503-0154-1.

[RRE91] James Rumbaugh, Jim Rumbaugh und Frederick Eddy: *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[Rum04] Bernhard Rumpe: *Agile Modellierung mit UML : Codegenerierung, Testfälle, Refactoring*. Springer, 2. Auflage, August 2004. ISBN 978-3540209058.

[Rum11] Bernhard Rumpe: *Modellierung mit UML*. Xpert.press. Springer, 2. Auflage, September 2011. ISBN 978-3-642-22412-6.

[SAV06a] BMW X5 SAV: *E70 Car Access System (CAS 3)*, Oktober 2006. `http://www.xoutpost.com/bmw-sav-forums/x5-e70-forum/`.

[SAV06b] BMW X5 SAV: *E70 Central Locking*, Oktober 2006. `http://www.xoutpost.com/bmw-sav-forums/x5-e70-forum/`.

[SAV06c] BMW X5 SAV: *E70 Comfort Access*, Oktober 2006. `http://www.xoutpost.com/bmw-sav-forums/x5-e70-forum/`.

[SAV06d] BMW X5 SAV: *E70 Exterior Lighting*, Oktober 2006. http://www.xoutpost.com/bmw-sav-forums/x5-e70-forum/.

[SAV06e] BMW X5 SAV: *E70 Information and Communication Technology (IKT)*, Oktober 2006. http://www.xoutpost.com/bmw-sav-forums/x5-e70-forum/.

[SAV06f] BMW X5 SAV: *E70 Interior Lighting*, Oktober 2006. http://www.xoutpost.com/bmw-sav-forums/x5-e70-forum/.

[SBPM08] Dave Steinberg, Frank Budinsky, Marcelo Paternostro und Ed Merks: *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, Dezember 2008. ISBN 978-0-321-33188-5.

[Sch05] S. Schlott: *Wahnsinn mit Methode: Zögerlicher Kampf gegen Variantenvielfalt*. AUTOMOBIL PRODUKTION, 1:38–42, Januar 2005.

[Sch10] Carsten Schmidt: *Modeling Function Variants in TargetLink*. TargetLink Application Note, dSPACE GmbH, Rathenaustraße 26, 33102 Paderborn, Deutschland, Juni 2010. http://www.dspace.de/.

[SD07] Marco Sinnema und Sybren Deelstra: *Classifying variability modeling techniques*. Information & Software Technology, 49(7):717–739, Juli 2007. ISSN 0950-5849.

[SDNB04] Marco Sinnema, Sybren Deelstra, Jos Nijhuis und Jan Bosch: *COVAMOF: A Framework for Modeling Variability in Software Product Families*. In: Robert L. Nord (Herausgeber): *Proceedings of the Third Software Product Line Conference (SPLC 2004)*, Band 3154 der Reihe *Lecture Notes in Computer Science*, Seiten 197–213, Boston, MA, USA, September 2004. Springer.

[SNS02] Patrik Simons, Ilkka Niemelá und Timo Soininen: *Extending and implementing the stable model semantics*. Artificial Intelligence, 138(1-2):181–234, Juni 2002. ISSN 0004-3702.

[STH11] Lambert M Surhone, Mariam T. Tennoe und Susan F. Henssonow: *Mylyn - Eclipse (software), Task Management, Task-Focused Interface*. Vdm Verlag Dr. Mller Ag & Co. Kg, 2011. ISBN 978-6132067371.

[SVEH07] Tom Stahl, Markus Völter, Sven Efftinge und Arno Haase: *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. dPunkt, 2. Auflage, Mai 2007. ISBN 978-3-89864-448-8.

[SZ06] Jörg Schäuffele und Thomas Zurawka: *Automotive Software Engineering - Grundlagen, Prozesse, Methoden und Werkzeuge effizient einsetzen*. Friedr. Vieweg & Sohn Verlag, GWV Fachverlage GmbH, Wiesbaden, 3. Auflage, März 2006. ISBN 978-3-8348-0051-0.

[Val04] Carlos Valcarcel: *Eclipse 3.0 Kick Start*. Sams, September 2004. ISBN 978-0672326103.

[vdB05] Michael von der Beeck: *Function Net Modeling with UML-RT: Experiences from an Automotive Project at BMW Group*. In: Nuno Nunes, Bran Selic, Alberto Rodrigues da Silva und Ambrosio Toval Alvarez (Herausgeber): *UML Modeling Languages and Applications*, Band 3297 der Reihe *Lecture Notes in Computer Science*, Seiten 94–104. Springer Berlin / Heidelberg, 2005. ISBN 3-540-25081-6.

[vdB07] Michael von der Beeck: *Development of logical and technical architectures for automotive systems*. Software and Systems Modeling, 6(2):205–219, Springer Berlin / Heidelberg, 2007. ISSN 1619-1366.

[vdLSR07] Frank J. van der Linden, Klaus Schmid und Eelco Rommes: *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007. ISBN 3540714367.

[vdML04] Thomas von der Maßen und Horst Lichter: *RequiLine: A Requirements Engineering Tool for Software Product Lines*. In: Frank van der Linden (Herausgeber): *Software Product-Family Engineering, 5th International Workshop (PFE 2003)*, Band 3014 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 168–180, Siena, Italien, November 2004. Springer. ISBN 3-540-21941-2.

[VM09] Das V-Modell: *V-Modell XT Gesamt*. Version 1.3, Februar 2009. `http://v-modell.iabg.de/`.

[W3C10] World Wide Web Consortium W3C: *XML Path Language (XPath) 2.0 (Second Edition)*. Version 2.0, Dezember 2010. `http://www.w3.org/TR/xpath20/`.

[Weba] BMW Website: *BMW Konfigurator*. `http://www.bmw.de/`.

[Webb] IBM Website: *Rational DOORS*. `http://www-142.ibm.com/software/products/de/de/ratidoor/`.

[Webc] MathWorks Website: *Simulink - Simulation and Model-Based Design*. `http://www.mathworks.com/products/simulink/`.

[Wei81] Mark Weiser: *Program Slicing*. In: *Proceedings of the 5th International Conference on Software Engineering (ICSE 1981)*, Seiten 439–449, San Diego, CA, USA, März 1981. IEEE Computer Society.

[Wei84] Mark Weiser: *Program Slicing*. IEEE Transactions on Software Engineering (TSE), 10(4):352–357, Januar 1984.

[Wei08] Jens Weiland: *Variantenkonfiguration eingebetter Automotive Software mit Simulink*. Dissertation, Wirtschaftswissenschaftliche Fakultät, Universität Leipzig, Juli 2008.

[Wes91] Bernhard Westfechtel: *Revisions- und Konsistenzkontrolle in einer integrierten Softwareentwicklungsumgebung*, Band 280 der Reihe *Informatik-Fachberichte*. Springer, 1991. ISBN 3-540-54432-1.

[WH06] Henning Wallentowitz und Konrad Reif (Hrsg.): *Handbuch Kraftfahrzeugelektronik - Grundlagen, Komponenten, Systeme, Anwendungen*. Friedr. Vieweg & Sohn Verlag, GWV Fachverlage GmbH, Wiesbaden, September 2006. ISBN 3-528-03971-X.

[WR05] Jens Weiland und Ernst Richter: *Konfigurationsmanagement variantenreicher Simulink-Modelle*. In: Armin B. Cremers, Rainer Manthey, Peter Martini und Volker Steinhage (Herausgeber): *INFORMATIK 2005 - Informatik LIVE! Band 2, Beiträge der 35. Jahrestagung der Gesellschaft für Informatik e.V. (GI)*, Band 68 der Reihe *Lecture Notes in Informatics (LNI)*, Seiten 176–180, Bonn, Deutschland, September 2005. Gesellschaft für Informatik e.V. (GI). ISBN 3-88579-397-0.

[zA11] Özgür Akcasoy: *Solution Patterns for Variability in Model-Based Product Lines*. Masterarbeit, Rheinisch-Westfälische Technische Hochschule (RWTH) Aachen, Mai 2011. Betreuer: Peter Manhart und Cem Mengi.

[Zim09] Ruben Zimmermann: *Werkzeuggestützte Variantenprogrammierung im Automobilbereich*. Diplomarbeit, Rheinisch-Westfälische Technische Hochschule (RWTH) Aachen, 2009. Betreuer: Christian Fuß und Cem Mengi.

Abkürzungsverzeichnis

ACSM	Advanced Crash Safety Management
AJDT	AspectJ Development Tooling
AL	Aktive Lenkung
API	Application Programming Interface
ARS	Active Roll Stabilization
ASP	Außenspiegel
AutoMoDe	Automotive Model-based Development
AUTOSAR	Automotive Open System Architecture
ANTLR	Another Tool for Language Recognition
BMW	Bayerische Motoren Werke
BSD	Bit-Serielle Datenschnittstelle
CA	Comfort Access
CAS	Car Access System
CBFM	Cardinality-Based Feature Model
CAN	Controller Area Network
CCC	Car Communication Computer
CDC	Compact Disk Changer
CDT	C/C++ Development Tooling
CFDL	CONSUL Family Description Language
CHAMP	Central Head unit and Multimedia Platform
CID	Central Information Display
CIDE	Colored Integrated Development Environment
CON	Controller
ConIPF	Configuration of Industrial Product Families
CONSUL	Configuration Support Library
COVAMOF	ConIPF Variability Modeling Framework
Crash-Sig	Crash-Signal
CVV	COVAMOF Varaibility View
CVVL	CVV Language
D-CAN	Diagnose-CAN
DME	Digitale Motorelektronik
DoI	Degree-of-Interest
DSC	Dynamic Stability Control
DSC-SEN	DSC-Sensor
DVD	Digital Video Disc

EBNF Erweiterte Backus-Naur-Form

EDC SHL Electronic Damper Control, Satellit hinten links

EDC SHR Electronic Damper Control, Satellit hinten rechts

EDC SVL Electronic Damper Control, Satellit vorne links

EDC SVR Electronic Damper Control, Satellit vorne rechts

EEF Extended Editing Framework

E/E Elektrik/Elektronik

EGS Elektronische Getriebesteuerung

EHC Electronic Height Control

EKP Elektronische Kraftstoffpumpe

EMF Elektro-mechanische Feststellbremse

EMF Eclipse Modeling Framework

EMP Eclipse Modeling Project

EWS Elektronische Wegfahrsperre

FAA Functional Analysis Architecture

F-CAN Fahrwerk-CAN

FD Fond Display

FDA Functional Design Architecture

FEAT Feature Exploration and Analysis Tool

FKA Fond Klimaautomatik

FLA Fernlichtassistent

FODA Feature-Oriented Domain Analysis

FRM Fußraummodul

FZD Funktionszentrum Dach

GEF Graphical Editing Framework

GMF Graphical Modeling Framework

GPS Global Positioning System

GWS Gangwahlschaltung

HB3SR Heizung/Belüftung 3. Sitzreihe

HiFi HiFi Verstärker

HKL Heckklappenlift

HUD Head-Up Display

IBOC In-Band On-Channel (HD Radio)

IBS Intelligenter Batteriesensor

IHKA Intergrierte Heiz- und Klimaautomatik

ISST Institut Software- und Systemtechnik

JB Junction Box

JDT Java Development Tooling

JMI Java Matlab Interface

K-Bus Karosserie-Bus

K-CAN Karosserie-CAN

Kfz Kraftfahrzeug

Kombi Kombinationsinstrument

LA	Logical Architecture
LIN	Local Interconnect Network
LIN-Bus	Local Interconnect Network Bus
LoCAN	Local CAN
M-ASK	Multi-Audio System Kontroller
MB	Mega Byte
MOSES	Modellbasierte Systementwicklung
MOST	Media Oriented Systems Transport
MOST WUP	MOST Wake-Up
OA	Operational Architecture
OC3	Seat Occupancy Sensor
OCL	Object Constraint Language
OSEK	Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug
OVM	Orthogonales Variabilitätsmodell
PDC	Park Distance Control
PKW	Personenkraftwagen
PT-CAN	Powertrain-CAN
QLT	Quality, Level, Temperature Sensor für Öl-Status
RDC	Reifendruck Control
RDC-SEN	RDC-Sensor
RFK	Rückfahrkamera
RLSS	Regen-/Fahrlicht Solar Sensor
RSE	Rücksitz-Entertainment
RSEB	Reuse-Driven Software Engineering Business
SA	Sonderausstattung
SBFA	Schalterblock Fahrer
SDARS	Satellite Digital Audio Radio Services
SG	Steuergerät
SINE	Sirene und Neigungssensor
SMBF	Sitzmodul Beifahrer
SMC	Schrittmotor Controller
SMFA	Sitzmodul Fahrer
SVBF	Sitzverstellung Beifahrer
SVFA	Sitzverstellung Fahrer
SysML	Systems Modeling Language
SZL	Schaltzentrum Lenksäule
TA	Technical Architecture
TAGE	Türaußengriffelektronik
TCU	Telematics Control Unit
TONS	Thermalöl Niveau Sensor
TOP-HIFI	Top HiFi Verstärker
TU	Technische Universität
UML	Unified Modeling Language
VEIA	Verteilte Entwicklung und Integration von Automotive-Produktlinien
VSL	Variability Specification Language

VDM	Vertikaldynamik Management
VDX	Vehicle Distributed eXecutive
VGSG	Verteilergetriebe Steuergerät
VVT	Variabler Ventiltrieb
VW	Volkswagen
WCRL	Weight Constraint Rule Language
WUP	Wake-Up
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XPath	XML Path Language
XSLT	Extensible Stylesheet Language Transformation

Lebenslauf

Cem Mengi

Geburtsdatum:	31. Mai 1980
Geburtsort:	Aachen
Geburtsname:	Mengi
Staatsangehörigkeit:	deutsch
seit März 2007	Wissenschaftlicher Angestellter am Lehrstuhl für Informatik 3 der RWTH Aachen; Beginn der Promotion
Februar 2007	Studienabschluss als Diplom-Informatiker (Dipl.-Inform.)
1999 – 2007	Informatikstudium an der RWTH Aachen
1999	Abitur am Rhein-Maas-Gymnasium Aachen

Related Interesting Work from the SE Group, RWTH Aachen

Agile Model Based Software Engineering

Agility and modeling in the same project? This question was raised in [Rum04]: "Using an executable, yet abstract and multi-view modeling language for modeling, designing and programming still allows to use an agile development process." Modeling will be used in development projects much more, if the benefits become evident early, e.g with executable UML [Rum02] and tests [Rum03]. In [GKRS06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum12] and [Rum11], the UML/P, a variant of the UML especially designed for programming, refactoring and evolution, is defined. The language workbench MontiCore [GKR+06] is used to realize the UML/P [Sch12]. Links to further research, e.g., include a general discussion of how to manage and evolve models [LRSS10], a precise definition for model composition as well as model languages [HKR+09] and refactoring in various modeling and programming languages [PR03]. In [FHR08] we describe a set of general requirements for model quality. Finally [KRV06] discusses the additional roles and activities necessary in a DSL-based software development project.

Generative Software Engineering

The UML/P language family [Rum12, Rum11] is a simplified and semantically sound derivate of the UML designed for product and test code generation. [Sch12] describes a flexible generator for the UML/P based on the MontiCore language workbench [KRV10, GKR+06]. In [KRV06], we discuss additional roles necessary in a model-based software development project. In [GKRS06] we discuss mechanisms to keep generated and handwritten code separated. In [Wei12] we show how this looks like and how to systematically derive a transformation language in concrete syntax. To understand the implications of executability for UML, we discuss needs and advantages of executable modeling with UML in agile projects in [Rum04], how to apply UML for testing in [Rum03] and the advantages and perils of using modeling languages for programming in [Rum02].

Unified Modeling Language (UML)

Many of our contributions build on UML/P described in the two books [Rum11] and [Rum12] are implemented in [Sch12]. Semantic variation points of the UML are discussed in [GR11]. We discuss formal semantics for UML [BHP+98] and describe UML semantics using the "System Model" [BCGR09a], [BCGR09b], [BCR07b] and [BCR07a]. Semantic variation points have, e.g., been applied to define class diagram semantics [CGR08]. A precisely defined semantics for variations is applied, when checking variants of class diagrams [MRR11c] and objects diagrams [MRR11d] or the consistency of both kinds of diagrams [MRR11e]. We also apply these concepts to activity diagrams (ADs) [MRR11b] which allows us to check for semantic differences of activity diagrams [MRR11a]. We also discuss how to ensure and identify model quality [FHR08], how models, views and the system under development correlate to each other [BGH+98] and how to use modeling in agile development projects [Rum04], [Rum02] The question how to adapt and extend the UML in discussed in [PFR02] on product line annotations for UML and to more general discussions and insights on how to use meta-modeling for defining and adapting the UML [EFLR99], [SRVK10].

Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use, but need appropriate tooling. The MontiCore language workbench [GKR+06], [KRV10], [Kra10] describes an integrated abstract and concrete syntax format [KRV07b] for easy development. New languages and tools

can be defined in modular forms [KRV08, Völ11] and can, thus, easily be reused. [Wei12] presents a tool that allows to create transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11]. A successful application has been carried out in the Air Traffic Management domain [ZPK+11]. Based on the concepts described above, meta modeling, model analyses and model evolution have been examined in [LRSS10] and [SRVK10]. DSL quality [FHR08], instructions for defining views [GHK+07], guidelines to define DSLs [KKP+09] and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

Modeling Software Architecture & the MontiArc Tool

Distributed interactive systems communicate via messages on a bus, discrete event signals, streams of telephone or video data, method invocation, or data structures passed between software services. We use streams, statemachines and components [BR07] as well as expressive forms of composition and refinement [PR99] for semantics. Furthermore, we built a concrete tooling infrastructure called MontiArc [HRR12] for architecture design and extensions for states [RRW13]. MontiArc was extended to describe variability [HRR+11] using deltas [HRRS11] and evolution on deltas [HRRS12]. [GHK+07] and [GHK+08] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants. Co-evolution of architecture is discussed in [MMR10] and a modeling technique to describe dynamic architectures is shown in [HRR98].

Compositionality & Modularity of Models

[HKR+09] motivates the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07] and algebraically underpinned in [HKR+07]. Semantic and methodical aspects of model composition [KRV08] led to the language workbench MontiCore [KRV10] that can even develop modeling tools in a compositional form. A set of DSL design guidelines incorporates reuse through this form of composition [KKP+09]. [Völ11] examines the composition of context conditions respectively the underlying infrastructure of the symbol table. Modular editor generation is discussed in [KRV07a].

Semantics of Modeling Languages

The meaning of semantics and its principles like underspecification, language precision and detailedness is discussed in [HR04]. We defined a semantic domain called “System Model” by using mathematical theory. [RKB95, BHP+98] and [GKR96, KRB96]. An extended version especially suited for the UML is given in [BCGR09b] and in [BCGR09a] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied on class diagrams in [CGR08]. [MRR11a, MRR11b] encode a part of the semantics to handle semantic differences of activity diagrams and [MRR11e] compares class and object diagrams with regard to their semantics. In [BR07], a simplified mathematical model for distributed systems based on black-box behaviors of components is defined. Meta-modeling semantics is discussed in [EFLR99]. [BGH+97] discusses potential modeling languages for the description of an exemplary object interaction, today called sequence diagram. [BGH+98] discusses the relationships between a system, a view and a complete model in the context of the UML. [GR11] and [CGR09] discuss general requirements for a framework to describe semantic and syntactic variations of a modeling language. We apply these on class and object diagrams in [MRR11e] as well as activity diagrams in [GRR10]. [Rum12] embodies the semantics in a variety of code and test case generation, refactoring and evolution techniques. [LRSS10] discusses evolution and related issues in greater detail.

Evolution & Transformation of Models

Models are the central artifact in model driven development, but as code they are not initially correct and need to be changed, evolved and maintained over time. Model transformation is therefore essential to effectively deal with models. Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04], refinement [PR99, KPR97, PR94], refactoring [Rum12, PR03], translating models from one language into another [MRR11c, Rum12] and systematic model transformation language development [Wei12]. [Rum04] describes how comprehensible sets of such transformations support software development, maintenance and [LRSS10] technologies for evolving models within a language and across languages and linking architecture descriptions to their implementation [MMR10]. Automaton refinement is discussed in [PR94, KPR97], refining pipe-and-filter architectures is explained in [PR99]. Refactorings of models are important for model driven engineering as discussed in [PR03, Rum12]. Translation between languages, e.g., from class diagrams into Alloy [MRR11c] allows for comparing class diagrams on a semantic level.

Variability & Software Product Lines (SPL)

Many products exist in various variants, for example cars or mobile phones, where one manufacturer develops several products with many similarities but also many variations. Variants are managed in a Software Product Line (SPL) that captures the commonalities as well as the differences. Feature diagrams describe variability in a top down fashion, e.g., in the automotive domain [GHK+08] using 150% models. Reducing overhead and associated costs is discussed in [GRJA12]. Delta modeling is a bottom up technique starting with a small, but complete base variant. Features are added (that sometimes also modify the core). A set of applicable deltas configures a system variant. We discuss the application of this technique to Delta-MontiArc [HRR+11, HRR+11] and to Delta-Simulink [HKM+13]. Deltas can not only describe spacial variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK+13] describes an approach to systematically derive delta languages. We also apply variability to modeling languages in order to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02]. And we specified a systematic way to define variants of modeling languages [CGR09] and applied this as a semantic language refinement on Statecharts in [GR11].

Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12] are complex, distributed systems which control physical entities. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12] and autonomous driving [BR12a] to processes and tools to improve the development as well as the product itself [BBR07]. In the aviation domain, a modeling language for uncertainty and safety events was developed, which is of interest for the European airspace [ZPK+11]. A component and connector architecture description language suitable for the specific challenges in robotics is discussed in [RRW13]. Monitoring for smart and energy efficient buildings is developed as Energy Navigator toolset [KPR12, FPPR12, KLPR12].

State Based Modeling (Automata)

Today, many computer science theories are based on state machines in various forms including Petri nets or temporal logics. Software engineering is particularly interested in using state machines for modeling systems. Our contributions to state based modeling can currently be split into three parts: (1) understanding how to model object-oriented and distributed software using statemachines resp. Statecharts

[GKR96, BCR07b, BCGR09b, BCGR09a], (2) understanding the refinement [PR94, RK96, Rum96] and composition [GR95] of statemachines, and (3) applying statemachines for modeling systems. In [Rum96] constructive transformation rules for refining automata behavior are given and proven correct. This theory is applied to features in [KPR97]. Statemachines are embedded in the composition and behavioral specifications concepts of Focus [BR07]. We apply these techniques, e.g., in MontiArcAutomaton [THR+13] as well as in building management systems [FLP+11].

Robotics

Robotics can be considered a special field within Cyber-Physical Systems which is defined by an inherent heterogeneity of involved domains, relevant platforms, and challenges. The engineering of robotics applications requires composition and interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers broad propagation of robotics applications. The MontiArcAutomaton language [RRW12] extends ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13] that perfectly fits Robotic architectural modelling. The LightRocks [THR+13] framework allows robotics experts and laymen to model robotic assembly tasks.

Automotive, Autonomic Driving & Intelligent Driver Assistance

Introducing and connecting sophisticated driver assistance, infotainment and communication systems as well as advanced active and passive safety-systems result in complex embedded systems. As these feature-driven subsystems may be arbitrarily combined by the customer, a huge amount of distinct variants needs to be managed, developed and tested. A consistent requirements management that connects requirements with features in all phases of the development for the automotive domain is described in [GRJA12]. The conceptual gap between requirements and the logical architecture of a car is closed in [GHK+07, GHK+08]. [HKM+13] describes a tool for delta modeling for Simulink [HKM+13]. [HRRW12] discusses means to extract a well-defined Software Product Line from a set of copy and paste variants. Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12a, BR12b], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in development and evolution of autonomous car functionality, and thus, enables us to develop software in an agile way [BR12a]. [MMR10] gives an overview of the current state-of-the-art in development and evolution on a more general level by considering any kind of critical system that relies on architectural descriptions. As tooling infrastructure, the SSElab storage, versioning and management services [HKR12] are essential for many projects.

Energy Management

In the past years, it became more and more evident that saving energy and reducing CO2 emissions is an important challenge. Thus, energy management in buildings as well as in neighbourhoods becomes equally important to efficiently use the generated energy. Within several research projects, we developed methodologies and solutions for integrating heterogeneous systems at different scales. During the design phase, the Energy Navigators Active Functional Specification (AFS) [FPPR12, KPR12] is used for technical specification of building services already. We adapted the well-known concept of statemachines to be able to describe different states of a facility and to validate it against the monitored values [FLP+11]. We show how our data model, the constraint rules and the evaluation approach to compare sensor data can be applied [KLPR12].

Cloud Computing & Enterprise Information Systems

The paradigm of Cloud Computing is arising out of a convergence of existing technologies for web-based application and service architectures with high complexity, criticality and new application domains. It promises to enable new business models, to lower the barrier for web-based innovations and to increase the efficiency and cost-effectiveness of web development. Application classes like Cyber-Physical Systems [KRS12], Big Data, App and Service Ecosystems bring attention to aspects like responsiveness, privacy and open platforms. Regardless of the application domain, developers of such systems are in need for robust methods and efficient, easy-to-use languages and tools. We tackle these challenges by perusing a model-based, generative approach [PR13]. The core of this approach are different modeling languages that describe different aspects of a cloud-based system in a concise and technology-agnostic way. Software architecture and infrastructure models describe the system and its physical distribution on a large scale. We apply cloud technology for the services we develop, e.g., the SSELab [HKR12] and the Energy Navigator [FPPR12, KPR12] but also for our tool demonstrators and our own development platforms. New services, e.g.,c collecting data from temperature, cars etc. are easily developed.

References

[BBR07] Christian Basarke, Christian Berger, and Bernhard Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 4(12):1158–1174, October 2007.

[BCGR09a] Manfred Broy, Maria Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In Kevin Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, 2009.

[BCGR09b] Manfred Broy, Maria Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the UML System Model. In Kevin Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, 2009.

[BCR07a] Manfred Broy, Maria Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, February 2007.

[BCR07b] Manfred Broy, Maria Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, February 2007.

[BGH+97] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Proceedings OOPSLA'97 Workshop on Object-oriented Behavioral Semantics*, TUM-I9737, TU Munich, 1997.

[BGH+98] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In M. Schader and A. Korthaus, editors, *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*. Physica Verlag, Heidelberg, 1998.

[BHP+98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In M. Broy and B. Rumpe, editors, *RTSE '97: Proceedings of the International Workshop on Requirements Targeting Software and Systems Engineering*, LNCS 1526, pages 43–68, Bernried, Germany, October 1998. Springer.

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007.

[BR12a] Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Proceedings of the 10th Workshop on Automotive Software Engineering (ASE 2012)*, pages 789–798, Braunschweig, Germany, September 2012.

[BR12b] Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*. Springer, 2012.

[CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, CfG Fakultät, TU Braunschweig, 2008.

[CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Model Driven Engineering Languages and Systems. Proceedings of MODELS 2009*, LNCS 5795, pages 670–684, Denver, Colorado, USA, October 2009.

[EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Meta-Modelling Semantics of UML. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*. Kluver Academic Publisher, 1999.

[FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008.

[FLP+11] Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. State-Based Modeling of Buildings and Facilities. In *Proceedings of the 11th International Conference for Enhanced Building Operations (ICEBO' 11)*, New York City, USA, October 2011.

[FPPR12] Norbert Fisch, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Proceedings of the 7th International Conference on Energy Efficiency in Commercial Buildings (IEECB)*, Frankfurt a. M., Germany, April 2012.

[GHK+07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based Modeling of Function Nets. In *Proceedings of the Object-oriented Modelling of Embedded Real-Time Systems (OMER4) Workshop*, Paderborn, Germany, October 2007.

[GHK+08] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, Toulouse, 2008.

[GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen (MBEFF)*, Informatik Bericht 2008-01, pages 76–89, CFG Fakultät, TU Braunschweig, March 2008.

[GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TUM, Munich, Germany, 1996.

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Technical Report 2006-04, CfG Fakultät, TU Braunschweig, August 2006.

[GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, and Martin Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Proceedings der Modellierung 2006*, Lecture Notes in Informatics LNI P-82, Innsbruck, März 2006. GI-Edition.

[GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TUM, Munich, Germany, 1995.

[GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems. 16th Monterey Workshop*, LNCS 6662, pages 17–32, Redmond, Microsoft Research, 2011. Springer.

[GRJA12] Tim Gülke, Bernhard Rumpe, Martin Jansen, and Joachim Axmann. High-Level Requirements Management and Complexity Costs in Automotive Development Projects: A Problem Statement. In *Requirements Engineering: Foundation for Software Quality. 18th International Working Conference, Proceedings, REFSQ 2012*, Essen, Germany, March 2012.

[GRR10] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Model Driven Engineering Languages and Systems, Proceedings of MODELS*, LNCS 6394, Oslo, Norway, 2010. Springer.

[HHK+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Proceedings of the 17th International Software Product Line Conference (SPLC), Tokyo*, pages 22–31. ACM, September 2013.

[HKM+13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-Class Variability Modeling in Matlab / Simulink. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, pages 11–18, New York, NY, USA, 2013. ACM.

[HKR+07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In D. H. Akehurst, R. Vogel, and R. F. Paige, editors, *Proceedings of the Third European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2007), Haifa, Israel*, pages 99–113. Springer, 2007.

[HKR+09] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In H. Arabnia and H. Reza, editors, *Proceedings of the 2009 International Conference on Software Engineeering in Research and Practice*, Las Vegas, Nevada, USA, 2009.

[HKR12] Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Proceedings of the 2nd International Workshop on Developing Tools as Plug-Ins (TOPI) at ICSE 2012*, pages 61–66, Zurich, Switzerland, June 2012. IEEE.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, Oct 2004.

[HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling Dynamic Component Interfaces. In Madhu Singh, Bertrand Meyer, Joseph Gil, and Richard Mitchell, editors, *TOOLS 26, Technology of Object-Oriented Languages and Systems*. IEEE Computer Society, 1998.

[HRR+11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Proceedings of International Software Product Lines Conference (SPLC 2011)*. IEEE Computer Society, August 2011.

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen, February 2012.

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VII, fortiss GmbH*, February 2011.

[HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208, Oxford, UK, March 2012. Springer.

[HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering 2012: Fachtagung des GI-Fachbereichs Softwaretechnik in Berlin*, Lecture Notes in Informatics LNI 198, pages 181–192, 27. Februar - 2. März 2012.

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM'09), Sprinkle, J., Gray, J., Rossi, M., Tolvanen, J.-P., (eds.)*, Techreport B-108, Helsinki School of Economics, Orlando, Florida, USA, October 2009.

[KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Proceedings of the Modelling of the Physical World Workshop MOTPW'12, Innsbruck, October 2012*, pages 2:1–2:6. ACM Digital Library, October 2012.

[KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Fourth IEEE Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*. P. Dini, IOS-Press, 1997.

[KPR12] Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering Band 14. Shaker Verlag Aachen, 2012.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering Band 1. Shaker Verlag, Aachen, Germany, 2010.

[KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Proceedings of the first International Workshop on Formal Methods for Open Object-based Distributed Systems*, pages 323–338. Chapmann & Hall, 1996.

[KRS12] Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, pages 113–116. VDI Verlag, 2012.

[KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In J. Gray, J.-P. Tolvanen, and J. Sprinkle, editors, *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling 2006 (DSM'06), Portland, Oregon USA*, Technical Report TR-37, pages 150–158, Jyväskylä University, Finland, 2006.

[KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM' 07), Montreal, Quebec, Canada*, Technical Report TR-38, pages 8–10, Jyväskylä University, Finland, 2007.

[KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, editors, *Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MODELS 2007), Nashville, TN, USA, October 2007*, LNCS 4735. Springer, 2007.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In R. F. Paige and B. Meyer, editors, *Proceedings of the 46th International Conference Objects, Models, Components, Patterns (TOOLS-Europe), Zurich, Switzerland, 2008*, Lecture Notes in Business Information Processing LNBIP 11, pages 297–315. Springer, 2008.

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

[LRSS10] Tihamer Levendovszky, Bernhard Rumpe, Bernhard Schätz, and Jonathan Sprinkle. Model Evolution and Management. In *MBEERTS: Model-Based Engineering of Embedded Real-Time Systems, International Dagstuhl Workshop, Dagstuhl Castle, Germany*, LNCS 6100, pages 241–270. Springer, October 2010.

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, May 2010.

[MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Proc. Euro. Soft. Eng. Conf. and SIGSOFT Symp. on the Foundations of Soft. Eng. (ESEC/FSE'11)*, pages 179–189. ACM, 2011.

[MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011.

[MRR11c] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Model Driven Engineering Languages and Systems (MODELS 2011), Wellington, New Zealand*, LNCS 6981, pages 592–607, 2011.

[MRR11d] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *Proc. 25th Euro. Conf. on Object Oriented Programming (ECOOP'11)*, LNCS 6813, pages 281–305. Springer, 2011.

[MRR11e] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Model Driven Engineering Languages and Systems (MODELS 2011), Wellington, New Zealand*, LNCS 6981, pages 153–167. Springer, 2011.

[PFR02] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product Line Annotations with UML-F. In G. J. Chastek, editor, *Software Product Lines - Second International Conference, SPLC 2*, LNCS 2379, pages 188–197, San Diego, 2002. Springer.

[PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In M. Naftalin, T. Denvir, and M. Bertran, editors, *FME'94: Industrial Benefit of Formal Methods*, LNCS 873. Springer, October 1994.

[PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In J. Davies J. M. Wing, J. Woodcock, editor, *FM'99 - Formal Methods, Proceedings of the World Congress on Formal Methods in the Development of Computing System*, LNCS 1708, pages 96–115. Springer, 1999.

[PR03] Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In H. Kilov and K. Baclawski, editors, *Practical foundations of business and system specifications*, pages 281–297. Kluwer Academic Publishers, 2003.

[PR13] Antonio Navarro Perez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In I. Ober, A. S. Gokhale, J. H. Hill, J. Bruel, M. Felderer, D. Lugato, and A. Dabholka, editors, *Proc. of the 2nd International Workshop on Model-Driven Engineering for High Performance and Cloud Computing. Co-located with MODELS 2013, Miami*, Sun SITE Central Europe Workshop Proceedings CEUR 1118, pages 15–24. CEUR-WS.org, 2013.

[RK96] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In H. Kilov and W. Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, pages 265–286. Kluwer Academic Publishers, 1996.

[RKB95] Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technical Report TUM-I9510, Technische Universität München, 1995.

[RRW12] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Requirements Modeling Language for the Component Behavior of Cyber Physical Robotics Systems. In N. Seyff and A. Koziolek, editors, *Modelling and Quality in Requirements Engineering: Essays Dedicated to Martin Glinz on the Occasion of His 60th Birthday*. Monsenstein und Vannerdat, Münster, 2012.

[RRW13] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Workshops and Tutorials Proceedings of the 2013 IEEE International Conference on Robotics and Automation (ICRA), May 6-10, 2013, Karlsruhe, Germany*, pages 10–12, 2013.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, ISBN 3-89675-149-2, 1996.

[Rum02] Bernhard Rumpe. Executable Modeling with UML - A Vision or a Nightmare? In T. Clark and J. Warmer, editors, *Issues & Trends of Information Technology Management in Contemporary Associations, Seattle*, pages 697–701. Idea Group Publishing, Hershey, London, 2002.

[Rum03] Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In F. de Boer, M. Bonsangue, S. Graf, W.-P. de Roever, editor, *Formal Methods for Components and Objects*, LNCS 2852, pages 380–402. Springer, November 2003.

[Rum04] Bernhard Rumpe. Agile Modeling with the UML. In M. Wirsing, A. Knapp, and S. Balsamo, editors, *Radical Innovations of Software and Systems Engineering in the Future. 9th International Workshop, RISSEF 2002. Venice, Italy, October 2002*, LNCS 2941. Springer, October 2004.

[Rum11] Bernhard Rumpe. *Modellierung mit UML*. Springer, second edition, September 2011.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring*. Springer, second edition, Juni 2012.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering Band 11. Shaker Verlag, Aachen, Germany, 2012.

[SRVK10] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling: State of the Art and Research Challenges. In *MBEERTS: Model-Based Engineering of Embedded Real-Time Systems, International Dagstuhl Workshop, Dagstuhl Castle, Germany*, LNCS 6100, pages 57–76, October 2010.

[THR+13] Ulrike Thomas, Gerd Hirzinger, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *Proceedings of the 2013 IEEE International Conference on Robotics and Automation (ICRA)*, pages 461–466, Karlsruhe, Germany, May 2013. IEEE.

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering Band 9. Shaker Verlag, Aachen, Germany, 2011.

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering Band 12. Shaker Verlag, Aachen, Germany, 2012.

[ZPK+11] Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In D. Schaefer, editor, *Proceedings of the SESAR Innovation Days*. EUROCONTROL, November 2011.