RWTH Aachen

Department of Computer Science
Technical Report

Integration und Analyse von Artefakten in der modellbasierten Entwicklung eingebetteter Software

Daniel Merschen

ISSN 0935–3232 · Aachener Informatik-Berichte · AIB-2014-02

RWTH Aachen · Department of Computer Science · Jan. 2014

The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

`http://aib.informatik.rwth-aachen.de/`

Integration und Analyse von Artefakten in der modellbasierten Entwicklung eingebetteter Software

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker
Daniel Merschen
aus Würselen-Bardenberg

Berichter: Universitätsprofessor Dr.-Ing. Stefan Kowalewski
Universitätsprofessor Dr. rer. nat. Bernhard Rumpe

Tag der mündlichen Prüfung: 07. Januar 2014

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, 2014)

Daniel Merschen
Lehrstuhl Informatik 11
`merschen@embedded.rwth-aachen.de`

Aachener Informatik Bericht AIB-2014-02

Herausgeber: Fachgruppe Informatik
RWTH Aachen University
Ahornstr. 55
52074 Aachen
GERMANY

ISSN 0935-3232

ISBN 978-3-8440-2601-6

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`

Zusammenfassung

Um heutigen Funktionalitäts- und Sicherheitsanforderungen an Automobile gerecht zu werden, ist eingebettete Software unerlässlich, da die rein elektronische Realisierung einerseits sehr komplex wäre und andererseits in einer hohen Anzahl elektrischer und elektronischer Komponenten münden würde. Dadurch würde das Automobil schwerer und folglich der Kraftstoffverbrauch höher. Für die Entwicklung eingebetteter Software hat sich in dieser Branche die modellbasierte Softwareentwicklung mit MATLAB/Simulink etabliert. Um dabei das Wiederverwendungspotenzial optimal auszuschöpfen, folgt man dabei dem Software-Produktlinienansatz.

In der Entwicklung ergeben sich nun neue Herausforderungen im Bereich des Komplexitäts- und Evolutionsmanagements der Software. So sind Abhängigkeiten im Simulink-Modell oft nicht offensichtlich. Auch der Zusammenhang zwischen Artefakten des Entwicklungsprozesses, z. B. zwischen dem Simulink-Modell und dem Lastenheft, ist häufig unklar. Dies erschwert insbesondere die Einarbeitung späterer Änderungen.

Diese Dissertation widmet sich derartigen Herausforderungen, indem zunächst ein allgemeines Lösungskonzept für die Artefaktintegration und -analyse erarbeitet wird, welches anschließend auf zwei Arten umgesetzt wird, die unterschiedlichen Paradigmen folgen.

Der modellbasierte Ansatz parst die Originalartefakte in Modelle des Eclipse Modeling Frameworks (EMF), um sie anschließend mit Hilfe von Modelltransformationen für spätere Analysen vorzubereiten. Dabei entsteht pro Artefakt ein Modell, der sogenannte *Repräsentant*. Die Artefaktintegration erfolgt über miteinander verknüpfte Metamodelle dieser Repräsentanten. Artefaktanalysen arbeiten auf diesen Repräsentanten und sind ebenfalls mit Modelltransformationen realisiert. Das Ergebnis ist daher abermals ein EMF-Modell, welches geeignet visualisiert werden kann.

Der zweite Ansatz integriert Artefakte über eine zentrale Datenbank. Dazu werden diese mit Hilfe einer Kombination aus werkzeugspezifischer und Java-Funktionalität in Java-Klassenmodelle transferiert, die anschließend in der Datenbank abgelegt und dort mit anderen Artefakten verknüpft werden können. Analysen sind hier ebenfalls in Java umgesetzt und operieren auf den Java-Klassenmodellen und der Datenbank.

Schließlich wird die Eignung der beiden Ansätze für die Artefaktintegration und Analyse hinsichtlich verschiedener Kriterien evaluiert. Dazu gehören zunächst die Laufzeiteffizienz und Skalierbarkeit für große Simulink-Modelle. Des Weiteren wird die Handhabbarkeit in der Praxis auf Grundlage von Fallstudien und Einschätzungen von Entwicklern bewertet. Da die Ansätze dynamisch um weitere Analysen erweiterbar sein müssen, um an neue bzw. unternehmensspezifische Bedürfnisse angepasst zu werden, werden dabei auch der Schwierigkeitsgrad der Analysenimplementierung und das notwendige Vorwissen in die Bewertung einbezogen.

Abstract

Nowadays, functional and safety requirements of vehicles can hardly be met without embedded software since a pure hardware-oriented realisation would be too complex and would result a huge number of electronic control units. Hence, the vehicle's weight would increase leading to a higher fuel consumption. During the last years, MATLAB/Simulink has become state of the art for the development of embedded software in the automotive domain. Furthermore, in order to make advantage of commonalities a software product line approach is applied.

This kind of development raises new challenges in the context of complexity and evolution management of embedded software. For instance, dependencies within Simulink models as well as links among design artefacts like Simulink model and requirements document are frequently not obvious. Hence, later change requests become more and more complicated to incorporate.

This thesis first presents a general concept for artefact integration and analysis in the context of embedded software development in the automotive domain. Second, the concept is realised with two implementation approaches following different paradigms.

The model-based approach first parses the original artefacts into models of the Eclipse Modeling Framework (EMF) which are subsequently structurally optimised by model transformations to facilitate later analyses. This results in one EMF model per artefact, the so-called *representative*. The artefact integration is realised by links among the representatives' meta models. Analyses of artefacts, which are again realised by model transformations, now operate on these representatives leading to a further EMF model representing the analysis result.

The second approach integrates artefacts via a central database. To this end, the original artefacts are first transferred into Java models applying tool-specific as well as Java functionality. In the next step, these models are persisted in the database where they can be linked to each other. Analyses operate on both Java models and the database and are, hence, implemented in Java.

After a detailed explanation, the suitability of the approaches for artefact integration and analysis in an industrial context is evaluated based on case studies and developers' opinions. To this end, the run time efficiency and the scalability to huge Simulink models are investigated. Furthermore, generalisability and extensibility are discussed. Since each approach has to be adaptable to company-specific needs, the complexity of analysis implementation and the necessary background knowledge are considered as well.

Danksagung

Die vorliegende Dissertation wäre ohne die Unterstützung durch andere nicht möglich gewesen, denen ich an dieser Stelle herzlich danken möchte.

Zunächst einmal gilt mein Dank Prof. Dr.-Ing. Stefan Kowalewski für die Unterstützung meines Promotionsvorhabens durch seine Rolle als Doktorvater und für seine hilfreiche, konstruktive Kritik sowie für die gute Zusammenarbeit während meiner gesamten Promotionszeit.

Des Weiteren bedanke ich mich bei Prof. Dr. rer. nat. Bernhard Rumpe für seine Bereitschaft, das Zweitgutachten zu verfassen, sowie bei Prof. Dr. rer. nat. Thomas Seidl und Prof. Dr. rer. nat. Jürgen Giesl für ihre Teilnahme an der Prüfungskommission.

Eine weitere bedeutsame Rolle kommt den Projektpartnern bei der Daimler AG zu. Für die stets kollegiale und kooperative Arbeitsatmosphäre, konstruktive Rückmeldungen und Verbesserungsvorschläge aus der industriellen Praxis danke ich insbesondere Dr.-Ing. Bernd Hedenetz, Jacques Thomas und Yves Duhr.

Auch danke ich meinen Kollegen am Lehrstuhl für das freundliche Arbeitsumfeld, ihre sachliche, konstruktive Kritik und ihre stete Hilfsbereitschaft. Für die Einführung in die zugrunde liegende Thematik am Anfang meiner Promotionszeit sowie für die Vorarbeiten, auf denen diese Dissertation aufsetzt, bedanke ich mich insbesondere bei Andreas Polzer.

Des Weiteren bedanke ich mich für die tatkräftige Unterstützung in der Implementierung und fruchtbare Fachdiskussionen bei meinen studentischen Hilfskräften und Abschlussarbeitern.

Mein Dank gilt ferner all jenen, die durch gründliches Gegenlesen zum Gelingen dieser Dissertation beigetragen haben.

Insbesondere danke ich schließlich meinen Eltern und meinem Bruder für ihre stete Unterstützung jeglicher Art in dieser arbeitsreichen Zeit.

Herzlichen Dank!

Aachen, den 6. Februar 2014

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Motivation

Die modellbasierte Entwicklung gewinnt im Bereich eingebetteter Software kontinuierlich an Bedeutung [21, 35]. Insbesondere im Automobilbereich hat sich mittlerweile MATLAB/Simulink [149] als Standard-Entwicklungswerkzeug etabliert [2, 19, 21, 33, 35, 159]. Die Gründe dafür sind vielfältig. In der Entwicklung eingebetteter Systeme werden häufig Steuerungs- und Regelungsalgorithmen implementiert. Die in Simulink übliche Modellierung von Blockschaltbildern stellt eine im Ingenieurswesen wohlbekannte Spezifikationsart für Algorithmen dar. Über Codegeneratoren werden so auch Personen ohne tiefgreifendes Hintergrundwissen in der Softwareentwicklung in die Lage versetzt, komplexe Algorithmen komfortabel zu realisieren. Dadurch können zudem die Codequalität im Vergleich zu bisherigen Softwareentwicklungsansätzen gesteigert und signifikante Effizienzsteigerungen erzielt werden [140, Kap. 2.2]. Ein weiterer Vorteil der modellbasierten Entwicklung liegt in der Simulierbarkeit der Modelle, wodurch Fehler frühzeitig im Entwicklungsprozess erkannt und behoben werden können. Schließlich lässt sich aus einem Simulink-Modell Code für verschiedene Hardware-Plattformen generieren, sodass der Entwickler von Hardware-Spezifika abstrahieren kann.

Des Weiteren ist insbesondere in der Automobilindustrie die Variabilität von Software aufgrund verschiedener Absatzmärkte mit unterschiedlicher Gesetzgebung, der Adressierung verschiedener Marktsegmente sowie dem breiten Produktangebot hoch [132]. So weisen beispielsweise Dziobek et al. [33] darauf hin, dass statistisch gesehen im Jahr 2006 keine zwei identischen Versionen der Mercedes C-Klasse ausgeliefert wurden. Aufgrund dieser Variantenvielfalt ergibt in dieser Branche häufig ein Softwareproduktlinienansatz Sinn, um durch systematische Wiederverwendung einerseits die Marktreife- und Entwicklungszeit zu reduzieren und andererseits dabei gleichzeitig die Software-Qualität zu steigern. In der Praxis bietet sich daher die Kombination von modellbasierter Softwareentwicklung und des Softwareproduktlinien-Konzepts an. Häufig wendet man dabei einen 150%-Ansatz an, d. h. ein Funktionsmodell beschreibt die gesamte Funktionalität der Produkt*linie*. Um ein *Produkt* davon abzuleiten, werden durch Konfiguration einzelne Teile davon deaktiviert.

Weitere erfolgskritische Faktoren in der Automobilindustrie sind die kontinuierliche Ausweitung des Angebots an neuen, innovativen Funktionen sowie das Mithalten mit dem Stand der sich weiterentwickelnden Technik. Dieser hardwaretechnische Fortschritt schlägt auch auf die Implementierung durch. Dies führt zu einem sehr dynamischen Entwicklungs- und Evolutionsprozess, der durch hohe Variabilität und Komplexität der Artefakte geprägt ist. Unter Artefakten versteht man im hier betrachteten Kontext die

Resultate verschiedener Entwicklungsprozessschritte, z. B. Lastenheft, Funktionsmodell und Testspezifikation.

Schließlich müssen die Artefakte konsistent zueinander sein in dem Sinne, dass beispielsweise das Lastenheft vollständig im Simulink-Modell umgesetzt sein muss und letzteres vollständig von Tests abgedeckt wird.

1.2 Ziele

1.2.1 Fragestellung

Im Rahmen der vorliegenden Dissertation wurde ein Lösungsansatz für die zuvor genannten Herausforderungen konzipiert und auf zwei Arten realisiert. Ziel der Arbeit ist neben der Ausarbeitung eines allgemeinen Lösungsansatzes zu untersuchen, welcher der beiden Ansätze sich für die Herausforderungen besser eignet, und wie gut sich die Ansätze in die industrielle Praxis eingliedern lassen.

1.2.2 Beiträge der Arbeit

Um das zuvor genannte Ziel zu erreichen, werden in dieser Dissertation folgende Beiträge geleistet:

Lösungskonzept Zunächst wird ein Lösungskonzept vorgestellt, welches umsetzungsunabhängig beschreibt, welche Wege beschritten werden müssen, um den Entwickler in der Handhabung von Komplexität zu unterstützen. Es sieht vor, Artefakte in ein gemeinsames Datenmodell zu integrieren, auf dem spätere Analysen operieren. Außerdem dient das Datenmodell der Anreicherung von Artefakten mit Informationen über den Entwicklungs- und Evolutionsprozess, über die diese indirekt miteinander verknüpft werden. Des Weiteren wird ein Ansatz für die Konsistenzprüfung im Kontext von Software-Produktlinien erläutert.

Modellbasierte Analyse Das Lösungskonzept wurde zunächst modellbasiert umgesetzt. Dieser Ansatz greift auf das Eclipse Modeling Framework (EMF) [38] zurück. Dazu wurden Metamodelle für Artefakte entwickelt und Parser erstellt, welche die Artefakte in EMF-Modelle überführen. Mit Hilfe von hybriden Modelltransformationen werden diese EMF-Modelle strukturell optimiert, um schließlich in weiteren Transformationen analysiert zu werden. Schließlich wird das Graphical Modeling Framework (GMF) [38] angewendet bzw. eine Rücktransformation nach MATLAB/Simulink durchgeführt, um die Analyseergebnisse zu visualisieren.

Dieser Ansatz baut auf Vorarbeiten auf, die ich in Kooperation mit meinem Kollegen, Andreas Polzer, umgesetzt habe. Der Parser für das Simulink-Modell entstammt seinen Vorarbeiten und wurde nur geringfügig modifiziert. Im Rahmen von Industrieprojekten wurden zudem Modelltransformationen für das Simulink-Modell gemeinsam implementiert.

Datenbankorientierte Analyse Der zweite Ansatz, der im Kontext der Dissertation umgesetzt wurde, verwendet eine relationale Datenbank als zentrale Artefaktablage. Dazu wurden Transferfunktionen entwickelt, die die Artefakte in eine relationale Datenbank importieren. Der Import geschieht dabei mit Hilfe einer Funktionsbibliothek, die unter Nutzung vorhandener Rahmenwerke in Java realisiert wurde. Anschließende Modellanalysen werden ebenfalls damit in Java umgesetzt, wobei Artefakte und ggf. weitere Informationen aus der Datenbank ausgelesen werden. Da für die praktische Anwendbarkeit häufige Werkzeugwechsel hinderlich sind, wird die Analysefunktionalität zudem in MATLAB/Simulink integriert.

Evaluierung Anhand von verschiedenen Fallstudien werden die beiden Ansätze miteinander verglichen. Die Fallstudien entstammen Industriekooperationen im Vorfeld dieser Dissertation. Dabei werden die Kriterien Effektivität, Effizienz, Skalierbarkeit, Komplexität in der Handhabung sowie die Erweiterbarkeit und Generalisierbarkeit und schließlich die Praxisrelevanz im industriellen Alltag betrachtet.

1.3 Bibliographische Hinweise

Teile der vorliegenden Dissertation entstammen Industriekooperationen und diversen durch mein Mitwirken entstandenen Veröffentlichungen sowie von mir betreuten studentischen Abschlussarbeiten. Deren Inhalte werden hier kurz zusammengefasst und in einen Zusammenhang gebracht.

In [118] wurde ein Teil des modellbasierten Analyseansatzes beschrieben. Insbesondere die Modell- und Metamodellerstellung für das Simulink-Modell und die darauf ausgeführten Analysen werde hier erstmals beschrieben. [88] und [117] setzen denselben Ansatz in den Kontext von Software-Produktlinien und beschreiben weitere Analysen, die mit Hilfe dieses Ansatzes implementiert wurden. Des Weiteren wird hier auch auf die Verknüpfung zwischen Artefakten eingegangen. Auch [86] beschäftigt sich mit der modellbasierten Analyse, fokussiert aber primär auf das Problem fehlender Meta-Informationen und stellt daher eine prototypische Implementierung für die Integration selbiger in die EMF-basierten Artefaktmodelle vor. Des Weiteren wird auf die Integration des Ansatzes in MATLAB/Simulink eingegangen. In [89] wird erstmals der datenbankorientierte Integrations- und Analyseansatz beschrieben. Die Schwerpunkte bilden hier die Architektur des Rahmenwerks sowie die Integration von Informationen in Simulink-Modelle mit dem Ziel, diese in Analysen zu integrieren und artefaktübergreifend wiederzuverwenden. In [87] werden strukturelle Analysen des Simulink-Modells auf Basis des dazu erweiterten und modifizierten datenbankorientierten Rahmenwerks aus [85] und [89] beschrieben und hinsichtlich Effizienz und Skalierbarkeit dem modellbasierten Ansatz gegenübergestellt. [163] integriert eine auf der Arbeit von Cmyrev et al. [26] aufbauende Konsistenzanalyse in das datenbankorientierte Rahmenwerk, die über Merkmalkonsistenzprüfung einen Beitrag zu einem durchgängigen Variantenmanagement leistet.

Im Rahmen der Diplomarbeit von Yves Duhr (Daimler AG) [32] wurde ein Ansatz entwickelt, zwischen EMF-basierten Artefaktmodellen über Signalnamen eine Verknüp-

fung zwischen Anforderungen und Simulink-Modell herzustellen. Diese ist somit dem modelltransformationsbasierten Ansatz zuzuordnen. Julian Pott beschäftigt sich in seiner Diplomarbeit [119] mit der Integration von Meta-Informationen in Simulink-Modelle. Dazu beschreibt er ein Annotationskonzept und stellt eine prototypische Implementierung eines datenbankorientierten Analyse- und Annotationswerkzeugs vor. In der Bachelorarbeit von Robert Gleis [49] wird der Import des Simulink-Modells in eine relationale Datenbank um Signale, die über Linien transportiert werden, erweitert. Die somit anschließend mögliche Analyse des Signalflusses erfolgt auf Basis dieser Datenbank mit Hilfe von Java-Funktionalität und MATLAB-Skripten. Schließlich werden die Ergebnisse dem modellbasierten Ansatz gegenübergestellt. Christian Bryllok und Norbert Wiechowski erweitern in ihren Diplomarbeiten das Potenzial des datenbankorientierten Analyseansatzes um Variabilitäts- und Konsistenzsicherungsaspekte für die Unterstützung eines durchgängigen Variantenmanagements [22, 162]. Christian Bryllok stellt dabei Funktionen für die Identifikation von Variabilitätsmustern im Simulink-Modell auf Basis seiner Verknüpfung mit dem Merkmalmodell bereit und ermöglicht die Variantenableitung. Norbert Wiechowski integriert die Methodik zur Konsistenzprüfung von Verknüpfungen zwischen Anforderungen und Testspezifikation aus [26] in das datenbankorientierte Analyserahmenwerk und berücksichtigt dabei zusätzlich die Verknüpfungen mit dem Simulink-Modell. Auch werden hier sogenannte *QuickFixes* zur semi-automatischen Auflösung von inkonsistenten Verknüpfungen implementiert. Die Abschlussarbeiten von Julian Pott, Robert Gleis, Norbert Wiechowski und Christian Bryllok fallen damit in die Kategorie des datenbankorientierten Analyseansatzes.

1.4 Gliederung

In Kapitel 2 werden die Grundlagen der Entwicklungsparadigmen, der industrielle Entwicklungsprozess, der im Kontext dieser Dissertation betrachtet wird, und die daraus resultierende Terminologie beschrieben. Im Anschluss stellt Kapitel 3 die daraus hervorgehenden Herausforderungen aus technischer Perspektive vor. Schließlich wird ein allgemeines Lösungskonzept, unabhängig von der Umsetzung, aus diesen Herausforderungen abgeleitet und in den Stand der Forschung eingeordnet. Die Kapitel 5 und 6 beschreiben zwei Umsetzungen des Konzepts, bevor Kapitel 7 die beiden Konzepte evaluiert. Kapitel 8 fasst die Erkenntnisse der Dissertation zusammen und leitet weiteren Forschungsbedarf ab.

2 Grundlagen und Stand der Technik

Das Ziel dieses Kapitels ist, dem Leser die Hintergründe der den vorgestellten Lösungsansätzen zugrunde liegenden Softwareentwicklungsparadigmen zu erläutern. Dazu wird zunächst in die modellbasierte Softwareentwicklung und die Software-Produktlinien-Entwicklung eingeführt. Sodann erfolgt die Betrachtung des für diese Dissertation relevanten industriellen Entwicklungsprozesses inklusive einer Vorstellung der verwendeten Werkzeuge und der aus dem Prozess resultierenden Terminologie.

2.1 Modellbasierte Softwareentwicklung

Für die Realisierung von Steuergerätesoftware setzt sich, insbesondere in der Automobilindustrie, zunehmend der modellbasierte Entwicklungsansatz durch [2, 21, 77, 150], der im Folgenden erläutert wird. Weitere Informationen über die modellbasierte Softwareentwicklung finden sich u. a. in [140] und [11].

2.1.1 Begriffsdefinitionen

Formales Modell Formale Modelle (im Folgenden *Modelle* genannt) spielen seit langem eine bedeutsame Rolle für die Struktur- und Verhaltensbeschreibung von Software und Systemen. In der Literatur finden sich verschiedene Definitionen eines Modells. Laut Stachowiak beschreibt ein Modell im Allgemeinen eine Abbildung eines Originals in verkürzter (abstrahierter) Form zu einem bestimmten Zweck [139, S. 131–133]. Balzert bezieht auch die Ablaufbeschreibung und die leichtere Untersuchungsmöglichkeit eines Systems als wesentliche Charakteristika in die Definition eines Modells ein [10, S.100]. Für die Informatik sind folglich beide Definitionen relevant, da Modelle hier sowohl dazu verwendet werden, die Struktur und Funktionalität zu beschreiben, als auch, um Abläufe zu integrieren und Analysen von Systemen zu ermöglichen.

Im Rahmen der modellbasierten Softwareentwicklung stellen Modelle zentrale Entwicklungsartefakte dar und werden für die automatische Codegenerierung und Analyse verwendet. Je nach Einsatzzweck kommen dabei unterschiedliche Modellierungsarten zum Einsatz.

Ein klassisches Beispiel eines Modells für die Verhaltens- und somit Ablaufsmodellierung stellen endliche Automaten dar, die im Jahre 1962 von Arthur Gill in [48] vorgestellt wurden. Auch Petri-Netze, deren Grundlagen Carl Adam Petri in seiner Dissertation ebenfalls im Jahr 1962 vorstellte [112], sind Modelle und werden insbesondere für die Modellierung *nebenläufiger* Prozesse verwendet.

Als Modellierungssprache in der Softwareentwicklung entwickelt sich mittlerweile die Unified Modeling Language (UML) [106] zum Standard [127]. Sie bietet für die Modellierungsaspekte Struktur, Funktionalität und Ablauf von Software unterschiedliche Modellierungsarten für Softwaresysteme an, z. B. über Klassen-, Aktivitäts- und Sequenzdiagramme.

Metamodell Das wesentliche Merkmal der modellbasierten Softwareentwicklung ist die automatische Generierung von Code auf Basis eines Modells. Ein damit verfolgtes Ziel ist, die Korrektheit des Codes sicherzustellen. Die Korrektheit des Modells ist folglich die Voraussetzung, um dieses Ziel zu erreichen. Jedes Modell muss daher einem Metamodell entsprechen, welches fachliche und technische Korrektheitsaspekte beschreibt. Ein Modelleditor, der auf einem Metamodell basiert, kann das modellierte Modell so bereits während der Modellierung durch Abgleich mit diesem Metamodell validieren, sodass sich Codegeneratoren und Interpreter auf die Korrektheit des Modells verlassen können. Ein Metamodell beschreibt die *abstrakte Syntax* und die *statische Semantik* der ihm entsprechenden Modelle. Unter der abstrakten Syntax versteht man eine Strukturbeschreibung, die ausdrückt, welche Entitäten miteinander in welcher Relation stehen, sowie deren Kardinalitäten. Ein Beispiel für die abstrakte Syntax ist die Assoziation zwischen Klassen in einem UML-Klassendiagramm. Es ist jedoch auf dieser strukturellen Ebene nicht möglich, Bedingungen anzugeben, die zusätzlich erfüllt sein müssen, damit zwei Entitäten über eine Assoziation miteinander assoziiert sein können. Derartige Bedingungen werden auch als Constraints bezeichnet und stellen die statische Semantik dar. Sie beziehen sich beispielsweise auf Werte der Entitätsattribute oder weitere Assoziationen zwischen Entitäten. Eine verbreitete Sprache für derartige Spezifikationen von Constraints in UML-Modellen ist die Object Constraint Language (OCL) [101] von der Object Management Group (OMG) [102]. Die *konkrete* Syntax ist hingegen kein Bestandteil eines Metamodells. Diese gibt an, wie ein modellierter Sachverhalt tatsächlich in einem Modell umgesetzt wird, zum Beispiel die Art der Darstellung einer Klasse im Klassendiagramm oder die textuelle Beschreibung einer Entität im Java-Code.

Jedes Modell bezieht sich auf eine fachliche Domäne, die sowohl technischer als auch fachlicher Natur sein kann [140, Kap. 3.1.1]. Ein Beispiel für technische Domänen sind architekturzentrierte Domänen wie Daten-, Komponenten-, und Ablaufmodelle. Eine fachliche Domäne legt die Semantik in Bezug auf ein Wissensgebiet fest. Domänenspezifische Zusammenhänge müssen in dem Metamodell abgebildet werden, um die Korrektheit von Modellen und somit des generierten Codes zu gewährleisten. Ein Modell stellt folglich eine Instanz des Metamodells dar. Da ein Metamodell selbst ein Modell darstellt, wird auch dieses wiederum durch ein weiteres Metamodell, das sogenannte Metametamodell beschrieben. Es ergibt sich also eine *Instanz-von*-Beziehung zwischen Modell und Metamodell und eine *beschreibt*-Beziehung zwischen Metamodell und Modell, die die OMG auf vier Ebenen betrachtet (vgl. Abbildung 2.1).

Da der Zusammenhang zwischen Modellen, Metamodellen und Metametamodellen unendlich fortgesetzt werden kann, existiert normalerweise ein Metamodell, welches

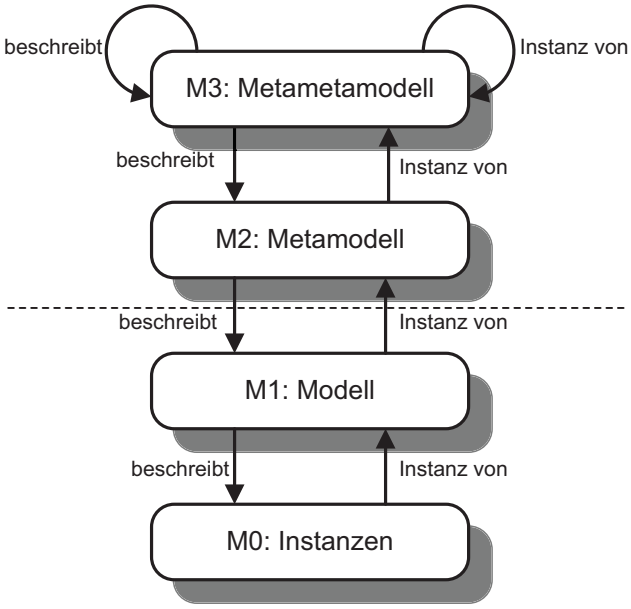


Abbildung 2.1: Meta-Ebenen der OMG [140, S.62]

sich selbst beschreibt. Im Standard der OMG ist dies die Meta Object Facility (MOF) [103] (auf Ebene M3 in Abbildung 2.1). Eine Instanz der MOF ist beispielsweise die Beschreibung von UML-Klassendiagrammen. Eine Klasse ist hier eine Instanz eines MOF-Classifiers und besitzt z. B. verschiedene Attribute und Assoziationen (Ebene M2). Auf Ebene M1 befindet sich das konkrete UML-Diagramm. Ihre Instanzen befinden sich schließlich auf Ebene M0.

Neben der Definition von Klassen für UML-Klassendiagramme definiert die Ebene M2 im OMG-Standard auch weitere in der UML verwendeten Elemente für die anderen UML-Diagrammarten, z. B. Komponentendiagramme, Aktivitätsdiagramme und Sequenzdiagramme. Komponentendiagramme modellieren die Struktur von Systemen. Aktivitätsdiagramme beschreiben Kontrollflüsse in der Software, wohingegen Sequenzdiagramme zeitliche Abläufe modellieren. Für weitere Informationen über die UML-Modellierung sei der Leser auf [127], [128] und [129] verwiesen.

Eine andere gängige und im weiteren Verlauf dieser Dissertation angewandte Metamodellierungstechnologie stellt das Eclipse Modeling Framework (EMF) [38] dar. Metamodelle werden hier selbst definiert. Nur das Metametamodell ist vorgegeben. So können Metamodelle für verschiedene Domänen erstellt werden, wodurch sich das Rahmenwerk aufgrund der verschiedenen Artefakte gut für die hier betrachteten Analysen anbietet. Aus dem Metamodell lässt sich Java-Code für Editoren entsprechender Modelle generieren.

Modelltransformation Im Rahmen der modellbasierten Softwareentwicklung spielen Modelltransformationen eine wichtige Rolle für die Validierung, Aufbereitung und Codegenerierung. Zum Beispiel können einfache Modellergänzungen durch Modelltransformationen übernommen werden. Als ein Beispiel für eine solche einfache Modellergänzung

beschreiben Stahl et al. das Einfügen einer Not-Aus-Transition zu allen Zuständen eines endlichen Automaten [140, S. 195]. Würde man die Verantwortung für solche Ergänzungen an den Benutzer übertragen, so würde sich eine neue Fehlerquelle ergeben.

Es existieren verschiedene Kategorien von Modelltransformationsansätzen – deklarative, imperative und hybride. Deklarative Ansätze arbeiten ausschließlich mit Hilfe von strukturellem Pattern Matching. Ein Vertreter dieser Kategorie sind beispielsweise Sprachen, die auf Triple-Graph-Grammatiken (TGGs) [134] basieren, wie beispielsweise das Story-Driven-Modelling [47, 156]. Die Transformationssprache QVT-Operational [39] ist hingegen ein Beispiel für eine rein imperative Transformationssprache. Die ATLAS Transformation Language (ATL) [36] stellt schließlich eine hybride Transformationssprache dar. Sie besteht aus Transformationsregeln, die über Pattern Matching angewendet werden. Die dabei erzeugten Elemente können anschließend noch imperativ modifiziert werden.

Codegenerierung Zentral für die modellbasierte Softwareentwicklung ist die automatische Codegenerierung. Im Unterschied zur herkömmlichen Softwareentwicklung, in der Modelle, insbesondere UML-Modelle, primär Dokumentationszwecken oder der initialen Generierung eines Codegerüstes dienen, welches anschließend manuell weiterentwickelt wird, soll hier bereits lauffähiger und höchstens geringfügig zu manipulierender Code erzeugt werden. Insbesondere sollen spätere Änderungen über das Modell erfolgen und nicht über die Modifikation von Quelltext. Die Codegenerierung ist eine spezielle Art der Modelltransformation, die aus dem Modell lauffähigen Code, also ein in konkreter Syntax beschriebenes Modell, generiert.

2.1.2 Ablauf

Um für eine Domäne modellbasiert Software entwickeln zu können, wird zunächst ein Metamodell erstellt, anhand dessen der Modelleditor bereits während der Modellierung die vom Entwickler modellierten Modelle validieren kann. Dabei sind sowohl softwaretechnische als auch domänenspezifische Aspekte zu berücksichtigen. Da spätere Korrekturen am Metamodell die Anpassung der entsprechenden Modelle sowie die erneute Codegenerierung erfordern und somit hohe Aufwände verursachen, ist die Korrektheit des Metamodells von essentieller Bedeutung. Daher sollten bei der Metamodellerstellung erfahrene Softwareentwickler und Domänenexperten gleichermaßen beteiligt sein [140].

Das Modell wird dann als Eingabe an einen Generator übergeben, der daraus Code generiert. Alternativ kann es auch an einen Interpreter übergeben werden, der auf Basis des Modells Aktionen ausführt [140, S. 12 ff.]. Im weiteren Verlauf dieser Dissertation wird jedoch davon ausgegangen, dass Code generiert wird.

2.1.3 Nutzen

Die modellbasierte Softwareentwicklung dient u. a. der Verbesserung der Software-Qualität und der Steigerung der Entwicklungseffizienz. Die Qualitätsverbesserung resultiert besonders aus der einheitlichen und dokumentierten Architektur [140, S. 16 ff.] sowie

der automatischen Codegenerierung. Auf längere Sicht kann die Entwicklungseffizienz gesteigert werden, da das System konzeptbedingt auch nach einigen Änderungen noch konform zu seiner Architektur (dem Modell) ist. Außerdem dient das Modell nicht nur als Codegenerierungsartefakt, sondern stellt darüberhinaus auch die Designdokumentation dar, wodurch spätere Änderungen vereinfacht und der Modifikationsprozess weiter beschleunigt werden können. Des Weiteren ist das Expertenwissen des Softwarearchitekten sowie das Domänenwissen im Modell gekapselt und kann so bei der Codegenerierung ausgenutzt werden. Broy et al. benennen zudem, dass durch den durchdringenden Gebrauch von Modellen das Abstraktionsniveau gegenüber der zugrunde liegenden Programmiersprache angehoben werden kann. Implementierungsdetails werden auf diese Art und Weise ausgeblendet, wodurch die Produktivität und Qualität steigen [21].

Durch den entwicklungsphasenübergreifenden Einsatz von Modellen sowie die Abstraktion von Zielplattformen wird in der modellbasierten Softwareentwicklung die Wiederverwendbarkeit unterstützt [35], insbesondere in Verbindung mit dem Software-Produktlinien-Konzept [140, Kap. 11.5]. Wie dies in der industriellen Entwicklung im Detail erfolgt, erläutert Abschnitt 2.3.3.

Schließlich werden durch die modellbasierte Softwareentwicklung bereits während der Modellierung Tests und Simulationen von Modellen ermöglicht, wodurch die Anzahl späterer Korrekturiterationen, die i. Allg. mit höheren Aufwänden bzw. Kosten verbunden sind, reduziert werden kann [90]. Durch die frühzeitige Test- und Simulationsmöglichkeit können außerdem die Entwicklungszeit reduziert und qualitativ hochwertige Funktionen realisiert werden [34].

2.2 Software-Produktlinien-Entwicklung

Bietet ein Unternehmen verschiedene Varianten einer Software an, so liegt nahe, dass sich viele Artefakte (z. B. Architektur, Softwarekomponenten und Anforderungen) stark ähneln und folglich systematisch wiederverwendet werden sollten. Die Software-Produktlinien-Entwicklung stellt einen systematischen Wiederverwendungsansatz dar und ist auf David Parnas zurückzuführen, der in [110] den Produktliniengedanken erstmals auf Softwareentwicklung überträgt. Da in eingebetteter Automobilsoftware extrem hohe Variabilität vorherrscht [33], bietet sich die Software-Produktlinien-Entwicklung hier besonders an und wird auch im weiteren Verlauf dieser Dissertation eine wichtige Rolle spielen. Daher werden hier ihre Hauptcharakteristika vorgestellt. Für detailliertere Informationen über die Software-Produktlinien-Entwicklung sei der interessierte Leser insbesondere auf [25] und [114] sowie auf die jeweils referenzierten Quellen verwiesen.

2.2.1 Begriffsdefinitionen

Produktlinie Clements und Northrop verstehen unter einer Software-Produktlinie eine auf ein bestimmtes Marktsegment ausgerichtete oder einem bestimmten Ziel dienende Menge software-intensiver Systeme, die eine Menge gemeinsamer Merkmale besitzen, die systematisch verwaltet werden [25, S. 5]. Aufgrund vieler Gemeinsamkeiten soll das

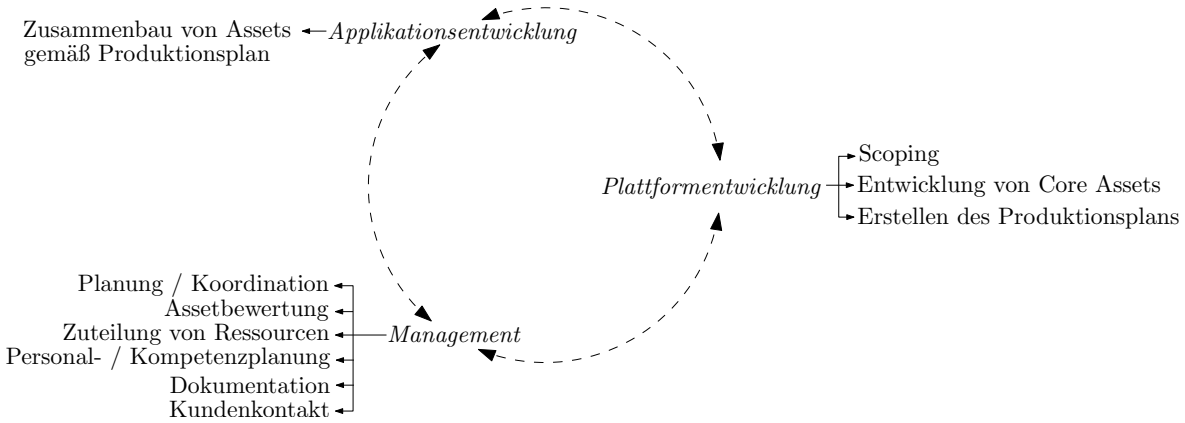


Abbildung 2.2: Drei Kernaktivitäten der Software-Produktlinien-Entwicklung

Wiederverwendungspotenzial dabei systematisch und bestmöglich ausgenutzt werden. Kennzeichnend für die Software-Produktlinien-Entwicklung ist daher die *systematische* Wiederverwendung von Code und anderen Artefakten im Gegensatz zur *opportunistischen* Wiederverwendung durch einfaches Kopieren. Im letzteren Fall wurden die wiederzuverwendenden Code-Abschnitte oder Funktionen nicht von Anfang an für die Wiederverwendung geplant und umgesetzt. Somit werden sie auch nicht in einer gemeinsamen Architektur gepflegt und weiterentwickelt. Zusammen mit häufig mangelhafter Dokumentation ist es dann oft günstiger, die Funktionen und Code-Abschnitte neu zu entwickeln anstatt bestehende wiederzuverwenden. In einer Software-Produktlinie hingegen werden alle wiederverwendbaren Elemente in einer *gemeinsamen* Architektur weiterentwickelt und gepflegt, wodurch jedes einzelne Produkt von Fehlerkorrekturen oder Erweiterungen profitiert.

Core Assets Produkte einer Software-Produktlinie werden aus einer Menge gemeinsamer *Core Assets* konstruiert. Core Assets stellen wiederverwendbare Bausteine dar, beispielsweise Architektur, Software-Komponenten, Dokumentation, Spezifikation und Testfälle. Die Menge der Core Assets bildet die sogenannte *Plattform* der Software-Produktlinie.

Domäne Eine Software-Produktlinie wird für ein bestimmtes Marktsegment entwickelt, mit dem ein spezifisches Fachwissen assoziiert ist, welches für die Produkte der Software-Produktlinie relevant ist und aus dem die abgedeckte Funktionalität resultiert. Ein solches Themen- und Wissensgebiet nennt man *Domäne*.

2.2.2 Funktionsweise

Die Software-Produktlinien-Entwicklung besteht im Wesentlichen aus den drei Kernaktivitäten *Plattformentwicklung*, *Applikationsentwicklung* und *Management*, die sich gegenseitig beeinflussen (vgl. Abbildung 2.2). In der Plattformentwicklung werden die Core Assets für die Produkte der Produktlinie entwickelt und weiterentwickelt, aus

denen in der Applikationsentwicklung die Produkte erstellt werden. Das begleitende Management soll systematisch die Evolution sowie die Entwicklung steuern und dazu auch das Wiederverwendungspotenzial von Assets untersuchen sowie Entwicklungsprozesse planen.

Plattformentwicklung In der Plattformentwicklung (engl. *Domain Engineering*) werden die sogenannten Core Assets der Produktlinie erstellt. Um Core Assets zu identifizieren, ist ein sogenanntes *Scoping* erforderlich. Dabei legt man fest, welche Produkte überhaupt zur Produktlinie gehören sollen, und analysiert deren Gemeinsamkeiten und Unterschiede, um die Variationspunkte in der Architektur zu identifizieren. Sodann muss das Wiederverwendungspotenzial aller Bausteine untersucht werden, um zu entscheiden, ob sie in die Plattform- oder Applikationsentwicklung übernommen werden sollen. Den Scope richtig zu bemessen, ist essentiell für den späteren Erfolg der Produktlinie. Bemisst man ihn zu groß, so wird aufgrund der breiteren Produktpalette die Menge der Gemeinsamkeiten und somit der Core Assets zu klein, wodurch zu viele Funktionalitäten in der Applikationsentwicklung erstellt werden müssen. Ein zu kleiner Scope resultiert in zu spezifischen bzw. zu konkreten Core Asset, wodurch das Wiederverwendungspotenzial sinkt.

Da neue Funktionen oft zunächst in höherwertige Produktkategorien Einzug halten und erst später in niedrigere Kategorien übernommen werden, ändert sich zudem das Wiederverwendungspotenzial von Artefakten evolutionsbedingt. Daher ist das Scoping regelmäßig zu wiederholen. Da somit Core Assets auch aus existierenden Produkten heraus extrahiert werden können, beeinflusst auch die Applikationsentwicklung die Plattformentwicklung.

Neben dem Scoping und der Entwicklung der Core Assets ist für jedes Asset zu dokumentieren, wie es in der Applikationsentwicklung zu verwenden ist, beispielsweise, ob die Variabilität durch Ableitung von Basisklassen, durch Konfiguration bzw. Parametrisierung oder durch Löschen von Komponenten implementiert werden soll. Zudem wird ein Produktionsplan erstellt, welcher beschreibt, wie die Assets *insgesamt* zusammengefügt werden sollen, und welche Werkzeuge dafür zu verwenden sind. Somit werden die Dokumentationen der einzelnen Assets hier zusammengeführt.

Applikationsentwicklung In der Applikationsentwicklung (engl. *Application Engineering*) werden aus den Core Assets der Plattformentwicklung konkrete Produkte erstellt. Neben den Core Assets selbst sind dazu auch die Unterschiede zwischen den Anforderungen an das *Produkt* und den Anforderungen an die Produkt*linie* zu identifizieren. Auf deren Basis ist zu entscheiden, ob das zu entwickelnde Produkt in den Scope der Produktlinie fällt. Ist dies der Fall, so werden die Core Assets dem Produktionsplan entsprechend zusammengefügt. Da die Entwicklung eines Produktes auch die Modifikation von Core Assets erfordern kann, wirkt sich die Applikationsentwicklung auch auf die Plattformentwicklung aus. Dies geschieht z. B., wenn das Produkt bei der Planung nicht berücksichtigt wurde.

Produktlinien-Management Um effizient zukünftige Anforderungen berücksichtigen zu können, ist eine vorausschauende Planung und Ausrichtung der Core Assets erforderlich. Dazu muss nachgehalten werden, welche Core Assets wie und für welche Produkte verwendet werden, um dadurch den Wert eines Assets für die Produktlinie bemessen zu können und einen Überblick über die Auswirkung von Änderungen auf die Produkte zu erlangen. Eine weitere Aufgabe des Produktlinien-Managements besteht in der Ressourcenzuteilung sowie der Koordination und Überwachung von Aktivitäten.

Man unterscheidet zwischen technischem und organisatorischem Management. Technisches Management steuert die Plattform- und Applikationsentwicklung und sorgt dafür, dass jeweils beide Entwicklungsgruppen in die notwendigen Aktivitäten involviert sind, z. B. sollten die Core-Asset-Experten involviert werden, wenn aus ihren Core-Assets Produkte erzeugt werden. Organisatorisches Management sorgt für die richtige Unternehmensstruktur, d. h. dass die richtigen Kompetenzen an der richtigen Stelle sitzen. Zudem legt es die Unternehmensstrategie fest und sorgt für die Dokumentation sowie den Aufbau, die Pflege und die Festigung von Kunden- und Zulieferungskontakten, da diese durch Produktlinien verstärkt erzeugt werden [25]. Schließlich fallen auch beim organisatorischen Management wiederverwendbare Core Assets an, z. B. Budget- und Zeitpläne. Insofern trägt auch das Management zur Plattformentwicklung bei.

2.2.3 Kosten und Nutzen

Durch die systematische Ausnutzung von Wiederverwendungspotenzial erzielt der Software-Produktlinien-Ansatz eine höhere Produktivität und somit eine Verkürzung der Marktreifezeit neuer Produktvarianten als die herkömmliche Softwareentwicklung. Aufgrund der zunächst zusätzlichen Aufwände für Scoping, Variabilitätsmodellierung und für die Konstruktion wiederverwendbarer Bausteine übersteigt die Marktreifezeit bei wenigen aus der Produktlinie konstruierten Produkte jene der durch herkömmliche Einzelproduktentwicklung entstandenen Produkte. Da ihre Architektur von besonderer Bedeutung für den dauerhaften Erfolg einer Produktlinie ist, sollten dabei erfahrene Architekten eingesetzt werden. Der daher höhere, initiale Aufwand zahlt sich längerfristig durch die bessere Nutzung des Wiederverwendungspotenzials gegenüber der Einzelproduktentwicklung und dadurch geringere Marktreifezeit und Kosten aus. Zahlreiche Untersuchungen wurden durchgeführt, um die Rentabilitätsschwelle der Software-Produktlinie-Entwicklung zu ermitteln. Clements und Northrop verweisen in [25, S. 226 ff.] diesbezüglich auf verschiedene Untersuchungen [58, 120, 124, 136, 152, 161], die nahe legen, dass sich ein Produktlinienansatz im Vergleich zum herkömmlichen Software-Entwicklungsansatz ab etwa drei Varianten rentiert. Generell hängt diese Schwelle jedoch von verschiedenen Parametern ab. So weisen McGregor et al. beispielsweise auf den Einfluss der angewandten Strategie für die Initiierung der Produktlinie hin [82]. Auch kann der Aufwand für die Entwicklung wiederverwendbarer Komponenten variieren. Eine detailliertere Kosten-Nutzen-Betrachtung findet sich neben den angegebenen Referenzen auch in [114, S. 10 ff.].

Durch die systematische Wiederverwendung der Core Assets kann zudem eine höhere Produktqualität gewährleistet werden, da diese ausgiebig getestet werden müssen und im

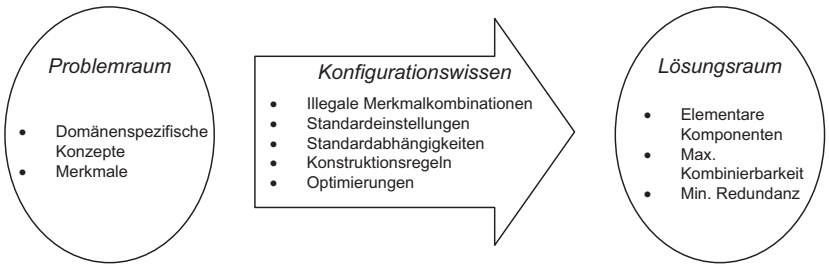


Abbildung 2.3: Elementes des generativen Domänenmodells (übersetzt aus [28, S.6])

Falle von Korrekturen alle Produkte gleichermaßen davon profitieren können. Außerdem können Produkte besser kundenspezifisch zugeschnitten werden. Somit ist insgesamt eine höhere Kundenzufriedenheit zu erwarten. Schließlich erlaubt der Software-Produktlinien-Ansatz durch das Konzept der wiederverwendbaren Bausteine, flexibel bestehende Produkte um neue Funktionalität zu erweitern.

Des Weiteren kann durch die von Experten für die gesamte Produktlinie entwickelte Architektur für jedes Produkt auf deren Expertise zurückgegriffen werden, da die Architektur für ein Produkt „nur noch“ instanziiert werden muss. Dies führt gemeinsam mit der systematischen Wiederverwendung sorgfältig getesteter Core Assets zu einem Qualitäts- und Zeitgewinn.

Auch im Bereich des Produktlinien*managements* werden Zeit und Kosten eingespart, da die bereits für andere Produkte erstellten und kontinuierlich überarbeiteten Zeit- und Budgetpläne eine gute Grundlage für die Planung neuer Produktentwicklungen darstellen.

Schließlich kann durch die Wiederverwendung Personal eingespart und somit die Produktivität erhöht werden. Da Personal aufgrund der Vorkenntnisse der Core Assets zudem effizient für verschiedene Produkte der Produktlinie eingesetzt werden kann, wird auch die Flexibilität des Entwicklungsprozesses gesteigert.

2.2.4 Negative Variabilität

Man unterscheidet bei der Software-Produktlinien-Entwicklung zwei verschiedene Ansätze. Beim (bisher beschriebenen) klassischen Ansatz werden einzelne Produkte der Produktlinie aus einer Menge gemeinsamer Core Assets entwickelt, d. h. hier werden die Komponenten zu einem Gesamtprodukt zusammengefügt. Ein Produkt entsteht also im Wesentlichen durch *Integration* von Bausteinen zur Realisierung von Funktionalität. Dieser Ansatz wird in dieser Arbeit in Anlehnung an [50] auch als Ansatz *positiver Variabilität* bezeichnet.

Ein weiterer Ansatz der Software-Produktlinien-Entwicklung basiert auf der *Generativen Softwareentwicklung* nach Czarnecki und Eisenecker [28]. Im Wesentlichen lässt sich Generative Softwareentwicklung beschreiben wie in Abbildung 2.3 dargestellt. Zunächst existiert ein Problemraum, der domänenspezifische Konzepte und eine funktionale Modellierung der Software-Produktlinie, z. B. in Form eines Merkmalmodells, beinhaltet. Außerdem existiert ein Lösungsraum, welcher die Komponenten für die

einzelnen Produkte der Produktlinie beinhaltet. Um nun ein Produkt zu erstellen, muss eine Konfiguration des Problemraums bestimmt werden, mit der dann aus den Bausteinen des Lösungsraums ein Produkt abgeleitet werden kann. Czarnecki und Antkiewicz beschreiben, wie man dieses Entwicklungsparadigma auf ein generisches Modell-Template anwenden und daraus spezifische Modell-Templates ableiten kann [27]. Das generische Modell-Template beinhaltet die Elemente aller gültigen Produkte der Produktlinie. Die speziellen Modell-Templates entstehen dann aus dem generischen Template durch Konfiguration. Hier wird also aus einem alle Produkte umfassenden Modell ein produktspezifisches Modell abgeleitet. Für die Entwicklung eines einzelnen Produktes wird also somit die nicht benötigte Funktionalität aus dem generischen Modell entfernt. Czarnecki und Antkiewicz bezeichnen diesen Ansatz als Konzept der *überlagerten Varianten* (engl. *Superimposed Variants*).

Für diesen Ansatz finden sich in der Literatur auch die Begriffe *Negative Variabilität* (z. B. in [19, 50]) oder auch *150%-Ansatz* (z. B. in [34]). Diese Synonyme sind der Tatsache geschuldet, dass hier im Gegensatz zur zuvor beschriebenen Methodik Funktionalität *entfernt* wird. Das generische Modell enthält also die Vereinigung aller möglichen Modellvarianten. Folglich wird es auch als 150%-Modell bezeichnet. Der Terminologie folgend ist das Funktionsmodell des spezifischen Produktes also als 100%-Modell aufzufassen.

2.2.5 Modellierung

Da im weiteren Verlauf auch das Verständnis der Merkmalmodellierung von Software-Produktlinien vorausgesetzt wird, soll hier auf die in dieser Dissertation verwendete Modellierungsart eingegangen werden. Sie basiert auf der sogenannten merkmalorientierten Domänenanalyse (engl. Feature-Oriented Domain Analysis (FODA)) nach Kang et al. [67], welche z. B. im Rahmen des Scopings vorgenommen wird. Üblicherweise verwendet man dazu *Merkmalmodelle* (auch *Feature-Bäume* oder *Feature-Modelle* genannt). Deren Merkmale bezeichnen Funktionen, die die Varianten einer Software-Produktlinie besitzen (können). Abbildung 2.4 zeigt ein beispielhaftes Merkmalmodell für eine Software-Produktlinie von Assistenten für autonomes Einparken aus [116].

Die Wurzel eines Merkmalmodells bildet stets die Produktvariante selbst, also hier die Variante eines Parkassistenten. Alle weiteren Knoten des Baums stellen Merkmale dar, die wiederum in Submerkmale unterteilt sein können. So besitzt etwa das Merkmal *Abstand* als Submerkmale Sensoren, die die Abstände des einzuparkenden Fahrzeugs von Hindernissen vor, hinter oder neben dem Fahrzeug messen. Ein ausgefüllter Kreis an einem Merkmal stellt ein *notwendiges* Merkmal einer Produktvariante dar, das ausgewählt werden *muss*, wenn das übergeordnete Merkmal ausgewählt ist. In Abbildung 2.4 muss beispielsweise jeder Parkassistent Aktuatoren, Sensoren und einen Steuerungsalgorithmus aufweisen, wohingegen die Simulationsstrecke optional ist, was durch einen offenen Kreis am Merkmal *Strecke* dargestellt wird. Merkmale, die über einen ausgefüllten Bogen miteinander verbunden sind, bilden eine *optionale* Gruppe, d. h. *mindestens ein* Merkmal ist auszuwählen. Um den rechtsseitigen Abstand eines Fahrzeugs von einem Hindernis zu messen, kann man zum Beispiel aus Sicherheitsgründen sowohl einen Infrarot- als auch einen Ultraschallsensor verwenden, benötigt jedoch mindestens

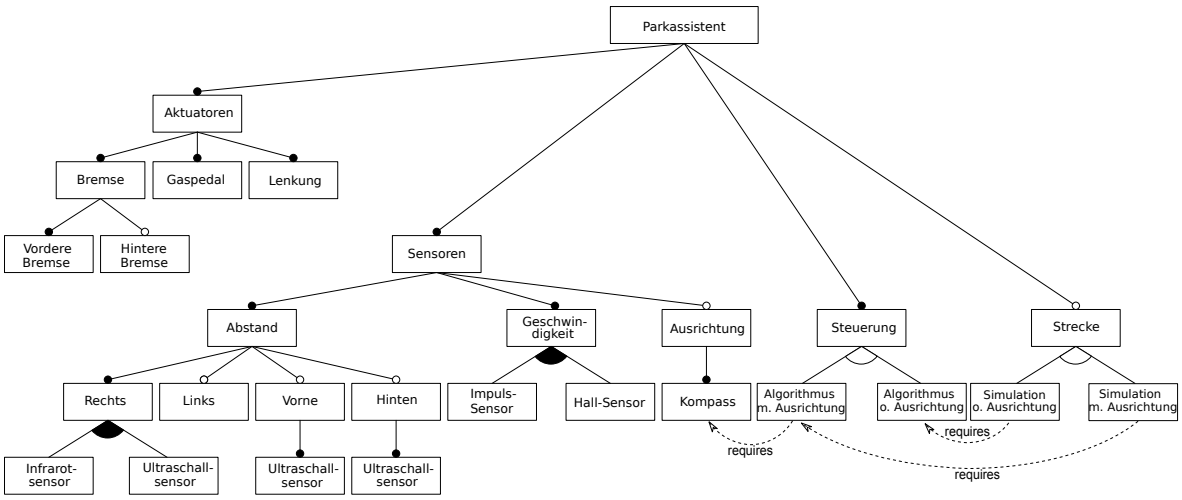


Abbildung 2.4: Ein Merkmalmodell für eine Produktlinie von Parkassistenten aus [116]

einen von beiden. Sind untergeordnete Merkmale über einen nicht ausgefüllten Bogen verbunden, so stellen sie eine *alternative* Gruppe dar, d. h. *genau ein* Merkmal der Gruppe muss ausgewählt werden. Ein Beispiel dafür ist die *Steuerung*, die entweder mit einem Algorithmus verfügbar ist, der auch die aktuelle Ausrichtung des Fahrzeugs berücksichtigt oder mit einem Algorithmus, der dies nicht kann. Wird ein Merkmal ausgewählt, so kann dies auch dann die Auswahl anderer Merkmale beeinflussen, wenn diese nicht in einer Eltern-Kind-Beziehung stehen. So erwartet beispielsweise einer der genannten Algorithmen die Information über die aktuelle Ausrichtung als Eingabe. Somit ist das Merkmal *Kompass* ebenfalls auszuwählen, welches diese Information zur Verfügung stellt. Im Merkmalmodell wird dies durch die *requires*-Relation ausgedrückt. Analog existiert noch eine *excludes*-Relation, die ausdrückt, dass sich zwei Merkmale gegenseitig ausschließen (nicht in Abbildung 2.4).

Neben der Variabilitätsmodellierung mit Merkmalmodellen existieren noch die Modellierung mit Entscheidungsmodellen, die Dhungana et al. beschreiben [30], sowie das orthogonale Variabilitätsmodell von Pohl et al. [23, 114]. Ein neueres Konzept stellt die Deltamodellierung von Schaefer et al. dar [130]. Da diese Modellierungstechniken im Rahmen der Dissertation jedoch keine Rolle spielen, sei der Leser diesbezüglich auf die referenzierte Literatur verwiesen.

2.3 Industrielle Entwicklung und Werkzeuge

Dieser Abschnitt beschreibt zunächst den üblichen Entwicklungsprozess für Steuergerätesoftware in der Automobilindustrie und erläutert dann, welche Werkzeuge für die einzelnen Phasen eingesetzt werden.

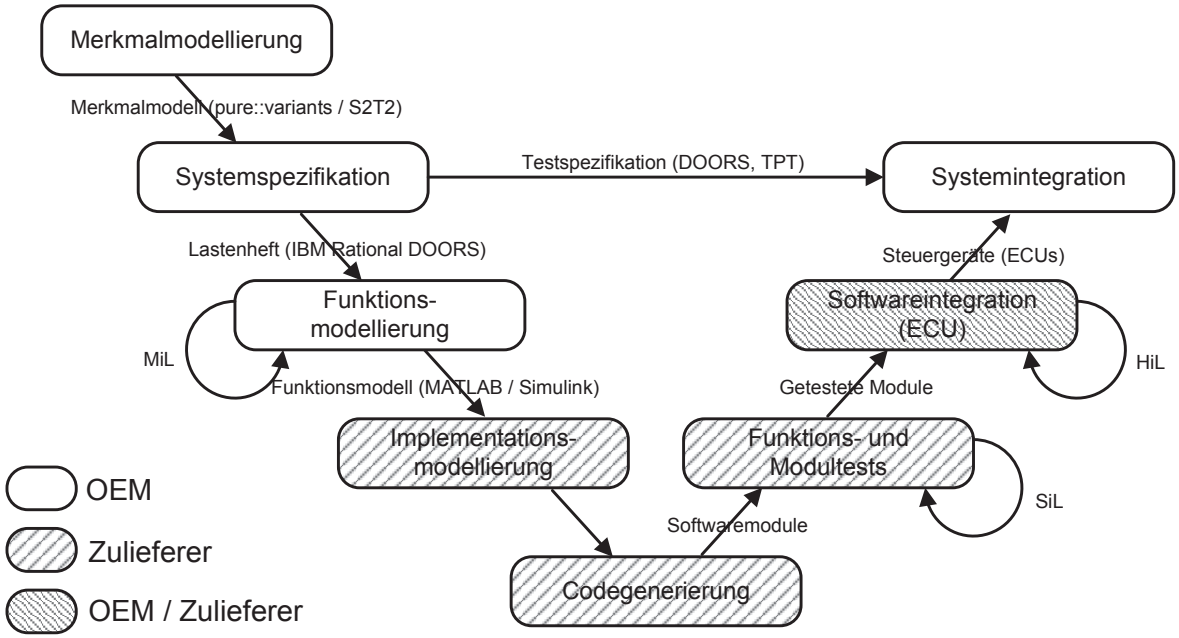


Abbildung 2.5: Industrieller Prozess modellbasierter Softwareentwicklung

2.3.1 Entwicklungsprozess

Die Entwicklung eingebetteter Softwareproduktlinien in der Automobilindustrie erfolgt nach dem auf Barry Boehm [15] zurückzuführendem V-Modell. Abbildung 2.5 veranschaulicht den Prozess und ist angelehnt an die Prozessbeschreibung der Daimler AG aus [90]. Zunächst werden die Merkmale der Produktlinie in einem Merkmalmodell spezifiziert (vgl. Abschnitt 2.2.5). Auf Basis dessen werden in der Systemspezifikationsphase die Anforderungen erfasst. Das resultierende Lastenheft in IBM Rational DOORS beschreibt die Funktionalität der Produktlinie und dient als Grundlage für die anschließende Funktionsmodellierung in MATLAB/Simulink. Das Funktionsmodell stellt ein formales Modell der gewünschten Funktionalität dar und dient gemäß dem Ansatz überlagerter Varianten (vgl. Abschnitt 2.2.4) als 150%-Modell bzw. Modell-Template, beschreibt also ebenfalls die gesamte Produktlinie. Technische Details über die Abbildung von Variabilität im Simulink-Modell werden in Abschnitt 2.3.3 beschrieben. Außerdem kann durch Bereitstellung entsprechender Schnittstellen im Funktionsmodell die Kompatibilität der generierten Software mit dem AUTOSAR-Standard [8] sichergestellt werden [131, 150].

Für die Ableitung einer Variante wird das 150%-Modell variantenspezifisch konfiguriert und in *Model-in-the-Loop-Tests (MiL)* so lange simuliert bis es korrekt ist und zur Implementierung an den Zulieferer übergeben werden kann. Diese frühzeitige Simulierbarkeit von Modellen stellt einen wesentlichen Vorteil in der Praxis dar, da dadurch die Anzahl kostspieliger, späterer Korrekturzyklen verringert werden kann. Der Zulieferer sorgt anschließend für die variantenspezifische Implementationsmodellierung und Codegenerierung und testet in sogenannten *Software-in-the-Loop-Tests (SiL)*

Artefakt	Elemente	Werkzeug
Merkmalmodell	Merkmal	S2T2 [79]
Lastenheft	Anforderung	IBM Rational DOORS [54]
Funktionsmodell	Subsystem	MATLAB/Simulink [149]
Testdokument	Test	IBM Rational DOORS, TPT [113]

Tabelle 2.1: Artefakte, ihre Elemente und Werkzeuge, aus denen sie stammen

Funktionen und Module. Schließlich integriert er die Software in die Steuergeräte (engl. Electronic Control Unit, ECU) und übergibt diese wiederum an den OEM. Der OEM führt sodann Komponententests (*Hardware-in-the-Loop, HiL*) durch. Schließlich erfolgen die Gesamtintegration ins Fahrzeug und entsprechende Fahrzeugintegrationstests. Für eine detailliertere Prozessbeschreibung sei der Leser auf [90, 131, 150] verwiesen. Eine genauere Beschreibung der MiL-, SiL- und HiL-Tests findet sich in [72].

2.3.2 Terminologie

Aus dem Entwicklungsprozess resultieren Begriffe, die für das Verständnis dieser Dissertation von Bedeutung sind. Deren Definitionen sind an die Terminologie der Diplomarbeit von Norbert Wiechowski [162] und die Arbeit von Cmyrev et al. [26] angelehnt.

Definition 2.3.1 ((Entwurfs-)Artefakt, Artefaktelement)
Unter einem **(Entwurfs-) Artefakt** versteht diese Dissertation Resultate der Entwicklungsschritte eingebetteter Software. Jedes Artefakt A besteht aus **Artefaktelementen** $a_i \in A$ und wird mit bestimmten Werkzeugen erstellt. Es lässt sich also vereinfacht als Menge von Artefaktelementen auffassen:

$$A = \{a_1, a_2, \ldots, a_n\}, n \in \mathbb{N}$$

Wir schreiben

- $M = \{m_1, m_2, \ldots, m_k\}, k \in \mathbb{N}$ für das Merkmalmodell (m_i: Merkmale),
- $R = \{r_1, r_2, \ldots, r_l\}, l \in \mathbb{N}$ für das Lastenheft (r_i: Requirements),
- $F = \{b_1, b_2, \ldots, b_m\}, m \in \mathbb{N}$ für das Funktionsmodell (b_i: Funktionsblöcke) und
- $T = \{t_1, t_2, \ldots, t_n\}, n \in \mathbb{N}$ für die Testspezifikation (t_i: Tests).

Tabelle 2.1 fasst zusammen, welche Artefakte (vgl. Definition 2.3.1) aus welchen Elementen bestehen und aus welchen Werkzeugen sie im Kontext dieser Dissertation stammen. Eine genauere Beschreibung der Artefakte erfolgt in Abschnitt 2.3.3.

Definition 2.3.2 (Clique)
Semantisch zusammengehörige Artefaktelemente $R' \subset R, F' \subset F, T' \subset T$ bezeichnen wir als **Clique**:

$$C = (R', F', T')$$

Da Artefakte dieselbe Software aus verschiedenen Blickwinkeln beschreiben, bestehen semantische Zusammenhänge zwischen ihnen bzw. ihren Elementen, die in dieser Dissertation gemäß Norbert Wiechowski [162] als Cliquen im Sinne von Definition 2.3.2 bezeichnet werden. Eine Anforderung zusammen mit den zugehörigen Tests und modellierten Funktionen bilden somit eine Clique.

Artefakte des Entwicklungsprozesses müssen hinsichtlich verschiedener Aspekte konsistent sein. Die erste Art der Konsistenz ist variantenunabhängig und wird durch Definition 2.3.3 beschrieben.

Definition 2.3.3 (Artefakt(in)konsistenz)
Artefakte $A_1, \ldots, A_n$ sind **artefaktkonsistent** genau dann, wenn alle dieselbe Funktionalität beschreiben bzw. umsetzen. Anderenfalls sind sie **artefaktinkonsistent**.

Artefaktinkonsistenz liegt also beispielsweise dann vor, wenn nicht jede Anforderung durch die Testspezifikation abgedeckt ist.

Eine weitere Art der Konsistenz im Variantenmanagement bezieht sich auf die Merkmalverknüpfungen von Cliquen. Den Variantenbegriff klärt Definition 2.3.4, während Definition 2.3.5 beschreibt, wann eine Clique hinsichtlich der Merkmalverknüpfungen konsistent ist.

Definition 2.3.4 (Artefaktvarianten und Validität)
Setzt man den 150%-Ansatz (vgl. Abschnitt 2.2.4) für alle Artefakte A voraus, so ist eine Artefaktvariante $V(A)$ aufzufassen als Teilmenge der Artefaktelemente von A:

$$V(A) \subseteq A, A \in \{M, R, F, T\}$$

Eine Variante ist **valide** genau dann, wenn sie dem Merkmalmodell (vgl. Abschnitt 2.2.5) der Produktlinie entspricht, d. h. wenn die ausgewählten Artefaktelemente die Merkmalabhängigkeiten berücksichtigen.

Definition 2.3.5 (Merkmalkonsistenz)
Sei

$$f : A \to \mathcal{P}(M), A \in \{R, F, T\}$$

die Zuordnung einer Menge $M' \in \mathcal{P}(M)$ von Merkmalen zu einem Artefaktelement $a \in A$. Ferner sei

$$M_A := \bigcup_{a \in A} f(a)$$

die Menge aller Merkmale, die mit den Elementen eines Artefaktes A der Clique verknüpft sind. Dann ist die Clique $C = (R', F', T')$ genau dann merkmalkonsistent, wenn gilt:

$$M_{R'} = M_{F'} = M_{T'}$$

Artefakte sind merkmalkonsistent genau dann, wenn alle Cliquen ihrer Elemente merkmalkonsistent sind.

Bemerkung 2.3.1
Für die Begriffe artefaktkonsistent und merkmalkonsistent verwendet diese Dissertation gleichermaßen die Kurzform **konsistent**. Eine Ausnahme bilden nur Fälle, in denen sich die jeweils gemeinte Konsistenzart nicht aus dem Kontext erschließen lässt.

2.3.3 Werkzeuge

Merkmalmodellierung mit S2T2

Für die Merkmalmodellierung einer Software-Produktlinie werden Merkmalmodelle eingesetzt. S2T2 vom Irish Software Engineering Research Centre [79] ist dazu ein geeignetes Werkzeug. Es ermöglicht auch die Konfiguration einer Variante und prüft bereits währenddessen ihre Konformität mit den im Merkmalmodell modellierten Constraints. Das Merkmalmodell wird von S2T2 als Modell des Eclipse Modeling Frameworks (EMF) [38] abgelegt und dient als Eingabe für den in Kapitel 6 vorgestellten Realisierungsansatz[1].

Anforderungsmanagement mit IBM Rational DOORS

Der Industriepartner, mit dem die der Dissertation vorangegangenen Projekte umgesetzt wurden, setzt zum Erfassen der Anforderungen IBM Rational DOORS ein. Damit erstellte Lastenhefte gliedern sich in Module, die die Anforderungen tabellarisch beschreiben. Einen Ausschnitt aus einem beispielhaften Lastenheft für den Parkassistenten aus Abbildung 2.4 zeigt Tabelle 2.2. Die dort erfassten Anforderungen werden anstelle des Original-Lastenhefts in DOORS als CSV-Datei exportiert.

Die linke Spalte beschreibt die Gliederung des Lastenheftes in Kapitel (z. B. 1), Unterkapitel (z. B. 1.1 oder 1.1.1) und konkrete Anforderungen. Letztere sind in IBM

[1] Es sei darauf hingewiesen, dass beim Industriepartner stattdessen das kommerzielle pure::variants [122] eingesetzt wird.

Hierarchieebene	Überschrift	Beschreibung
1	Sicherheit	
1.1	Sensorik	
1.1.1	Ausrichtung	
1.1.1.0-1		Um die Ausrichtung des Autos zu bestimmen, soll es in der Luxus-Variante einen Kompass geben. Andere Varianten müssen diesen Sensor nicht besitzen.
1.1.2	Geschwindigkeitssensoren	
[. . .]		

Tabelle 2.2: Anforderungen in IBM Rational DOORS

Rational DOORS mit einem an die Hierarchieebene angehängten Strich gefolgt von einer Nummer gekennzeichnet (z. B. 1.1.1.0-1).

Modellbasierte Software-Produktlinien-Entwicklung mit MATLAB/Simulink

Eingebettete Systeme sind häufig regelungstechnischer Natur, d. h. auf Basis gemessener Sensorwerte werden Aktuatoren so beeinflusst, dass eine bestimmte Ausgangsgröße auf einem bestimmten Niveau gehalten bzw. auf ein bestimmtes Niveau gebracht wird. Besonders in der Automobilindustrie setzt sich daher die Entwicklung mit MATLAB/-Simulink immer mehr durch [2, 19, 21, 33, 35, 159]. UML und EMF spielen hier aktuell praktisch (noch) keine bzw. nur eine untergeordnete Rolle.

Funktionsweise MATLAB/Simulink ist eine domänenspezifische Sprache, die es ermöglicht, grafisch aus einer umfangreichen und erweiterbaren Bibliothek funktionaler und über Parameter im Verhalten beeinflussbarer Blöcke ein Blockschaltbild im regelungstechnischen Sinne zu modellieren. So wird auch programmiertechnisch weniger erfahrenen Anwendern die Entwicklung von Regelungsalgorithmen erleichtert, sowie die Codegenerierung für unterschiedliche Hardwareplattformen erlaubt. Die Blöcke realisieren dabei die Signalverarbeitung und logische sowie arithmetische Operationen, d. h. sie berechnen aus einer beliebigen Menge von Eingangssignalen ein oder mehrere Ausgangssignale oder dienen dem Routing von Signalen. Ein fiktives und absichtlich besonders einfach gehaltenes Modell zeigt Abbildung 2.6. In der Realität implementieren funktionale Blöcke auch regelungstechnisch relevante Übertragungsfunktionen, z. B. für die Realisierung von PID-Reglern aus Integratoren, Derivatoren und Verstärkern. Im Beispiel werden einfache, arithmetische Funktionen modelliert. Ebenfalls in Abbildung 2.6 erkennbar ist der rekursive Aufbau von Modellen durch die Kapselung von Funktionalität mit Hilfe sogenannter Subsysteme. So wird innerhalb von *Subsystem 1* wiederum mathematische Funktionalität umgesetzt (vgl. Einblick in *Subsystem 1* oben

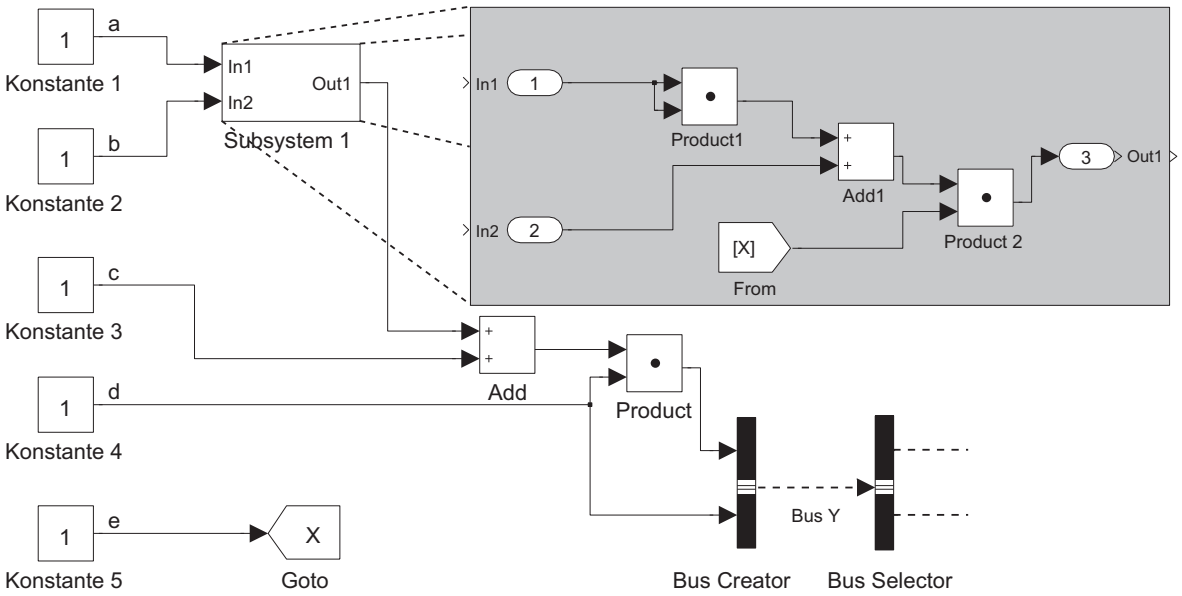


Abbildung 2.6: Graphische Spezifikation mathematischer Funktionen in Simulink-Syntax

rechts). Die abgerundeten Blöcke sind Schnittstellenblöcke (sogenannte *Inports* bzw. *Outports*), die keine eigene Funktionalität besitzen und nur das Ein- bzw. Ausgangssignal von außen nach innen bzw. von innen nach außen weiterleiten. Die spitz zulaufenden Blöcke besitzen ebenfalls keine Funktionalität, sondern beschreiben Sprünge innerhalb des Modells, die auch über verschiedene Subsysteme hinweg erfolgen können. Typisch für Simulink-Modelle ist schließlich die Bündelung von Signalen in Bussen, um die Anzahl an Verbindungslinien zwischen Blöcken gering zu halten und dem Anwender so einen besseren Überblick zu verschaffen. Ein Signalbus wird durch einen *Bus Creator* erzeugt und über einen *Bus Selector* wieder aufgelöst.

Aus einem so modellierten Blockschaltbild lässt sich schließlich lauffähiger Code für die zugrunde liegende Hardware durch Codegeneratorwerkzeuge wie *Simulink Coder* (ehemals *Real Time Workshop*) von Mathworks [145] oder *TargetLink* von dSpace [31] erzeugen.

Vergleicht man die modellbasierte Softwareentwicklung in MATLAB/Simulink mit den in Abschnitt 2.1 beschriebenen allgemeinen Konzepten der modellbasierten Softwareentwicklung, so ist das Metamodell für Simulink-Modelle implizit vorgegeben. Der Editor (Simulink) sorgt für die Konformität der erstellten Modelle mit diesem Metamodell. Die konkrete Syntax ist abhängig von der Programmiersprache und der Hardware, für die Code erzeugt werden soll, und wird erst durch die Codegeneratoren berücksichtigt, die die Modell-zu-Code-Transformationen vornehmen. So kann ein und dasselbe Modell für unterschiedliche Hardware verwendet werden. Eine weitere konkrete Syntax ist die textuelle Repräsentation des Modells in der zugehörigen MDL-Datei (vgl. Abbildung 5.6a).

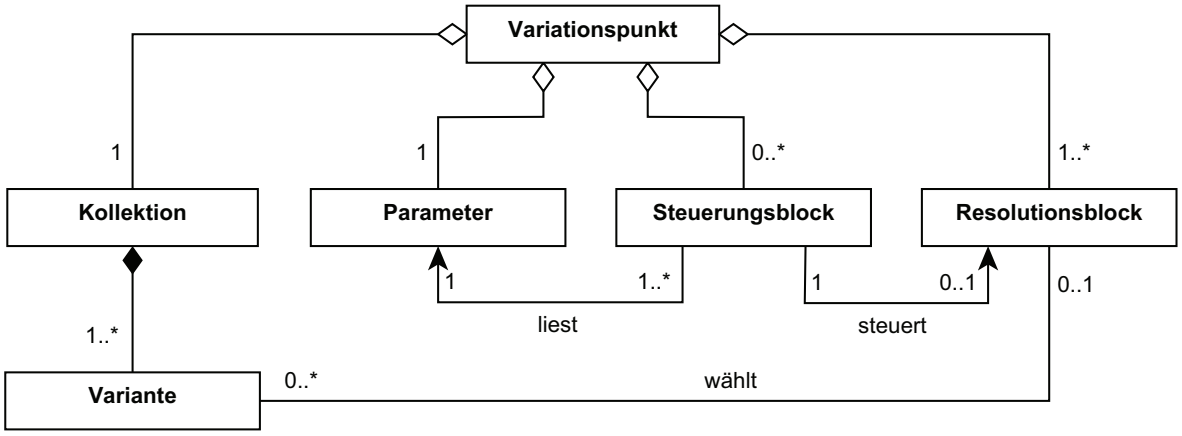
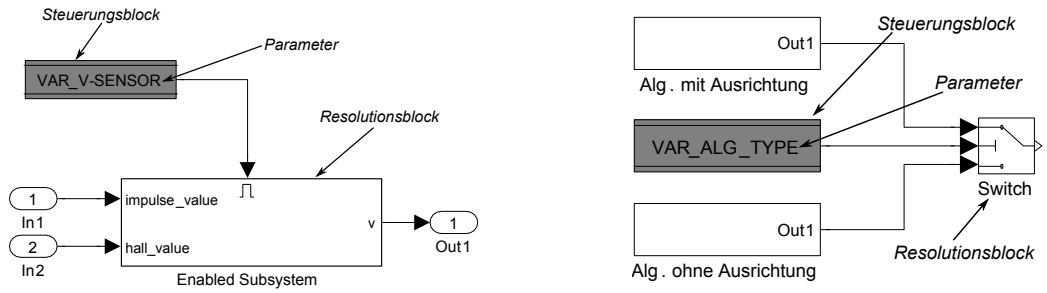


Abbildung 2.7: Variabilität in MATLAB/Simulink (angelehnt an [35])

Abbildung von Variabilität Aufgrund vieler Gemeinsamkeiten der eingebetteten Software folgt man häufig dem Ansatz der Software-Produktlinien-Entwicklung. Entsprechend Dziobek, Weiland und Richter basiert zum Beispiel der bei der Daimler AG angewandte Ansatz zur Variabilitätsmodellierung und -konfiguration eingebetteter Software auf dem Software-Produktlinien-Konzept und der Generativen Software-Entwicklung nach Czarnecki und Eisenecker [28], d. h. das 150%-Simulink-Modell-Template stellt die Vereinigung aller Modelle für die Softwarevarianten dar [35, 159]. Ein Vorteil dieses Ansatzes im Vergleich zum Ansatz positiver Variabilität liegt darin, dass nur ein Modell für die Produktlinie zu warten ist. Zudem lässt sich die Rückwärtskompatibilität neuer Modelle zu älteren Produktlinien leichter bewerkstelligen. Um das Modell zu einem 100%-Modell konfigurieren zu können, also Variabilität zu binden, müssen die Variationspunkte aus dem Merkmalmodell in das Simulink-Modell übertragen werden. Über Parameter können dann bestimmte Funktionalitäten aktiviert bzw. deaktiviert werden. Abbildung 2.7 beschreibt konzeptionell die Modellierung von Variationspunkten in MATLAB/Simulink. Demnach besitzt jeder Variationspunkt eine Menge von Modellvarianten, die zu einer *Variantenkollektion* zusammengefasst werden. Der Begriff *Variante* bezeichnet in dem hier betrachteten Kontext im Gegensatz zu Definition 2.3.4 einen an einem Variationspunkt gemäß Konfiguration zu aktivierenden bzw. zu deaktivierenden *Modellteil*. Um ein Modell für zukünftig geplante Varianten erweiterbar zu gestalten, ist nur mindestens *eine* anstelle von mindestens *zwei* Varianten mit einer Variantenkollektion assoziiert. Über den *Variantenparameter* kann eine Variante aus der Kollektion ausgewählt werden. Dazu wird der Parameter von einer Menge von *Steuerungsblöcken* in Simulink verarbeitet, die über ein entsprechendes Signal einen *Resolutionsblock* ansteuern, der entsprechend die Varianten aktiviert bzw. deaktiviert und so die Variabilität bindet. Da sich, z. B. bedingt durch Abhängigkeiten von Merkmalen, die Konfiguration an verschiedenen Stellen im Modell auswirken kann, kann es mehrere Kontroll- und Resolutionsblöcke für einen Variationspunkt geben.

Es existieren verschiedene Möglichkeiten, Variationspunkte in Simulink-Modelle zu integrieren. Dazu zählen spezielle Arten von Subsystemen sowie Blöcke, die logische



(a) Optionale Gruppe

(b) Alternative Gruppe

Abbildung 2.8: Optionale bzw. alternative Gruppen von Funktionen im Simulink-Modell (angelehnt an [35])

Gatter oder Signal-Routing implementieren. Ein an Dziobek et al. [35] angelehntes und auf den Parkassistenten aus Abbildung 2.4 übertragenes Beispiel für die Umsetzung einer optionalen Gruppe von Merkmalen in einem Simulink-Modell mit Hilfe eines sogenannten *Enabled-Subsystem* als Resolutionsblock zeigt Abbildung 2.8a. Der Steuerungsblock verarbeitet hier den Wert des Parameters *VAR_V-SENSOR*, der über eine Konfigurationsdatei, z. B. ein MATLAB-Skript, eingelesen wird. Dementsprechend sendet er ein Signal, das die selektierten Varianten beschreibt, an den Resolutionsblock *Geschwindigkeitsmessung*, der hier mit einem sogenannten *Enabled-Subsystem* realisiert ist. Innerhalb dieses Subsystems befinden sich die verschiedenen Varianten, also jene Modellteile, die die verschiedenen Merkmale des Variationspunktes umsetzen. Aus welchen der beiden Eingangssignale *impulse_value* und *hall_value* innerhalb des Subsystems das Ausgangssignal v berechnet wird, hängt also von der gewählten Variante ab.

Für die Abbildung alternativer Merkmale im Simulink-Modell bietet sich ein sogenannter *Switch*-Block als Resolutionsblock an (vgl. Abbildung 2.8b). Er setzt den Variationspunkt *Steuerung* aus dem Merkmalmodell in Abbildung 2.4 um. Hier verarbeitet der Steuerungsblock den Wert des Parameters *VAR_ALG_TYPE* und sendet ein boolesches Signal an den Switch-Block, der je nach Wert den oberen oder unteren Eingang zu seinem Ausgangssignal weiterleitet. Die verschiedenen Varianten sind dazu mit den beiden Eingängen des Switch-Blocks verbunden. Durch Kombination mit anderen Switch-Blöcken oder die Verwendung eines *Multi-Switch*-Blocks lässt sich somit aus einer Menge von Varianten genau eine auswählen.

Für weitere Modellierungsmöglichkeiten von Variationspunkten in MATLAB/Simulink sei der Leser auf [35] verwiesen.

Im Paradigma der generativen Programmierung von Czarnecki und Eisenecker [28] stellt das so modellierte generische Modell das Modell-Template dar, aus dem die Modelle für die Produktvarianten der Produktlinie abgeleitet werden. Dazu müssen jedoch die Varianten im Modell mit den von ihnen realisierten Merkmalen des Merkmalmodell verknüpft werden. Dazu bietet sich das Werkzeug pure::variants [13, 122] an. Auf Basis einer variantenspezifischen Merkmalkonfiguration können dann die nötigen Parameter für das Modell-Template gesetzt und so die Konfiguration auf dieses übertragen werden, um das produktspezifische Modell zu erzeugen [35, 150]. Ein Nachteil dieses Ansatzes

liegt insbesondere in der hohen Komplexität der 150%-Modell-Templates. Details zu diesem Aspekt beschreibt Kapitel 3.

Kubica beschreibt eine alternative Umsetzung des Software-Produktlinien-Ansatzes in MATLAB/Simulink bei der Audi AG [72]. Der Problemraum wird auch hier gemäß Czarnecki und Eisenecker (vgl. Abbildung 2.3) vom Merkmalmodell aufgespannt. Der Lösungsraum besteht jedoch nicht aus einem 150%-Modell, sondern setzt sich aus Modellen zusammen, die bestimmte Merkmale implementieren. Bei der Ableitung einer Variante werden diese Modelle gemäß Konfiguration zusammengefügt und um notwendige Schnittstellen ergänzt. Dieses Vorgehen sorgt dafür, dass dadurch, dass es keine deaktivierten Modellteile eines 150%-Modells gibt, weniger Code generiert und somit weniger Speicherplatz auf dem Steuergerät benötigt wird. Andererseits wirft dieser Ansatz u. a. Schwierigkeiten bei der Wartung und Aktualisierung der Modellbibliothek auf, da Änderungen in verschiedenen Modellen auf unterschiedliche Weise umgesetzt werden müssen. Somit wird der durch Wiederverwendung gewonnene Nutzen durch höhere Wartungsaufwände eingeschränkt [72, S. 32]. Da dieser Ansatz im weiteren Verlauf der Dissertation nicht weiter betrachtet wird, wird der Leser für Details über diesen Ansatz auf die referenzierte Literatur verwiesen.

Testspezifikation mit TPT

Time Partition Testing (TPT) [113] ist ein Werkzeug für das modellbasierte Testen eingebetteter Systeme basierend auf deren Ein- und Ausgangssignalen. Es lässt sich für die MiL-, SiL- und HiL-Tests (vgl. Abschnitt 2.3.1) anwenden und ist somit bereits frühzeitig im Entwicklungsprozess einsetzbar. Tests werden hier graphisch als hybride Zustandsautomaten spezifiziert, die das zu testende Simulink-Modell stimulieren. Neben der Spezifikation werden auch die Testdurchführung, -auswertung und -dokumentation unterstützt. TPT ist echtzeitfähig und ISO-26262-zertifiziert, wodurch es insbesondere in sicherheitskritischem Kontext angewendet werden kann. Auch lässt es sich in MATLAB/Simulink und IBM Rational DOORS integrieren und kann für den Test von AUTOSAR-Applikationen verwendet werden. Aufgrund von Variabilität des zu testenden Systems können auch Zustände des Automaten Varianten aufweisen, z. B. aufgrund von unterschiedlichen Eingangssignalen je nach Variante. Für eine detailliertere Beschreibung sei der Leser jedoch auf die Webseite der Werkzeugs [113] verwiesen, da für das Verständnis dieser Dissertation das Wissen um die Modellierungsart und den Zusammenhang zwischen Variabilität und Automat ausreicht.

3 Herausforderungen

Der in Abschnitt 2.3.1 vorgestellte Entwicklungsprozess adressiert bereits Anforderungen hinsichtlich Effizienz- und Qualitätsaspekten durch das angewandte Entwicklungsparadigma sowie durch frühzeitige Simulations- und Testmöglichkeiten. Dennoch offenbart er weitere Herausforderungen in Bezug auf die Analyse von Abhängigkeiten und Zusammenhängen sowohl innerhalb ein und desselben Artefakts als auch über Artefaktgrenzen hinweg. Außerdem ergeben sich Anforderungen an ein konsistentes Varianten- und Merkmalmanagement sowie an einheitliche und dadurch automatisch analysierbare Dokumentation.

3.1 Abhängigkeiten und Zusammenhänge

Der Funktionsumfang und die kontinuierliche Weiterentwicklung der Software wirkt sich auf die Komplexität aller Artefakte sowie auf deren Zusammenhänge aus. Des Weiteren entstehen Abhängigkeiten, die die effiziente Umsetzung späterer Änderungen an der Produktlinie bzw. den Produkten sowie deren Wartung und funktionale Erweiterung erschweren.

3.1.1 Funktionsmodell

Insgesamt stellt die modellbasierte Softwareentwicklung mit MATLAB/Simulink in Kombination mit dem 150%-Ansatz einen erfolgversprechenden Ansatz für die Entwicklung großer, variantenreicher, eingebetteter Software dar. Durch den kontinuierlich zunehmenden Funktionsumfang eingebetteter Software im Automobilbereich wächst jedoch das 150%-Simulink-Modell stetig an. Da neue Funktionalität nicht direkt in allen Baureihen eingeführt wird, wächst dadurch gleichzeitig die Anzahl der Variationspunkte, die in das Simulink-Modell übertragen werden müssen. Andere Variationspunkte entfallen dadurch, dass Funktionalität, die zunächst nur in Luxusvarianten verfügbar ist, im Laufe der Zeit auch in tiefere Klassen Einzug hält. Auch variiert die zugrunde liegende Hardware. Dies resultiert in kontinuierlicher Evolution und Anwachsen des Modells. So ist zum Beispiel bei der Daimler AG ein Simulink-Modell mit ca. 30.000 Blöcken entstanden [150]. Auch können angewandte Entwurfspraktiken, die eigentlich der Komplexitätsreduktion dienen, in das Gegenteil umschlagen, wie auch das einfache Beispiel in Abbildung 2.6 veranschaulicht. So erschwert beispielsweise die Verwendung von Signalbussen und rekursive Schachtelung von Subsystemen die Identifikation von Zusammenhängen oder Abhängigkeiten innerhalb der Modelle, obwohl sie zunächst einmal der Übersichtlichkeit des Modells dienen. Die Verwendung der Sprungblöcke

Goto und *From* kann ebenfalls die Übersichtlichkeit erhöhen, z. B. wenn die alternativ notwendigen direkten Verbindungslinien Überhand nähmen. Jedoch ist es damit auch möglich, über die durch die rekursive Schachtelung von Subsystemen hergestellte Subsystemhierarchie hinweg Signale zu transportieren, ohne dass dies an den Schnittstellen der Subsysteme erkennbar ist. In Abbildung 2.6 realisiert beispielsweise *Subsystem 1* eine ternäre, arithmetische Funktion, die die Eingabesignale a, b und e verarbeitet. Signal e wird jedoch über einen Goto-Block in das Subsystem transportiert. Navigiert der Anwender nicht explizit in das Subsystem hinein, so ist es wahrscheinlich, dass er annimmt, *Subsystem 1* realisiere eine binäre Funktion mit den Eingabesignalen a und b. Die Abhängigkeit des Ausgangssignals von Signal e wird er vermutlich erst durch eventuelle Fehlermeldungen während der Simulation erkennen. Analog zu Sprungblöcken können auch verschiedene Steuerungsblöcke für denselben Variationspunkt an verschiedenen Stellen in unterschiedlichen Rekursionstiefen (bezüglich Subsystemen) im Simulink-Modell liegen. Dies ist der Tatsache geschuldet, dass die Aktivierung eines Merkmals bzw. der damit assoziierten Modellteile auch die automatische Aktivierung (im Falle von *requires*-Beziehungen) bzw. Deaktivierung (im Falle von *excludes*-Beziehungen) eines anderen Merkmals bzw. eines anderen Modellteils erfordert, die aber einem anderen Variationspunkt zugeordnet ist und somit in einem anderen Teilbaum des Modells (hinsichtlich der Subsystemrelation) liegt. Wirkzusammenhänge in einem 150%-Modell zu erkennen, stellt somit eine Herausforderung für Entwickler im Rahmen der (Weiter-) Entwicklung einer Software-Produktlinie dar [34, 150].

3.1.2 Lastenheft

Das Lastenheft beschreibt die Anforderungen an die gesamte Produktlinie. Es stellt also analog zum 150%-Simulink-Modell die Vereinigung aller Anforderungen an die Einzelprodukte dar. Evolutionsbedingt ist das Lastenheft beim Industriepartner im Laufe der Zeit auf mittlerweile ca. 2.000 Anforderungen angewachsen [150]. Neben textueller Beschreibung in natürlicher Sprache sind auch Berechnungsvorschriften für Signale hier spezifiziert. Dadurch, dass Signale aus anderen Signalen berechnet werden, folgen Zusammenhänge zwischen verschiedenen Anforderungen. Das häufig verwendete Werkzeug IBM Rational DOORS erlaubt zwar die Verknüpfung von Anforderungen im Lastenheft, jedoch muss diese manuell erfolgen. Auch geht die Übersichtlichkeit des Lastenheftes mit zunehmender Größe verloren.

Eine weitere Art der Inkonsistenz resultiert aus der Konfiguration des Lastenhefts für einzelne Varianten. So werden zum Beispiel die Schnittstellen von Modulen[1] ebenfalls im Lastenheft beschrieben. Nach der Konfiguration kann es aber passieren, dass bestimmte Signale, die in der Schnittstellenbeschreibung existieren, nicht mehr vorhanden sind, da das Merkmal und somit die Anforderung, welche das Signal spezifiziert, deaktiviert wurde. Somit entsteht durch die Konfiguration ein unvollständiges Lastenheft bzw. das Lastenheft wird in sich inkonsistent.

[1] Dies sind spezielle Arten von Subsystemen, die je ein bestimmtes Merkmal implementieren.

Schließlich ist oft nicht offensichtlich, welche Anforderung von welchen Teilen des Simulink-Modells umgesetzt wird. Die Identifikation des Zusammenhangs zwischen Anforderungen und Funktionsmodell ist also schwierig. Sie wird zusätzlich erschwert durch solche Signale, welche im Simulink-Modell rein technisch bedingt vorhanden, aber daher nicht im Lastenheft erwähnt sind, sowie durch inkonsistente Signalbenennungen in Lastenheft und Simulink-Modell.

3.1.3 Artefaktübergreifende Zusammenhänge und Abhängigkeiten

Die zuvor genannten Zusammenhänge beziehen sich primär auf einzelne Artefakte. Eine weitere Herausforderung betrifft die Verknüpfungen zwischen den Artefakten. Diese ist oft nur den Entwicklern bekannt, wodurch der Einarbeitungsaufwand für neues Personal unnötig steigt und die Effizienz der Weiterentwicklung begrenzt wird, da z. B. bei einer Anforderungsänderung nicht mehr offensichtlich ist, welche Modellteile davon betroffen sind. Zwar lassen sich Anforderungen aus IBM Rational DOORS mit Blöcken in MATLAB/Simulink verknüpfen, jedoch können konstruktionsbedingt auch Modellteile davon betroffen sein, die wegen Signal- oder Merkmalabhängigkeiten von den verknüpften Blöcken abhängen. Zudem müssen die Verknüpfungen explizit manuell erstellt werden. *Automatische, artefaktübergreifende* Analysen, die auch indirekte Abhängigkeiten berücksichtigen, sind daher zumeist nicht möglich, da dazu Werkzeuggrenzen überwunden werden müssen und eine artefaktübergreifende Datenbasis fehlt, auf der derartige Analysen automatisiert werden können. Auch lassen sich Auswirkungen von Änderungen auf verschiedene Produktvarianten daher nicht leicht automatisch analysieren. Da sich somit Änderungsauswirkungen nur schwer analysieren lassen, kann auch der Aufwand, Änderungen oder neue Funktionalität umzusetzen, nicht bzw. nur schwer bestimmt werden.

3.2 Inkonsistenz

Schließlich wird die konsistente Umsetzung von Änderungsanträgen durch die isolierte Bearbeitung verschiedener Artefakte und die fehlende integrative Datenbasis erschwert. Dadurch kommt es zu Artefaktinkonsistenz gemäß Definition 2.3.3. Im Falle einer Software-Produktlinie kann dabei zudem Merkmalinkonsistenz im Sinne von Cmyrev et al. [26] (vgl. Definition 2.3.5) verursacht werden. Wie Abbildung 3.1 veranschaulicht, kann es beispielsweise passieren, dass eine Anforderung A mit genau zwei Testfällen T_1 und T_2 verknüpft ist, die Testfälle aber nicht alle Merkmale testen, die die Anforderung beschreibt. Die Anforderung A bezieht sich hier auf die Merkmale M_1, M_2 und M_3. Die mit der Anforderung verknüpften Tests hingegen beziehen sich auf die Merkmale M_1 und M_3. Hier ist also entweder die Clique nicht vollständig oder ihre Artefaktelemente sind nicht korrekt mit Merkmalen verknüpft. Verlässt man sich also auf die Merkmalverknüpfung, so besteht die Gefahr, dass die Anforderung A nicht vollständig getestet wird, wenn diese Inkonsistenz nicht behoben wird. Die Inkonsistenzen

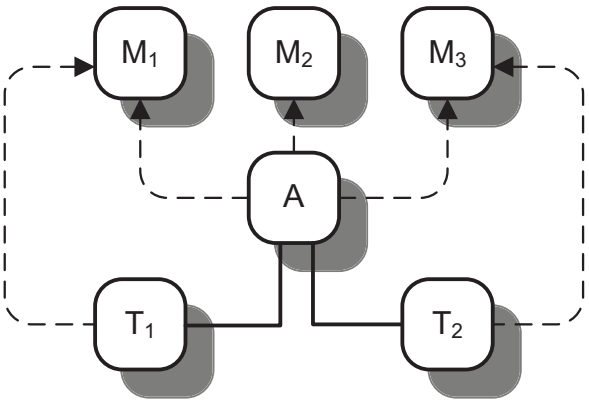


Abbildung 3.1: Beispiel für eine Inkonsistenz aufgrund unvollständiger Merkmalverknüpfung (durchgezogene Linie = Clique, gestrichelte Linie = Merkmalverknüpfung)

sind manuell zu identifizieren und dementsprechend teuer und aufwändig. Daher sollten die Identifikation und Auflösung von Inkonsistenzen (semi-) automatisch erfolgen.

3.3 Verteilte, uneinheitliche Dokumentation

Um Software verständlich zu halten, ist sorgfältige Dokumentation unerlässlich. In der Praxis ist diese jedoch oft unzureichend vorhanden. Dadurch steigen die Wartungs- und Evolutionskosten, die Sommerville für langlebige Softwaresysteme zum Beispiel drei bis viermal so hoch schätzt wie die eigentlichen Entwicklungskosten [137]. Des Weiteren bemerken Mens und Demeyer, dass etwa 50% der Entwicklungskosten nur durch das Verstehen von vorhandenem Code verursacht werden [83]. Bedingt durch die stetig steigende Komplexität sind auch Simulink-Modelle im Laufe der Zeit immer weniger selbsterklärend, wodurch der Bedarf nach Dokumentation auch hier gegeben ist. Somit ist oft kein oder zu wenig Wissen über Entwurfsentscheidungen und Gründe für frühere Änderungen vorhanden, wodurch die effiziente Weiterentwicklung von Systemen erschwert wird.

Teilweise werden solche Informationen beim Industriepartner bereits in das Simulink-Modell eingepflegt. Dazu werden spezielle Dokumentationsblöcke verwendet, die auf der Ebene des Subsystems, welches dokumentiert werden soll, in das Simulink-Modell eingefügt werden. Sie dienen ausschließlich der Dokumentation und sind für die Funktionalität irrelevant. Die dort abgelegte Information wird von den Entwicklern zum Großteil als Freitext annotiert.

Dieses Vorgehen verursacht zwei Probleme. Einerseits liegen diese Informationen zwar genau dort, wo sie relevant sind, wodurch Entwickler diese nicht in einem Gesamtdokument suchen müssen. Jedoch gelangt der Entwickler an diese Informationen auch nur, wenn er durch das Modell in das betroffene Subsystem navigiert. Dies impliziert, dass er bereits weiß, wo die ihn interessierenden Informationen zu finden sind. Aufgrund der Komplexität und der Abhängigkeiten des Modells (vgl. Abschnitt 3.1.1) ist dies jedoch

```
11.02.2010  - YD -  v0.1
initial version in 999, R 1.0
~
16 Apr 2010 - JT -  Version 1.0 line 999, Release R2.0~
~
- 11.02.2011  - YD -  1.1
added new Signal X ~
=> LOP1222
added new Signal Y ~
=> CR101, LOP1234~
~
- 10.03.2011  - YD -  1.1.1
Signal X changed
=> LOP2345
~
- 16-APR-2011 - JT -  Version 1.2 - line 999, Release R3.0
--> Added new Subsystem A~

[...]
```

Abbildung 3.2: Ein fiktives Beispiel für einen Dokumentationsblock in TargetLink

nicht immer der Fall. Es ist also für den Entwickler nicht möglich, die Informationen zentral abzufragen.

Andererseits wird durch Freitextdokumentation dieselbe Semantik in verschiedener Syntax beschrieben. So finden sich in der beispielhaften Dokumentation eines Subsystems in Abbildung 3.2 verschiedene Datumsformate, verschiedene Versionsspezifikationen und Angaben zu von Änderungen betroffenen Baureihen (*line*). Die Abkürzung *CR* steht für *Change Request*, also einen Änderungsantrag, *LOP* für *Liste offener Punkte*, also diejenigen Aufgaben, die noch zu erledigen sind, während ein Änderungsantrag eine größere Änderung am Gesamtsystem beschreibt und damit mehrere offene Punkte umfasst. Derartige Freitextinformationen lassen sich nur schwer automatisiert verarbeiten, z. B. um abzufragen, welche Subsysteme im Rahmen eines Änderungsantrags an einem bestimmten Datum modifiziert wurden und für welche Baureihen diese relevant sind.

3.4 Fazit

Um die zuvor genannten Herausforderungen zu adressieren, sollten Entwickler durch ein Werkzeug unterstützt werden, das automatisierte Analysen von Abhängigkeiten und die zentrale Abfrage von Dokumentation ermöglicht. Für Simulink-Modelle müssen strukturelle Analysen ermöglicht werden, um Sichten auf diese zu erstellen und dem Anwender so Hilfestellung für die Beherrschung der Komplexität zu geben. Damit die Dokumentation auch für die implizite Verknüpfung von Artefakten verwendet werden kann, muss diese artefaktübergreifend einheitlich angewendet werden. Zwecks Sicherstellung von Merkmalkonsistenz der Artefakte ist zudem die automatische Identifikation sowie die (semi-) automatische Auflösung von Merkmalinkonsistenz zu ermöglichen. Da aufgrund der Dynamik des Entwicklungs- und Evolutionsprozesses zu erwarten ist, dass im Laufe der Zeit weitere Analysen automatisiert werden müssen, ist das Werkzeug

so zu gestalten, dass neue Analysen möglichst komfortabel unter Wiederverwendung vorhandener Funktionalität vom Anwender implementiert werden können.

4 Lösungsmethodik

Um die Anforderungen aus Abschnitt 3.4 zu erfüllen, ist es sinnvoll, die Artefakte in ein gemeinsames Datenmodell zu integrieren. Das Datenmodell sollte nicht nur die Artefakte selbst beschreiben, sondern auch Verknüpfungen zwischen ihnen ermöglichen, um auf diese Weise Artefaktgrenzen zu überwinden. Auch kann ein solches integratives Datenmodell helfen, die Informationen, mit denen Artefakte zu Dokumentationszwecken angereichert werden, zu vereinheitlichen und zur artefaktübergreifenden Verwendung zur Verfügung zu stellen. Dadurch werden die Artefakte im Datenmodell im Laufe der Evolution mit immer mehr Informationen angereichert, wodurch implizit die Anzahl indirekter Verknüpfung zwischen ihnen zunimmt. Das auf diese Weise stetig wachsende Datenmodell ermöglicht so die Automatisierung aller Analysearten. Um ein Analysewerkzeug erweiterbar zu gestalten, sollte zusätzlich eine Menge vordefinierter Funktionalität zur Verfügung gestellt werden. Idealerweise ergibt sich daraus eine Art Bibliothek, mit deren Funktionen sich weitere Analysen komfortabel umsetzen lassen.

In diesem Kapitel werden zunächst ein ideales und ein pragmatisches Lösungskonzept für die Artefaktintegration vorgestellt. Letzteres orientiert sich stärker an industriellen Bedürfnissen. Nachdem ein Annotationskonzept erläutert wurde, welches der artefaktübergreifenden, einheitlichen Dokumentation und somit der impliziten Verknüpfung von Artefakten dient, werden Methodiken für die Identifikation und die semi-automatische Auflösung von Merkmalinkonsistenzen beschrieben. Schließlich erfolgt eine Einordnung der Konzepte in den aktuellen Stand der Forschung und Praxis.

4.1 Integrationskonzepte

Im Folgenden werden zwei Konzepte vorgestellt, die sich für die genannten Herausforderungen anbieten. Ersteres ist zwar wünschenswert, aber für die industrielle Praxis ungeeignet. Daher wird es nur der Vollständigkeit halber vorgestellt. Die im Rahmen der Dissertation realisierten Ansätze orientieren sich am zweiten, pragmatischen Konzept.

4.1.1 Ideales Konzept

Ein ideales Konzept zur Adressierung der Herausforderungen aus Kapitel 3 abstrahiert von den tatsächlich verwendeten Werkzeugen und definiert zunächst unabhängig davon ein Datenmodell, welches alle Artefakte umfasst (vgl. Abbildung 4.1). Es beschreibt zum Beispiel die Struktur von Blockschaltbildern und Lastenheften sowie deren Verknüpfungen, d. h. die Daten*struktur* wird definiert. Wie diese Daten hingegen *verarbeitet* werden, obliegt den auf diesem Datenmodell operierenden Werkzeugen. Da diese hier

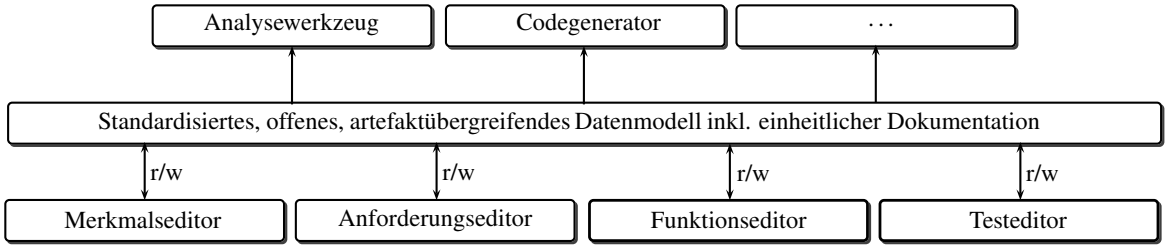


Abbildung 4.1: Das ideale Lösungskonzept

auf Grundlage des Datenmodells entwickelt werden, folgt das Konzept einem Bottom-Up-Ansatz. Der Vorteil liegt darin, dass das Datenmodell als Standard fungieren kann, ähnlich dem bereits für Anforderungen existierenden Requirements Interchange Format (ReqIF) [104], und die Werkzeugisolation so überwunden wird. Jedes Werkzeug, das dieses Datenmodell unterstützt, kann sich auf ein bestimmtes Artefakt fokussieren, kennt aber auch die Semantik anderer dort abgelegter Daten. Es existiert also nur noch ein Modell, auf dem alle Entwickler aus allen Phasen arbeiten. Die speziellen Werkzeuge aus den zugehörigen Phasen stellen dann lediglich eine Sicht auf das Modell dar (z. B. die Visualisierung eines Blockschaltbilds analog zu MATLAB/Simulink), die beliebig mit einheitlichen Meta-Informationen aus den anderen Phasen ergänzt werden können. Damit ist es nicht nur möglich, zielgerichtete Anfragen an das Datenmodell zu richten, sondern auch Werkzeuge beliebig auszutauschen und parallel zu verwenden, ohne dass eine Adaption vorhandener Artefakte erforderlich ist.

Beispiele für diese Werkzeuge sind die in Abbildung 4.1 genannten Artefakteditoren. Außerdem können verschiedene Generatoren Code aus dem Blockschaltbild generieren. Für diese Dissertation hingegen ist insbesondere das Analysewerkzeug von Interesse. Dieses kann auf Basis des Datenmodells entwickelt werden, sodass die tatsächlich verwendeten Werkzeuge für Analysen irrelevant werden. Schließlich können integrierte Entwicklungsumgebungen, ähnlich Eclipse, realisiert werden, die die zuvor genannten Editoren, Codegeneratoren und Analysewerkzeuge als Plugins integrieren.

4.1.2 Pragmatisches Konzept

In der industriellen Softwareentwicklung ist das ideale Konzept aus verschiedenen Gründen nicht praktikabel. So werden selten Systeme vollständig neu entwickelt. Vielmehr bauen neue Systeme häufig auf vorhandenen Systemen auf, die „nur“ weiterentwickelt werden. Da die Neuentwicklung auf Basis des gemeinsamen Datenmodells oft zu teuer wäre, müsste die Migration vorhandener Artefakte automatisiert erfolgen. Um diese fehlerfrei zu ermöglichen, wäre die Kooperationsbereitschaft der Werkzeughersteller erforderlich, da nur diese die proprietären Dateiformate im Detail kennen. Selbst wenn diese gegeben wäre, müssten die aktuellen Werkzeuge entweder an das Datenmodell angepasst werden, wenn man diese weiterhin verwenden will, oder es müssten neue Werkzeuge entwickelt werden, die auf dem neuen Datenmodell operieren. Somit wäre der initiale Aufwand groß und finanziell aus Sicht der Hersteller nicht wirtschaftlich.

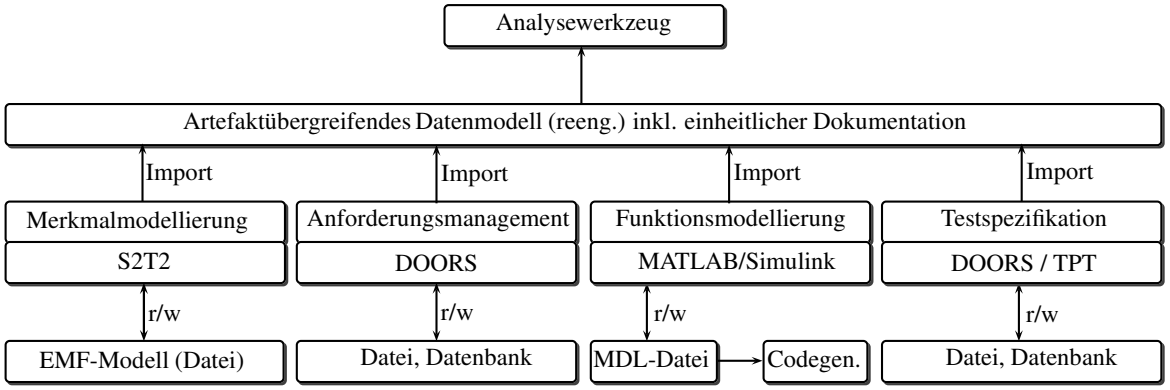


Abbildung 4.2: Das pragmatische Lösungskonzept

Schließlich kooperieren bereits heute verschiedene Werkzeuge miteinander. So lassen sich in MATLAB/Simulink Blöcke mit Anforderungen in IBM Rational DOORS verknüpfen. TargetLink stellt Blockbibliotheken und Codegeneratoren als Erweiterung für MATLAB/Simulink zur Verfügung, und TPT ermöglicht die Verknüpfung von Zustandsautomaten mit MATLAB/Simulink und IBM Rational DOORS.

Tabelle 4.1 fasst die Vor- und Nachteile des idealen Konzeptes im Hinblick auf praktische Aspekte zusammen.

Daher wurde für diese Dissertation ein pragmatisches Konzept entworfen, welches sich möglichst genau am idealen Konzept orientiert und zugleich beabsichtigt, zwar gängige Werkzeuge und ihre Schnittstellen zueinander beizubehalten, aber dennoch ein möglichst hohes Maß an Werkzeugunabhängigkeit zu wahren. Abbildung 4.2 visualisiert dieses Konzept.

Das artefaktübergreifende Datenmodell wird hier aus vorhandenen Artefakten heraus extrahiert und analog zum idealen Konzept zur Adressierung der in Abschnitt 3.3 genannten Herausforderungen um ein einheitliches Dokumentationskonzept ergänzt. Für die Funktionsmodellierung und Anforderungs- und Testspezifikation wurden die industriell gebräuchlichen Werkzeuge MATLAB/Simulink und IBM Rational DOORS zugrunde gelegt. Die Merkmalmodellierung erfolgt mit S2T2. Jedes Werkzeug besitzt sein eigenes Datenformat. Die Artefakte werden über die Import-Funktionalität in das Datenmodell integriert, das somit der Artefaktintegration dient und auf dem Analysewerkzeuge analog zum idealen Konzept aufsetzen. Die Analysen können also werkzeugunabhängig[1] auf dem gemeinsamen Datenmodell erfolgen. Auch wird die Informationsanreicherung der Artefakte über dieses Datenmodell vereinheitlicht, um implizit die Rückverfolgbarkeit zwischen Artefakten herzustellen. Abgesehen von der Informationsanreicherung arbeiten die Werkzeuge selbst jedoch unabhängig vom Datenmodell. So kann beispielsweise der Code weiterhin aus dem Simulink-Modell generiert werden anstatt, wie beim idealen Konzept, aus dem gemeinsamen Datenmodell. Ein Nachteil dieses Konzepts besteht darin, dass für jedes Werkzeug eine Schnittstelle zum Datenmodell erstellt werden

[1] in dem Sinne, dass die Werkzeuge nicht notwendig sind, um Analysen durchzuführen. Durch das Reengineering beeinflussen die verwendeten Werkzeuge das Datenmodell natürlich schon.

Pro	Contra
• Schaffen von Werkzeugunabhängigkeit • Struktur von Artefakten ist allen Werkzeugen bekannt • Generischer Ansatz, da die Funktionalität im Modell nur beschrieben wird; die Interpretation der deskriptiven Daten obliegt den darauf aufsetzenden Werkzeugen • Hauptsächlich leichtgewichtige Werkzeuge nötig, die jeweils nur einen kleinen Teil der zur Entwicklung nötigen Funktionalität umsetzen; idealerweise als Plugins beliebig zu umfassenden integrativen Rahmenwerken erweiterbar	• Entwicklung neuer Werkzeuge oder Anpassung von Werkzeugen an das Datenmodell nötig • Migration existierender Artefakte in das Datenmodell oder Neuentwicklung der Artefakte nötig • Kooperation von Werkzeugherstellern aus ökonomischen Gründen unwahrscheinlich • Bei Werkzeugwechsel Schulung von Mitarbeitern nötig; Verlust des Vorteils erworbener Erfahrung

Tabelle 4.1: Vor- und Nachteile des idealen Konzepts

muss. An dieser Stelle lässt sich also Werkzeugunabhängigkeit nicht realisieren. Jedoch ist auf diese Art und Weise immerhin pro Werkzeug nur eine Schnittstelle zu implementieren. Im herkömmlichen Ansatz (ohne gemeinsames Datenmodell) benötigt jedes Werkzeug Schnittstellen zu jedem anderen. Jedoch können dadurch auch inkonsistente Ausprägungen zwischen persistenten Originalartefakten (z. B. dem Simulink-Modell) und deren Repräsentation im Datenmodell verursacht werden. Folglich ist ein Synchronisationsmechanismus erforderlich, um diese zu vermeiden. Dieses Problems nimmt sich die vorliegende Dissertation jedoch nicht an. Gemäß dem idealen Konzept würde hingegen bereits die Persistierung über das gemeinsame Datenmodell erfolgen, wodurch diese Art von Inkonsistenz konstruktionsbedingt ausgeschlossen wäre und die separate Implementierung von Schnittstellen entfiele.

Da die Dissertation auf verschiedenen Kooperationen mit der Industrie beruht, war jedoch die Beibehaltung der aktuellen Werkzeuglandschaft von besonderem Interesse. Daher wurde das pragmatische Konzept umgesetzt.

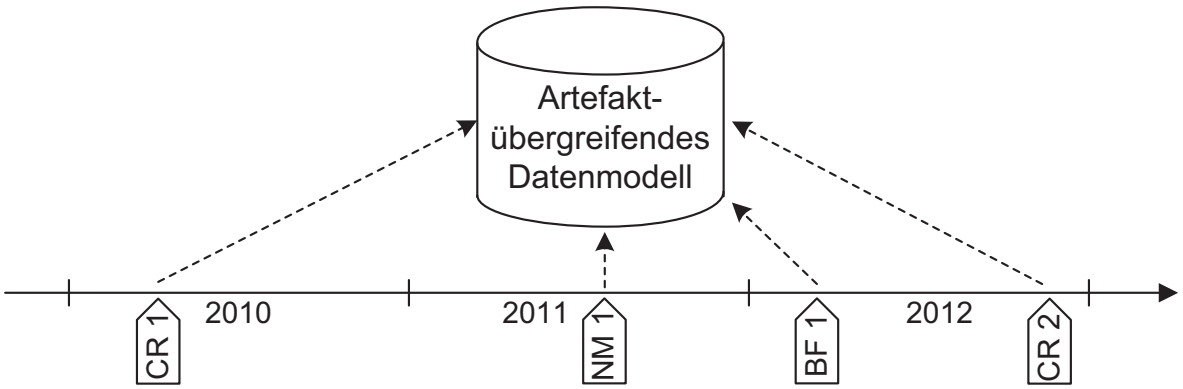


Abbildung 4.3: Integrationskonzept für Artefakte und Informationen über die Entwicklung und Evolution (CR = Change Request (Änderungsantrag), NM = Neues Merkmal, BF = Bugfix (Fehlerkorrektur))

4.2 Einheitliche Dokumentation

Um große Software-Produktlinien effizient weiterentwickeln zu können, ist es wichtig, Architekturentscheidungen, Zusammenhänge zwischen Artefaktmodifikationen und Änderungsanträgen sowie zwischen Artefakten und realisierten Merkmalen sorgfältig zu dokumentieren, da diese Informationen für die effiziente Einarbeitung späterer Änderungen relevant sein können. Möchte man beispielsweise eine früher festgelegte Architektur modifizieren, so ist es wichtig zu verstehen, weshalb bestimmte Entwurfsentscheidungen getroffen wurden, um die Auswirkung der Modifikation auf die Produktlinie besser abschätzen zu können.

Dabei ist es sinnvoll, derartige Informationen dort abzulegen, wo sie relevant sind, also beispielsweise im betroffenen Subsystem eines Simulink-Modells. Dies ist auch in der industriellen Entwicklung gängige Praxis (vgl. Abschnitt 3.3). Daher können auch verschiedene Subsysteme mit identischen Informationen annotiert sein. Beispielsweise betrifft in Abbildung 4.3 die Information, dass ein Subsystem im Rahmen des Änderungsantrags *CR 1* modifiziert wurde, nicht notwendigerweise nur ein einziges Subsystem. Ist man jedoch daran interessiert, welche Subsysteme im Rahmen eines Änderungsantrags modifiziert wurden (z. B. da dieser nicht korrekt umgesetzt wurde), so ist diese im Modell mehrfach an unterschiedlichen Stellen abgelegte Information mühsam vom Entwickler zusammenzutragen oder von vornherein in einem separaten Dokument festzuhalten. Möchte man nicht auf den Vorteil lokal gehaltener Informationen verzichten und die parallele Wartung eines Änderungsdokumentes vermeiden, so sollte es möglich sein, die Informationen über automatisierte Abfragen bei Bedarf aus dem Modell zu extrahieren. Dafür ist es jedoch wichtig, dass diese automatisiert auf semantische Gleichheit überprüft werden können. Dies ist für die heutzutage meist angewandte Freitextspezifikation nicht der Fall. Daher sollte eine in Syntax und Semantik vereinheitlichte Spezifikationsmethodik für diese Informationen angewendet werden, die im artefaktübergreifenden Datenmodell verankert wird.

Die Relevanz der Informationen beschränkt sich zudem nicht auf ein bestimmtes Artefakt. So können neben Subsystemen beispielsweise auch Anforderungen oder Tests von Änderungsanträgen betroffen sein. Daher bietet es sich an, auch diese in das in Abschnitt 4.1.2 vorgestellte artefaktübergreifende Datenmodell zu übernehmen, um sie dort mit den Artefakten zu verknüpfen. Dieses Konzept veranschaulicht Abbildung 4.3. Hier kommt es beispielsweise im Jahr 2010 zu einem Änderungsantrag *CR1*. Dieser wirkt sich potenziell auf alle Artefakte aus, die sich im gemeinsamen Datenmodell befinden. Jeder Artefaktverantwortliche annotiert die dabei modifizierten Artefaktelemente mit diesem Änderungsantrag in der vom Datenmodell vorgegebenen Syntax (Typ und Wert). Die Tatsache, dass dieselbe Annotation in derselben Syntax von Verantwortlichen anderer Artefakte vorgenommen wird, stellt implizit einen Zusammenhang zwischen den Elementen her. Im Laufe der Zeit ergeben sich durch weitere Informationsanreicherungen folglich immer mehr potenzielle Zusammenhänge zwischen Artefaktelementen, die für die Identifikation von Auswirkungen späterer Änderungsanträge nützlich sein können.

Zusammengefasst sollten anstelle der in Abschnitt 3.3 beschriebenen (Freitext-) Dokumentationsblöcke vereinheitlichte Annotationen verwendet werden, die dem Benutzer möglichst wenig Variationsspielraum für die Spezifikation von Informationen lassen, d. h. möglichst nur eine Spezifikationssyntax für dieselbe Semantik erlauben, und artefaktübergreifend verwendet werden können.

Der Annotationsbegriff orientiert sich an der Variablentypisierung in Programmiersprachen und ist definiert durch Definition 4.2.1.

Definition 4.2.1 (Annotation)
Unter einer **Annotation** A versteht diese Dissertation eine in Syntax und Semantik vereinheitlichte Information über den Entwicklungs- und Evolutionsprozess, die artefaktübergreifend verwendet werden kann und definiert ist durch

$$A = \{(t, v) \mid t \in T \wedge v \in V(t)\}$$

wobei T die Menge der zulässigen Annotations*typen* und $V(t)$ die Menge der für einen Annotationstyp $t \in T$ zulässigen *Werte* darstellt.

Die einheitliche Syntax stellt sicher, dass die Annotationen automatisiert ausgewertet werden können. Durch die artefaktübergreifende Verwendung können potenzielle Zusammenhänge zwischen Artefaktelementen besser identifiziert werden.

Abbildung 4.4 beschreibt das Konzept am Beispiel von Simulink-Modell und Lastenheft. Eine Annotation wird also einem Artefaktelement (hier Anforderung oder Block/Subsystem) zugewiesen. Jedem Artefakt können zulässige Annotationstypen zugeordnet werden (Assoziation *relevante Annotationstypen*), da nicht jede Art von Information für jedes Artefakt relevant sein muss.

Um dem Benutzer artefaktübergreifend dieselben Annotationstypen zur Verfügung zu stellen, werden diese ebenfalls im zentralen Datenmodell abgelegt und verwaltet. Die Verwendung anderer Annotationstypen als der im Datenmodell für ein Artefakt als zulässig ausgewiesenen Typen sollte ausgeschlossen sein, da diese anderen Werkzeugen nicht bekannt wären. Folglich wäre ihnen die Semantik nicht klar. Gleichzeitig sollte

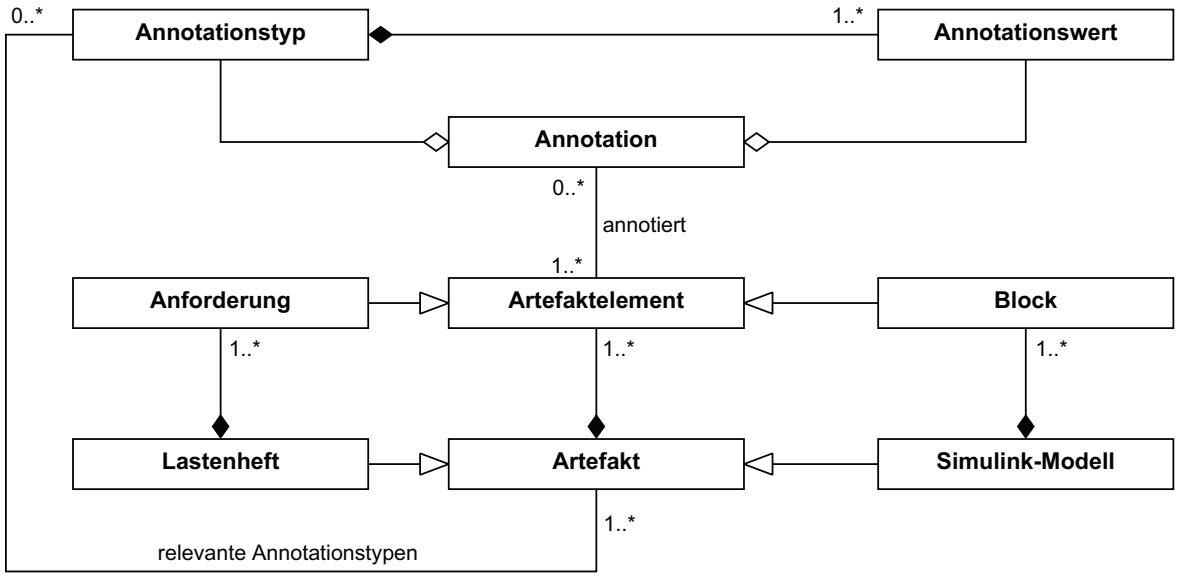


Abbildung 4.4: Annotationskonzept für einheitliche, artefaktübergreifende Anreicherung mit Informationen zu Dokumentations- und Analysezwecken am Beispiel der Artefakte Lastenheft und Simulink-Modell

die Menge zugelassener Annotationen bei Bedarf erweiterbar sein. Anderenfalls besteht die Gefahr, dass wichtige Informationen nicht festgehalten werden, da kein passender Annotationstyp dazu existiert. Die Einführung neuer Annotationstypen bedarf jedoch einer Abstimmung zwischen den Artefaktverantwortlichen, um Duplikate oder Widersprüche zu bereits existierenden Annotationstypen auszuschließen. Dabei muss für jeden Annotationstyp festgelegt werden, welche Werte gültig sind und für welche Artefakte er zulässig ist.

4.3 Automatisierte Analysen

Für die Handhabung der in Kapitel 3 beschriebenen Komplexität und die Identifikation von Abhängigkeiten und möglichen artefaktübergreifenden Zusammenhängen sind automatische Analysen der Artefakte erforderlich. Je nachdem, welche Fragestellung eine solche Analyse behandelt, können dazu unterschiedliche Arten von Sichten erzeugt werden. Um zum Beispiel dem Entwickler Überblick über ein Simulink-Modell zu verschaffen, ist es sinnvoll, bestimmte Teile, die die Fragestellung nicht betreffen, herauszufiltern. Andere Analysen dienen der Abfrage von Annotationen eines Artefakts, die speziellen Suchkriterien entsprechen, z. B., um einen Überblick über Subsysteme eines Simulink-Modells zu erstellen, die mit einem bestimmten Merkmal oder Änderungsantrag annotiert wurden.

Da auch artefakt*übergreifende* Analysen möglich sein sollen und um möglichst werkzeugunabhängig zu sein, sollte jede Analyse auf dem in Abschnitt 4.1.2 beschriebenen Datenmodell basieren. Dazu ist es sinnvoll, dieses strukturell so zu gestalten, dass häufig

zu identifizierende Eigenschaften leicht abfragbar sind und so eine möglichst einfache Implementierung effizienter, aussagekräftiger Analysen ermöglicht wird. Beispielsweise ist zu erwarten, dass Abhängigkeitsanalysen des Simulink-Modells die Information, welche Blöcke miteinander verbunden sind und welche Signale transportiert werden, häufig benötigen. Folglich ist es sinnvoll, derartige Informationen möglichst explizit im Datenmodell zu modellieren, sodass für spätere Analysen keine komplizierten Algorithmen mehr auszuführen sind.

Um schließlich auch den Anwender in die Lage zu versetzen, selbst Analysen zu implementieren, ist es wünschenswert, dass diesem vordefinierte Analysefunktionalitäten als Bausteine an die Hand gegeben werden.

4.4 Inkonsistenzen und ihre Auflösung

Um Artefaktkonsistenz im Sinne von Definition 2.3.3 zu prüfen, müssen Artefakte semantisch miteinander verglichen werden. Dies ist werkzeuggestützt aktuell praktisch nicht möglich, da dafür die Semantik natürlichsprachlicher Anforderungen und Tests algorithmisch erfasst und mit formalen Spezifikationen, z. B. dem Simulink-Modell, abgeglichen werden muss. Folglich ist eine manuelle Überprüfung durch einen Produktlinienexperten erforderlich. Dennoch kann dieser dabei durch ein Werkzeug unterstützt werden, welches zu diesem Zweck die Merkmalkonsistenz der Cliquen der zu prüfenden Artefakte untersucht. Vorausgesetzt, dass alle Artefaktelemente korrekt untereinander sowie mit den Merkmalen, auf die sie sich beziehen, verknüpft sind (d. h. die Cliquen und Merkmalverknüpfungen sind korrekt), können dabei identifizierte Merkmalinkonsistenzen (Definition 2.3.5) von Cliquen Aufschluss darüber geben, dass Artefakte nicht konsistent zueinander sind und auf welche Merkmale und Artefaktelemente die Inkonsistenz zurückzuführen ist.

Das Werkzeug, welches in Abschnitt 6.7 vorgestellt wird, prüft dazu die Merkmalkonsistenz. Im Falle von Merkmalinkonsistenz stellt ein Assistent dem Anwender Korrekturmaßnahmen zur Auswahl (daher *semi*-automatische Korrektur). Dieser entscheidet, welchen Vorschlägen er folgt und welchen nicht. Nach Anwendung dieses Assistenten sollten alle Cliquen und Merkmalverknüpfungen vollständig sein, sodass ggf. verbliebene Inkonsistenzen Aufschluss über Ursachen der Artefaktinkonsistenz geben.

Um Artefakte auf Merkmalkonsistenz zu überprüfen und ggf. zu korrigieren, wendet das im Rahmen der Diplomarbeit von Norbert Wiechowski [162] entstandene Werkzeug das mengentheoretische Konsistenzanalysekonzept von Cmyrev et al. [26] an, welches dazu auf Simulink-Modelle sowie um Kritikalitätsstufen von Inkonsistenzen erweitert und in das datenbankorientierte Rahmenwerk aus Kapitel 6 integriert wurde. Im Folgenden wird die zugrunde liegende Theorie der Merkmalkonsistenzprüfung und den vorgeschlagenen Korrekturmaßnahmen genauer erläutert.

Cmyrev et al. gruppieren Inkonsistenzen, die auf Verknüpfungen zwischen Artefakten und Merkmalen basieren, in verschiedene Kategorien, die simultan in der beispielhaften Clique aus Abbildung 4.5 zum Ausdruck kommen. Bevor diese definiert werden können,

werden die dafür zentralen Begriffe der *Merkmalbasis* und des *Bezugsmengenpaars* erläutert.

Definition 4.4.1 (Merkmalbasis)
Unter einer Merkmalbasis $B(C) \in \{R', F', T'\}$ einer Clique $C = (R', F', T')$ versteht man diejenige Menge der in C enthaltenen Elemente eines Artefaktes, die als korrekt mit Merkmalen verknüpft angenommen werden.

Definition 4.4.2 (Bezugsmengenpaar)
Die Menge $P_B(C)$ der Bezugsmengenpaare einer Clique $C = (R', F', T')$ mit Merkmalbasis $B(C) \in \{R', F', T'\}$ ist definiert als

$$P_B(C) = \{\langle B(C), A\rangle \mid B(C) \neq A \in \{R', F', T'\}\}.$$

Definition 4.4.3 (Kategorien von Merkmalinkonsistenzen)
Seien $C = (R', F', T')$ eine Clique und $B(C) \in \{R', F', T'\}$ ihre Merkmalbasis. Dann ist ein Bezugsmengenpaar $p = \langle B(C), A\rangle \in P_B(C)$ inkonsistent wegen

(i) Unvollständigkeit, wenn gilt

$$M_{B(C)} \setminus M_A \neq \emptyset \wedge M_{B(C)} \cap M_A \neq \emptyset,$$

(ii) Redundanz, wenn gilt

$$M_A \setminus M_{B(C)} \neq \emptyset \wedge M_{B(C)} \cap M_A \neq \emptyset,$$

(iii) Widerspruch, wenn gilt

$$M_{B(C)} \cap M_A = \emptyset \wedge |M_{B(C)} \cup M_A| \geq 1.$$

Die Mengen M_A und $M_{B(C)}$ sind definiert gemäß Definition 2.3.5.

Entsprechend lässt sich $P_B(C)$ einteilen in (nicht notwendigerweise disjunkte) Mengen von Bezugsmengenpaaren mit unterschiedlichem Konsistenzstatus:

- $U(P_B(C)) := \{p \in P_B(C) \mid p \text{ ist inkonsistent wg. Unvollständigkeit}\}$
- $R(P_B(C)) := \{p \in P_B(C) \mid p \text{ ist inkonsistent wg. Redundanz}\}$
- $W(P_B(C)) := \{p \in P_B(C) \mid p \text{ ist inkonsistent wg. Widerspruch}\}$
- $K(P_B(C)) := P_B(C) \setminus (U(P_B(C)) \cup R(P_B(C)) \cup W(P_B(C)))$ (Menge konsistenter Bezugsmengenpaare)

Eine Clique C ist inkonsistent wegen Unvollständigkeit (Redundanz, Widerspruch) genau dann, wenn ein Bezugsmengenpaar $p \in P_B(C)$ inkonsistent wegen Unvollständigkeit (Redundanz, Widerspruch) ist.

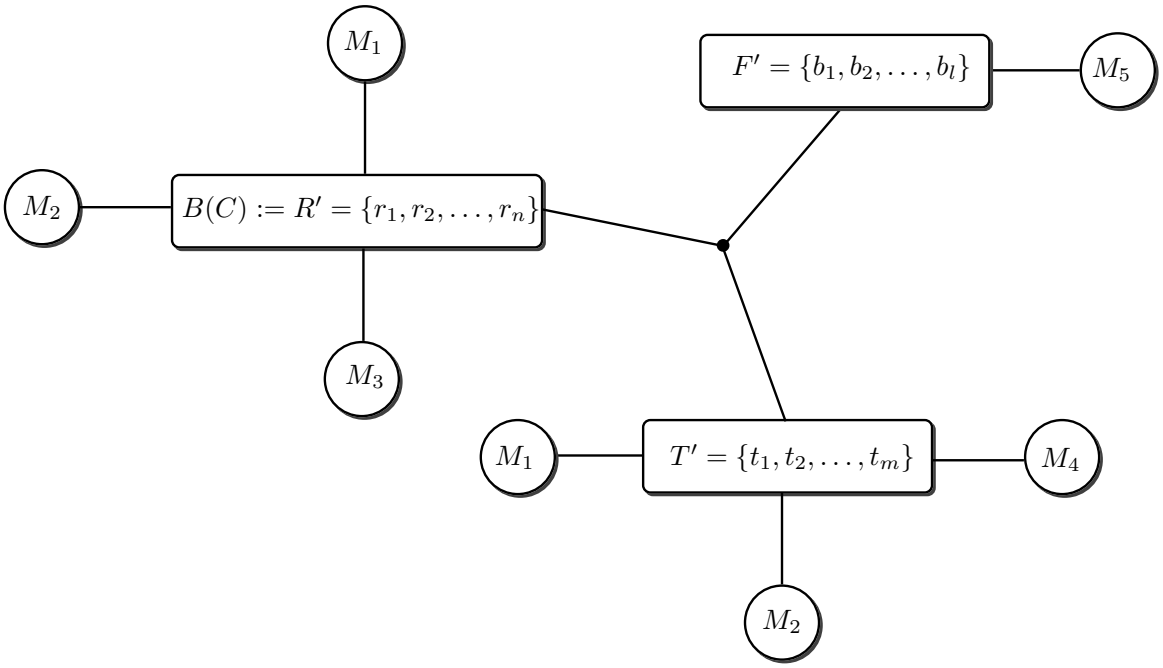


Abbildung 4.5: Eine Clique mit verschiedenen Kategorien von Inkonsistenzen

Korollar 4.4.1
Eine Clique kann hinsichtlich mehrerer Inkonsistenzkategorien zugleich inkonsistent sein. Des Weiteren kann ein Bezugsmengenpaar zugleich inkonsistent wegen Unvollständigkeit und inkonsistent wegen Redundanz sein.

Hinsichtlich des Bezugsmengenpaars $\langle R', F' \rangle$ ist die Clique in Abbildung 4.5 inkonsistent wegen Widerspruch, da $M_{R'} \cap M_{F'} = \emptyset$ und $|M_{R'} \cup M_{F'}| \geq 1$. Bezüglich des Bezugsmengenpaars $\langle R', T' \rangle$ ist sie sowohl inkonsistent aufgrund von Unvollständigkeit als auch aufgrund von Redundanz, da $M_{R'} \setminus M_{T'} = \{M_3\} \neq \emptyset$ und $M_{T'} \setminus M_{R'} = \{M_4\} \neq \emptyset$ und $M_{T'} \neq \emptyset$ und $M_{R'} \neq \emptyset$.

Bemerkung 4.4.1
Man beachte, dass eine merkmalkonsistente Clique nicht notwendigerweise vollständig ist. Beispielsweise kann eine Anforderung A, die genau mit einem Merkmal M verknüpft ist, in einer Clique mit genau einem Test T für genau dieses Merkmal M sein. Dann sind A und T zwar merkmalkonsistent, jedoch kann ein weiterer Test für M existieren, der nicht zur Clique gehört. Dies kann auf eine fehlende Verknüpfung mit der Anforderung hindeuten, muss aber nicht, da durchaus unterschiedliche Anforderungen und Tests für dasselbe Merkmal existieren können, die nicht logisch zusammenhängen und folglich keine Clique bilden. Daher kann das Werkzeug aus Abschnitt 6.7 nur dann korrekte Ergebnisse liefern, wenn die Korrektheit der Cliquen und Merkmalverknüpfungen gegeben ist.

Bevor die Korrekturaktionen für Merkmalinkonsistenz formalisiert werden, wird ein Maß für die Kritikalität selbiger gemäß [162, 163] eingeführt, das für die spätere Priorisierung von Korrekturaktionen verwendet werden kann.

Definition 4.4.4 (Kritikalität von Inkonsistenz)
Die Kritikalität der Inkonsistenz eines Bezugsmengenpaars $\langle B(C), A\rangle =: p \in P_B(C)$ einer Clique $C = (R', F', T')$ mit Merkmalbasis $B(C)$ ist definiert durch

$$k : P_B(C) \to \mathbb{N} \cup \{0\}$$

mit

$$k(p) = \begin{cases} |M_{B(C)} \setminus M_A| + \alpha_u, & \text{falls } p \in U(P_B(C)) \\ |M_A \setminus M_{B(C)}| + \alpha_r, & \text{falls } p \in R(P_B(C)) \\ |M_{B(C)}| + |M_A| + \alpha_w, & \text{falls } p \in W(P_B(C)) \\ |M_{B(C)} \setminus M_A| + |M_A \setminus M_{B(C)}| + \alpha_u + \alpha_r, & \text{falls } p \in U(P_B(C)) \cap R(P_B(C)) \\ 0 & \text{falls } p \in K(P_B(C)) \end{cases}$$

wobei $\alpha_r \ll \alpha_u \ll \alpha_w$ Parameter für die Inkonsistenzkategorien Unvollständigkeit (α_u), Redundanz (α_r) und Widerspruch (α_w) darstellen.

Auf die Kritikalität wirkt sich also neben der Anzahl der unterschiedlichen Merkmalverknüpfungen der Artefaktelementmengen eines Bezugsmengenpaars über die Parameter $\alpha_r \ll \alpha_u \ll \alpha_w$ auch die Inkonsistenz*kategorie* aus. Dies ist zum Beispiel für ein Bezugsmengenpaar $p = \langle R', T'\rangle$ sinnvoll, da beispielsweise fehlende Merkmale in $M_{T'}$ darauf hindeuten können, dass Tests für die Anforderungen $r \in R'$ fehlen. Dies kann dazu führen, dass nicht hinreichend getestet wird. Um zu verhindern, dass viele unkritische, redundante Merkmalverknüpfungen zu einer höheren Kritikalität führen als wenige widersprüchliche oder unvollständige Verknüpfungen, müssen die Differenzen zwischen den Parametern hinreichend groß gewählt werden. Norbert Wiechowski wählt daher die Werte $\alpha_r = 0, \alpha_u = 10.000$ und $\alpha_w = 20.000$ [162, 163].

Merkmalinkonsistenzen entstehen durch fehlerhafte Merkmalverknüpfungen oder fehlerhafte Cliquen, die der Anwender folglich zu prüfen hat, wenn Inkonsistenzen identifiziert wurden. Im Falle einer fehlerhaften Clique sind die in ihr enthaltenen Artefaktelemente nicht korrekt mit anderen Elementen verknüpft. Somit kann hier die Korrektur durch Ergänzen bzw. Entfernen von entsprechenden Artefaktelementverknüpfungen erfolgen, wodurch indirekt die Merkmalmengen der Clique beeinflusst werden, da weitere Merkmale mit den neuen bzw. entfernten Elementen verknüpft sein können. Ist die Clique hingegen korrekt, so ist die Merkmalverknüpfung ihrer Elemente zu prüfen.

Sowohl die direkte Modifikation von Merkmalverknüpfungen als auch Änderungen an der Clique wirken sich letztendlich auf die Merkmalverknüpfungen einer Clique aus, die für die Betrachtung von Merkmalkonsistenz von Interesse sind. Tabelle 4.2 fasst mögliche Korrekturaktionen hinsichtlich der Merkmalmengen eines Bezugsmengenpaares $p = \langle B(C), A\rangle \in U(P_B(C))$ zusammen, das inkonsistent aufgrund von Unvollständigkeit ist. Die Menge $\Delta := M_{B(C)} \setminus M_A$ stellt dabei die Menge derjenigen Merkmale dar, die die Unvollständigkeit verursachen und die folglich aus $M_{B(C)}$ entfernt oder in M_A eingefügt werden müssten, um die Inkonsistenz zu beseitigen. X beschreibt diejenige Menge von Merkmalen, die durch eine Korrekturmaßnahme tatsächlich diesen Mengen hinzugefügt

Aktion	Geänderte Merkmalmengen			
	$X \subset \Delta$	$X = \Delta$	$X \supset \Delta$	$X \cap (M_{B(C)} \cup M_{A'}) = \emptyset$
$M_{B(C)} \setminus X$	A	B	C	(D)
$M_A \cup X$	E	F	G	H

Tabelle 4.2: Betrachtung von Korrekturaktionen hinsichtlich der Merkmalmengen eines Bezugsmengenpaars $p = \langle B(C), A\rangle \in U(P_B(C))$ ($\Delta := M_{B(C)} \setminus M_A$).

bzw. aus diesen Mengen entfernt werden. Ob dies durch die Modifikation der Clique oder eine Änderung der Merkmalverknüpfungen erfolgt, ist für die Untersuchung der Inkonsistenz irrelevant.

So treten die Fälle A und E beispielsweise nicht nur auf, wenn die Merkmale aus X direkt mit Artefaktelementen verknüpft bzw. direkt von diesen entfernt werden. Auch die Entfernung von Elementen aus der Clique (A) bzw. das Hinzufügen (E) kann sich gleichermaßen auf die Merkmalmengen auswirken. Hier ist dies der Fall, wenn diese Artefaktelemente weniger Merkmalverknüpfungen besitzen als aus $B(C)$ entfernt bzw. zu A ergänzt werden müssten, um die Unvollständigkeit auszugleichen. Analoges gilt für die Fälle C und G, da hier mehr Merkmale entfernt bzw. ergänzt werden als notwendig. In den Fällen B und F enthalten die hinzugefügten bzw. entfernten Artefaktelemente genau diejenigen Mengen, die nötig sind, um die Inkonsistenz aufzulösen. Fall D ist irrelevant, da er sich nicht auf Merkmalmengen auswirkt. Fall H beschreibt die Aktion, dass Merkmale, welche zuvor weder mit A noch mit $B(C)$ verknüpft waren, mit A verknüpft werden. Dies kann geschehen, wenn Merkmalverknüpfungen der Elemente aus A fehlen und diese daher ergänzt werden müssen oder wenn in der Clique Artefaktelemente fehlen und daher ergänzt werden müssen.

Jede so implizit (über Cliquenmodifikation) oder direkt hinzugefügte bzw. entfernte Merkmalverknüpfung kann neue Inkonsistenzen sowohl des betroffenen Bezugsmengenpaares als auch der Clique insgesamt verursachen, da auch die Menge der Merkmalverknüpfungen der Merkmalbasis modifiziert worden sein können. Somit wirken sich die Aktionen aus Tabelle 4.2 auf den Konsistenzstatus und somit auf die Kritikalität der Inkonsistenz des Bezugsmengenpaars aus. Die Auswirkung der Aktionen und möglicher Kombinationen von ihnen sind in Tabelle 4.3 zusammengefasst. Korrekturaktionen und ihre Auswirkungen für den Fall von Inkonsistenz eines Bezugsmengenpaares aufgrund von redundanten Merkmalverknüpfungen werden hier nicht behandelt, da diese sich durch den Tausch der Merkmalbasis $B(C)$ und A aus den Tabellen 4.2 und 4.3 ergeben. Sie werden in der Diplomarbeit von Norbert Wiechowski [162, S. 58 ff.] und in [163] beschrieben.

Im Falle von Inkonsistenz aufgrund von widersprüchlichen Merkmalverknüpfungen eines Bezugsmengenpaars $p = \langle B(C), A\rangle$ sind die Elemente von $B(C)$ mit völlig anderen Merkmalen verknüpft als die Elemente von A. Insofern liegt hier die Vermutung nahe, dass entweder die Clique unberechtigt ist oder die Merkmalverknüpfungen der Elemente

	Fall								
Index	A	B	C	E	F	G	H	Resultat	Kritikalität
1	×							U	↘
2		×						K	0
3			×					R	↘
4				×				U	↘
5					×			K	0
6						×		R	↘
7							×	$U \wedge R$	↗
8	×			×				$K \vee R \vee U$	↻
9	×				×			R	↘
10	×					×		R	↘
11		×		×				R	↘
12		×			×			R	↘
13		×				×		R	↘
14			×	×				R	↘
15			×		×			R	↘
16			×			×		R	↘

Tabelle 4.3: Auswirkung der Aktionen aus Tabelle 4.2

aus A nicht korrekt bzw. nicht vorhanden sind. Sind die Merkmalverknüpfungen von A korrekt, so empfiehlt sich die Auflösung der Clique, d. h. die Verbindungen zwischen den Elementen der Clique werden gelöst. Anderenfalls sollten die Merkmalverknüpfungen von A gelöscht und durch geeignete Merkmale, die mit $B(C)$ verknüpft sind, ersetzt werden. Da die Merkmalverknüpfung der Elemente aus $B(C)$ als korrekt angenommen wird (vgl. Definition 4.4.1), wird die direkte Modifikation selbiger hier nicht erwogen.

Nachdem alle Korrekturaktionen abgeschlossen sind, wird die Konsistenzprüfung erneut angestoßen. Sofern noch immer Merkmalinkonsistenzen vorhanden sind, liegt es nun nahe, dass Artefaktelemente für die problematischen Merkmale nicht existieren. Beispielsweise fehlt ein Subsystem im Simulink-Modell, welches ein bestimmtes Merkmal umsetzt. Somit ist ein Indiz für die Artefaktinkonsistenz gemäß Definition 2.3.3 gegeben.

Die hier betrachteten Arten der Inkonsistenzidentifikation und -auflösung wurden im Rahmen von Norbert Wiechowskis Diplomarbeit in das datenbankorientierte Analyserahmenwerk integriert. Das dabei entstandene Werkzeug beschreibt Abschnitt 6.7.

4.5 Verwandte Arbeiten

Dieser Abschnitt dient dazu, dem Leser einen Überblick über Arbeiten zu verschaffen, die sich dieser Dissertation ähnlichen Herausforderungen widmen. Sofern die Arbeiten

aufgrund ihrer praktischen Umsetzung einem der Lösungsansätze in den Kapiteln 5 oder 6 zuzuordnen sind, werden sie in den Abschnitten 5.8 bzw. 6.8 auch umsetzungsbezogen beschrieben und hier fachlich eingeordnet.

4.5.1 Artefaktintegration, Rückverfolgbarkeit und Informationsanreicherung

Eine umfassende Beschreibung des Stands der Technik und der verbleibenden Herausforderungen im Kontext der Artefakt- bzw. Werkzeugintegration geben Broy et al. [21]. Eine wesentliche Ursache für Inkonsistenz und unzureichende Rückverfolgbarkeit stellt demnach die Werkzeugisolation dar. Auswirkungen von Artefaktmodifikationen auf andere Artefakte automatisiert zu identifizieren, ist dadurch nicht bzw. nur schwer möglich. Abhilfe schaffen nach Ansicht der Autoren eine umfassende Modellierungstheorie und ein integratives Architekturmodell, in das sich die Werkzeuge eingliedern, um einen durchgängigen, modellbasierten Entwicklungsprozess zu etablieren. Das in Abschnitt 4.1.1 vorgeschlagene ideale Konzept entspricht somit dieser Forderung.

Der Forderung nach einem integrativen Architekturmodell schließen sich auch Dziobek et al. an [34]. Sie skizzieren den aktuellen AUTOSAR[1]-orientierten Entwicklungsprozess einer eingebetteten Software-Produktlinie bei der Daimler AG und die daraus resultierenden Anforderungen an dessen Unterstützung und an die Entwicklungsumgebung [34]. Neben einer gemeinsamen Ablage (engl. Repository) für Artefakte und einem artefaktübergreifenden Management wird eine bessere Verknüpfung der Rollen beim Fahrzeughersteller und Zulieferer gefordert. Die Artefaktintegration muss nach Ansicht der Autoren zwingend vollautomatisch erfolgen, um in der Praxis akzeptiert zu werden, und dient neben der Artefaktverknüpfung einem durchgängigen Variantenmanagement. Auch hier wird die Werkzeugisolation als Kern des Problems identifiziert. Über standardisierte und artefaktübergreifende Annotationen gemäß Abschnitt 4.2 sowie über die Verknüpfung der Artefakte mit den Merkmalen trägt das Lösungskonzept dieser Dissertation auch der Anforderung von Dziobek et al. nach Einbezug von Rollen in das integrative Datenmodell Rechnung.

Zhang et al. weisen auf Probleme aktueller Ansätze zur Werkzeugintegration hin. Werden diese mit Hilfe der proprietären APIs realisiert, so führen spätere Werkzeugwechsel zu hohen Aufwänden, da die Verknüpfungen neu erstellt werden müssen. Das abstraktere Vorgehen über ein werkzeugübergreifendes Metamodell hingegen erweist sich als zu kompliziert und wartungsintensiv. Daher stellen sie einen Ansatz vor, der mit Hilfe eines Integrationsmodells vom werkzeugspezifischen Metamodell abstrahiert. In diesem Integrationsmodell besitzen die Originale einen Repräsentanten, der mit ihnen über einen eindeutigen Bezeichner verknüpft ist.

Auf eine weitere Gefahr durch mangelhafte Werkzeugintegration weisen Holtmann et al. hin [52]. Sie sehen die Einhaltung von Sicherheitsstandards durch die mangelhafte

[1] Der AUTOSAR-Standard wurde von der Automobilindustrie und ihren Zulieferern definiert, um u. a. einheitliche Schnittstellen und Modularität von Steuergerätesoftware sicherzustellen und so z. B. den Austausch und die Aktualisierung von Software zu erleichtern.

Durchgängigkeit des Entwicklungsprozesses gefährdet. In diesem Kontext befassen sie sich mit der Problematik, dass Tests und Validierungen von Systemen meist erst spät im Entwicklungsprozess erfolgen und dadurch teure Korrekturzyklen verursachen, die zu vermeiden gewesen wären, wenn man die Validierung bereits auf Artefakten früherer Entwicklungsprozessschritte durchgeführt hätte.

Auch Biehl et al. beschäftigen sich mit Sicherheitsaspekten in der Entwicklung eingebetteter Systeme im Automobilbereich [14]. Sie weisen darauf hin, dass durch den zunehmenden Einsatz eingebetteter Software in dieser Domäne traditionelle Sicherheitsanalysemethoden in die Werkzeugkette zu integrieren sind. Dazu müssen Werkzeuge des Entwicklungsprozesses mit jenen für Sicherheitsanalysen verbunden werden. Zu diesem Zweck stellen die Autoren einen Ansatz vor, der mit Hilfe von Modelltransformationen Architekturbeschreibungen in der domänenspezifischen Sprache EAST-ADL mit dem Sicherheitsanalysewerkzeug HiP-HOPS verknüpft. So kann der Anwender auch nach einer Architekturmodifikation die Sicherheitsprüfung in HiP-HOPS beliebig oft ohne großen Aufwand wiederholen.

Mader et al. beschäftigen sich mit Sicherheitsanalysen, fokussieren dabei jedoch den Arbeitsablauf im Bereich automobiler Antriebssysteme [80]. Bedingt durch zunehmende Anzahl elektronischer Steuerungssysteme steigt auch hier die Anzahl von Sensoren stetig an. Dies resultiert in steigender Komplexität eingebetteter Software. Da die betrachteten Systeme sicherheitskritisch sind, müssen die zugrunde liegenden Architekturen entsprechenden Anforderungen und Normen, z. B. ISO 26262, genügen, deren Einhaltung mit Hilfe verschiedener Methoden, z. B. Fehlerbaumanalysen, validiert werden muss. Die Autoren präsentieren einen Ansatz, der notwendige Eigenschaften des Arbeitsprozesses in der domänenspezifischen Sprache EAST-ADL beschreibt. Um zu prüfen, ob der Arbeitsprozess den Anforderungen genügt, werden die Artefakte des Prozesses in ein Modell integriert, welches anschließend zu validieren ist. Werden Anforderungen verletzt, so schlägt das Rahmenwerk Korrekturen vor und führt sie ggf. durch. Die Autoren evaluieren ihren Ansatz im Rahmen einer Fallstudie über die Entwicklung hybrider Fahrzeuge.

Wolf et al. stellen einen Annotationsansatz vor, der dazu dient, dass sich verteilte Teammitglieder innerhalb eines Modells austauschen können [164]. Des Weiteren werden auch hier Annotationen für die implizite Herstellung von Rückverfolgbarkeit eingesetzt. Dadurch, dass die zu annotierenden Elemente und Annotationen in einer n:m-Relation stehen, können so auch kompliziertere Zusammenhänge abgebildet werden. Im Gegensatz zu dem hier präsentierten Konzept sind die Annotationen jedoch nicht in Syntax und Semantik vereinheitlicht. Annotationstypen sind also keine festen Werte zugeordnet. Benutzer können sich jedoch für automatische Benachrichtigungen über Annotationen an bestimmten Elementen registrieren. Das vorgestellte Konzept ist also insbesondere der Entwicklung in verteilten Teams gewidmet und bezieht sich auf Entwurfsartefakte „klassischer“ Softwareentwicklung wie Lastenheft und UML-Diagramme. Somit ist es nicht direkt im Kontext der Entwicklung mit MATLAB/Simulink anwendbar.

Steinbauer et al. beschreiben ein Annotationskonzept für Simulink-Modelle, beschränken sich aber auf Freitextinformationen und unterscheiden nicht zwischen verschiedenen

Annotationstypen [141]. Ziel der Annotationen ist hier die Anreicherung von Simulationsmodellen um Informationen über deren Entwicklung und Verwendung.

Zdrahal, Mulholland et al. beschreiben Clockwork, ein europäisches Projekt, das sich mit dem Wissensaustausch zwischen Entwicklern in kollaborativen Entwicklungsumgebungen beschäftigt [165, 95]. Dabei wird Entwicklern mit einer Kombination aus Technologien des semantischen Webs und informaler Beschreibung von Modellen ermöglicht, ihr Wissen über Entwurfsentscheidungen effizient in Modellen abzulegen. Um zusätzliche Aufwände durch Annotationen auf das Wesentliche zu beschränken, sollen dabei nur Entwurfsentscheidungen von professionellen Entwicklern annotiert werden, die kein Allgemeinwissen darstellen, sondern auf den individuellen Fall bezogene Entscheidungen. Um die Informationen aus dem Modell abzufragen, werden Browser-basierte Anfragedialoge zur Verfügung gestellt. Da gemäß Mens und Demeyer 50% der Entwicklungskosten auf das Verstehen vorhandenen Codes entfallen [83], ist anzunehmen, dass diese durch die Wissensanreicherung reduziert werden können. Der wesentliche Unterschied zum Annotationskonzept dieser Dissertation besteht darin, dass die Annotationen artefaktübergreifend und in einheitlicher Form in einer zentralen Datenbank abgelegt werden und das Ziel verfolgen, somit implizite Verknüpfungen zwischen diesen zu ermöglichen.

Anquetil et al. führen eine detaillierte Bestandsaufnahme der Fähigkeiten aktueller Werkzeuge im Hinblick auf Rückverfolgbarkeit von Entwicklungsartefakten durch und schlussfolgern, dass aktuelle Werkzeuge sich auf bestimmte Arten der Rückverfolgbarkeit (z.B. Anforderungen) beschränken, zu werkzeugspezifisch sind, zu wenige Analysemöglichkeiten bieten und nicht hinreichend skalierbar sind [5]. Insbesondere identifizieren sie Bedarf an Unterstützung der Rückverfolgbarkeit im Kontext der Software-Produktlinien-Entwicklung, den auch Berg et al. sehen [12]. Dort liegen besondere Herausforderungen darin, dass für alle Artefakte ein grundlegendes Verständnis über die Auswirkung von Variabilität notwendig ist, Verknüpfungen zwischen Produktlinienmitgliedern existieren und wenig Werkzeugunterstützung für das Anforderungsmanagement gegeben ist. Die Autoren unterscheiden die Rückverfolgbarkeit von Verfeinerung (engl. *Refinement Traceability*), Ähnlichkeit (engl. *Similarity Traceability*), Variabilität (engl. *Variability Traceability*) und Versionen (engl. *Versioning Traceability*). Das vorgestellte Rahmenwerk AMPLE Traceability Framework (ATF) adressiert insbesondere die Rückverfolgbarkeit von Variabilität. Der zweite Beitrag der Publikation besteht in der Vorstellung eines Metamodells für Rückverfolgbarkeitsverknüpfungen. Dazu werden ähnlich dem Ansatz dieser Dissertation die Artefakte mit Informationen angereichert, die kontextabhängig angezeigt werden können. Im Gegensatz zu den in dieser Dissertation verfolgten Zielen, liegt hier der Schwerpunkt auf der Modellierung von Rückverfolgbarkeitsinformation, sowie deren Verwaltung, Visualisierung und Persistierung. Diese Dissertation beabsichtigt hingegen, durch Artefaktintegration und -annotation eine Grundlage für die automatische Identifikation *potenzieller* Zusammenhänge zu schaffen und fokussiert weniger auf deren Modellierung. Des Weiteren betrachtet diese Dissertation insbesondere die speziell im Kontext der Entwicklung regelungstechnischer, eingebetteter Systeme mit MATLAB/Simulink entstehenden Herausforderungen, während Anquetil et al. primär die „klassische“ Softwareentwicklung betrachten und UML-Modelle anstelle von signal-

flussorientierten Blockschaltbildern berücksichtigen. Schließlich werden hier mit Hilfe der Verknüpfung von Merkmalen und Artefaktelementen Konsistenzuntersuchungen ermöglicht.

Matthias Riebisch beschreibt, wie Merkmale verwendet werden können, um die Rückverfolgbarkeit zwischen Abstraktionsebenen von Artefakten herzustellen und so die konsistente und vollständige Umsetzung von Änderungen an Software-Systemen sicherzustellen [126]. Auch das Lösungskonzept dieser Dissertation sieht die Verknüpfung von Artefakten mit Merkmalen vor, um Variantenmanagement zu unterstützen. Dennoch werden auch direkte Verknüpfungen von Artefaktelementen berücksichtigt, da Merkmale durch ihren hohen Abstraktionsgrad zwar die Anzahl der Verknüpfungen reduzieren, aber dadurch die Gefahr steigt, dass zu viele Artefaktelemente als zusammengehörig identifiziert werden.

Auch in der aktuellen Werkzeuglandschaft wird die Artefaktintegration bereits in verschiedener Hinsicht adressiert. So berücksichtigt die OSGi- und webtechnologiebasierte Plattform IBM Jazz [53] beispielsweise über die Integration von Personen und Ressourcen in den Entwicklungsprozess einen Teil der von Dziobek et al. in [34] genannten Herausforderungen. Außerdem wird die Entwicklung in verteilten Teams unterstützt.

Auch die Produktfamilie IBM Rhapsody [55] dient der kooperativen Softwareentwicklung im Team mit speziellem Fokus auf die graphische, modellbasierte Entwicklung eingebetteter Systeme und Echtzeitanwendungen. Die enthaltenen Produkte dienen dem effizienten Entwurf von Anwendungen im Team und unterstützen Entwickler in der Analyse und Validierung von Anforderungen. Außerdem wird durch die Anwendung der Modellierungssprachen Systems Modeling Language (SysML) [105] und Unified Modelling Language (UML) [106] Hilfestellung in der Entwicklung konsistenter Applikationen gegeben.

Artisan Studio [7] dient ebenfalls der modellbasierten Entwicklung hochqualitativer, konsistenter Modelle in verteilten Teams. Neben Spezifikationswerkzeugen für UML- und SysML-Diagramme stellt es Generatoren zur Verfügung und unterstützt die Rückverfolgbarkeit im Entwicklungsprozess.

4.5.2 Konsistenzprüfung und Variabilitätsmanagement

Ein oft adressiertes Thema in der aktuellen Forschung betrifft die Konsistenz von Artefakten des Entwicklungsprozesses, wobei dabei häufig die Konsistenz von UML-Modellen untereinander oder im Zusammenhang mit Code im Vordergrund steht [20, 24, 81, 98, 74]. Konsistenzbedingungen werden oft mit der Object Constraint Language (OCL) [101] definiert, z.B. in [44] und [45], obwohl diese keine Möglichkeit bietet, Inkonsistenzen zu korrigieren. Abhilfe schafft hier die Eclipse Validation Language (EVL) [70, 71] von Kolovos et al. [71]. Wie bei der OCL handelt es sich dabei um eine textuelle Spezifikationssprache, die auf EMF-Modelle angewendet werden kann.

Mens et al. beschreiben einen deklarativen Spezifikationsansatz für die Identifikation und Auflösung von Inkonsistenz [84], der auf Graphtransformationen basiert. Auch Anne-Thérèse Körtgen spezifiziert Konsistenzregeln deklarativ als *Triple Rules* und

wendet diese an, um Inkonsistenzen zwischen UML-Modellen und Code zu identifizieren [73].

Man unterscheidet Ansätze, die Inkonsistenzen nur identifizieren und jenen, die sie auch beheben. Von letzterer Sorte existieren gemäß Reder nur wenige [123]. Mögliche Seiteneffekte der Korrekturen werden ebenfalls selten behandelt. Daher beschreibt Reder einen Ansatz für die Identifikation und Korrektur von Inkonsistenzen, der auf alle Modellierungssprachen und prädikatenlogische Konsistenzdefinitionen angewendet werden kann. Die Regeln formuliert Reder prototypisch durch OCL und wendet sie auf UML-Diagramme an.

Auch ihre parallele Bearbeitung führt zu Inkonsistenzen von Artefakten. So sehen unterschiedliche Teams oft ein Gesamtsystem aus unterschiedlichen Perspektiven, die jedoch nicht notwendigerweise disjunkt sind. Dadurch können simultan verschiedene Änderungen an denselben Elementen durchgeführt werden. Mougenot et al. präsentieren eine Methode, die derartige Inkonsistenzen erkennen kann [94]. Dazu bedient sie sich des Ansatzes partieller Replikation aus der künstlichen Intelligenz und baut auf operationsbasierter Repräsentation von Modellen auf.

Im Software-Produktlinien-Kontext ergeben sich weitere Inkonsistenzarten, wie u. a. von Lauenroth und Pohl sowie von Cmyrev et al. beschrieben [26, 75, 76]. So können durch die Unterteilung in Domänen- und Applikationsentwicklung Inkonsistenzen dadurch entstehen, dass in der Domäne widersprüchliche Anforderungen existieren, die jedoch mit dem Binden der Variabilität automatisch aufgelöst werden, da die widersprüchlichen Anforderungen niemals gleichzeitig für dieselbe Variante gelten [75, 76]. Herkömmliche Konsistenzprüfungen sind daher nicht im Produktlinienkontext anwendbar. Lauenroth und Pohl stellen daher einen Ansatz vor, der für die Inkonsistenzuntersuchung zusätzlich die Variantenzugehörigkeit von Artefaktelementen berücksichtigt. Cmyrev et al. beschäftigen sich mit einem konsistenten Variantenmanagement [26]. Hier beziehen sich Artefaktelemente auf bestimmte Merkmale des Merkmalmodells. Werden die Artefaktelemente mit den Merkmalen verknüpft, so ist sicherzustellen, dass zusammengehörige Elemente unterschiedlicher Artefakte auch mit denselben Merkmalen assoziiert sind. Cmyrev et al. beschreiben zu diesem Zweck einen Identifikationsansatz für Inkonsistenzen der Merkmalverknüpfungen zwischen Anforderungen und Testfällen und unterteilen Inkonsistenzen in verschiedene Kategorien [26].

Farkas et al. stellen ein Rahmenwerk vor, welches die artefaktübergreifende Konformität von Artefakten mit Richtlinien prüft [44, 45, 43]. Dazu werden die Artefakte in eine gemeinsame Ablage integriert, auf deren Basis Richtlinien geprüft werden können, die in der vorgestellten Spezifikationssprache Query-Based Rule Description Language (QRDL) spezifiziert wurden. Das Rahmenwerk ist im Kontext des MESA[1]-Projektes beim Fraunhofer FOKUS in Kooperation mit der Carmeq GmbH entstanden und besonders durch Herausforderungen in der Automobilindustrie inspiriert. Das Rahmenwerk wird am Beispiel der Integration von Anforderungen aus IBM Rational DOORS und MATLAB/Simulink vorgestellt.

[1] Metamodellierung zur Automatisierung von Analyse- und Entwicklungsmethoden für Software im Automobil

Nach Ansicht von Nöhrer und Egyed sind Inkonsistenzen die Folge von Fehlern, weshalb es sinnvoll ist, diese in umfassenderem Kontext zu betrachten, sich also nicht auf individuelle Inkonsistenzen zu beschränken [100]. Dazu schlagen die Autoren die Gruppierung von miteinander verbundenen Inkonsistenzen vor, wodurch die umfassende Fehlersuche erleichtert werden könnte.

Variabilitätsbezogene Herausforderungen, die in dieser Dissertation adressiert werden, beziehen sich primär auf den Ansatz der generativen Programmierung bzw. der überlagerten Varianten nach Czarnecki et al. [27, 28]. Wie beim modellbasierten Analyseansatz dieser Dissertation wenden Botterweck et al. beispielsweise Modelltransformationen in der hybriden Modelltransformationssprache ATLAS Transformation Language (ATL) [36, 63, 64] an, um ein Simulink-Modell in ein EMF-Modell zu überführen [19]. Dort wird dieses Modell mit den Merkmalen des Merkmalmodells verknüpft, sodass ein Produktlinienmodell im EMF entsteht. Dieses dient als Basis für den vorgestellten S2T2 Configurator [79], der die Zusammenhänge zwischen der Implementierung und den funktionalen Merkmalen visualisiert und auch automatisch Konsequenzen von Konfigurationsentscheidungen analysieren kann (z. B. welche Implikationen bezüglich weiterer Merkmale existieren). Schließlich lässt sich gemäß dem Konzept negativer Variabilität aus dem integrierten Produktlinienmodell ein 100%-Modell ableiten. In [16] wird dieser Ansatz auf aspektorientierte Programmierung angewendet.

In der Praxis reicht die statische Betrachtung funktionaler Merkmale nicht aus, um Variabilität erfolgreich zu handhaben. Aus diesem Grund halten Thomas et al. die Einführung eines abstrakteren Prozesses – des *Produktlinienlebenszyklusses* – für notwendig, um die Evolution und Wartung von Produktlinien über die Zeit zu ermöglichen [150]. Das in dieser Dissertation präsentierte Lösungskonzept greift somit durch die Integration des Merkmalmodells in das artefaktübergreifende Datenmodell sowie durch die Möglichkeit, durch Modellanalysen den Entwickler bei der evolutionsbedingten Modifikation der Produktlinie zu unterstützen, die Forderung nach Evolutionsunterstützung von Thomas et al. auf. Durch die Unterstützung des Anwenders bei der Sicherung von Merkmalkonsistenz wird zudem ein durchgängiges Variantenmangement ermöglicht.

Auf funktionaler Ebene beschäftigen sich Botterweck et al. mit der Berücksichtigung von Evolutionsaspekten im Produktlinienmanagement [17, 18]. Die Autoren identifizieren einen Mangel an Unterstützung für die langfristige Planung von Software-Produktlinien in der aktuellen Forschung. Diese ist jedoch von besonderer Bedeutung, da viele Unternehmen, die erfolgreich den Produktlinienansatz verfolgen, ihre Produktlinien langfristig ausrichten, aktuelle Forschung sich aber primär mit der *Entwicklung* und weniger mit der *Evolution* beschäftigt. Daher wird hier das Rahmenwerk *EvoFM* vorgestellt, welches ein Konzept für die Evolution von Merkmalen in die Merkmalmodellierung integriert. Die Autoren veranschaulichen EvoFM an Beispielen aus der Automobilindustrie.

Polzer et al. wenden den Ansatz negativer Variabilität im Kontext von Variabilität der einer eingebetteten Software zugrunde liegenden Hardware an [116]. Die adressierte Herausforderung liegt hier darin, dass die Verwendung unterschiedlicher Sensoren und Aktuatoren für Mikrocontroller einer Produktlinie die Modifikation des Verhaltens des Mikrocontrollers erfordert. Dazu wird ein Ansatz vorgestellt, der den Modifikationsprozess mit Hilfe von modellbasierter Entwicklung und Anwendung des Rapid Control

Prototyping durch Modularisieren von Parametrisierungen beschleunigt. Die Anpassung des Implementationsmodells selbst ist hier nicht notwendig. Für die Realisierung wird der Simulink Connector von pure::variants [13, 122] verwendet, um das Merkmalmodell mit dem Implementationsmodell in MATLAB/Simulink zu verknüpfen. Der vorgestellte Ansatz ermöglicht einen effizienteren Entwicklungsprozess, da testbare Applikationen schneller zur Verfügung gestellt werden können.

4.5.3 Artefaktanalyse und Modelltransformationen

Ein häufig adressiertes Thema in Theorie und Praxis ist die Überprüfung von Modellierungsrichtlinien von Simulink-Modellen [2, 47, 77, 93, 125, 142, 143, 157]. Solche Modellierungsrichtlinien werden für die modellbasierte Entwicklung mit MATLAB/Simulink u. a. vom MATLAB Automotive Advisory Board [148] (MAAB) definiert und oft durch firmenspezifische Richtlinien ergänzt. Sie dienen der besseren Verständlichkeit und Wartbarkeit von Simulink-Modellen sowie der Steigerung von Effizienz und Sicherheit von generiertem Code.

Stürmer et al. berichten über ihre Erfahrungen mit Modellierungsrichtlinien und Werkzeugen, die deren Einhaltung überprüfen und ggf. automatisch korrigieren [142]. Sie berücksichtigen die Werkzeuge Model Advisor, MINT [125], Model Examiner (MXAM) [93] und MATE [2, 77, 143, 157].

Die Werkzeuge Model Advisor und MINT prüfen die Konformität von Modellen mit Modellierungsrichtlinien über die Ausführung von MATLAB-Skripten, die Reporte generieren, die die Ergebnisse der Konformitätsprüfung zusammenfassen und mit dem untersuchten Modell verknüpfen. Die Korrektur obliegt jedoch dem Anwender. Nach Stürmer et al. ist dieser Prozess jedoch aufwändig und fehleranfällig. Besonders die fehlende Möglichkeit, Verletzungen der Richtlinien automatisch zu korrigieren, machen den Einsatz der beiden Werkzeuge nach Ansicht der Autoren schwierig. Besser schneiden in dieser Hinsicht die Werkzeuge MXAM und MATE ab, die diese Funktionalität bieten. Neben der Möglichkeit, Richtlinienverletzungen zu korrigieren, lassen sich mit MXAM generische Überprüfungen sowohl in den Rahmenwerken selbst als auch als MATLAB- oder Batch-Skripte ausführen. Wie die identifizierten Richtlinienverletzungen werden auch deren Korrekturen protokolliert. Dies ist insbesondere im sicherheitsrelevanten Umfeld von hoher Bedeutung. MXAM unterstützt sowohl die Prüfung von MISRA-Regeln [92] als auch die Einhaltung von Sicherheitsvorschriften gemäß ISO 26262[1] [57] für Simulink-, Stateflow-, TargetLink- und ASCET-Modelle. Stürmer et al. berichten, dass durch die frühzeitige Anwendung von MXAM Aufwände für Modell-Reviews signifikant reduziert werden können, da formale Überprüfungen und Korrekturen bereits im Vorfeld möglich sind. Das Werkzeug MATE bietet über Analysen von Richtlinienkonformanz hinaus auch vordefinierte Modell-Muster, z. B. *if-then-else* oder *switch*, die richtlinienkonform sind und vom Anwender bereits während der Entwicklung verwendet werden können. Des Weiteren werden Operationen für die visuelle Optimierung von Simulink-Modellen

[1] Dieser Standard definiert internationale Regeln für die funktionale Sicherheit von elektrischen / elektronischen Systemen im Automobil.

bereitgestellt. Die Reparaturoperationen sind teilweise vollautomatisch, zum Teil auch semi-automatisch, d. h. sie unterstützen den Anwender in der Korrektur von Richtlinienverletzungen, benötigen aber Informationen von diesem und handeln folglich nicht autonom. MATE basiert im Gegensatz zu MXAM, MINT und Model Advisor nicht auf MATLAB-Skripten, sondern auf Modelltransformationen, die deklarativ mittels Story-Driven Modelling (SDM) in Fujaba [47, 156] definiert werden, da MATLAB-Skripte kompliziert strukturiert und daher schwer verständlich und fehleranfällig sind. Gemäß den Erfahrungen von Amelunxen et al. können Analysen so etwa vier mal effizienter implementiert werden als auf dem direkten Weg mittels MATLAB-Skripten [2]. Um das Modell analysieren und insbesondere modifizieren zu können, muss dennoch ein Zugriff auf das Modell mittels MATLAB-Skripten erfolgen. Dazu besitzt MATE eine Schnittstelle zwischen MOFLON [1, 153, 160], dem MOF-basierten Metamodellierungsplugin von Fujaba, und MATLAB. Neben der direkten Manipulation des Simulink-Modells über die API besteht die Möglichkeit, die Modifikationen in einem externen Rahmenwerk vorzunehmen. Dies ist insbesondere im Falle komplexerer Analysen sinnvoll. Der in der vorliegenden Arbeit verfolgte Ansatz erweitert im Gegensatz dazu die Reichweite der Analysen, da auch andere Artefakte als das Simulink-Modell betrachtet werden können, läuft aber aufgrund der Komplexität der betrachteten Analysen grundsätzlich auf Basis eines externen Rahmenwerks. Auch der MESA-Ansatz [44, 45] verwendet eine separate Ablage für die Definition von Analysen der Richtlinienkonformität und Konsistenz von Artefakten. Das Metamodell dieser Ablage folgt dem MOF-Standard [103]. Die Richtlinien werden hier textuell per OCL definiert. Im Gegensatz zu den zuvor vorgestellten Werkzeugen können hier auf Basis der integrativen Modell-Ablage auch artefaktübergreifende Richtlinien definiert werden. Wie in dieser Dissertation auch, werden insbesondere die Werkzeuge MATLAB/Simulink und IBM Rational DOORS betrachtet. Im Gegensatz zu der vorliegenden Dissertation beschränken sich die Autoren jedoch auf die Richtlinienprüfung. Sichtenextraktionen zur Handhabung der Komplexität von Simulink-Modellen, artefaktübergreifende Informationsanreicherung und die Integration von Variabilitätsaspekten, z. B. durch die Integration und Verknüpfung der Artefakte mit Merkmalmodellen hingegen werden hier nicht adressiert. Daher sind Konsistenzanalysen im Hinblick auf Merkmalverknüpfungen und Validierung von Varianten auf Basis von Merkmalmodellen nicht möglich. Das von MATE verwendete Metamodellierungsplugin MOFLON dient als Werkzeug für die Entwicklung von Werkzeugen und unterstützt dazu laut Weisemöller et al. neben der zuvor beschriebenen SDM-Methodik auch Modelltransformationen gemäß MOF/QVT-Standard und Triple Graph Grammatiken (TGGs) [160]. Die Meta Object Facility (MOF) der Object Management Group (OMG) [103] ist ein Standard für die plattformunabhängige Metamodellierung von Modellen in der modellbasierten Entwicklung und erlaubt die Integration von Daten. QVT (Queries Views Transformation) [99] ist Teil des Standards und beschreibt analog Sprachen für die Transformation von MOF-basierten Modellen. Diese können sowohl deklarativer als auch imperativer Natur sein. TGGs wurden erstmalig im Jahr 1995 von Schürr vorgestellt [134] und stellen eine Erweiterung von Paar-Graph-Grammatiken gemäß Pratt [121] um kontextsensitive Produktionsregeln dar. TGGs ermöglichen damit komplexe, deklarative Graph-zu-Graph-Transformationen über die Spezifikation von Tripeln zusammengehöri-

ger Graphen, wovon einer den sogenannten Korrespondenzgraphen darstellt, der die Relation der beiden anderen Graphen zueinander definiert. Eine Anleitung zum Umgang mit TGGs findet sich in [69]. Manfred Nagl beschreibt Grundlagen der Theorie und die Anwendung von Graph-Grammatiken [97]. Heckel führt schließlich allgemeiner in die Grundlagen der Graphtransformationen ein und stellt deren historische Entwicklung und weiterführende Ansätze dar [51].

Neben deklarativen Modelltransformationen existieren imperative sowie hybride Transformationssprachen. Eine gute Einordnung von Modelltransformationssprachen in verschiedene Kategorien sowie eine Bewertung selbiger liefern Czarnecki et al. [29]. Ein weiterer Überblick über Transformationssprachen sowie ein Benchmarking finden sich auch in der Diplomarbeit von Kerstin Falkowski [42]. Patzina et al. vergleichen den zuvor genannten deklarativen SDM-Ansatz mit der hybriden Spezifikation von Transformationsregeln in der Modelltransformationssprache ATL, die auch für diese Dissertation angewendet wurde [111].

Varró et al. setzen Graphtransformationen mit einer relationalen Datenbank um [158]. Dazu werden deklarative Transformationsregeln in SQL-Anweisungen übersetzt, indem für jede Regel Datenbanksichten (sog. *Views*) erzeugt werden und Pattern Matchings über innere Joins, negative Anwendungsbedingungen über äußere Join-Operationen realisiert werden. Modellmanipulationen werden über *Insert*-, *Delete*- und *Update*-Anweisungen ermöglicht. Die im Kontext der vorliegenden Dissertation betrachteten Analysen lassen sich jedoch nicht rein deklarativ umsetzen, weshalb der Ansatz in dieser Form nicht übernommen werden kann. Dennoch legt die von Varró vorgestellte Effizienzuntersuchung nahe, dass die Verwendung einer relationalen Datenbank auch die Effizienz der hier betrachteten Analysen verbessern kann.

Auch Kalnins et al. identifizieren das Problem der effizienten Implementierung von Pattern Matchings im Kontext von Modelltransformationen und untersuchen die Effizienzsteigerung deklarativer Graphtransformationen durch eine zugrunde liegende relationale Datenbank [66].

Betrachtet man den in dieser Dissertation betrachteten Analysebegriff etwas abstrakter, so ist an dieser Stelle auch ConQAT [154] der TU München zu nennen. Dabei handelt es sich um eine Menge von Werkzeugen, die insbesondere der Qualitätssicherung und -analyse in der Softwareentwicklung dienen. Dazu gehören die effiziente Erstellung von Qualitätsüberwachungs-Dashboards, die Identifikation von Klonen im Code, die graphische Visualisierung von Qualitätsmerkmalen, die automatische Analyse von Architekturübereinstimmung sowie eine Trendanalyse zur Überwachung des Qualitätsstatus. Neben der Untersuchung von klassischem Code ist auch die Analyse von Simulink-Modellen möglich, die über einen Java-Parser in das Rahmenwerk importiert werden können. Im Gegensatz zu den in dieser Dissertation betrachteten Analysen mit Sichtenerstellung analysiert ConQAT allerdings auch hier Qualitätsmerkmale.

Joshi und Heimdahl stellen modellbasierte Sicherheitsanalysen für Simulink-Modelle vor, da manuelle Analysen durch den Ingenieur häufig fehleranfällig, unvollständig und inkonsistent sind [62]. Dazu verwenden sie formale Modelle, die vom Entwickler für das System und die Sicherheitsanalyse entworfen, mit SCADE [41] simuliert und über statische Analysemethoden automatisiert geprüft werden können. SCADE ist eine inte-

grierte Entwicklungsumgebung, welche insbesondere für die modellbasierte Entwicklung sicherheitskritischer, eingebetteter Systeme eingesetzt wird und nach verschiedenen Sicherheitsstandards zertifiziert ist. Es basiert auf grafischen, datenflussorientierten Diagrammen, die es formal verifizieren kann und aus denen C- oder ADA-Code generiert wird.

Die Untersuchung der Abhängigkeiten eines Blocks von vorangehenden Modellteilen, wie in Abschnitt 7.1.1 dieser Dissertation vorgestellt, erstellt sogenannte *Slices* im Sinne von Frank Tip [151]. Dort bezeichnen *Program Slices* Teile von Programmen, die Auswirkungen auf bestimmte Stellen im Code haben, wohingegen hier keine Wertentwicklung von Variablen im Code, sondern der Signalfluss im Simulink-Modell analysiert wird.

5 Modellbasierte Analyse

Der erste Ansatz für die Umsetzung des in Kapitel 4 vorgestellten pragmatischen Lösungskonzepts basiert auf Modellen für die Artefakte, die als Dateien persistiert werden, und implementiert Analysen mittels Modelltransformationen. Dazu werden im Folgenden zunächst die angewandten Werkzeuge vorgestellt. Sodann werden relevante Metamodelle eingeführt sowie die Modellerstellung erläutert. Nachdem die Umsetzung des Lösungskonzepts erläutert wurde, werden Optimierungsansätze skizziert und der Ansatz von verwandten Arbeiten abgegrenzt.

5.1 Werkzeuge und Rahmenwerke

In diesem Abschnitt werden die für die Realisierung verwendeten Werkzeuge und die Entwicklungsumgebung zusammengestellt.

5.1.1 Entwicklungsumgebung

Als integrierte Entwicklungsumgebung für den modellbasierten Analyseansatz wurden die Eclipse Modeling Tools [37] verwendet. Die wichtigsten der benötigten Plugins werden im Folgenden kurz vorgestellt.

Eclipse Modeling Framework Grundlage der Modellanalysen sind Modelle des Eclipse Modeling Frameworks (EMF) [38]. Dabei handelt es sich um ein frei verfügbares, quelloffenes Rahmenwerk für die modellbasierte Softwareentwicklung in Eclipse. Beispiele für Metamodelle, die im Kontext dieser Dissertation damit entwickelt wurden, werden in Abschnitt 5.2 beschrieben. Modelle werden hier im XML Metadata Interchange Format (XMI) [102] beschrieben. Dabei handelt es sich um einen XML-basierten Standard der Object Management Group (OMG), der ein werkzeugunabhängiges Datenaustauschformat beschreibt.

Graphical Modeling Framework Um Analyseergebnisse darzustellen, werden EMF-Modelle in graphischen Editoren visualisiert. Diese werden mit dem Graphical Modeling Framework (GMF) [38] umgesetzt, welches auf EMF aufsetzt. Das GMF folgt einem *Model-View-Controller*-Prinzip. Um damit einen graphischen Editor zu konstruieren, müssen verschiedene Modelle spezifiziert werden, deren Zusammenhang in Abbildung 5.1 dargestellt ist. Das *fachliche* Modell, das mit einem graphischen Editor erstellt bzw. von diesem dargestellt werden soll, wird durch das sogenannte *Domänenmodell* beschrieben. Ein Beispiel für ein Domänenmodell ist die Repräsentation des Simulink-Modells als

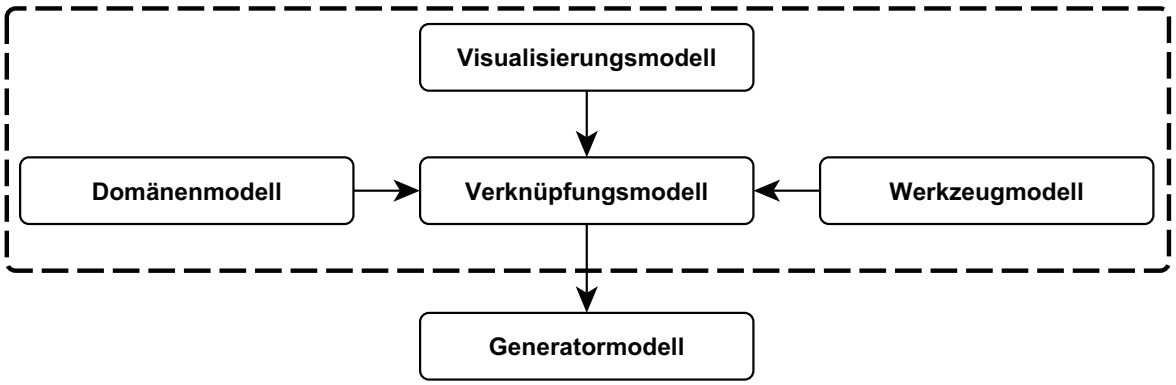


Abbildung 5.1: Modelle für die Erstellung eines graphischen Editors mit GMF

EMF-Modell bestehend aus verschiedenen Blocktypen, Linien und Parametern (vgl. Abschnitt 5.3.1). Im Gegensatz dazu beschreibt ein *Visualisierungsmodell* die grafischen Elemente, die der Editor bieten soll, z. B. ein rot ausgefülltes Rechteck mit gestricheltem Rand. Dieses Modell besitzt keinerlei Kenntnis der Domäne. Das *Werkzeugmodell* beschreibt, welche Werkzeuge der Editor anbieten soll, kennt aber weder die damit modellierte Domäne noch deren Visualisierung. Damit dient das Werkzeugmodell als *Controller* des Domänenmodells. Das *Verknüpfungsmodell* verknüpft die drei Modelle miteinander. Beispielsweise kann hier definiert werden, dass ein funktionaler Block aus dem Domänenmodell mit dem roten Rechteck mit gestricheltem Rand aus dem Visualisierungsmodell dargestellt wird und über einen Button des Werkzeugmodells in das Modell eingefügt werden kann. Auf dem Verknüpfungsmodell setzt das Generatormodell auf, aus welchem der Code für einen grafischen Editor generiert wird. Dieser kann selbst wiederum als Plugin in Eclipse integriert werden.

Xtext / MontiCore Für die Übersetzung der in Dateien abgelegten Artefakte in EMF-Modelle wurden Xtext [38] bzw. MontiCore [78] verwendet. Diese Rahmenwerke erlauben u. a. die Generierung eines Parsers und Lexers auf Basis einer zu spezifizierenden Grammatik für eine domänenspezifische Sprache (DSL). Auch bieten beide die Möglichkeit, aus einem Modell, welches zu dem der Grammatik entsprechenden Metamodell konform ist, wieder das entsprechende DSL-Dokument zu generieren. Für diese Dissertation ist zum einen der nach dem Parsen vorhandene abstrakte Syntaxbaum (AST) von Interesse, da dieser nach einer weiteren, strukturellen Optimierung als Grundlage für Analysen mittels Modelltransformationen dient. Zum anderen wird der Generator von Xtext (Xpand) dazu verwendet, aus Analyseergebnissen wieder ein Simulink-Modell zwecks Visualisierung zu generieren.

ATLAS Transformation Language / Apache Ant Für die Modell-zu-Modell-Transformationen von EMF-Modellen wurde die ATL eingesetzt. ATL bringt eine integrierte Entwicklungsumgebung mit, die sich nahtlos in die Eclipse-Umgebung integriert. Neben der Transformationsmaschine selbst ist auch ein Editor für die Modelltransformati-

onsspezifikation enthalten, der auch gängige Features (z. B. Syntaxhervorhebung und -validierung) enthält.

Bei der ATL handelt es sich um eine hybride Modelltransformationssprache (vgl. Abschnitt 2.1.1), d. h. Transformationsregeln können sowohl deklarative als auch imperative Elemente enthalten. Die deklarativen Elemente spezifizieren dabei einerseits das Muster, welches in dem zu transformierenden Modell gematcht werden muss, und wie dieses in das Ergebnismodell überführt werden soll. Zusätzlich existieren imperative Regelelemente, die beispielsweise im Anschluss an die deklarative Transformation noch angewendet werden sollen.

Für jede Spezifikation einer Transformation wird ein Automat (engl. abstract state machine (ASM)) generiert, der die Transformation vornimmt.

Um eine Transformation in Eclipse durchzuführen, muss eine Laufzeitkonfiguration (engl. Run Configuration) erstellt werden, die die in der Spezifikation deklarierten Modell- und Metamodellvariablen mit tatsächlichen Modellen belegt und festlegt, wo das Transformationsergebnis abgelegt wird. Da die Modelltransformationen auch für die Wiederverwendung als Analysebausteine durch Anwender gedacht sind, können und wurden für Modellanalysen mehrere Modelltransformationen hintereinander geschaltet. Jede einzelne modifiziert das Eingabemodell dazu hinsichtlich eines speziellen Aspekts (vgl. Abschnitt 5.4.1). Aus Bequemlichkeitsgründen wurden die zu transformierenden Modelle, die Transformationen und ggf. weitere Parameter in Ant-Build-Skripte integriert. Ant ist eine von der Apache Software Foundation zur Verfügung gestellte Java-Bibliothek, die primär zur Steuerung von Build-Prozessen in der Softwareentwicklung mit Java eingesetzt wird. Da Ant jedoch ansonsten keine weitere Rolle für die Dissertation spielt, sei es hier nur der Vollständigkeit halber erwähnt und für Details auf die Webseite [6] verwiesen.

5.1.2 MATLAB/Simulink

Die Implementationsmodelle liegen als Modelle in MATLAB/Simulink 2007b vor. Neben Blöcken der Standard-Bibliothek von Simulink-Blöcken sind auch Blöcke aus TargetLink-Bibliotheken [31] im Simulink-Modell enthalten.

5.1.3 IBM Rational DOORS

Die Anforderungen liegen in Form eines Lastenhefts in IBM Rational DOORS [54] in Version 9.2 vor. Jedoch wurde für die Umwandlung des Lastenhefts in ein EMF-Modell auf dem CSV-Export aufgebaut. Technische Details über den Transfer des Lastenhefts in ein EMF-Modell beschreibt Abschnitt 5.4.

5.2 Ablauf

Abbildung 5.2 beschreibt den Ablauf, modellbasiert Artefakte und Annotationen in ein gemeinsames Datenmodell zu integrieren und zu analysieren. Um artefaktübergreifend

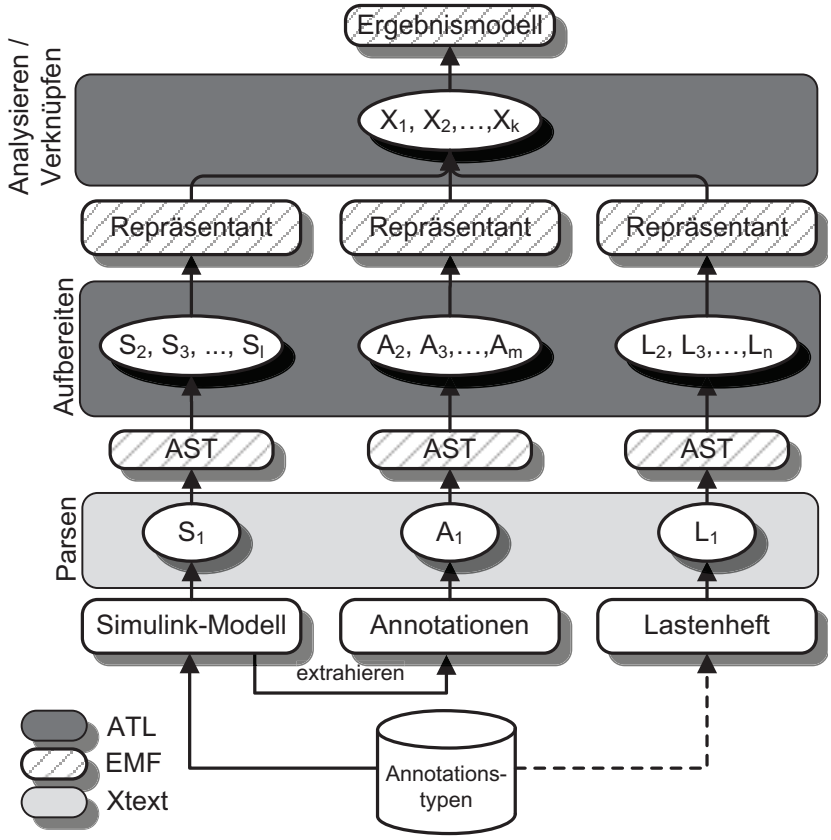


Abbildung 5.2: Ablauf des modellbasierten Ansatzes

dieselben Annotationstypen verwenden zu können, liegen diese in einer separaten Datenbasis. Da darin befindliche Annotationen aktuell nur aus MATLAB/Simulink abgefragt werden können, stellt der gestrichelte Pfeil zwischen der Datenbasis und dem Lastenheft eine Erweiterungsmöglichkeit dar. Annotationen werden zwar in MATLAB/Simulink erstellt, jedoch im weiteren Verlauf nicht gemeinsam mit dem Simulink-Modell, sondern separat weiterverarbeitet[1]. Der modellbasierte Ansatz ist auf das Simulink-Modell, die daraus extrahierten Annotationen und das Lastenheft beschränkt.

Um die Artefakte und Annotationen in ein artefaktübergreifendes Datenmodell im Sinne von Abschnitt 4.1.2 zu überführen, werden die zugrundeliegenden Dateien mit Xtext geparst (Schritte S_1, A_1 und L_1 in Abbildung 5.2). Die daraus resultierenden ASTs sind EMF-Modelle, die einem Teil dieses Datenmodells entsprechen. Diese werden strukturell in mehreren Modelltransformationen aufbereitet, um effizienter analysierbar zu sein. Dies geschieht in den Schritten $S_2, \ldots, S_l$, $A_2, \ldots, A_m$ bzw. $L_2, \ldots, L_n$. Mit dem Vorliegen der dann aufbereiteten EMF-Modelle, den sogenannten *Repräsentanten*, ist der Artefaktimport in das gemeinsame Datenmodell abgeschlossen. Diese Modelle dienen als Eingabe für weitere Transformationen, die der Modellanalyse dienen und ihrerseits wiederum in beliebig vielen Schritten $X_1, \ldots, X_k$ realisiert werden. Schließlich

[1] Details und Gründe für dieses Vorgehen erläutert Abschnitt 5.5.2.

ergibt sich ein weiteres EMF-Modell, welches das Ergebnis der Analyse darstellt und nun noch visualisiert werden kann. Die folgenden Abschnitte beschreiben die einzelnen Schritte detaillierter.

Der Parser (S_1) für das Simulink-Modell wurde aus Vorarbeiten von Andreas Polzer übernommen und modifiziert. Die Modelltransformationen für die Aufbereitung ($S_2, \ldots, S_l$) und Analysen ($X_1, \ldots, X_k$) des Simulink-Modells wurde in Kooperation mit diesem im Rahmen von gemeinsam durchgeführten Industrieprojekten implementiert.

5.3 Metamodelle

Das für diesen Ansatz verwendete, gemeinsame Datenmodell besteht aus miteinander verbundenen EMF-Modellen für die Artefakte, d. h. für jeden Artefakttyp existiert ein eigenes Metamodell, welches mit den Metamodellen anderer Artefakttypen verbunden ist. Jedes nach dem Parsen und Aufbereiten gemäß Abbildung 5.2 entstandene EMF-Modell beschreibt somit einen Teil dieses Datenmodells, auch wenn an dieser Stelle die Verknüpfungen noch nicht gesetzt sind. Dies kann jedoch im Anschluss erfolgen, da die Verbindung im Metamodell vorgesehen ist. Die Verknüpfungen kann man dann über einen EMF-Modelleditor manuell erstellen oder über eine Modelltransformation, die die dafür anzuwendenden Kriterien beschreibt, z. B. Namensgleichheit zwischen Anforderungen und funktionalen Blöcken.

Die folgenden Abschnitte erläutern für jedes betrachtete Artefakt einen Ausschnitt aus dem jeweils relevanten Metamodell und beschreiben den Weg vom Originalartefakt über Parser und Modelltransformationen zum finalen, analysier- und verknüpfbaren Repräsentanten. Auch Farkas präsentiert Metamodelle für Simulink-Modelle und Lastenhefte aus IBM Rational DOORS [43, S. 133 f.]. Aufgrund anderer Schwerpunkte sind hier jedoch andere Informationen von Interesse, sodass sich insbesondere das Metamodell für das Lastenheft von dem hier präsentierten unterscheidet (vgl. Abschnitt 5.3.2).

5.3.1 Simulink-Modell

Abbildung 5.3 zeigt einen Ausschnitt aus dem Metamodell für das aus dem Simulink-Modell durch den Parser erstellte EMF-Modell. Die Notation entspricht dem Standard der UML. In der Abbildung wurde aus Verständlichkeitsgründen auf technische Details (z. B. Optionen und Parameter von Blöcken) verzichtet.

Die Wurzel eines solchen EMF-Modells ist die Entität *Simulink-Modell*. Sie ist mit genau einem *System* assoziiert, welches die modellierte Funktionalität kapselt. Dieses besteht im Wesentlichen aus funktionalen *Blöcken*, *Linien* und Voreinstellungen (*Defaults*). Blöcke wiederum können *Ports* besitzen, über die sie mit anderen Blöcken Signale austauschen. Jede Linie hat genau eine Quelle und mindestens ein Ziel, welches im zugehörigen *SrcPointer* bzw. *DstPointer* durch die Angabe des Blocknamens sowie der Portnummer spezifiziert ist. Sofern eine Linie mehr als ein Zielobjekt besitzt, wird dies über (ggf. geschachtelte) Abzweigungen (engl. *Branches*) abgebildet, die dazu je genau ein Zielobjekt besitzen.

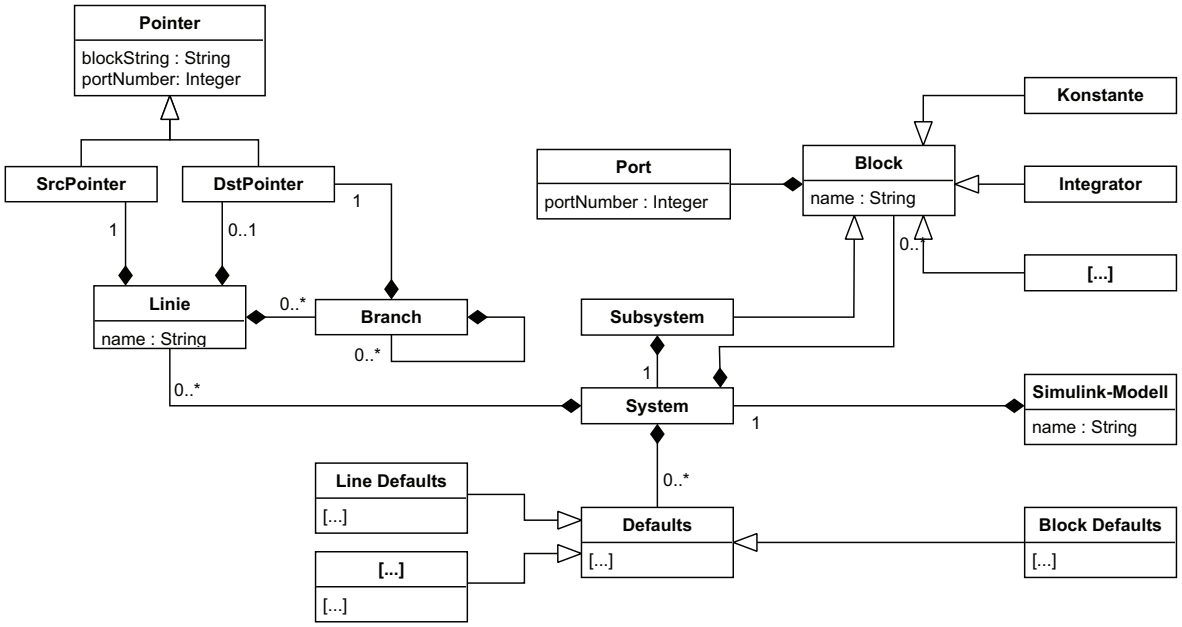


Abbildung 5.3: Metamodell von Simulink-Modellen nach dem Parsen (Ausschnitt)

Ein besonderes Merkmal ist der rekursive Aufbau von Simulink-Modellen durch Subsysteme, die eine spezielle Art von Blöcken darstellen, da sie ein weiteres System kapseln. Daher besitzen sie selbst ein *System*, wodurch die Rekursion möglich wird.

5.3.2 Lastenheft

Wie in Abschnitt 2.3.3 beschrieben, liegen die Anforderungen in Form eines Dokuments in IBM Rational DOORS vor. Die vorliegende Dissertation beschränkt sich im Gegensatz zu dem von Farkas [43, S. 133] vorgestellten Metamodell auf ein einziges Modul des Lastenheftes, welches die Anforderungen tabellarisch beschreibt und für die weitere Aufbereitung als CSV-Dokument exportiert wird. Dieses Dokument besteht folglich aus Zeilen, Spalten und kommagetrennten Einträgen pro Zeile, die sich auf jeweils eine Spalte der Tabelle beziehen. Diesen Sachverhalt bilden die im Metamodell in Abbildung 5.4 grau markierten Entitäten ab. Die anderen Entitäten beschreiben das baumartige Metamodell des DOORS-Lastenheftes, welches sich auf die Tabelle übertragen lässt wie die *is-a*-Beziehungen zu den grauen Entitäten ausdrücken. Das Wurzelelement *Lastenheft* entspricht dem Wurzelelement *Tabelle*. Es besteht aus einer Menge von Elementen, die in der Tabelle den Zeilen entsprechen. Elemente können entweder (Sub-) Kapitel sein, oder konkrete Anforderungen. Jedes (Sub-) Kapitel besteht u. U. aus weiteren Subkapiteln oder Anforderungen. Sowohl (Sub-) Kapitel als auch Anforderungen entsprechen somit Zeilen in der DOORS-Tabelle. Sie erben von ihrer Oberklasse *Element* eine Menge von Attributen, die sich aus Typ und Wert zusammensetzen, die in der Tabelle den Spalten (-überschriften) bzw. Zelleneinträgen entsprechen.

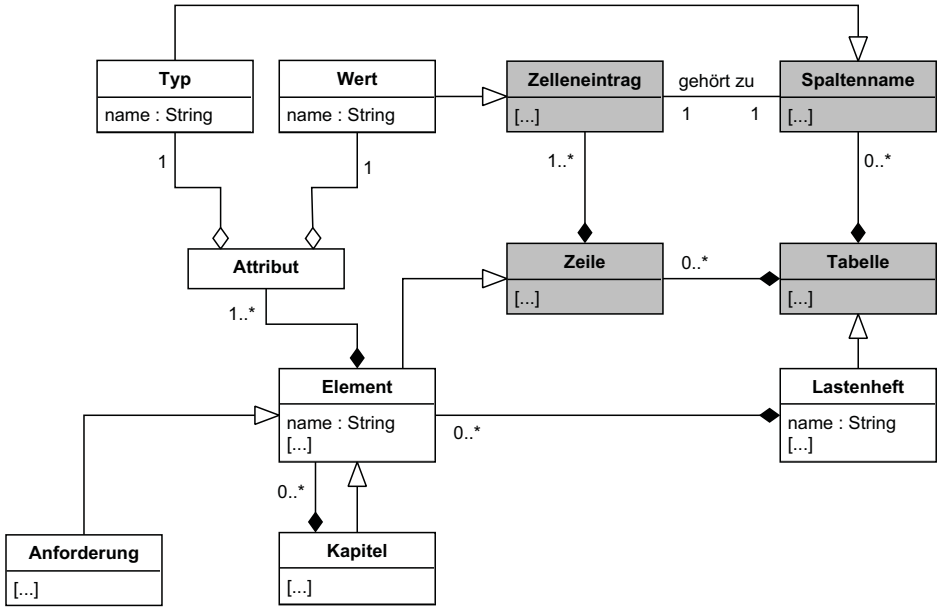


Abbildung 5.4: Strukturen von Anforderungen und der Tabellenstruktur (grau)

Im Beispiel von Tabelle 2.2 entsprechen also alle Tabellenzeilen, deren Hierarchieebene nicht mit einem Bindestrich gefolgt von einer Zahl enden, den Kapiteln bzw. Subkapiteln, z. B. 1.1.1. Die anderen Elemente sind Anforderungen, die den (Sub-) Kapiteln zugewiesen sind, z. B. 1.1.1.0-1. Die Attribute des Elements 1.1.1.0-1 stellen die Spalteneinträge der Zeile dar, wobei der Titel dem Namen der Spalte und der Wert dem Zelleneintrag entspricht.

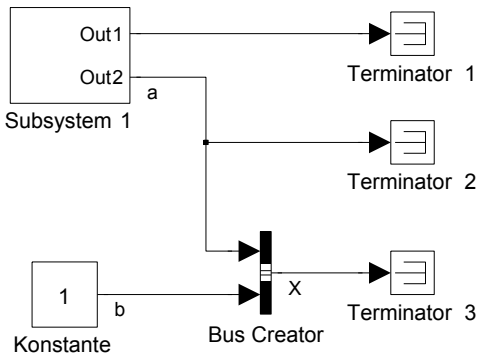
Abbildung 5.4 ist aus Verständlichkeitsgründen nicht in allen technischen Details dargestellt. Auch die Benennung der Entitäten entspricht daher nicht genau der Implementierung. Inhaltlich spiegelt es jedoch den Aufbau des Metamodells wieder.

5.4 Artefaktimport

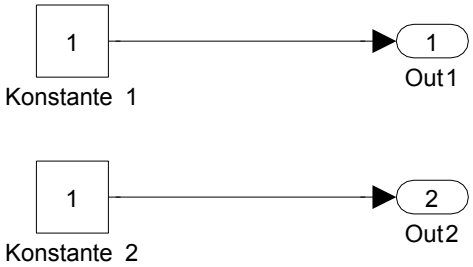
Wie in Abbildung 5.2 dargestellt, besteht der Import der Artefakte aus einer Parser- und einer Anreicherungsphase. Im Folgenden wird für das Simulink-Modell und das Lastenheft die Funktionsweise der beiden Phasen erläutert.

5.4.1 Simulink-Modell

Parser Um das Simulink-Modell in ein EMF-Metamodell zu überführen, dessen Metamodell ausschnittsweise in Abbildung 5.3 beschrieben wurde, wird zunächst die textuelle Beschreibung des Simulink-Modells (MDL-Datei) geparst. Um einen Parser zu erstellen, ist genaue Kenntnis der *konkreten* Syntax nötig. Abbildung 5.6a stellt dazu einen Ausschnitt für das beispielhafte Simulink-Modell aus Abbildung 5.5 dar.



(a) Oberste Hierarchieebene



(b) Inhalt von *Subsystem 1*

Abbildung 5.5: Ausschnitte eines beispielhaften Simulink-Modells

Jedes Element des Simulink-Modells (Block, Port, Linie, ...) wird durch ein Schlüsselwort (fett gedruckt) eingeleitet. Daraufhin folgt die Angabe der ggf. darin enthaltenen Elemente und Parameter in einem Block, der in geschweiften Klammern eingeschlossen ist. So ist beispielsweise der Block *Subsystem 1* in den Zeilen 12-36 spezifiziert. Er besitzt keine Eingangs- und zwei Ausgangsports (Z. 15). Einer der Ports ist in den Zeilen 17-20 dargestellt. Das vom Subsystem gekapselte Modell (vgl. Entität *System* in Abbildung 5.3) wird innerhalb des *System*-Blocks beschrieben (Z. 21-35). Es besitzt u. a. zwei Schnittstellenblöcke nach außen (*Out1* und *Out2*, Z. 24-33).

Ab Zeile 41 befinden sich eine Liniendefinition. Der Quellblock (*SrcBlock*) und -port (*SrcPort*) einer Linie werden dabei über ihren Namen bzw. ihre Nummer als Attribut einer Linie spezifiziert (Z. 43, 44). Analog erfolgt die Spezifikation des Zielblocks (*DstBlock*) bzw. -ports (*DstPort*). Besitzt eine Linie mehrere Ziele, so werden Zielblock und -port gemäß Abschnitt 5.3.1 innerhalb eigener *Branch*-Elemente (Z. 46, 50) spezifiziert. Anderenfalls erfolgt dieselbe Spezifikation direkt auf Ebene der Linie.

Um das Simulink-Modell in ein EMF-Modell zu übersetzen, wurde ein Parser mit Xtext für die zugrunde liegende Grammatik entwickelt. Nach dem Parsen liegt der AST als EMF-Modell vor, welches dem Metamodell aus Abbildung 5.3 entspricht.

Aufbereitung Durch das Parsen des Codes in Abbildung 5.6a wird zwar ein EMF-Modell gemäß Abbildung 5.3 generiert, jedoch sind die Möglichkeiten, die Modellstruktur dabei bereits für spätere Analysen zu optimieren, beschränkt. So lässt sich zwar von Linienquellen und -zielen durch *Pointer* abstrahieren. Die so verknüpften Blöcke bzw. Ports können jedoch noch nicht durch Links, z. B. in Form von Navigationspfaden, sondern nur durch Angabe ihrer Namen bzw. Nummern in den Zeigern abgelegt werden, die nur innerhalb ihres Subsystems aber nicht modellweit eindeutig sind. Dies erschwert die effiziente Durchführung späterer Analysen, die beispielsweise Abhängigkeiten zwischen Blöcken identifizieren.

Das Resultat zeigt Abbildung 5.6b. So wird die Linie aus Abbildung 5.6a nach dem Parsen beschrieben durch die Zeilen 43-57 in Abbildung 5.6b. Auch hier werden werden also die über die Linie verbundenen Blöcke über ihren Namen und die Nummer des

```
Block {
  BlockType  BusCreator
  Name   "Bus Creator"
  Ports   [2, 1]
  Port {
    PortNumber   1
    Name   "X"
    PropagatedSignals  "a, b"
    [...]
  }
}
Block {
  BlockType   SubSystem
  Name   "Subsystem 1"
  Ports   [0, 2]
  [...]
  Port {
    PortNumber   2
    Name   "a"
  }
  System {
    Name   "Subsystem 1"
    [...]
    Block {
      BlockType   Outport
      Name   "Out1"
    }
    Block {
      BlockType   Outport
      Name   "Out2"
      Port   "2"
      [...]
    }
    [...]
  }
}
Block {
  BlockType   Terminator
  Name   "Terminator 2"
}
Line {
  Name   "a"
  SrcBlock   "Subsystem 1"
  SrcPort   2
  [...]
  Branch {
    DstBlock   "Terminator 2"
    DstPort   1
  }
  Branch {
    [...]
    DstBlock   "Bus Creator"
    DstPort   1
  }
}
[...]
```

(a) Vor dem Parsen

```
<blocks xsi:type="emdl:BusCreator"
  name="Bus Creator">
<ports portNumber="1">
  <options xsi:type="emdl:SimpleOption"
    name="PropagatedSignals" value="a, b"/>
  [...]
</ports>
<portsNumber>
  <numbers>2</numbers>
  <numbers>1</numbers>
</portsNumber>
</blocks>
<blocks xsi:type="emdl:SubSystem"
  name="Subsystem 1">
  [...]
  <ports portNumber="2">
  [...]
  </ports>
  [...]
  <system name="Subsystem 1">
    [...]
    <blocks xsi:type="emdl:BlockOutPort"
      name="Out1">
      [...]
    </blocks>
    <blocks xsi:type="emdl:BlockOutPort"
      name="Out2">
      <options xsi:type="emdl:SimpleOption"
        name="Port" value="2"/>
      [...]
    </blocks>
    [...]
  </system>
  <portsNumber>
    <numbers>0</numbers>
    <numbers>2</numbers>
  </portsNumber>
</blocks>
<blocks xsi:type="emdl:Terminator"
  name="Terminator 2">
  [...]
</blocks>
<mdlLines name="a">
  [...]
  <branches>
    <destinationPointer
      blockString="Terminator 2"
      portNumber="1"/>
  </branches>
  <branches>
    <destinationPointer
      blockString="Bus Creator"
      portNumber="1"/>
    [...]
  </branches>
  <sourcePointer blockString="Subsystem 1"
      portNumber="2"/>
</mdlLines>
[...]
```

(b) Nach dem Parsen

Abbildung 5.6: Textuelle Darstellung des Simulink-Modells aus Abbildung 5.5

betroffenen Ports identifiziert. Möchte man also verbundene Blöcke identifizieren, so muss man über die Linien iterieren und die Quell- bzw. Zielblöcke bzw. -ports im Modell suchen. Diese Aufgabe würde durch die Verwendung eines Navigationspfads der Blöcke bzw. Ports anstelle ihrer Namen deutlich vereinfacht, da ihre Suche im Modell entfallen würde.

Auch empfiehlt es sich, die noch unveränderte Linienstruktur zu optimieren. Die oben genannte Linie besitzt auch nach dem Parsen noch Abzweigungselemente (Z. 45, 50 in Abbildung 5.6b).

Eine weitere Schwierigkeit, die durch das Parsen selbst noch nicht gelöst wird, ist die je nach Blocktyp und Anzahl ein- und ausgehender Linien unterschiedliche Block-Port-Struktur. So hat beispielsweise der Block *Subsystem 1* (Z. 12-36 in Abbildung 5.6a bzw. Z. 13-38 in Abbildung 5.6b) einen Port (Z. 17-20 in Abbildung 5.6a bzw. Z. 16-18 in Abbildung 5.6b), während der Block *Out1* keinen Port besitzt (Z. 24-27 in Abbildung 5.6a bzw. Z. 22-25 in Abbildung 5.6b). *Subsystem 1* besitzt außerdem nicht für jede ausgehende Linie einen Port (z. B. für die Linie zwischen *Subsystem 1* und *Terminator 1* (Abbildung 5.5a). Der laut Zeile 15 in Abbildung 5.6a bestehende erste Ausgangsport wird also implizit angenommen, aber nicht im Code spezifiziert.

Aus dem Modell geht außerdem nicht hervor, welche Ports eines Subsystems mit welchen Schnittstellenblöcken verbunden sind. So drückt Zeile 15 in Abbildung 5.6a zwar aus, dass das in Abbildung 5.5 dargestellte *Subsystem 1* keine Eingangs- und zwei Ausgangsports besitzt. Die Zugehörigkeit der Schnittstellenblöcke *Out 1* und *Out 2* (Zeilen 24, 28) zu den beiden Ausgangsports ist nicht explizit angegeben. Für *Out 1* wird kein Port angegeben, weshalb zu vermuten ist, dass er sich auf den ersten Port bezieht. *Out 2* hingegen wird über das entsprechende Attribut in Zeile 31 dem Ausgangsport 2 des Subsystems zugeordnet. Doch auch die Repräsentation der beiden Blöcke nach dem Parsen (Zeilen 22, 26 in Abbildung 5.6b) vereinfacht die Identifikation des Zusammenhangs zwischen ihnen und den Ausgangsports nicht. Eine direkte Verknüpfung über einen Navigationspfad würde also auch hier die Signalverfolgung über Subsystemgrenzen hinweg vereinfachen.

Schließlich werden auch Signale nicht als eigene Elemente beschrieben, die mit den sie transportierenden Linien und Signalbussen verknüpft werden. Sowohl vor als auch nach dem Parsen werden Signale nur über Liniennamen beschrieben. In Abbildung 5.6a wird zum Beispiel die Linie von *Subsystem 1* zu *Terminator 2* und *Bus Creator* mit dem Signalnamen *a* beschriftet (Z. 42). Der Ausgangsport des *Bus Creators* spezifiziert die Signale im Attribut *PropagatedSignals* ebenfalls über die Auflistung ihrer Namen (Z. 8). Auch das Parsen ändert an dieser Tatsache nichts. (vgl. Z. 5, 43 in Abbildung 5.6b). Um festzustellen, welche Signale in einem Signalbus enthalten sind, muss daher bis zu dem Bus Creator zurück navigiert werden, der den Signalbus erzeugt. Nur diejenigen Linien, die aus einem Block ausgehen, der ein Signal generiert, sind also über die Namensangabe direkt mit diesem assoziiert. Um Signalabhängigkeiten besser analysieren zu können, ist daher die Ablage von Signalen und Signalbussen als eigenständige Elemente sinnvoll, die bidirektional mit den sie transportierenden Linien und Signalbussen über Navigationspfade verknüpft sind.

Schritt	Bezeichnung	Beschreibung
1	Erstellen logischer Ports	Pro aus- und eingehender Linie in bzw. aus einem Block wird ein logischer Port erzeugt
2	Erstellen logischer Linien	Verbinden von Ports über logische Linien
3	Erstellen der Signalstruktur	Signale im Modell werden extrahiert und mit den sie transportierenden Linien bzw. Signalbussen verknüpft

Tabelle 5.1: Aufbereitungsschritte des ASTs für das Simulink-Modell

Um bei der Entwicklung späterer Analysen von diesen Details abstrahieren zu können, wurde in mehreren aufeinander aufbauenden Schritten mit Hilfe von Modelltransformationen in ATL die Modellstruktur optimiert und um zusätzliche Verknüpfungen und Elemente ergänzt. Die Schritte fasst Tabelle 5.1 zusammen[1]. Der Ausschnitt aus Abbildung 5.6b wird dabei zu dem in Abbildung 5.7 gezeigten Ausschnitt aus dem EMF-Modell transformiert. Es entspricht dem Metamodell aus Abbildung 5.8. Die folgenden Abschnitte erläutern die einzelnen Schritte.

Erstellen und Verknüpfen logischer Ports Der erste Schritt besteht darin, eine einheitliche Block-Port-Struktur zu schaffen. Dazu wird jedem Block pro ein- bzw. ausgehender Linie genau ein Port zugewiesen. Damit entfallen spätere Fallunterscheidungen, wodurch zu erwarten ist, dass sowohl die Implementierung späterer Analysen vereinfacht als auch deren Effizienz erhöht wird. So wird beispielsweise jetzt nicht mehr nur für den zweiten Port von *Subsystem 1* ein entsprechendes *outPort*-Element generiert, sondern für beide Ausgangsports (Z. 3/4 von Abbildung 5.7).

Eine weitere Verknüpfung, die die Signalverfolgung über Subsystemgrenzen hinweg erleichtern soll, betrifft die Modellierung des Zusammenhangs zwischen Schnittstellenblöcken und den Ports von Subsystemen (s. o.). Daher werden die Ports in diesem Schritt noch mit den zugehörigen Schnittstellenblöcken verbunden, indem deren Navigationspfad in das Attribut *accordingBlock* geschrieben wird (Z. 7). Zwecks bidirektionaler Verknüpfung wird auch der umgekehrte Weg im Attribut *accordingPort* der Schnittstellenblöcke angegeben (Z. 11). Damit ist das Modell so vorbereitet, dass die Linien mit den Ports verbunden werden können.

Erstellen logischer Linien Für die Verbindung von Ports werden gemäß Abbildung 5.8 für die bereits aus dem Parsen vorhandenen, technischen Linien (Schlüsselwort *mdlLines* in Abbildung 5.6b) logische Linien mit leichter analysierbarer Struktur erstellt. Dazu wird von Abzweigungen abstrahiert und stattdessen pro Linie ein Quell- und mindestens

[1] Es sei angemerkt, dass es sich bei den Schritten um eine thematische Gliederung handelt. Einzelne Schritte sind technisch bedingt teilweise selbst wiederum in mehreren Modelltransformationen umgesetzt.

```
1  <blocks xsi:type="emdl:SubSystem"
2    name="Subsystem 1">
3    <outPorts number="1" [...] />
4    <outPorts number="2"
5      logicalSignals="//@simulinkModel/
           @logicalSignals.0"
6      linkingLine="//@simulinkModel/@system/
           @logicalLines.0"
7      accordingBlock="//@simulinkModel/@system/
           @blocks.0/@system/@blocks.0" [...]/>
8    <system name="Subsystem 1">
9      <blocks xsi:type="emdl:BlockOutPort"
10       name="Out2"
11       accordingPort="//@simulinkModel/@system/
             @blocks.0/@outPorts.1">
12       [...]
13     </blocks>
14     [...]
15   </system>
16   [...]
17 </blocks>
18 <blocks xsi:type="emdl:BusCreator"
19   name="Bus Creator">
20   <outPorts name="Bus Creator"
21     logicalSignals="//@simulinkModel/
           @logicalSignals.1"
22   [...] >
23   <inPorts name="Bus Creator"
24     logicalSignals="//@simulinkModel/
           @logicalSignals.0"
25     linkingLine="//@simulinkModel/@system/
           @logicalLines.0"
26     [...] />
27   [...]
28 </blocks>
29 <blocks xsi:type="emdl:Terminator" name="
       Terminator 2">
30   <inPorts name="Terminator 2"
31     logicalSignals="//@simulinkModel/
           @logicalSignals.0"
32     linkingLine="//@simulinkModel/@system/
           @logicalLines.0"
33     [...] />
34   [...]
35 </blocks>
36 <logicalLines logicalSignals="//@simulinkModel/
       @logicalSignals.0"
37   target="//@simulinkModel/@system/@blocks.1/
       @inPorts.0
38     //@simulinkModel/@system/@blocks.2/@inPorts
           .0"
39   source="//@simulinkModel/@system/@blocks.0/
       @outPorts.1" [...]
40 />
41 <logicalSignals xsi:type="emdl:AtomicSignal"
42   name="a"
43   signalTransporters="
44     //@simulinkModel/@system/@blocks.1/@inPorts
           .0
45     //@simulinkModel/@system/@blocks.2/@inPorts
           .0
46     //@simulinkModel/@system/@blocks.0/
           @outPorts.1
47     //@simulinkModel/@system/@logicalLines.0"
48   busSignals=
49     "//@simulinkModel/@logicalSignals.1"/>
50 <logicalSignals xsi:type="emdl:BusSignal"
51   name="X"
52   atomicSignals="//@simulinkModel/
       @logicalSignals.0
53   [...] />
54 [...]
```

Abbildung 5.7: Die Spezifikation des Inhaltes aus Abbildung 5.6b nach dem Durchlaufen der Aufbereitungsschritte

ein Zielport referenziert. Die Zeiger (*SrcPointer* bzw. *DstPointer* aus Abbildung 5.3), die technische Linien für das Quell- oder Zielobjekt benötigen, können entfallen, da jetzt generell auf Ports verwiesen wird. Beispielsweise wird für die zuvor beschriebene Linie zwischen *Subsystem 1*, *Terminator 2* und *Bus Creator* nun die logische Linie in Zeile 36 erzeugt. Quelle und Ziele dieser Linie werden wiederum als Navigationspfade in den Attributen *source* bzw. *target* spezifiziert. Auch hier wird eine Rückwärtsreferenz von den Quell- bzw. Zielblöcken zur Linie erstellt (Z. 6, 25 und 32).

Erstellen logischer Signale Die Verknüpfungen zwischen Blöcken, Ports und Linien können im letzten Schritt verwendet werden, um zu analysieren, welche Linie welche Signale transportiert und welcher Signalbus welche Signale beinhaltet. Dazu werden neue Entitäten für Signale erstellt und diese mit den Linien und Ports verknüpft, die diese transportieren (vgl. Abbildung 5.8).

Um festzustellen, welche Signale auf einer Linie transportiert werden, muss diese daher durch das Modell rückverfolgt werden bis zum Ursprung der Signale. Dazu wurde

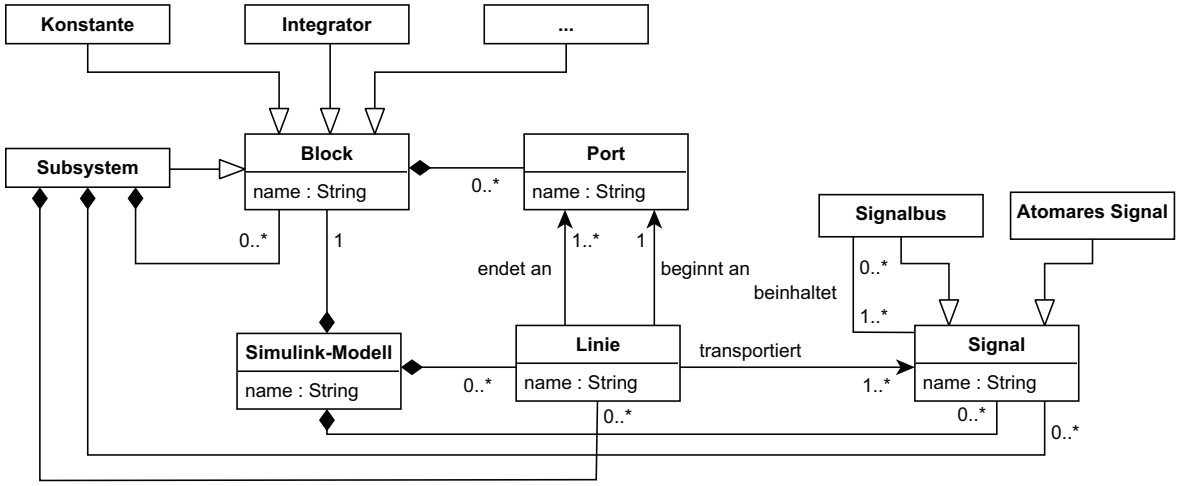


Abbildung 5.8: Ein Ausschnitt aus dem Metamodell für strukturell aufbereitete EMF-Repräsentanten des Simulink-Modells

angenommen, dass eine Linie, die von einem Block ausgeht, den Namen des vom Block erstellten Signals trägt. Des Weiteren kann man zwischen atomaren Signalen und Signalbussen unterscheiden. Atomare Signale transportieren tatsächliche Informationen, während Signalbusse im Simulink-Modell verwendet werden, um Signale zusammenzufassen. Beim Erstellen logischer Signale wird daher überprüft, welche Signale von einem *Bus Creator* zu einem Signalbus zusammengefasst werden und so in die in Abbildung 5.8 dargestellte Signalstruktur eingefügt.

In Abbildung 5.7 findet sich ein atomares Signal in Zeile 41. Der Pfad zu diesem Signal wird von den Linien und Ports, die es transportieren, durch das Attribut *logicalSignals* referenziert (Z. 5, 24, 31 und 36). Umgekehrt referenziert der Pfad im Signalattribut *signalTransporters* die Linien und Ports, über die es transportiert wird (Z. 43).

Die Signalstruktur aus atomaren Signalen und Signalbussen wird durch die Attribute *busSignals* bzw. *atomicSignals* ausgedrückt. In Abbildung 5.7 stellt Zeile 50 beispielsweise einen Signalbus dar. Über das Attribut *atomicSignals* referenziert ein Signalbus die atomaren Signale, die er transportiert (Z. 52). Dort erfolgt die Rückreferenz auf diesen wiederum im Attribut *busSignals* (Z. 48).

5.4.2 Lastenheft

Wie in Abschnitt 5.3.2 beschrieben, wird das Lastenheft zunächst als CSV-Datei aus IBM Rational DOORS exportiert, um es in ein EMF-Modell umzuwandeln. Dazu wird die Grammatik der CSV-Datei analog zur Grammatik des Simulink-Modells in Xtext beschrieben und der zugehörige Parser damit generiert. Es ergibt sich ein AST, der dem grau hinterlegten Teil in Abbildung 5.4 entspricht, d. h. es liegt eine Liste von Zeilen der Tabelle vor, an die jeweils die Spaltenwerte als Attribute gehängt sind. Da jedoch jede Zeile eine Hierarchietiefe in Bezug auf die Dokumentstruktur hat, wird dieses Modell nun noch in die baumartige Struktur des Lastenheftes transformiert, sodass sich die in

Abbildung 5.4 weiß hinterlegte Struktur ergibt. Die Struktur wird wiederum analog zur Aufbereitung des Simulink-Modells über Pfade zwischen den Elementen modelliert.

5.5 Annotationen

5.5.1 Integration in MATLAB/Simulink

Gemäß dem in Abschnitt 4.2 vorgestellten Konzept sollte die Dokumentation in möglichst einheitlicher Form durch formale Annotationen anstatt durch Freitext gemäß Abbildung 3.2 erfolgen. Diese sollten in einem zentralen Datenmodell abgelegt werden, um für verschiedene Artefakte verwendet werden zu können. Außerdem sollte es möglich sein, den Annotationsprozess möglichst nahtlos in die verwendeten Werkzeuge einzufügen, um möglichst wenige neue Aufwände für Anwender zu schaffen. Im Rahmen der Dissertation wurde zwar ein artefaktübergreifendes Konzept erstellt, jedoch nur prototypisch für die Verwendung in MATLAB/Simulink realisiert. Die Verbindung zwischen Datenbasis der Annotationstypen und Lastenheft ist in Abbildung 5.2 daher nur gestrichelt dargestellt. Annotationen wurden also für den modellbasierten Ansatz nicht *artefaktübergreifend* verwendet, da der Fokus hier primär auf der Analyse von Simulink-Modellen lag und für derartige Analysen der datenbankorientierte Analyseansatz geeigneter schien. Im Folgenden beschränke ich mich daher auf die Annotation von Simulink-Modellen. Des Weiteren wurde für diesen Ansatz nur die Menge der Annotations*typen* vorgegeben, während die Werte frei spezifiziert werden.

Als zentrale Ablage für Annotationstypen wurde eine relationale Datenbank verwendet. Die Entscheidung für eine relationale Datenbank anstelle einer dateibasierten Ablage von Annotationen in einem EMF-Modell fiel einerseits, da diese eine etablierte Art des Datenmanagements darstellen und leicht über die Database Toolbox von MATLAB/Simulink [146] abgefragt werden können. Andererseits war bereits geplant, im nächsten Schritt den gesamten Ansatz auch datenbankorientiert umzusetzen und die Annotationen in einer Datenbank mit den ebenfalls dort abgelegten Artefakten zu verknüpfen (vgl. Kapitel 6).

Der Anwender kann beliebige Subsysteme im Modell zur Annotation selektieren. Um die möglichen Annotationstypen auf jene in der Datenbank zu beschränken, liest ein MATLAB-Skript diese aus der Datenbank aus und stellt sie dem Anwender zur Auswahl. Welche Annotationstypen möglich sind, wurde auf Basis der Dokumentationsblöcke (vgl. Abschnitt 3.3) aus beispielhaften Modellen des Industriepartners festgelegt, jedoch ist die Datenbank beliebig um weitere Typen erweiterbar.

Nach der Auswahl des Annotationstyps erfolgt noch die Eingabe des Wertes als Freitext durch den Benutzer. Diesen Vorgang wiederholt der Anwender so lange bis er alle Annotationen für das gewählte Subsystem spezifiziert hat.

Die dann vorhandenen Tupel aus ausgewähltem Typ und frei spezifiziertem Wert werden als Array mit Hilfe der *set_param*-Methode der MATLAB API [147] in den Parameter *UserData* des Subsystems geschrieben.

Grundsätzlich könnte man auch Annotationen für beliebige Blöcke zulassen. Dies geschieht jedoch aktuell nicht (vgl. Abschnitt 3.3). Für eine Modellgröße von ca. 30.000 Blöcken, wie beim Industriepartner der Fall, ist ein solcher Ansatz zu feingranular, sodass der Aufwand für das Annotieren von Blöcken zu groß wird. Daher wurde als kleinste annotierbare Einheit im Simulink-Modell das Subsystem gewählt.

5.5.2 EMF-Integration der Annotationen

Nun sind die Annotationen im beschriebenen Typ-Wert-Format in das Simulink-Modell integriert. Um sie für modellbasierte Analysen verwenden zu können, müssen sie allerdings in den EMF-Repräsentanten des Simulink-Modells integriert werden, da die zugehörigen Transformationen auf diesen basieren. Dies kann entweder dadurch geschehen, dass man den Modellimport des Simulink-Modells – also den Parser – anpasst oder durch Übertragung vom Simulink-Modell auf seinen bereits vorhandenen EMF-Repräsentanten. Im ersten Fall ergeben sich zwei Schwierigkeiten: Der *UserData*-Parameter enthält die annotierten Werte (hier ein Array aus Typ-Wert-Paaren) in verschlüsselter Form, also nicht als Klartext. Dadurch sind sie vom Parser nicht sinnvoll zu verarbeiten. Außerdem müsste für jede hinzugefügte Annotation der EMF-Repräsentant des Simulink-Modells vollständig neu erzeugt werden. Daher wurde vom zweiten Ansatz Gebrauch gemacht, dessen Ablauf in Abbildung 5.2 integriert ist.

Zunächst durchläuft ein MATLAB-Skript die Subsysteme des Simulink-Modells, extrahiert die Annotationen aus dem Parameter *UserData* und generiert ein Dokument, das die Annotationen in einer einfachen, domänenspezifischen Annotationssprache beschreibt (vgl. Abbildung 5.1).

```
1 AnnotationModel AnnotModel {
2   annotations {
3     Annotation 0 {
4       path    "parkingAssistant/Sensors/Distance/LeftDistance/LeftInfraredSensor"
5       type    "Merkmal"
6       value   "Infrarotsensor"
7       [...]
8     },
9     [...]
10 }
```

Listing 5.1: Ein Ausschnitt aus einem beispielhaften Annotationsdokument

Ein solches Annotationsdokument beginnt mit dem Schlüsselwort *AnnotationModel* gefolgt von einem Bezeichner. Nach dem Schlüsselwort *annotations* werden die Annotationen in einem Block von geschweiften Klammern aufgelistet, wobei jede Annotation mit dem Schlüsselwort *Annotation* beginnt gefolgt von einer numerischen ID. Ihre Attribute werden dann ebenfalls in einem Block von geschweiften Klammern aufgelistet. Das Attribut *path* enthält den Simulink-Pfad zu dem Subsystem, aus dem die Annotation stammt. Anschließend werden weitere Attribute, insbesondere Annotationstyp und -wert spezifiziert. Im Beispiel wird aus dem Subsystem *LeftInfraredSensor* die Annotation mit Typ *Merkmal* und Wert *Infrarotsensor* ausgelesen (vgl. Abbildung 2.4).

Der nach dem Parsen dieses Dokumentes vorhandene AST beschreibt somit den grau hinterlegten Teil des in Abbildung 5.9 beschriebenen Metamodells. Alle Annota-

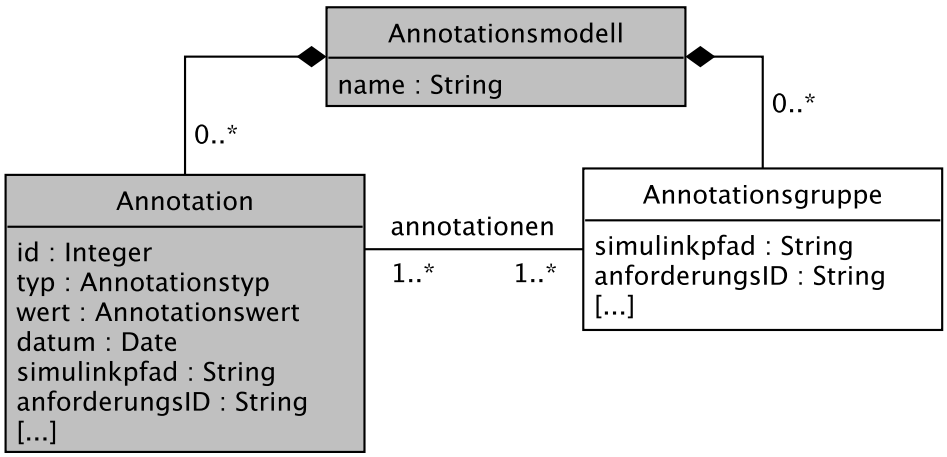


Abbildung 5.9: Das Annotationsmetamodell (grau: Metamodell des ASTs nach dem Parsen des Annotationsdokuments

tionen werden also dem Wurzelelement Annotationsmodell zugeordnet und besitzen Attribute für die Spezifikation von Typ und Wert sowie die Identifizierer der Artefaktelemente, auf die sie sich beziehen. Dies kann der Pfad zu einem Subsystem im Simulink-Modell sein (Attribut *simulinkPfad*), wie in Zeile 4 in Abbildung 5.1, aber theoretisch auch eine Anforderung (Attribut *anforderungsID*). Auch können anhand dieser Attribute während der folgenden Aufbereitung (vgl. Schritte $A_2 \dots A_m$ in Abbildung 5.2) Annotationsgruppen erstellt werden, die alle für ein Subsystem oder eine Anforderung relevanten Annotationen zusammenfassen. So können Annotationsduplikate eliminiert werden und in späteren Modelltransformationen (vgl. Schritte $X_1 \dots X_k$ in Abbildung 5.2) die EMF-Repräsentanten der Subsysteme bzw. Anforderungen (vgl. Abbildung 5.8 bzw. Abbildung 5.4) komfortabel mit den für sie relevanten Annotationen bzw. Annotationsgruppen aus dem Annotationsmodell verknüpft werden.

Nun können die als Annotationen beschriebenen Information über den Entwicklungs- oder Evolutionsprozess auch in automatische Analysen (vgl. Abschnitt 5.6) einbezogen werden, da die Annotationen eines Subsytems über den entsprechenden Link von den Repräsentanten ins Annotationsmodell ausgelesen werden können.

5.6 Analysen und ihre Integration in MATLAB/Simulink

Die Analysen des modelbasierten Ansatzes bestehen ebenfalls aus einer Hintereinanderausführung von Modelltransformationen in ATL. Diese können in Ant-Skripten [6] zusammengefasst werden, um dem Anwender die einzelnen Schritte zu ersparen.

Um den Arbeitsfluss in der Praxis nicht durch häufiges Wechseln der Werkzeuge zu stören, ist es zudem sinnvoll, die Analysen direkt aus den verwendeten Werkzeugen starten und im Falle von Simulink-Modellen auch deren Ergebnisse dort visualisieren zu können.

Für die Dissertation habe ich mich dafür auf die Integration in MATLAB/Simulink beschränkt.

Die Integration von Analysen in MATLAB/Simulink ist über entsprechende Kontextmenüeinträge im Simulink-Modell möglich. Sodann wird eine TCP/IP-Verbindung zu einem Modelltransformationsserver aufgebaut, der anhand der übergebenen Parameter entscheidet, welche Analyse auszuführen ist, und das zu der Analyse gehörende ANT-Skript von Modelltransformationen startet.

Jede Analyse resultiert in einem weiteren EMF-Modell, welches das Analyseergebnis darstellt. Um dieses zu visualisieren, wird das Graphical Modeling Framework (GMF) [38] verwendet. Handelt es sich strukturell wieder um ein Simulink-Modell (z. B. einen Ausschnitt aus dem analysieren Modell), so kann das EMF-Modell auch wieder in ein Simulink-Modell umgewandelt werden[1]. Dadurch kann man auch das Problem der recht aufwändigen, übersichtlichen Visualisierung der EMF-Ergebnismodelle mit Hilfe des GMFs umgehen, da man die Positionierungsinformationen aus MATLAB/Simulink für die Anordnung der Blöcke verwenden kann. Dadurch erkennt außerdem der Ingenieur Strukturen aus dem untersuchten Modell leichter wieder. Diese Generierung eines Simulink-Modells ist der Grund dafür, dass bei diesem Analyseansatz alle Informationen des analysierten Simulink-Modells in dessen EMF-Repräsentanten übernommen werden (vgl. Abschnitt 7.2.1).

5.7 Optimierungsansätze

Der Vollständigkeit halber sei an dieser Stelle bereits darauf hingewiesen, dass sich der modellbasierte Ansatz in der vorgestellten Form als wenig effizient, speicherintensiv und nicht hinreichend skalierbar erweist (vgl. Abschnitt 7.2.1). Daher wurde der Ansatz besonders im Hinblick auf zwei Aspekte verbessert.

Der erste Ansatz reduziert die Modellgröße der EMF-Modelle durch Entfernen von Informationen, die für die hier betrachteten Ergebnisse irrelevant sind (z. B. Attribute und Parameter des Simulink-Modells und seiner Elemente).

Auf gleichermaßen reduzierten Modellen operiert die zweite Optimierungsmethode, erzeugt aber zusätzlich nur die Grundstruktur (d. h. den Aufbau aus Blöcken, Linien und Subsystemen) über Parser und Modelltransformationen. Für die Erstellung von Signalen wird auf ein Java-Rahmenwerk für Graphverarbeitung zurückgegriffen. Dieses Vorgehen ist sinnvoll, da sich sowohl bei Betrachtung der Laufzeiten der Einzelschritte als auch hinsichtlich ihrer Komplexität in der Handhabung die Erstellung der Signalstruktur als besonders aufwändig herausstellt.

Aufgrund der reduzierten Modellinformationen haben jedoch beide Optimierungsansätze den Nachteil, dass aus den Ergebnismodellen keine Simulink-Modelle mehr ohne Hinzunahme des Originalmodells generiert werden können, also deren Visualisierung in MATLAB/Simulink erschwert wird. Um dieses Problem zu umgehen, wird daher der erstellte Repräsentant so aufbereitet, dass er mit Hilfe des datenbankorientierten

[1] Dieses ist ausschließlich zur Visualisierung gedacht. Es wurde nicht geprüft, ob sich daraus fehlerfreier Code generieren lässt.

Ansatzes in MATLAB/Simulink integriert werden kann und somit der Rückgriff auf das Originalmodell möglich wird.

Details und die dadurch erzielten Vorteile dieser Optimierungsmöglichkeiten gegenüber dem ursprünglichen Ansatz werden im Zuge der Evaluation in Abschnitt 7.2.1 beschrieben.

5.8 Verwandte Arbeiten

Fachlich verwandte Arbeiten wurden bereits in Abschnitt 4.5 beschrieben. Hier beschränke ich mich daher einerseits auf die Beschreibung von Arbeiten, die (u. U. in fachlich verschiedenem Kontext) ähnliche Techniken anwenden wie der in diesem Kapitel beschriebene, modellbasierte Ansatz. Andererseits wird auf die unterschiedliche Umsetzung fachlich verwandter Arbeiten eingegangen.

In Abschnitt 4.5.3 wurden verschiedene existierende Analysewerkzeuge für Simulink-Modelle vorgestellt. Allen gemeinsam ist, dass sie entweder die Modelle nach verschiedenen Kenngrößen untersuchen und entsprechende Reporte erzeugen oder die Originalmodelle so modifizieren, dass sie den allgemeinen Modellierungsrichtlinien des MAAB oder unternehmensspezifischen Richtlinien entsprechen. Dem Autor ist jedoch aktuell kein Analysewerkzeug bekannt, welches die Sichtenextraktion auf Simulink-Modelle erlaubt. Die benannten Arbeiten implementieren die Modellprüfungen zumeist mittels Matlab-Skripten (z. B. MXAM [93], MINT [125] und Model Advisor) oder deklarativ, graphtransformationsbasiert mittels Story-Driven Modelling (SDM) (z. B. MATE [157]). Erstere Möglichkeit weist gemäß Amelunxen et al. den Nachteil auf, dass die Abstraktionsebene sehr gering ist, wodurch die Implementierung insbesondere evolutionsbedingt kompliziert wird [2]. Daher präferieren sie die SDM-Methode für strukturelle Analysen. Diese basiert auf Triple-Graph-Grammatiken von Schürr et al. [134], die wiederum die Paar-Graph-Grammatiken von Pratt [121] um kontextsensitive Produktionsregeln erweitern, die auch $n : m$-Relationen modellieren können. Eine Transformationsregel wird hier strukturell als Tripel (LG, CG, RG) beschrieben, wobei LG und RG die linke bzw. rechte Seite der Regel beschreiben und der Korrespondenzgraph CG, beschreibt, wie diese miteinander in Relation stehen. Diese werden dabei grafisch vom Benutzer spezifiziert und von einer Modelltransformationsmaschine operationalisiert. Dadurch muss der Anwender keine zeitliche Abfolge von Transformationsschritten berücksichtigen. Einen beispielhaften TGG-Interpreter stellt die Universität Paderborn im Internet bereit [155]. Für die Sichtenextraktion auf Simulink-Modelle dieser Dissertation bietet sich das SDM nicht an, da sich, wie ebenfalls von Amelunxen et al. herausgestellt, diese Spezifikationsart nicht für die Nachverfolgung komplexer Pfade durch das Modell anbietet. Dies ist für die hier betrachteten Modellanalysen häufig nötig. Beispielsweise kann die Information, welche Signale über eine Linie transportiert werden, aus dem Graphen nur durch Zurückverfolgen von Pfaden extrahiert werden. Dies ist zum Teil kompliziert, da nicht alle Blöcke, zwischen denen Signale fließen, auch unbedingt über Kanten miteinander verbunden sind. So ermöglichen spezielle Sprung-Blöcke derartige Signalflüsse zwischen beliebigen Teilen des Simulink-Modells.

Auch für die Prüfung regulärer Ausdrücke oder arithmetischer Berechnungen eignet sich die SDM-Methode nicht [2]. Daher fiel die Wahl für die vorliegende Dissertation auf die hybride Transformationssprache ATL, um auch imperative Spezifikationselemente für Modelltransformationen nutzen zu können.

Ein modellbasiertes Konzept für die durchgängige Entwicklung mit Fokus auf die Automotive-SPICE[1]-konforme Entwicklung in der Automobilindustrie beschreiben Holtmann et al. [52]. Zentral sind hier die meist späten Tests und Validierungen im Entwicklungsprozess. Dies gefährdet die Einhaltung von Sicherheitsstandards. Auch wird die Spezifikation von Anforderungen in natürlicher Sprache als problematisch betrachtet, da diese informal, oft mehrdeutig und somit nicht automatisch zu verarbeiten ist. Abhilfe soll die vorgestellte Controlled Natural Language (CNL) schaffen, die die Ausdrucksstärke und Mehrdeutigkeit natürlicher Sprache einschränkt, indem sogenannte Requirements Patterns gebildet werden. Dies sind vordefinierte Phrasen, die z. B. durch Namen von Subsystemen und Signalen ergänzt werden und die automatisch validiert werden können. Auch können die Anforderungen so durch Parsen in Modelle transformiert werden, die mit Hilfe von Modelltransformationen weiter verarbeitet werden können. Das so geschaffene Modell kann folglich zu Analysezwecken im Hinblick auf Rückverfolgbarkeit verwendet werden.

Polzer et al. beschäftigen sich damit, welche Herausforderungen sich ergeben, wenn der Software-Produktlinien-Ansatz auf die modellbasierte Softwareentwicklung im Umfeld regelungstechnischer Anwendungen übertragen wird [115, 116]. Die Autoren identifizieren insbesondere, dass dadurch nicht die Vorzüge modellbasierter Softwareentwicklung (z. B. Codegenerierung und Simulationsfähigkeit) verloren gehen dürfen. Des Weiteren wird durch Variabilität zusätzliche Komplexität in Modelle integriert. Schließlich kann die Ableitung einer Variante auch andere Varianten beeinflussen, sodass Gefahr von Inkonsistenzen besteht. Für die Adressierung dieser Herausforderungen stellen die Autoren ein EMF-basiertes Rahmenwerk vor und berichten über dessen prototypische Anwendungen. Auch hier wird ein Simulink-Modell geparst und in ein EMF-Modell geschrieben.

Botterweck et al. befassen sich mit der Konfiguration und Ableitung von Varianten eines Simulink-Modells [19] bei Anwendung des 150%-Ansatzes auf Basis von Modelltransformationen in ATL und dem EMF. Während andere Ansätze sich mit der isolierten Konfiguration eines Merkmalmodells beschäftigen, ohne das zugrunde liegende Implementationsmodell dabei in Betracht zu ziehen, zielt der hier präsentierte Ansatz darauf ab, auch die aus dem Simulink-Modell resultierenden Abhängigkeiten bereits bei der Konfiguration zu berücksichtigen. Das Simulink-Modell wird dazu auch hier mit Xtext geparst, während das Merkmalmodell bereits als EMF-Modell vorliegt, da das Werkzeug S2T2 [79] dieses erzeugt, wodurch kein Parsen notwendig ist. Modelltransformationen werden hier verwendet, um bereits während der Variantenkonfiguration nicht nur das zugehörige Merkmalmodell, sondern auch Abhängigkeiten innerhalb des Simulink-Modells

[1] Automotive-SPICE (Software Process Improvement and Capability Determination) ist eine Norm für die Bewertung des Entwicklungsprozesses bei Zulieferern der Automobilindustrie. Für Details sei der Leser auf [96] verwiesen.

zu berücksichtigen, wobei ein Merkmal A von einem Merkmal B abhängt, wenn im Simulink-Modell eine Kante zwischen einem mit B assoziierten Block und einem mit A verbundenen Block existiert. Die Aufbereitung des EMF-Repräsentanten des Simulink-Modells zum Zwecke von Modellanalysen ist hier folglich nicht erforderlich, da keine Signale betrachtet und keine transitiven Abhängigkeiten berücksichtigt werden müssen. Schließlich lässt sich gemäß dem Konzept negativer Variabilität aus dem integrierten Produktlinienmodell ein 100%-Modell ableiten.

Um den Anforderungen von Thomas et al. [150] gerecht zu werden, genügt es nicht, Produktlinien statisch, d. h. ohne Berücksichtigung der Evolution, zu betrachten. Insbesondere können vorhandene Variationspunkte evolutionsbedingt aufgelöst oder neue ergänzt werden, die bei der Initiierung der Produktlinie nicht vorhersehbar waren. Daher strebt die Daimler AG ein nahtloses, zentrales Variantenmanagement an, welches nicht nur Artefakte und Merkmale verknüpfen kann, sondern den Anwender durch Analysen auch bei der evolutionsbedingten Überarbeitung der Produktlinie unterstützt. Botterweck et al. greifen dazu die Planungsphase der Produktlinie auf. Sie merken an, dass existierende Produktlinienansätze die Evolution zwar oft schon methodisch unterstützen (z. B. durch Scoping), jedoch dabei die Werkzeugunterstützung vernachlässigen [17, 18]. Andere Ansätze betrachten isoliert die Werkzeugunterstützung, berücksichtigen aber die Planung nicht hinreichend. Das vorgestellte Werkzeug *EvoFM* setzt daher für die Merkmalmodellierung auf der Modellebene an, da dadurch nach Ansicht der Autoren ein hinreichend ausgewogenes Verhältnis zwischen Abstraktion und Konkretisierung gegeben ist, um vorausschauend zu planen und dennoch Werkzeugunterstützung zu berücksichtigen. Für die Realisierung wird auch hier auf EMF und ATL zurückgegriffen. Abgesehen von der Unterstützung bei der Planung ist der Benutzer nach Angaben von Thomas et al. [150] aber auch bei der Artefaktanalyse zu unterstützen, um Änderungen und deren Auswirkungen leichter überblicken zu können. Diesem Punkt widmet sich die vorliegende Dissertation.

6 Datenbankorientierte Analyse

Zwar erweist sich der in Kapitel 5 vorgestellte, modellbasierte Analyseansatz beim Industriepartner auf Basis des produktiv eingesetzten 150%-Simulink-Modells als grundsätzlich anwendbar, jedoch zeigt sich, dass der Ansatz für den regelmäßigen Einsatz zu kompliziert zu handhaben ist. Des Weiteren offenbart er Mängel hinsichtlich Speicherbedarf, Fehleranfälligkeit, Effizienz und Skalierbarkeit (vgl. Kapitel 7). Daher wurde im Rahmen der Dissertation ein zweiter Analyseansatz implementiert, der durch die Kombination einer Datenbank als artefaktübergreifendes Datenmodell mit Java-Funktionalität die Handhabbarkeit verbessert.

Da der hier beschriebene Ansatz sich während der Entwicklung als komfortabler erwies (vgl. Kapitel 7), wurde er von Anfang an stärker als integrative Lösung für die Artefaktintegration und -verknüpfung sowie für die Umsetzung des Annotationskonzeptes und die Konsistenzprüfung konzipiert und besitzt daher einen deutlich größeren Funktionsumfang als der modellbasierte Ansatz.

Nach einer kurzen Vorstellung der verwendeten Werkzeuge und Rahmenwerke wird ein Überblick über den Ablauf und die zugrunde liegende Architektur präsentiert. Anschließend werden die Datenmodelle der Artefakte erläutert, bevor detaillierter auf die Funktionsweise ihrer Integration, Annotation und Analyse eingegangen wird. Abschnitt 6.8 grenzt den Ansatz schließlich von verwandten Arbeiten ab.

6.1 Werkzeuge und Rahmenwerke

Die Realisierung für den datenbankorientierten Analyseansatz wurde mit den Eclipse Modeling Tools [37] vorgenommen.

Das zugrunde liegende Implementationsmodell stammt aus MATLAB/Simulink 2009b [149]. Der wesentliche, relevante Unterschied zu 2007b, welches für den modellbasierten Ansatz verwendet wurde, beschränkt sich aber hier auf ein weiteres Attribut *SID*, welches einen Block im Simulink-Modell und (nach dem Import) dessen Repräsentanten in der Datenbank eindeutig identifiziert. Um einen Block eindeutig zu identifizieren, könnte man alternativ auch den Navigationspfad durch die Subsysteme im Modell wählen. Da diese Zeichenkette für hohe Hierarchietiefen extrem lang werden kann, ist jedoch aus Effizienzgründen davon abzuraten.

Als zugrunde liegende Datenbank wurde eine relationale MySQL-Datenbank [109] gewählt. Gemäß Herstellerangaben ist diese Datenbank dazu in der Lage, auf Datenmengen mehrerer TB effizient zu arbeiten[1]. Alternativ hätte man auch eine objektorientierte

[1] Dies ist selbstverständlich u. a. abhängig vom zur Verfügung stehenden Arbeitsspeicher des Rechners.

Datenbank wählen können. Für strukturelle Analysen des Simulink-Modells wären ebenfalls Graphdatenbanken geeignet. Diese arbeiten mit Traversierungsalgorithmen, die es ermöglichen, ausgehend von Knoten eines Graphen den Transitionen zu folgen und so gezielt Informationen aufzusammeln. Werden ausschließlich derartige Analysen beabsichtigt, so eignet sich also auch diese Datenbankgattung. Da relationale Datenbanken aber nach wie vor am weitesten verbreitet sind, ist in der industriellen Praxis aufgrund der damit einhergehenden proprietären Unterstützung durch herkömmliche Werkzeuge (z. B. über die Database Toolbox von MATLAB/Simulink [146]) eine höhere Akzeptanz zu erwarten als bei Verwendung anderer Datenbankarten. Derartige Werkzeuge können somit leichter mit dem beschriebenen Ansatz kombiniert werden. Zudem werden auch andere Artefakte in die Datenbank integriert, die keine Graphstruktur besitzen, z. B. Tabellen von Anforderungen. Schließlich sollen auch Analysen durchgeführt werden können, für die die Graphtraversierung nicht von Vorteil ist (z. B. einfache Abfragen von Annotationen). Daher fiel die Entscheidung auf eine relationale Datenbank, die den besten Kompromiss hinsichtlich der hier adressierten Anforderungen darstellt.

Die gesamte Funktionalität des Rahmenwerks wurde mit Java entwickelt. Für den Datenbankzugriff wurde auf ausgereifte Rahmenwerke zurückgegriffen. Für das objektrelationale Mapping von Java-Objekten auf eine relationale Datenbankstruktur wurde Hibernate [60] verwendet. Des Weiteren wurde Gebrauch von der Java Persistence API [107] (JPA) gemacht. Die graphische Oberfläche wurde teilweise mit dem Standard Widget Toolkit (SWT) [40] und teilweise mit Swing umgesetzt.

Für strukturelle Analysen des Simulink-Modells und die Darstellung von Analyseergebnissen wurde neben MATLAB/Simulink Gebrauch von JGraphX [61] gemacht, einem Java-Rahmenwerk für die Verarbeitung und Visualisierung von Graphen.

Für den hier beschriebenen Analyseansatz wurden auch Testfallbeschreibungen aus dem Werkzeug TPT [113] in Version 4.0 in die Datenbank importiert. Dabei handelt es sich um ein Werkzeug der Firma PikeTec, welches zum modellbasierten Testen eingebetteter Systeme eingesetzt wird. Es unterstützt insbesondere die Testablaufbeschreibung durch Zustandsautomaten sowie die automatisierte Durchführung, Auswertung und Dokumentation.

Das Lastenheft liegt in derselben Form wie in Kapitel 5 vor.

6.2 Ablauf und Architektur

Dieser Abschnitt beschreibt den allgemeinen Ablauf und den Aufbau des datenbankorientierte Rahmenwerks, ohne die Einzelschritte im Detail zu erläutern. Dies erfolgt nach der Präsentation der Datenmodelle im weiteren Verlauf dieses Kapitels.

6.2.1 Konzeptbeschreibung

Abbildung 6.1 beschreibt den datenbankorientierten Ansatz konzeptionell. Die „Außenwelt“ bilden die herkömmlichen Entwicklungswerkzeuge, d. h. MATLAB/Simulink, IBM Rational DOORS, S2T2 und TPT. Der Import der Artefakte erfolgt über das

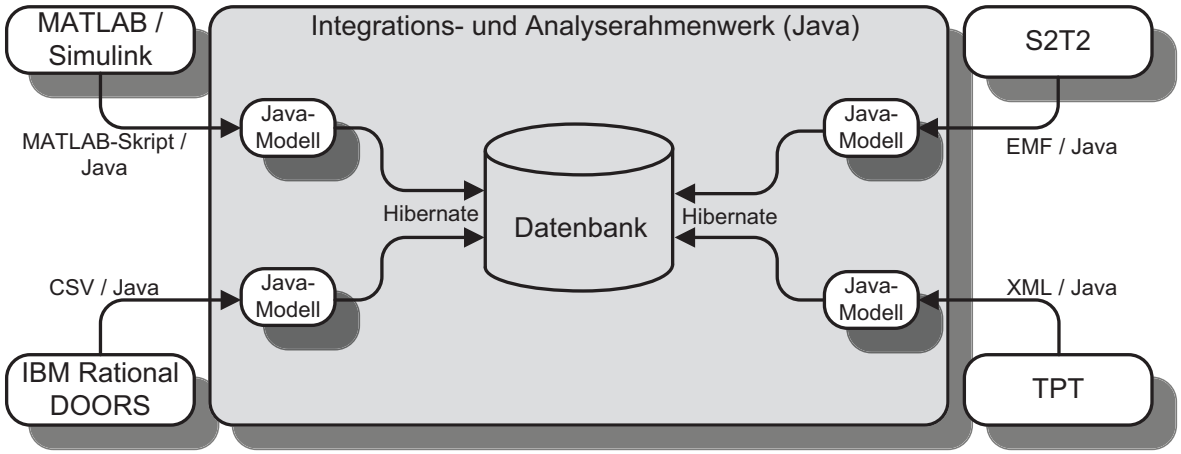


Abbildung 6.1: Datenbankorientiertes Artefaktintegrations- und -analysekonzept

in Java implementierte Integrations- und Analyserahmenwerk, das die Daten in einer relationalen Datenbank ablegt, in der diese miteinander verknüpft und annotiert werden können. Die detaillierte Erläuterung des Artefaktimports erfolgt in Abschnitt 6.4. Ein direkter Zugriff auf die Datenbank ist zwar nicht vorgesehen aber möglich. Dies kann nützlich sein, wenn ein Werkzeug eigene Funktionalität für den Zugriff auf eine relationale Datenbanken besitzt. Einen solchen Direktzugriff bietet die Database Toolbox von MATLAB/Simulink an, die zum Beispiel genutzt werden kann, um einfache Abfragen (z. B. die Abfrage von Subsystemen, die auf eine bestimmte Art annotiert sind) an die Datenbank zu richten. Die Funktionen des Rahmenwerks können teilweise direkt aus MATLAB/Simulink heraus aufgerufen werden, insbesondere die Analyse-, Annotations- und Importfunktionalität für Simulink-Modelle. Für den Import und die Annotation von Lastenheft, Tests und Merkmalmodellen sowie für die Verwaltung von Annotationstypen und die Durchführung von Konsistenzuntersuchungen besitzt das Rahmenwerk ein eigenes Verwaltungswerkzeug. Im Folgenden wird die Architektur des Rahmenwerks kurz beschrieben.

6.2.2 Architektur

Die wesentlichen Bestandteile des entwickelten Rahmenwerks sind in Abbildung 6.2 dargestellt. Demnach besteht es aus fünf Komponenten. Die Verwaltungs- und Importkomponente dient dem Import von Artefakten aus den Originalwerkzeugen in das Rahmenwerk sowie der Verwaltung der Datenbank. Nach dem Import liegen die Artefakte als Java-Modelle vor, die über die Datenbankzugriffskomponente persistiert werden können. Ebenso können einmal importierte Modelle aus der Datenbank ausgelesen werden. Die Analyse- und Annotationskomponente führt die Analysen durch und ermöglicht die Annotation der Artefakte auf Basis der Modelle sowie der in der Datenbank abgelegten Annotationstypen. Analyseergebnisse von Simulink-Modellen werden über die Visualisierungskomponente dargestellt, die sowohl über MATLAB/Simulink als auch mit Hilfe von Funktionen des JGraph-Rahmenwerks [61] erfolgen kann. Die

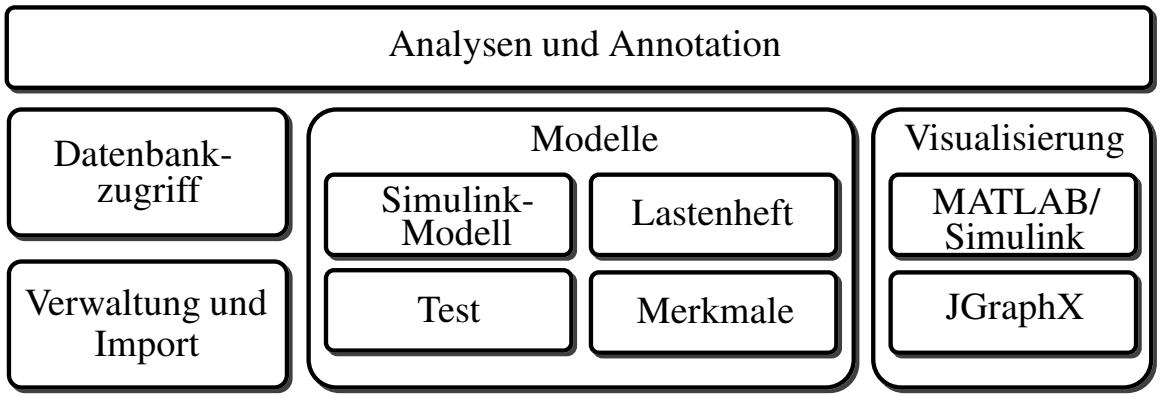


Abbildung 6.2: Architekturbestandteile des datenbankorientierten Integrations- und Analyserahmenwerks

Funktionsweisen von Import und Analysen sowie die einzelnen Komponenten werden im weiteren Verlauf dieses Kapitels jeweils dort erläutert, wo diese relevant werden.

6.3 Datenmodell

Der in Abbildung 6.3 dargestellte Auszug aus dem Datenmodell der Datenbank veranschaulicht die Artefaktintegration über ein gemeinsames Wurzelelement in UML-Notation. Der in diesem Kapitel beschriebene Ansatz berücksichtigt mehr Artefakte als der modellbasierte Ansatz in Kapitel 5. So werden hier nicht nur die Anforderungen und das Simulink-Modell integriert, sondern zusätzlich das Merkmalmodell und die Testspezifikation berücksichtigt. Jedes Artefakt bezieht sich gemäß dem in Abschnitt 2.2.4 beschriebenen 150%-Ansatz auf die gesamte Produktlinie. Daher werden sie in der Datenbank einem gemeinsamen Wurzelelement *Produktlinie* zugewiesen. Zwar wurde der Datenbankansatz bisher nur für die Analyse einer einzigen Produktlinie implementiert, jedoch ließe sich das Rahmenwerk dadurch leicht auf die Verwaltung mehrerer Produktlinien erweitern. Durch den Einbezug des Merkmalmodells der Produktlinie wird das Potenzial des Analyserahmenwerkes über die Verknüpfung von Merkmalen mit Elementen der verschiedenen Artefakte auch auf variabilitäts- und konsistenzbezogene Analysen erweitert, die mit dem modellbasierten Ansatz aktuell nicht möglich sind (vgl. Fallstudie in Abschnitt 7.1.6). Die Beziehungen zwischen den Artefakt*elementen* werden über Cliquen integriert (vgl. Abschnitt 6.7.1).

Da sich das Datenmodell der Anforderungen nicht wesentlich vom dem weißen Teil des Metamodells des modellbasierten Ansatzes in Abbildung 5.4 unterscheidet, wird es an dieser Stelle nicht mehr detailliert beschrieben.

Die natürlichsprachliche Beschreibung der Tests erfolgt auf dieselbe Art und Weise wie die Beschreibung der Anforderungen in IBM Rational DOORS. Da folglich das Datenmodell der Testbeschreibungen dem der Anforderungen entspricht, gehe ich an dieser Stelle nicht mehr darauf ein. Die Testabläufe werden über Zustandsautomaten

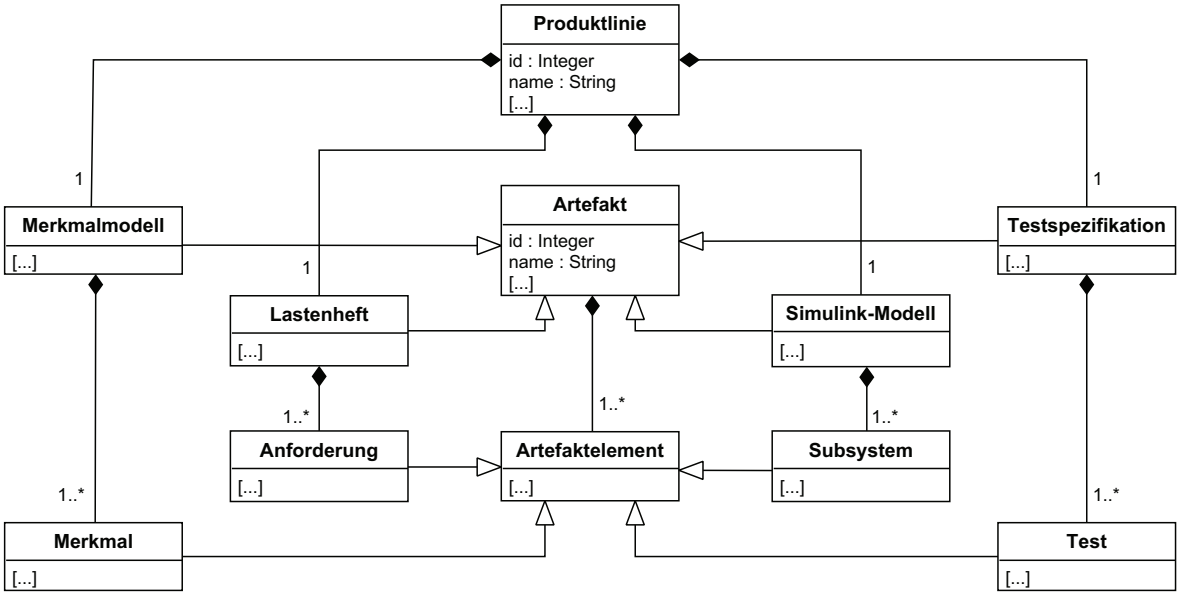


Abbildung 6.3: Zusammenhang zwischen Artefakten und der Produktlinie

im Werkzeug TPT definiert. Da diese Automaten für die vorliegende Dissertation ausschließlich im Kontext von Merkmalkonsistenzanalysen eine Rolle spielen und dort nur als eine Gesamtentität mit den Attributen *ID* und *Name* repräsentiert werden, ergibt sich ein triviales Datenmodell, auf dessen Darstellung ich daher ebenfalls verzichte.

Im Folgenden werden daher nur Ausschnitte aus den Datenmodellen für das Simulink-Modell und das Merkmalmodell beschrieben. Anschließend wird erläutert, inwiefern das Konzept einheitlicher Annotationen im Datenmodell abgebildet ist.

6.3.1 Simulink-Modell

Auch das Simulink-Modell wird in der Datenbank durch ein dem Metamodell aus Abbildung 5.8 ähnliches Datenmodell repräsentiert. Die wesentlichen Unterschiede zeigt Abbildung 6.4. Sie bestehen darin, dass

a) jede Linie nur eine Quelle und ein Ziel besitzt und

b) Linien direkt mit Blöcken anstatt mit Ports verbunden sind.

Eine Linie aus MATLAB/Simulink mit mehr als einem Ziel wird daher in diesem Datenmodell durch mehreren Linien mit derselben Quelle und unterschiedlichen Zielen dargestellt. Um die Information, an welchem Port eine Linie beginnt bzw. endet, nicht zu verlieren, wird die Portnummer als Attribut (*srcPort* bzw. *dstPort*) in die Linie im Datenmodell geschrieben.

Wie aus Abschnitt 5.4.1 bekannt, wird die Signalverfolgung über Subsystemgrenzen oder Sprungblöcke hinweg durch die fehlende direkte Verknüpfung zwischen den Schnittstellenblöcken eines Subsystems und der Außenwelt bzw. zwischen Sprungblöcken unnötig verkompliziert. Daher werden Linien im Modellaufbereitungsschritt während

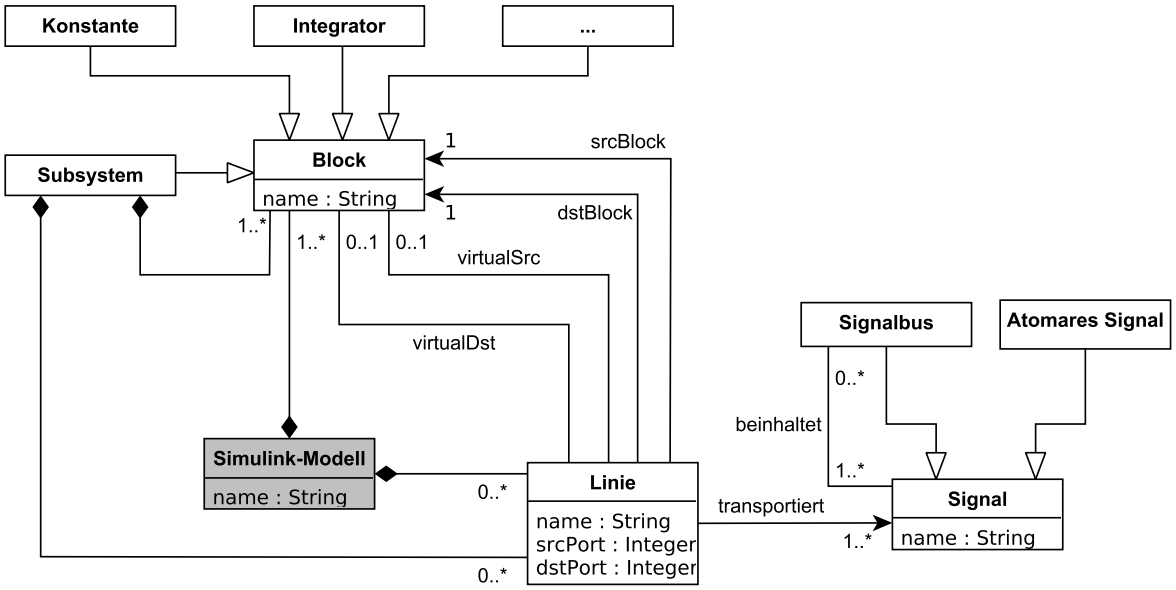


Abbildung 6.4: Das Datenmodell von Repräsentanten des Simulink-Modells in der Datenbank

des Imports (vgl. Abschnitt 6.4.1) sogenannte virtuelle Quellen (*virtualSrc*) bzw. Ziele (*virtualDst*) zugewiesen. Der in Abbildung 5.5b vom Block *Konstante 1* ausgehenden Linie wird beispielsweise als virtuelles Ziele direkt der Block *Terminator 1* zugewiesen. Für die Verfolgung des aus *Konstante 1* ausgehenden Signals ist es somit später nicht mehr erforderlich, zu prüfen, über welche Schnittstellenblöcke dieses Signal das Subsystem verlässt. Stattdessen kann ein direkter Sprung zum relevanten Block der Außenwelt erfolgen.

6.3.2 Merkmalmodell

Der Import des Merkmalmodells in die Datenbank erlaubt variabilitätsbezogene Analysen sowie Merkmalkonsistenzprüfungen auf Basis der Datenbank. Dessen Datenmodell ist in Abbildung 6.5 dargestellt. Der weiß hinterlegte Teil repräsentiert dabei im Wesentlichen die aus Abschnitt 2.2.5 bekannte FoDA-Notation. Die Wurzel des Baums wird durch die Entität *Merkmalmodell* repräsentiert, der die Merkmale des Modells zugeordnet werden. Ein Merkmal kann für sein Elternmerkmal notwendig sein (boolesches Attribut *mandatory*) oder nicht. Die Eltern-Kind-Relation wird hier über die reflexive Kompositionsassoziation der Merkmale ausgedrückt. Des Weiteren werden optionale und alternative Gruppen durch entsprechende Entitäten und die Zugehörigkeit der Merkmale zu diesen über die entsprechenden *gehört-zu*-Assoziation ausgedrückt. Auch der gegenseitige Ausschluss und die *benötigt*-Assoziation werden durch reflexive Assoziationen der Merkmale modelliert.

Damit auch Varianten der Produktlinie in der Datenbank verwaltet werden können, wird deren konfiguriertes Merkmalmodell über die gleichnamige Entität in die Datenbank

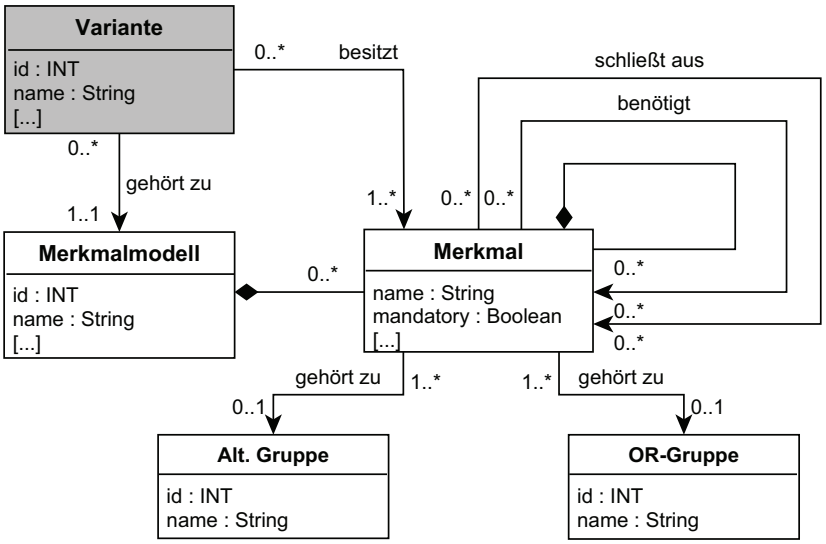


Abbildung 6.5: Datenmodell von Merkmalmodellen in der Datenbank

integriert. Eine Variante beschreibt stets eine Teilmenge des Merkmalmodells (*besitzt*-Assoziation). Um die Zugehörigkeit einer Variante zu einer Produktlinie auszudrücken, wird sie mit deren Merkmalmodell assoziiert (*gehört zu*-Assoziation). Diese Modellierung erlaubt die Gültigkeitsprüfung einer Variante auf Basis des Merkmalmodells, sofern dies nicht bereits vor dem Import (z. B. im Merkmalmodellierungswerkzeug S2T2) erfolgt ist.

6.3.3 Weitere Modellanreicherungen

Abbildung 6.6 zeigt das Konzept für die Integration von einheitlichen Annotationen. Die Skizze knüpft über die Entität *Artefaktelement* an Abbildung 6.3 an. Jedes Artefaktelement kann mit beliebig vielen *Annotations-Session*s[1] verknüpft sein, die Annotationen zusammenfassen, die sich gemäß Abschnitt 4.2 aus einem Typ t und einem Wert v zusammensetzen. Auch kann eine *Annotations-Session* beliebig viele Artefaktelemente annotieren. Dadurch wird es möglich, Subsysteme, Anforderungen, Tests und Merkmale mit denselben Informationen zu verknüpfen und so implizit potenzielle Zusammenhänge zwischen Artefakten zu modellieren.

Damit das Spezifikationsformat einheitlich ist, also möglichst keine Freitextannotationen möglich sind, kann jeder Annotationstyp mit einer Wertequelle verbunden

[1] Auf die Verwendung von *Annotations-Session*s könnte man auch verzichten, da sie für die Ziele des Annotationskonzepts nicht relevant sind. Stattdessen würde man Annotationen direkt mit den Artefaktelementen verknüpfen, die sie annotieren. Der Vorteil einer *Annotations-Session* liegt darin, dass damit einerseits mehrere gleichzeitig annotierte Informationen als zusammengehörig gekennzeichnet werden. Dies kann für die Abbildung eines Annotationskontextes sinnvoll sein. Auch können die in einer Session abgelegten Informationen über Datum und Urheber der Annotationen Hilfestellung bei der Verfolgung von Änderungen geben. Hier wurde jedoch nur aus Erweiterbarkeitsgründen auf diese Architektur zurückgegriffen.

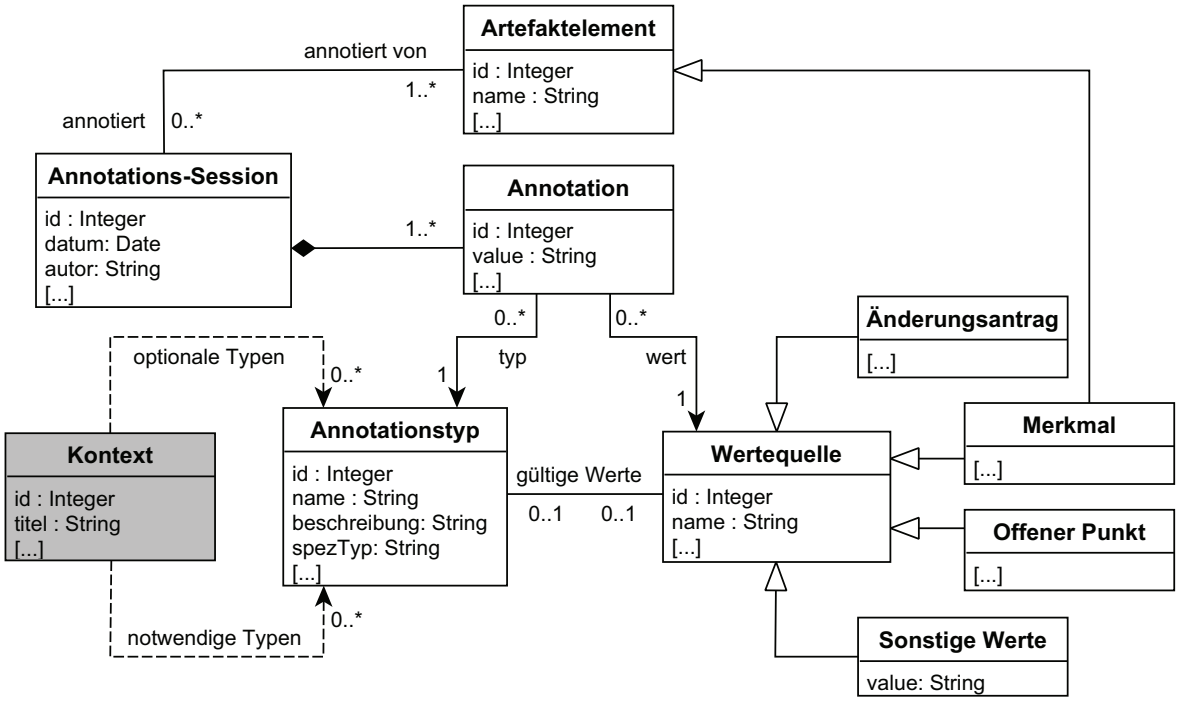


Abbildung 6.6: Das Annotationskonzept für Artefaktelemente

werden. Dabei handelt es sich um eine Tabelle, die eine Spalte besitzt, die die Menge gültiger Werte für den Typ beinhaltet. Beispielsweise kann die Wertequelle für einen Annotationstyp *Änderungsantrag (CR)*, der die Annotation von Artefaktelementen mit Änderungsanträgen, von denen sie bereits betroffen waren, ermöglicht, die gleichnamige Datenbanktabelle sein, die die gültigen Werte in der ID-Spalte enthält. Ein im weiteren Verlauf besonders relevanter Annotationstyp beschreibt die Zugehörigkeit eines Artefaktelements zu einem Merkmal. Die gültigen Werte dieses Typs sind die in der Tabelle *Merkmal* abgelegten Namen. Über die auf diese Weise spezifizierte Zugehörigkeit von Artefaktelementen zu Merkmalen können Artefakte einerseits indirekt verknüpft und andererseits über die im Merkmalmodell abgelegten Abhängigkeiten auf Variabilitätsaspekte hin analysiert werden. Die Tabelle *Sonstige Werte* kann verwendet werden, um die für einen Annotationstyp gültigen Werte zu definieren, der sich nicht auf ohnehin in der Datenbank vorhandene Daten bezieht, also für beliebige Aufzählungstypen.

Ein Annotationstyp besitzt neben den gültigen Werten zusätzliche Attribute, die beschreiben, wie die zugehörigen Werte spezifiziert werden. Dies wird im Attribut *spezTyp* festgehalten. Dabei kann festgelegt werden, wie viele der gültigen Werte ausgewählt werden können, oder ob es sich um eine Freitextspezifikation handelt. Dürfen Elemente eines Artefakts beispielsweise mit mehreren Merkmalen verknüpft werden, so ist der Artefakttyp, wie oben beschrieben, mit der Merkmalentität als Quelle verknüpft und *spezTyp* auf den Wert *Multiple* gesetzt. Schließlich lässt sich die Anwendbarkeit eines Annotationstyps über entsprechende boolesche Attribute auf bestimmte Artefakte

einschränken. Das Verwaltungswerkzeug in Abschnitt 6.5 ermöglicht die flexible und unternehmensspezifische Konfiguration möglicher Annotationstypen.

Um Annotationen in einen weiteren semantischen Kontext zu integrieren, wurde das Datenmodell um die in Abbildung 6.6 grau hinterlegte Entität *Kontext* erweitert. Diese wird aktuell noch nicht berücksichtigt, ist aber als Erweiterungspunkt gedacht, um Annotationen in einen sachlichen Kontext setzen zu können, der bestimmte Annotationen *erfordert* (Assoziation *Notwendige Typen*) und weitere Annotationen *ermöglicht* (Assoziation *Optionale Typen*). Beispielsweise kann die Einarbeitung eines Änderungsantrags als ein solcher Kontext aufgefasst werden, für den die Angabe der ID des zugehörigen Änderungsantrags notwendig ist, während die Angabe eines Kommentars optional ist.

6.4 Artefaktintegration

Um die Originalartefakte in die Datenbank zu importieren, verfährt man gemäß Abbildung 6.1. Für jedes der verwendeten Werkzeuge wird eine werkzeugspezifische Importfunktionalität der Verwaltungs- und Importkomponente (vgl. Abbildung 6.2) bereitgestellt, die aus jedem Artefakt ein Java-Klassenmodell konstruiert, welches anschließend in der Datenbank persistiert wird. Mit der Datenbankzugriffskomponente wird mit Hilfe des Hibernate-Rahmenwerks das Java-Klassenmodell auf das Datenmodell der Datenbank übertragen. Die dabei relevanten Datenmodelle wurden in Abschnitt 6.3 ausschnittsweise beschrieben. Im Folgenden wird die Erstellung des Java-Modells für die Artefakte genauer beschrieben.

6.4.1 Simulink-Modell

Im Falle eines Simulink-Modells erfolgt der Import mittels eines MATLAB-Skripts, welches verschiedene Informationen aus dem Simulink-Modell ausliest und diese als Parameter an Java-Funktionen der Verwaltungs- und Importkomponente übergibt, die daraus ein Java-Klassenmodell der Modellkomponente konstruieren. Im Vergleich zum modellbasierten Vorgehen aus Kapitel 5 lassen sich so Fehler, die auf eine modifizierte, konkrete Syntax der MDL-Datei zurückzuführen sind (z. B. bei einem Versionswechsel von MATLAB/Simulink) vermeiden. Aktuell beschränkt sich der Import auf die Blockattribute *Name*, *Typ* und *SID*, das Linienattribut *Name*, Quellblock bzw.-port und Zielblock bzw.-port von Linien sowie die Subsystemhierarchie.

Dadurch dass die SIDs der Simulink-Blöcke importiert werden, sind deren Repräsentanten in der Datenbank eindeutig und komfortabel auffindbar. Dies ist beispielsweise für die Abfrage von Annotationen von Subsystemen von Interesse (vgl. Abschnitt 6.5), die konzeptgemäß ja in der Datenbank und nicht im Simulink-Modell abgelegt werden. Alternativ könnte man jeden Block analog zu einer Datei im Dateisystem auch über den eindeutigen Pfad im Modell identifizieren. Da dieser jedoch bei einer hohen Rekursionstiefe von Subsystemen sehr lang werden kann, wurde stattdessen die SID verwendet.

Im Gegensatz zum modellbasierten Ansatz wird direkt im ersten Schritt die Linienstruktur vereinfacht. Dazu wird aus einer Linie in Simulink eine Linie pro Ziel erstellt, sodass eine Linie mit zwei Zielen zu zwei Linien mit derselben Quelle aber verschiedenen Zielen wird. In der Datenbank erfolgt die Verknüpfung zwischen Linien und Blöcken über Fremdschlüssel in den entsprechenden Tabellen. Um die Information über den Quell- bzw. Ziel*port* dabei nicht zu verlieren, werden deren Portnummern als Attribute der Linien festgehalten. (vgl. Abschnitt 6.3.1)

Diese Informationen reichen aktuell aus, um die vom Industriepartner benannten Analysen durchführen zu können, jedoch wird analog zum modellbasierten Ansatz auf Basis des jetzt vorhandenen Java-Modells noch eine Modelloptimierung vorgenommen, um spätere Analysen zu beschleunigen und die benutzerdefinierte Implementierung von Analysen zu vereinfachen. Diese Optimierung betrifft die Linienstruktur, den Signaltransport und die Signalstruktur (atomare Signale und Signalbusse) (vgl. Abbildung 6.4). Die Linienstruktur wird durch die zusätzliche Spezifikation von virtuellen Quellen und Zielen gemäß Abschnitt 6.3.1 zur Vereinfachung der Signalverfolgung aufbereitet. Für Verbindungen über Sprungblöcke (vgl. Abschnitt 2.3.3) wird hier eine zusätzliche Linie eingefügt, welche ausschließlich eine virtuelle Quelle (Goto) und ein virtuelles Ziel (From) besitzt.

Für die Erstellung der Signalstruktur bestehend aus atomaren Signalen und Signalbussen sowie für die Verknüpfung von Signalen mit den Linien, die sie transportieren, wird ein Worklist-Algorithmus auf das Java-Klassenmodell des Simulink-Modells angewendet.

Das nun aufbereitete Java-Modell wird über eine Funktionalität der Datenbankzugriffskomponente (vgl. Abbildung 6.2) in der Datenbank persistiert, um es für spätere Analysen direkt in optimierter Form aus der Datenbank laden zu können.

6.4.2 Import weiterer Artefakte

Der Import des Lastenheftes, der Testspezifikation und des Merkmalmodells erfolgt über den dateibasierten Import in dem in Abschnitt 6.2.1 erwähnten Verwaltungswerkzeug. Dazu wird die Produktlinie (das Wurzelelement aus Abbildung 6.3), der das Artefakt zugewiesen werden soll, über den in Abbildung 6.7 dargestellten Dialog ausgewählt und der Pfad im Dateisystem zur Artefaktdatei angegeben. Für den Import des Lastenhefts ist der Pfad zu der CSV-Datei anzugeben, die dafür zuvor aus IBM Rational DOORS exportiert werden muss. Diese wird dann mit Hilfe der OpenCSV-Parser-Bibliothek [138] in ein Java-Klassenmodell umgewandelt, das gemäß der in Abschnitt 5.3.2 vorgestellten Struktur erzeugt wird. Dieser Baum wird anschließend wiederum von der Datenbankzugriffskomponente (vgl. Abbildung 6.2) in der Datenbank persistiert.

Analog ist für den Import der Testspezifikation aus TPT und für das Merkmalmodell aus S2T2 zu verfahren, die als XML-Datei bzw. EMF-Modell bereitgestellt, anschließend geparst und persistiert werden.

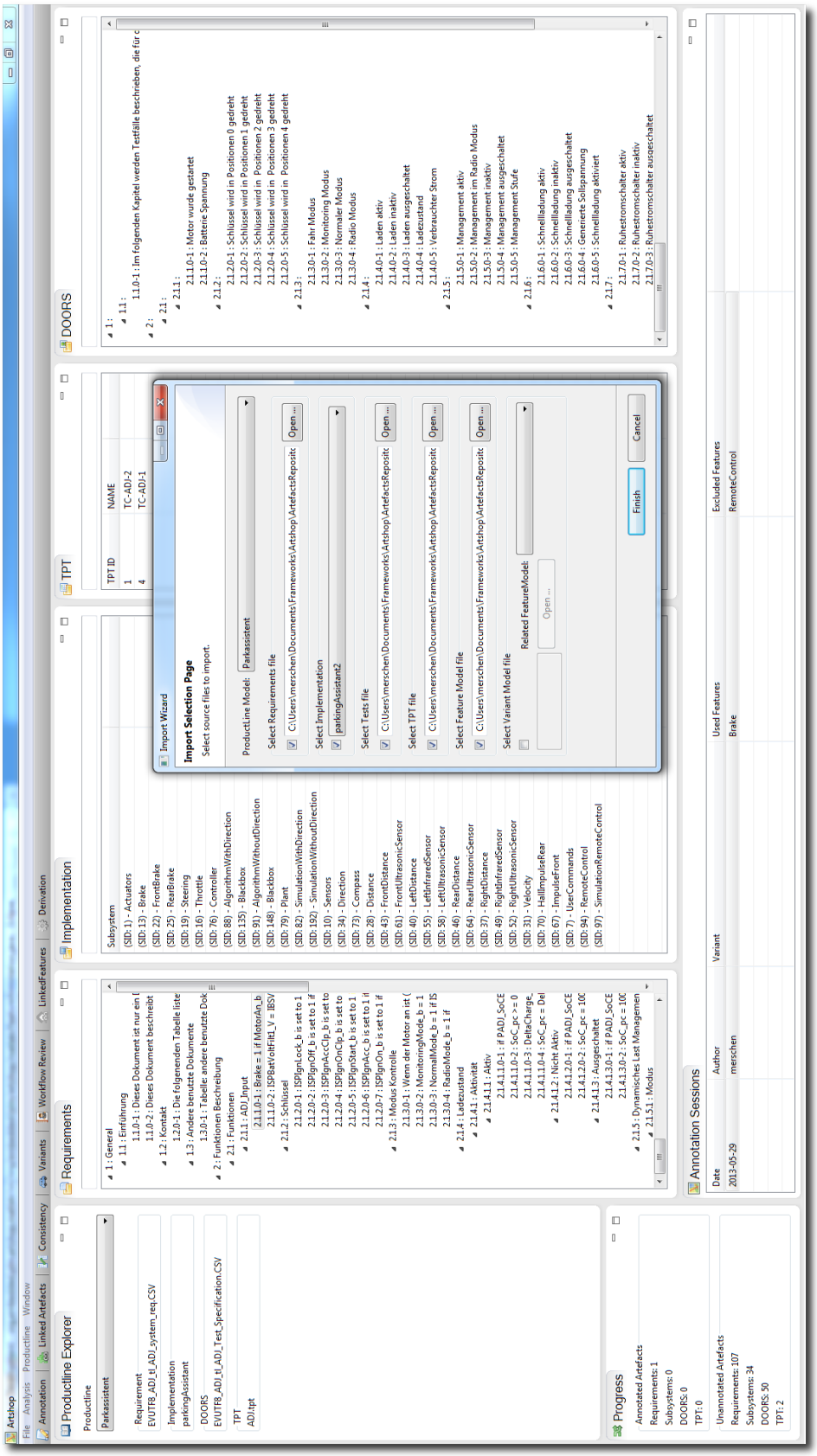


Abbildung 6.7: Screenshot des Verwaltungswerkzeug Artshop aus Abschnitt 6.2.1: Vordergrund: Der Import-Dialog, Hintergrund: Bereits importierte Artefakte

6.5 Umsetzung des Annotationskonzeptes

Für die Umsetzung des Annotationskonzepts wurde das in Abschnitt 6.2.1 genannte Verwaltungswerkzeug mit Funktionalität zum Verwalten von Annotationstypen und Annotieren von Anforderungen, Tests und Merkmalen versehen. Dabei wird die Datenbank um die entsprechenden Informationen gemäß dem in Abschnitt 6.3.3 beschriebenen Datenmodell angereichert. Die beiden Funktionalitäten werden im Folgenden beschrieben.

6.5.1 Verwaltung von Annotationstypen

Da der Entwicklungs- und Evolutionsprozess unternehmensspezifisch ist, ist zu erwarten, dass auch die diesbezüglichen Annotationstypen variieren. Daher wurden diese nicht fest in das Rahmenwerk integriert, sondern können über einen Dialog verwaltet werden. Er ermöglicht das Erstellen, Modifizieren und Löschen von Annotationstypen. Dazu wird ggf. die Wertequelle des Typs eingestellt. Außerdem wird über die Angabe eines Spezifikationstyps festgelegt, wie viele dieser Werte ausgewählt werden dürfen. Da nicht jeder Annotationstyp sinnvoll auf jedes Artefakt anwendbar ist, wird zudem über entsprechende boolesche Attribute definiert, für welche Artefakte ein Annotationstyp gültig ist (vgl. Abschnitt 6.3.3). Um dem Entwickler zu erklären, welchen Sachverhalt ein Annotationstyp beschreibt, kann eine kurze Erläuterung ergänzt werden.

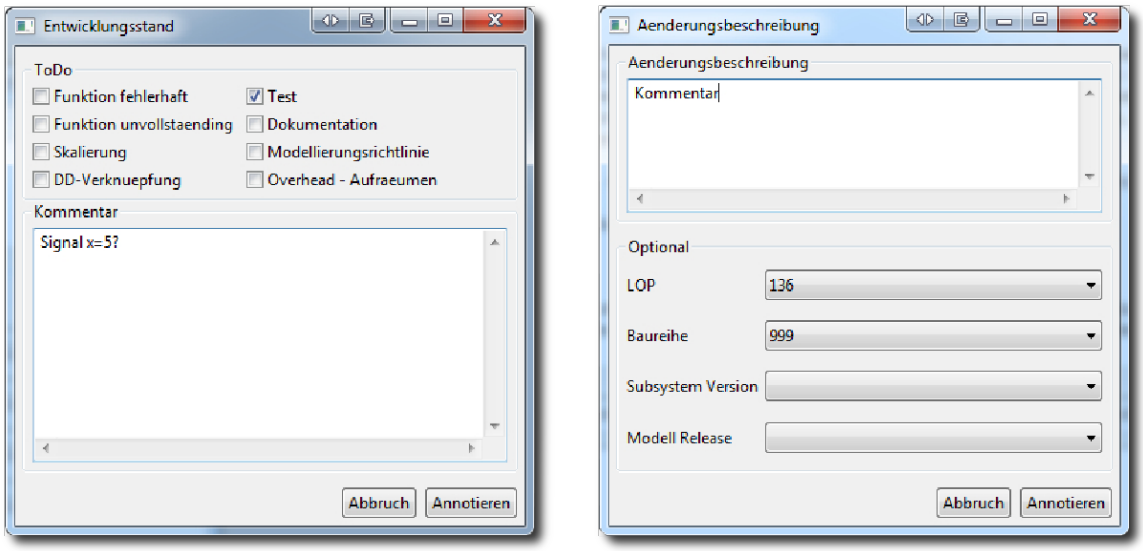
Die artefaktübergreifende Anwendung und einheitliche Syntax von Annotationen wird dadurch erreicht, dass für die spätere Annotation von Artefaktelementen ausschließlich die auf diese Weise definierten Annotationstypen und zugehörige Werte zur Auswahl gestellt und die resultierenden Annotation ebenfalls in der Datenbank abgelegt werden. Daher darf die in Abschnitt 6.5.2 beschriebene Annotation von Artefaktelementen auch nur über die vom entwickelten Rahmenwerk zur Verfügung gestellte Annotationsfunktionalität erfolgen.

Für die in Abschnitt 6.3.3 erwähnte Erweiterung um einen Annotationskontext ist es sinnvoll, das Werkzeug um einen entsprechenden Dialog zu erweitern. Die Zuordnung von Annotationstypen zu einem vordefinierten Kontext würde dann dort erfolgen.

6.5.2 Annotation von Artefaktelementen

Subsysteme in MATLAB/Simulink Die Annotationsfunktionalität von Subsystemen fügt sich nahtlos in MATLAB/Simulink ein. Dennoch erfolgt sie ausschließlich auf Basis der Datenbank über die Funktionalität der Analyse- und Annotationskomponente des Rahmenwerks. Dazu wurden beispielhaft für verschiedene Kontexte Annotationsdialoge konstruiert, die über ein Kontextmenü auf dem Subsystem aufgerufen werden können[1]. Zwei Beispiele für diese Annotationsdialoge zeigt Abbildung 6.8. Über den Dialog in

[1] Diese Kontexte stellen zwar die in Abschnitt 6.3.3 beschriebenen Kontexte dar, werden aber bisher nicht in die Datenbank geschrieben, sondern „hart“ codiert. Setzt man das Konzept des Kontextes um, so ist es sinnvoll, einen generischen Standard-Dialog zu erstellen, der kontextabhängig die Angabe von Annotationen bestimmter Typen verlangt (notwendige Typen) und andere Annotationen ermöglicht (optionale Typen).



(a) Entwicklungsstand (b) Offene Punkte und betroffene Baureihen

Abbildung 6.8: Annotation von Subsystemen in verschiedenen Kontexten [119]

Abbildung 6.8a annotiert der Entwickler den Entwicklungsstand eines Subsystems, beispielsweise, dass ein Subsystem noch zu testen ist, und schreibt eventuell noch einen erläuternden Kommentar dazu. In diesem Kontext sind somit die Annotationen vom Typ *Entwicklungsstatus* und *Kommentar* relevant. Der Dialog in Abbildung 6.8b ermöglicht die Ablage von Informationen, die sich aus der Einarbeitung eines Änderungsantrags ergeben. Dazu zählen hier u. a. die damit assoziierten offenen Punkte (vgl. Abbildung 6.6) und die betroffene Baureihe.

Alternativ zur kontextbezogenen Umsetzung des Annotationskonzepts ist ein Dialog denkbar, mit dem der Anwender beliebige Annotationstypen und -werte aus Auswahlfeldern selektiert. Dabei könnten die über die Annotationsverwaltung (Abschnitt 6.5.1) eingepflegten Informationen den Anwender unterstützen, beispielsweise als ToolTipText.

Die einfache Abfrage von Subsystemen eines Simulink-Modells, die auf bestimmte Art und Weise annotiert sind, kann über einen ebenfalls in MATLAB/Simulink integrierten Suchdialog erfolgen (Abbildung 6.9). Somit wird dem Problem der über das gesamte Modell verteilten Information begegnet (vgl. Abschnitt 3.3). Im Beispiel werden Subsysteme abgefragt, die noch nicht vollständig umgesetzt sind und sich auf das Merkmal *Sensors* beziehen. Die Subsysteme, auf die die dort genannten Kriterien zutreffen, werden tabellarisch im unteren Teil des Dialogs aufgelistet.

Weitere Artefakte Die Annotation von Anforderungen und Tests wurde noch nicht in die zugehörigen Werkzeuge integriert. Dies erfolgt daher über das Verwaltungswerkzeug aus Abbildung 6.7, welches die Artefakte aus der Datenbank ausliest und visualisiert. Nachdem beispielsweise eine Anforderung aus dem Anforderungsbaum (vgl. Hintergrund links in Abbildung 6.7) ausgewählt wurde, öffnet sich der Dialog in Abbildung 6.10, der

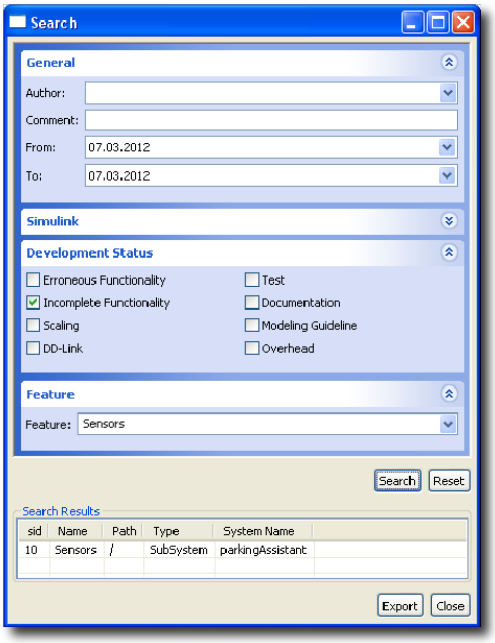


Abbildung 6.9: Abfragedialog für Subsysteme des Simulink-Modells [119]

die Spezifikation von Annotationen auf Basis der Datenbank ermöglicht. Im Hintergrund werden die bereits vorhandenen Annotationen der Anforderung angezeigt. In der Auswahlbox des Annotationstyps werden die für Lastenhefte zugelassenen Annotationstypen zur Auswahl gestellt. Diese Informationen wurden beim Erstellen des Annotationstyps festgelegt (vgl. Abschnitt 6.5.1). Hier hat der Anwender den Annotationstyp *Merkmal* gewählt, sodass das Werkzeug nun die gültigen Werte aus der Wertequelle ausliest und in einer weiteren Auswahlbox dem Anwender zur Annotation anbietet. In diesem Fall entstammen die Werte also der Spalte *Name* der Tabelle *Merkmal*.

Sobald die Annotation gespeichert wurde, wird diese in der Datenbank mit dem entsprechenden Artefaktelement verknüpft, sodass sie in Analysen leicht automatisiert abgefragt werden kann.

6.6 Analysen

6.6.1 Funktionsweise

Analysen werden in Java implementiert, wobei eine Menge vordefinierter Funktionen aus einer Funktionsbibliothek der Analyse- und Annotationskomponente verwendet werden kann. Dazu gehören zum Beispiel Funktionen, die die atomaren Signale eines Signalbusses extrahieren oder Funktionen, die eine neue Linie im Java-Ergebnis-Modell einfügen und dabei ggf. notwendige Attribute in verbundenen Blöcken setzen. Bei der Implementierung kann der Anwender folglich von Details abstrahieren und auf dem Datenmodell in Abbildung 6.4 operieren.

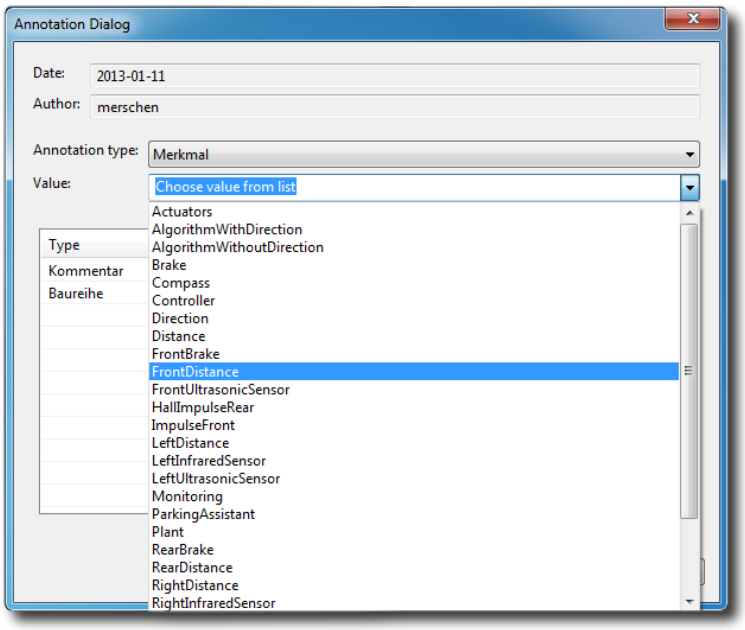


Abbildung 6.10: Annotation über das Verwaltungswerkzeug aus Abbildung 6.7

Im Allgemeinen arbeitet eine solche Analyse auf Java-Modellen aus der Modellkomponente (vgl. Abbildung 6.2), die dazu nach Bedarf aus der Datenbank ausgelesen werden. Es ist aber auch möglich, direkte Anfragen an die Datenbank zu stellen. Dazu können Funktionen des Hibernate-Frameworks oder der Java-eigenen JDBC/ODBC-Funktionalität [108] verwendet werden. Auch wären direkte Datenbankabfragen aus MATLAB/Simulink möglich, wovon jedoch im Rahmen der Disseration kein Gebrauch gemacht wurde. Die Art der Implementierung ist also frei wählbar. Erfahrungsgemäß bietet sich für strukturelle Analysen des Simulink-Modells die Verwendung des Java-Modells an, da dieses für graphentheoretische Fragestellungen besser geeignet ist als die relationale Struktur der Datenbank. Für einfache Abfragen bezüglich Annotationen und Verknüpfungen hingegen kann der Direktzugriff sinnvoller sein. Strukturelle Analysen von Simulink-Modellen resultieren in einem neuen Java-Modell, welches anschließend visualisiert werden kann.

6.6.2 Visualisierung

Für die Visualisierung der aus strukturellen Analysen resultierenden Java-Modelle bestehen zwei Möglichkeiten.

Die erste Möglichkeit sieht vor, für das Java-Modell unter Zuhilfenahme des originalen analysierten Simulink-Modells mit Methoden der MATLAB API [147] ein Simulink-Modell zu erstellen, welches ausschließlich der Visualisierung dient. Da alle Blockinstanzen im Java-Modell dasselbe Attribute *SID* besitzen wie die Simulink-Blöcke, die sie repräsentieren (vgl. Abschnitt 6.4.1), können die Simulink-Blöcke anhand dessen aus dem analysierten Modell ausgelesen und in ein neues Simulink-Modell kopiert

werden. Blöcke, die im Rahmen einer Analyse neu erstellt werden müssen[1], besitzen eine SID, die im Originalmodell nicht vorkommt. Diese werden in das neue Simulink-Modell ebenfalls über die MATLAB API neu eingefügt. Die Verbindungslinien werden aufgrund der fehlenden *SID* nicht aus dem analysierten Simulink-Modell übernommen. Daher werden ihre Quell- und Zielblöcke bzw.-ports aus dem Java-Modell ausgelesen und über MATLAB-Funktionen auf die Blöcke des neuen Modells übertragen. Dies ist möglich, da die notwendigen Informationen beim Import des Originalmodells in die Datenbank importiert wurden (vgl. Abschnitt 6.4.1). Der Vorteil dieser Visualisierungsmöglichkeit liegt insbesondere darin, dass die Anwender mit MATLAB/Simulink bereits vertraut sind und durch das Kopieren von Blöcken auch deren Position im Modell übernommen wird, wodurch das Ergebnis dem Original ähnelt. Nachteil dieser Visualisierungsmethodik ist jedoch ihre wenig effiziente Laufzeit, die wahrscheinlich durch den Wechsel zwischen MATLAB- und Java-Funktionalität verursacht wird.

Alternativ wurde daher ein zweiter Visualisierungsansatz entwickelt, der unter Nutzung des JGraphX-Rahmenwerkes eine eigene Visualisierungsmethodik besitzt. Damit erstellte Visualisierungen spiegeln zwar den Aufbau aus Blöcken, Linien und Subsystemen wieder, haben aber ansonsten keine Ähnlichkeit mit Simulink-Modellen. Dafür könnte man jedoch auch neue graphische Elemente definieren und so auch andere Artefaktelemente abbilden, wovon allerdings noch kein Gebrauch gemacht wurde.

Die Visualisierung von inkonsistenten Merkmalverknüpfungen erfolgt tabellarisch im Verwaltungs- und Analysewerkzeug gemäß Abschnitt 6.7.2.

6.7 Konsistenzprüfung

Artefaktkonsistenz im Sinne von Definition 2.3.3 sicherzustellen, erfordert einen semantischen Abgleich der Artefakte, der nicht unmittelbar automatisierbar ist. Daher unterstützt das hier vorgestellte Werkzeug den Anwender dabei, Cliquen auf Merkmalkonsistenz zu überprüfen. Dazu berücksichtigt es sowohl die über eine Clique verknüpften Artefaktelemente als auch deren Merkmalverknüpfungen und bietet ggf. Korrekturvorschläge an. Welchen Vorschlägen der Anwender folgt, bleibt diesem überlassen. Schlussendlich verbleibende *Merkmal*inkonsistenzen geben Aufschluss über potenzielle *Artefakt*inkonsistenzen und ihre Ursachen, die manuell zu lösen sind. Die Funktionalität wurde im Rahmen der Diplomarbeit von Norbert Wiechowski implementiert und in das datenbankorientierte Rahmenwerk integriert. Sie basiert auf der in Abschnitt 4.4 beschriebenen Theorie, die ihrerseits auf dem mengentheoretischen Konsistenzkonzept von Cmyrev et al. [26] aufsetzt.

[1] Dieser Fall tritt zum Beispiel auf, wenn ein Signalbus durch die atomaren Signale, die er transportiert, ersetzt werden soll (vgl. Abschnitt 7.1.2). In diesem Fall müssen ggf. die Schnittstellen von Subsystemen derart angepasst werden, dass pro atomarem Signal ein Schnittstellenblock erzeugt wird. Entsprechend ist die innere Struktur an die geänderte Schnittstelle anzupassen.

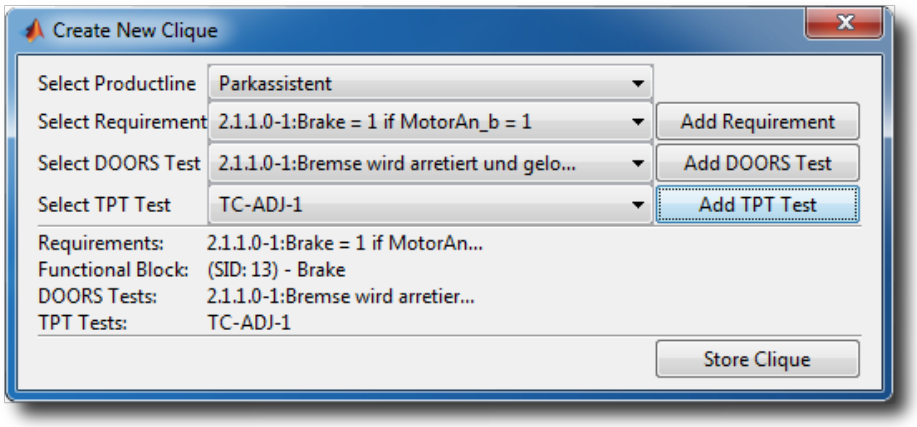


Abbildung 6.11: Erstellen einer Clique für ein vorausgewähltes Subsystem

6.7.1 Erstellen von Merkmalverknüpfungen und Cliquen

Bevor Cliquen erstellt werden können, müssen zunächst die zu verknüpfenden Artefakte aus MATLAB/Simulink heraus bzw. über den in Abbildung 6.7 dargestellten Dialog gemäß Abschnitt 6.4 in die Datenbank importiert werden. Anschließend kann innerhalb der Datenbank die Verknüpfung der Artefaktelemente mit Merkmalen oder die Verknüpfung der Elemente untereinander zu Cliquen erfolgen. Merkmale werden als Annotation vom Typ *Merkmal* mit den Artefaktelementen verknüpft (vgl. Abschnitt 6.5.2). Die zugehörigen Annotationswerte sind die Namen der Merkmale, die aus der Datenbank ausgelesen und zur Auswahl gestellt werden. Im Falle von Subsystemen des Simulink-Modells erfolgt dies direkt aus MATLAB/Simulink. Die anderen Artefakte werden im Verwaltungs- und Analysewerkzeug über den Dialog in Abbildung 6.10 mit Merkmalen verknüpft. Nach dem Artefaktimport und der Merkmalannotation bietet das Werkzeug eine Übersicht über die Artefakte und zeigt im Reiter *Annotation Sessions* (vgl. Abschnitt 6.3.3) die vorhandenen Merkmalannotationen eines selektierten Artefaktelements an (vgl. Abbildung 6.7). Die Spalte *Author* kennzeichnet den Benutzer, der die Annotation vorgenommen hat. Die Zugehörigkeit des Artefaktelements zu einer Produktvariante ist in der Spalte *Variant* angegeben. *Used Feature* kennzeichnet die Merkmale, mit denen es annotiert wurde, während über *Excluded Feature* auch spezifiziert werden kann, welche Merkmale das Artefaktelement ausschließt. Für das weitere Verständnis ist nur die Spalte *Used Feature* von Interesse.

Über den Dialog aus Abbildung 6.11, der sich über ein Kontextmenü direkt für ein Subsystem in MATLAB/Simulink aufrufen lässt und zusätzlich über das Verwaltungswerkzeug aufgerufen werden kann, fügt der Anwender komfortabel Artefaktelemente zu Cliquen im Sinne von Definition 2.3.2 zusammen. Eine semi-automatische Cliquenerstellung ist nur über einen entsprechenden Button aus dem Verwaltungswerkzeug möglich. Dabei werden Elemente verschiedener Artefakte, die mit denselben Merkmalen annotiert sind, identifiziert und dem Anwender zur Verknüpfung angeboten. Dies ist sinnvoll, da die gleiche Merkmalverknüpfung darauf hindeutet, dass die Elemente sich

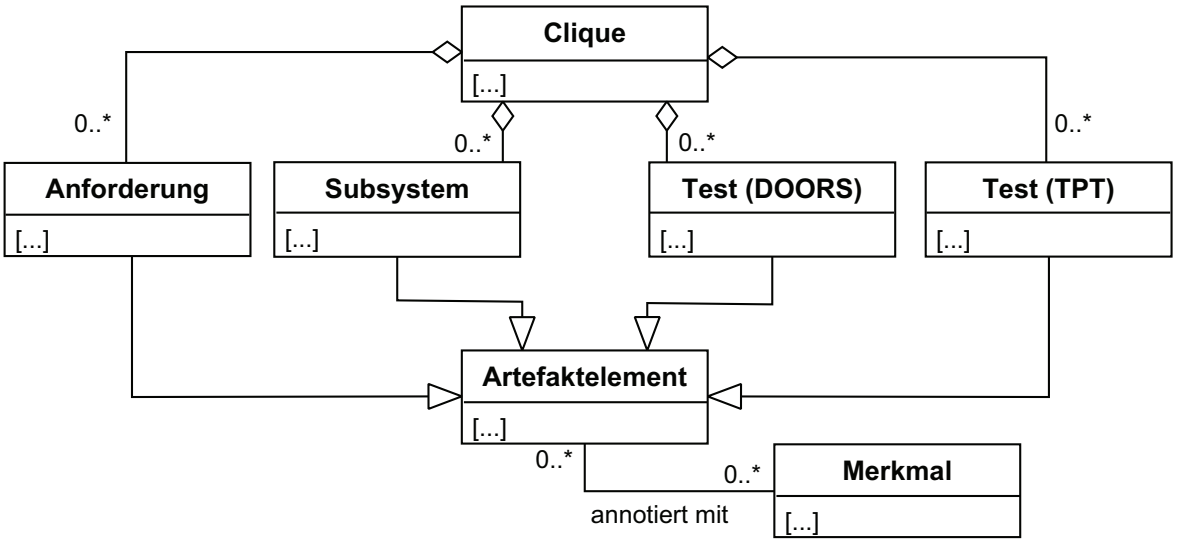


Abbildung 6.12: Struktur einer Clique und Merkmalverknüpfungen ihrer Elemente

auf dieselbe Funktionalität beziehen und somit evtl. miteinander verbunden werden sollten.

Abbildung 6.12 zeigt vereinfacht die Struktur einer Clique und die Annotation der in ihr enthaltenen Artefaktelemente mit Merkmalen in der zentralen Datenbank, die sich über die Entität *Artefaktelement* in das Datenmodell aus Abbildung 6.6 integriert.

6.7.2 Konsistenzanalyse und Korrekturmaßnahmen

Auf Basis der nun in der Datenbank vorhandenen Artefakte, Cliquen zwischen ihren Elementen und deren Merkmalverknüpfungen können im Verwaltungs- und Analysewerkzeug gemäß Abschnitt 4.4 Merkmalkonsistenzprüfungen angestoßen werden. Dabei identifizierte inkonsistente Cliquen werden in unterschiedlichen Reitern je nach Inkonsistenzkategorie angezeigt. Ein Beispiel für eine identifizierte Clique, die inkonsistent aufgrund unvollständiger Merkmalverknüpfung ist, visualisiert Abbildung 6.13. Auf der linken Seite werden im oberen Teil die Produktlinie mit den zugehörigen 150%-Artefakten angezeigt, deren Cliquen auf Merkmalkonsistenz geprüft wurden. Der untere, linke Teil zeigt den Status der Datenbankverbindung und den Abschluss der Konsistenzprüfungen an. Im dargestellten Beispiel wurde nur eine Clique identifiziert, die aufgrund unvollständiger Merkmalverknüpfung inkonsistent ist. Diese wird im Reiter *Incomplete* angezeigt. Wären weitere Inkonsistenzarten identifiziert worden, so würde es dafür entsprechende Reiter geben. Selektiert der Anwender die inkonsistente Clique, so wird unter *Details* dargestellt, welches Artefakt die Ursache der Inkonsistenz ist. In diesem Fall ist dies der Test in TPT, der nicht mit allen Merkmalen verknüpft ist wie die Anforderungen des als Merkmalbasis voreingestellten Lastenhefts[1] (Spalte *Inconsistent Artefacts*).

[1] Die Merkmalbasis lässt sich über den Reiter *Options* umstellen.

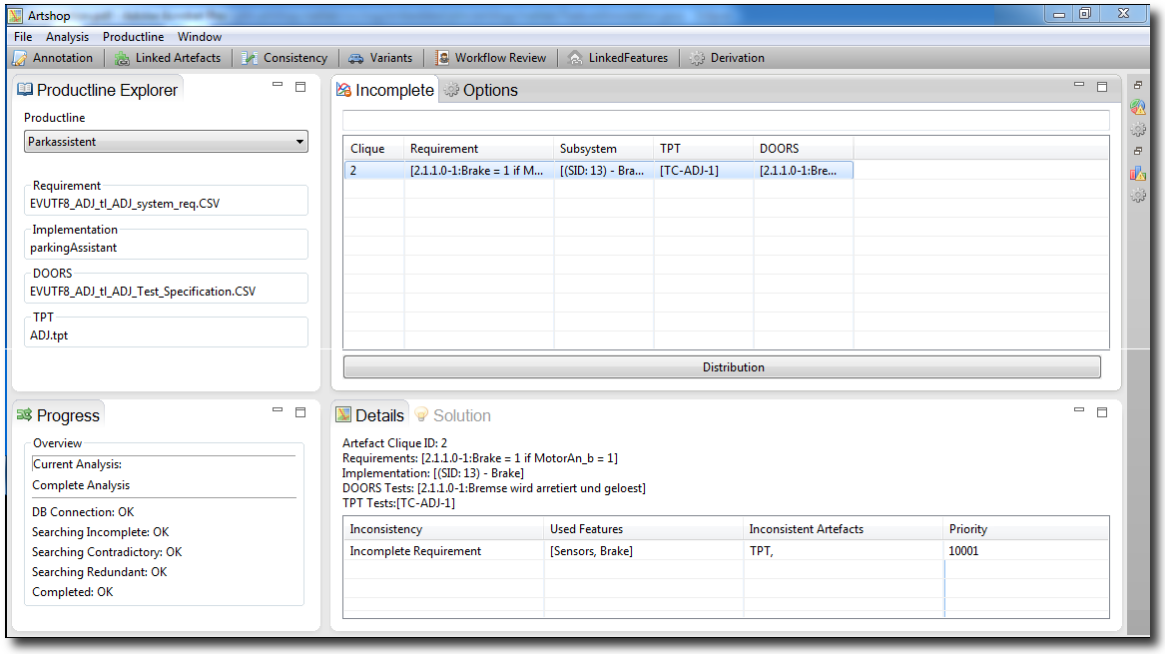


Abbildung 6.13: Das Resultat einer beispielhaften Konsistenzanalyse im Verwaltungs- und Analysewerkzeug des datenbankorientierten Ansatzes

Für die Auflösung der Inkonsistenz kann sich der Anwender im Reiter *Solution* Korrekturvorschläge unterbreiten lassen (vgl. Abbildung 6.14). Der erste Vorschlag basiert auf der Annahme, dass die Clique nicht korrekt ist. Der Anwender kann diese entweder löschen[1] (Button *Delete Clique*), die Clique manuell korrigieren (Button *Edit Clique*) oder die semi-automatische Korrektur der Clique verwenden (Button *Quick Fix*). Im letzteren Fall sucht das Werkzeug nach Testfällen, die mit den fehlenden Merkmalen verknüpft sind, und bietet sie dem Anwender zum Hinzufügen an.

Der zweite Vorschlag geht davon aus, dass die Clique korrekt ist, aber die Merkmalverknüpfungen fehlerhaft sind. Deren Korrektur erfolgt ebenfalls entweder manuell über den *Edit*-Button des jeweiligen Artefaktes oder semi-automatisch über den *Quick-Fix*-Button. Für die semi-automatische Korrektur werden fehlende Merkmale zur Verknüpfung mit den Elementen der Clique angeboten.

6.8 Verwandte Arbeiten

Viele Arbeiten, die sich denselben bzw. ähnlichen Herausforderungen widmen wie der in diesem Kapitel beschriebene Ansatz, wurden bereits in Abschnitt 4.5 beschrieben. An dieser Stelle gehe ich daher einerseits auf Arbeiten ein, die dem Ansatz *umsetzungstechnisch* verwandt sind. Der fachliche Kontext spielt in diesem Fall keine Rolle.

[1] Dies bedeutet selbstverständlich nur, dass die Verknüpfungen zwischen den Artefaktelementen der Clique entfernt werden. Die Elemente selbst bleiben erhalten.

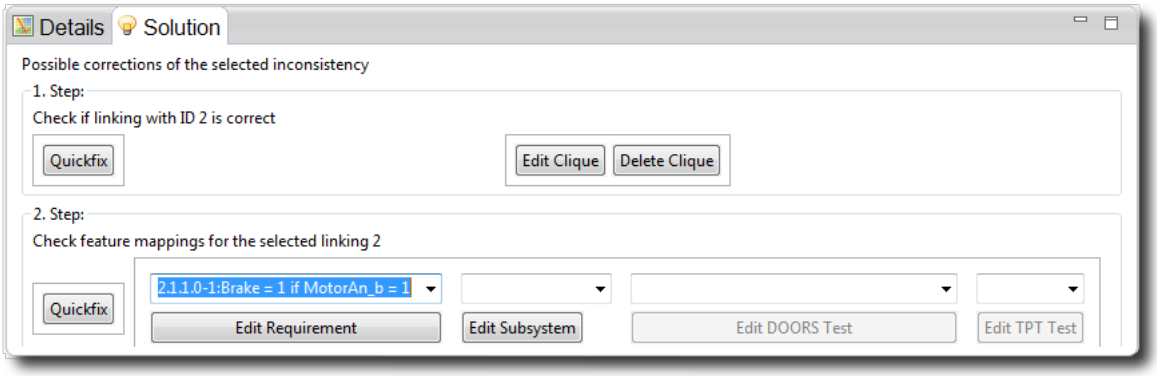


Abbildung 6.14: Möglichkeiten semi-automatischer Korrekturmaßnahmen im Verwaltungs- und Analysewerkzeug [162, S. 98]

Andererseits wird die Umsetzung der bereits in Abschnitt 4.5 beschriebenen, *fachlich* verwandten Arbeiten mit dem in diesem Kapitel vorgestellten Ansatz verglichen, sofern dies nicht bereits in Abschnitt 5.8 erfolgt ist.

Um die Werkzeugintegration gemäß Broy et al. [21] zu ermöglichen, baut der in diesem Kapitel beschriebene Lösungsansatz auf einer zentralen, artefaktübergreifenden, relationalen MySQL-Datenbank auf, die somit als integratives Architekturmodell fungiert, in das über die Verwaltungs- und Importkomponente die Artefakte aus verschiedenen Werkzeugen integriert werden. Durch die zusätzliche Integration des Merkmalmodells in die Datenbank können Elemente unterschiedlicher Artefakte mit denselben Merkmalen verknüpft werden, wodurch diese auf funktionaler Ebene verknüpft und bei konsequenter Anwendung sinnvoll auf Konsistenz geprüft werden können und die Datenbank dem Variantenmanagement dienen kann, wie von Dziobek et al. [34] gefordert. Annotationen im Sinne von Wolf et al. [164] integriert der vorgestellte Ansatz ebenfalls in die Datenbank. Damit diese leichter automatisiert verarbeitet und für Analysen verwendet werden können, werden sie über ein Verwaltungswerkzeug in die gemeinsame Datenbasis eingepflegt, wodurch die Anzahl der Annotationstypen und zugehörige Werte flexibel zur Laufzeit modifiziert und erweitert werden können. Somit kann zwar keine vollautomatische Verknüpfung von Artefakten, wie von Dziobek et al. [34] gefordert, erreicht werden, jedoch können einheitliche, artefaktübergreifende Annotationen Anhaltspunkte für die Zusammengehörigkeit von Artefaktelementen geben. Über die Integration der Java-basierten Implementierung in MATLAB/Simulink wird der Ansatz auch für die modellbasierte Entwicklung eingebetteter Software nutzbar, während Wolf et al. sich auf „klassische“ Software-Entwicklungsartefakte beziehen. Zudem wird ein zusätzlicher Werkzeugwechsel im Entwicklungsprozess vermieden. Die von Wolf et al. beschriebene Unterstützung der verteilten Arbeit im Team adressiert der vorgestellt Ansatz dadurch, dass die Daten in einer zentralen Datenbank abgelegt sind, auf die über beliebige Rechner zugegriffen werden kann. Jedoch steht hier nicht die Kooperation verteilter Teams im Vordergrund, sodass auch keine Benachrichtigung von Mitarbeitern erfolgt, wenn Änderungen an einem Artefakt umgesetzt werden. Dadurch, dass die Annota-

tionen in einer artefaktübergreifenden Datenbank und nicht direkt in die Werkzeuge integriert sind, lassen sich verschiedene Rollen dadurch unterstützen, da sie auch anderen Prozessbeteiligten zur Verfügung stehen. Somit wäre dies ein Ansatzpunkt, um auch verschiedene Rollen im Entwicklungsprozess nach Dziobek et al. zu integrieren. Aufgrund der aktuellen Beschränkung auf die Artefakte des Automobilherstellers (vgl. Abschnitt 2.3.1) können jedoch aktuell keine beim Zulieferer angesiedelten Rollen integriert werden. Die Integration in eine gemeinsame Datenbank ermöglicht Abfragen und Modellanalysen und unterstützt somit die Evolution der Produktlinie, wie von Thomas et al. [150] gefordert.

Wie der von Wolf et al. in [164] vorgestellte Ansatz auch, bezwecken die von Steinbauer, Zdrahal, Mulholland et al. in [95, 141, 165] vorgestellten Clockwork-Werkzeuge *Clockwork Enrich Tool* und *Clockwork Scenario Tool* ebenfalls die Unterstützung des Informationsaustausches im Team. Die Werkzeuge sind Simulink-spezifisch implementiert und folglich nahtlos integriert. Jedoch sind die Informationen weder in Syntax und Semantik vereinheitlicht noch stehen sie für andere Artefakte zur Verfügung. Der Ansatz dieser Dissertation schafft Abhilfe, indem er vollständig auf Basis der Datenbank arbeitet, deren Struktur die vereinheitlichte und artefaktübergreifende Annotationsspezifikation unterstützt, und mit Java Simulink-unabhängig aber dennoch nahtlos in MATLAB/Simulink integrierbar implementiert wurde.

Effizienzprobleme von Modelltransformationen inspirieren Varró et al. zur Implementierung eines Ansatzes, der Graphtransformationen auf Basis von Datenbanken durchführt [158]. Hier wird erstmals untersucht, wie Datenbanken für Modelltransformationen verwendet werden können, während andere Arbeiten sich bis dato mit der Anwendung von Modelltransformationen zur Unterstützung des Datenbankmanagements beschäftigen (z. B. [3, 4, 59, 68]). Dazu werden die linke Seite der Transformationsregeln sowie die negative Anwendungsbedingung und der Pre-Graph durch Datenbanksichten (Views) spezifiziert, die die Kriterien für Matchings definieren, z. B. Werte von Entitätsattributen bzw. deren Relationen. Die Autoren evaluieren die Effizienz und Skalierbarkeit ihres Ansatzes gegenüber den Graphtransformationsmaschinen AGG [144] und PROGRES [133, 135], die ohne Zuhilfenahme einer Datenbank arbeiten. Dabei kann eine signifikante Effizienz- und Skalierbarkeitsverbesserung durch den vorgestellten Ansatzes nachgewiesen werden.

Auch Kalnins et al. können über eine datenbankorientierte Implementierung von Modelltransformationen signifikante Effizienzsteigerungen erreichen [66]. Hier wird zwecks Effizienzsteigerung die Implementierung des Pattern Matchings der Transformationssprache MOLA [56, 65] durch die Übersetzung der Regeln in SQL-Anfragen an eine MySQL-Datenbank modifiziert. Die Spezifikations*methodik* für den Anwender ändert sich also wie bei Varró et al. nicht. Die Autoren evaluieren in Benchmark-Tests die Effizienz des Ansatzes und vergleichen ihn ebenfalls mit dem zuvor genannten AGG-Ansatz von Taenzer et al. [144].

Anquetil et al. befassen sich mit Rückverfolgbarkeit, fokussieren aber insbesondere die Rückverfolgbarkeit von *Variabilität* über Artefaktgrenzen hinweg [5]. Das dazu vorgestellte AMPLE Traceability Framework (ATF) besitzt u. a. einen Persistenz- und Anfragemanager für Verknüpfungen zwischen Artefakten. Deren Struktur wird durch

ein Metamodell des EMFs beschrieben und entweder in einer XML-Datei oder einer relationalen Datenbank abgelegt. Für die Persistierung in der Datenbank verwenden die Autoren EMF Teneo [38]. Über den Anfragemanager können die Verknüpfungen gemäß bestimmter Kriterien abgefragt werden. Die Artefakte selbst werden hier offenbar nicht in der Datenbank persistiert. Stattdessen werden die zu verknüpfenden Artefaktelemente über sogenannte *Traceable Artefacts* mit einer URI zum Originalartefakt im Metamodell repräsentiert. Dies hat den Vorteil, dass nicht, wie beim Ansatz dieser Dissertation, Inkonsistenzen zwischen Originalartefakt und dessen Repräsentanten in der Datenbank entstehen können. Jedoch könnten Analysen eines einzelnen Artefaktes so auch nur unter Rückgriff auf das Original erfolgen. Der in dieser Dissertation verfolgte Ansatz, um z. B. die Struktur von Simulink-Modellen zu analysieren, kommt in diesem Fall mit weniger Rückgriffen auf das originale Simulink-Modell aus. Da das ATF aber derartige Analysen nicht beabsichtigt, ist dieser Ansatz hier sinnvoller.

Alexander Reder identifiziert einen Mangel an Werkzeugen, die die Problematik von Inkonsistenzen zwischen Entwurfsregeln und Modellen umfassend adressieren [123]. Dazu gehört neben der Identifikation von Inkonsistenzen und deren Reparaturen auch die Berücksichtigung möglicher Seiteneffekte auf andere Regeln. Beispielsweise kann die Korrektur einer Regelverletzung gleichzeitig andere Regelverletzungen korrigieren oder hervorrufen. Der von ihm vorgestellte, inkrementelle Ansatz kombiniert dazu eine Laufzeitanalyse des Verhaltens von Entwurfsregeln sowie eine statische Analyse ihrer Struktur. Die Konsistenzregeln werden in Prädikatenlogik formuliert und können auf alle Arten von Modellierungssprachen angewendet werden. Teile dieser Regeln werden dabei mit Modellteilen verknüpft, sodass auf Basis dieser Mappings Teile der Regeln, auf die die Inkonsistenz zurückzuführen ist, identifiziert und entsprechende Reparaturmöglichkeiten abgeleitet werden können. Zudem können die Mappings für die Identifikation möglicher Seiteneffekte genutzt werden. Im Gegensatz zur vorliegenden Dissertation betrachtet Reder also die Inkonsistenz zwischen Entwurfsregeln und deren Umsetzung, während sich diese Dissertation mit Inkonsistenz zwischen Artefakten bzw. Merkmalverknüpfungen im Variantenmanagement beschäftigt. Die Seiteneffekte von Inkonsistenzkorrekturen bestehen hier darin, dass z. B. durch die Korrektur von Merkmalverknüpfung verschiedene Cliquen beeinflusst werden können.

Lauenroth und Pohl präsentieren einen Ansatz für die Konsistenzprüfung von Verhaltensinvarianten in der Software-Produktlinien-Entwicklung [76]. Der Ansatz verwendet Model-Checking-Methoden, um unter Berücksichtigung der Variantenzugehörigkeit die Konsistenz von Domänenanforderungen zu überprüfen. Der Ansatz basiert auf dem Orthogonalen Variabilitätsmodell [114, S. 72-86]. Jede Domänenanforderung wird dazu mit einer Variante assoziiert. Zwecks automatisierter Prüfung von Konsistenz wird für die Anforderungsspezifikation eine Spezifikationssprache für Verhalten aus endlichen Automaten und aussagenlogischen Invarianten über den Automatenzuständen definiert. Transitionen im Automaten unterteilen sich dadurch in gemeinsame und variantenspezifische Transitionen. Während erstere bei der Ableitung jedes produktspezifischen Lastenhefts übernommen werden, erfolgt die Übernahme der variantenspezifischen Transitionen nur, wenn die assoziierte Variante ausgewählt wurde. Der vorgestellte Algorithmus für die Konsistenzprüfung der Domänenanforderungen besteht aus zwei

Phasen. Zunächst werden *potenzielle* Inkonsistenzen identifiziert. Dabei handelt es sich um Paare zwischen einer Menge von Anforderungen und dem Variantenvektor (aktivierte bzw. deaktivierte Varianten), für den die Inkonsistenz besteht. Für diesen Schritt ist die zuvor genannte Assoziation zwischen Anforderungen und Varianten nötig. Im zweiten Schritt werden daraus die *tatsächlichen* Inkonsistenzen gesucht, indem geprüft wird, ob der Variantenvektor aufgrund des Variabilitätsmodells überhaupt gültig ist. Das vorgestellte Rahmenwerk wurde in Java implementiert und verwendet einen SAT-Solver für die Gültigkeitprüfung des Variantenvektors.

Cmyrev et al. stellen einen Ansatz für die Unterstützung des merkmalbasierten Variantenmanagements durch Konsistenzprüfung der Merkmalverknüpfungen von Anforderungen und Tests vor, der, wie in Abschnitt 6.7 beschrieben, für die vorliegende Dissertation erweitert und in den datenbankorientierten Ansatz integriert wurde [26]. Bei korrekter Merkmalverknüpfung der Artefaktelemente werden hier verbliebene Inkonsistenzen als Indiz für Artefaktinkonsistenz betrachtet. Cmyrev et al. gruppieren Inkonsistenzen in die Kategorien Widerspruch, Redundanz und Unvollständigkeit. Die Inkonsistenzen werden formal beschrieben und durch ein in der Entwicklung befindliches Eclipse-basiertes Plugin für pure::variants [13, 122] identifiziert. Die Variabilität ist hier als Merkmalmodell in FODA-Notation [67] beschrieben.

Tibor Farkas (et al.) befasst sich mit Herausforderungen, die sich insbesondere in der Entwicklung von Elektrik/Elektronik-Systemen stellen [43, 44, 45]. Er identifiziert einen Mangel an Abstraktion in existierenden Ansätzen zur regelbasierten Konformitätsprüfung, sowie deren zu starke Fokussierung auf bestimmte Werkzeuge. Farkas beschreibt ein Microsoft .NET-basiertes [91] Rahmenwerk, das im Rahmen des Forschungsprojektes MESA[1] [46] entstanden ist und eine werkzeugübergreifende Richtlinienkonformitätsprüfung von Artefakten beabsichtigt [44, 45]. Für den Einbezug des Entwicklungsprozesses werden die Artefakte auch hier in eine gemeinsame Ablage integriert, die über eine Datenbank realisiert ist. Die in [43] vorgestellte, erweiterbare Sprache namens *Query-Based Rule Description Language (QRDL)* dient der Spezifikation artefaktübergreifender Richtlinienüberprüfungen. Die in QRDL spezifizierten Regeln werden in die Ausdrucks- bzw. Programmiersprachen OCL, LINQ [9] und M-Skript übersetzt. Die Sprache erlaubt neben primitiven, aussagenlogischen und arithmetischen Ausdrücken auch die Spezifikation komplexerer prädikatenlogischer, relationenalgebraischer und regulärer Ausdrücke. Die zugrunde liegenden Metamodelle der Artefakte finden sich in [43, 45]. Die vorliegende Dissertation legt im Gegensatz zu den Arbeiten von Farkas (et al.) den Fokus stärker auf Modellanalyse von Simulink-Modellen, Konsistenzsicherung im Kontext des Variantenmanagements und Informationsanreicherung, weshalb sich die hier verwendeten Metamodelle auch nur auf die für derartige Analysen relevanten Elemente beschränken. Zudem wird die Zusammengehörigkeit von Artefakten anhand gleicher Bezeichner identifiziert, während die vorliegende Dissertation beabsichtigt, über die Integration einheitlicher Informationen anderweitige Anhaltspunkte für potenziell zusammengehörige Elemente zu geben.

[1] Metamodellierung zur Automatisierung von Analyse- und Entwicklungsmethoden für Software im Automobil

7 Evaluierung

Für die Evaluierung der Analyseansätze werden in Abschnitt 7.1 verschiedene Fallstudien vorgestellt, die praktisch relevante und prototypisch implementierte Anwendungsfälle beschreiben. Auf deren Basis werden die Ansätze anschließend in Abschnitt 7.2 hinsichtlich ihrer Anwendbarkeit in der Praxis bewertet. Dazu gehören neben der Effizienz und Skalierbarkeit der Artefaktintegration im Hinblick auf die Modellgröße auch die Integrationsmöglichkeit in MATLAB/Simulink und die Generalisierbarkeit sowie ihre Handhabung in der Entwicklung. Schließlich werden die Erkenntnisse zusammengefasst und Einschränkungen der Bewertung erläutert.

7.1 Fallstudien

Um die vorgestellten Analyseansätze zu bewerten, wurden mit ihnen verschiedene Analysen implementiert, die als relevant für die industrielle Praxis identifiziert wurden. Diese sollen in den folgenden Abschnitten kurz beschrieben werden.

7.1.1 Signalverfolgung

Im Zuge der Evolution von Simulink-Modellen ist es häufig von Interesse, zu identifizieren, welche Modellteile von einem Signal abhängen bzw. von welchen Modellteilen ein Signal abhängt, um die Auswirkung einer Modellmodifikation abschätzen zu können. Zu diesem Zweck wurde eine Analyse implementiert, die auf Basis eines im Simulink-Modell auszuwählenden Blocks gestartet wird und eine Sicht erzeugt, die alle Elemente des Modells entfernt, die nicht von den aus dem Block ausgehenden Signalen abhängen. Abbildung 7.1 veranschaulicht die Analyse schematisch am Beispiel der Verfolgung der aus dem Block *Konstante 1* in Abbildung 2.6 ausgehenden Signale. Wie rechts oben ersichtlich, wird diese Analyse rekursiv innerhalb von Subsystemen fortgesetzt.

Es sei darauf hingewiesen, dass es eine Möglichkeit der Signalverfolgung in MATLAB/Simulink bereits gibt. Dabei werden auf Basis einer Linie die von ihr beeinflussten Modellteile (also Blöcke und Linien) farbig markiert, jedoch keine Modellteile ausgefiltert. Durch die Verwendung der implementierten Analyseansätze, insbesondere des datenbankorientierten Ansatzes und der dazu erstellten Java-Funktionsbibliothek besteht aber die Möglichkeit, mehr Einfluss auf die Analyse zu nehmen. Beispielsweise könnte man die Signalverfolgung auch konfigurierbar gestalten, sodass verschiedene Blocktypen unterschiedlich behandelt würden, dass beim Start einer Analyse auf einem Signalbus die zu verfolgenden Signale aus der Liste der vom Bus transportierten Signale

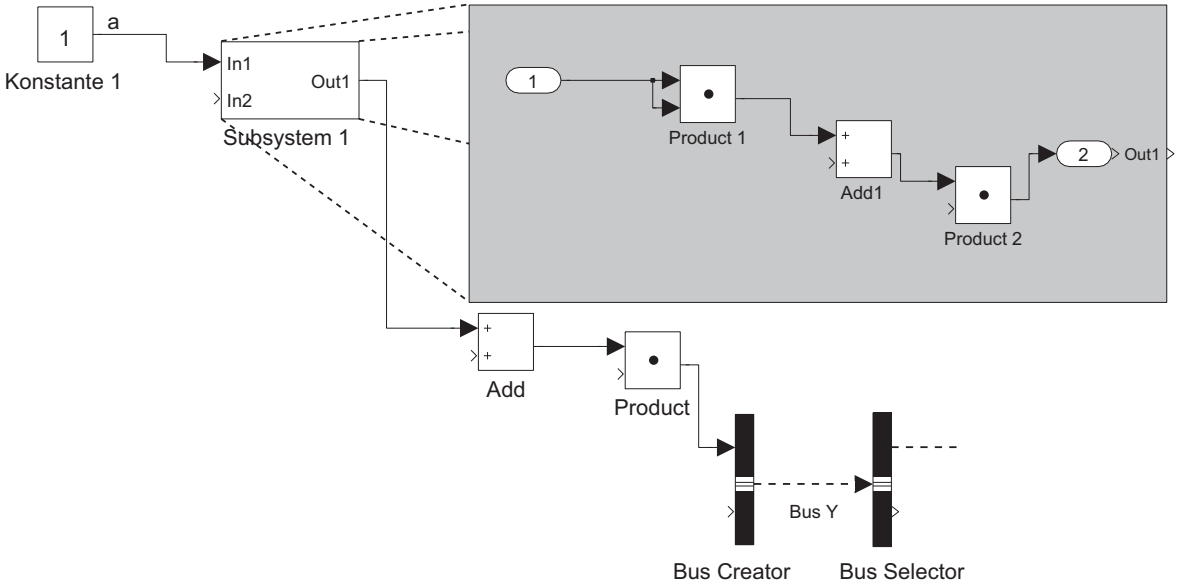
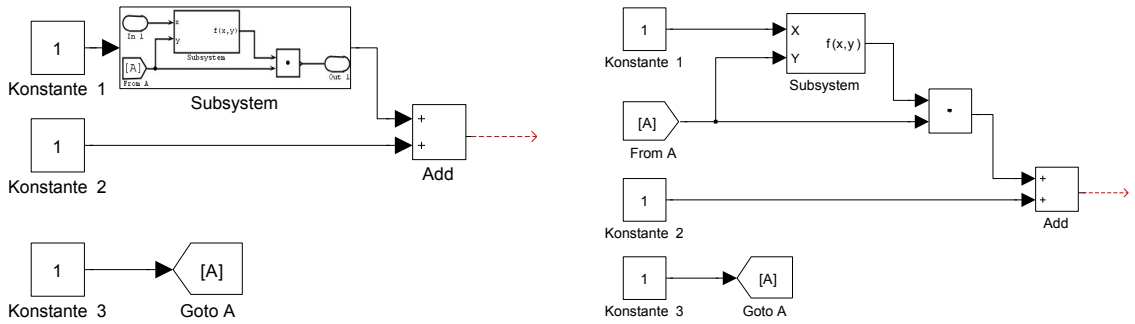


Abbildung 7.1: Die Ergebnissicht einer vorwärts gerichteten Signalverfolgungsanalyse auf Basis des Blocks Konstante 1 aus Abbildung 2.6

ausgewählt werden oder die maximale Rekursionstiefe der zu berücksichtigenden Subsystemhierarchie festgelegt werden könnte. So wird aktuell z. B. die Signalverfolgung im Gegensatz zur Analyse in MATLAB/Simulink auch hinter Switch-Blöcken fortgesetzt, um auch *potenzielle* Abhängigkeiten zu identifizieren. Zwar lassen sich solche Analysen auch mittels der MATLAB-Skriptsprache implementieren, jedoch eignet diese sich nicht für die Implementierung komplizierterer Analysen, wie auch aus verwandter Literatur bekannt (vgl. Abschnitt 4.5.3).

7.1.2 Auflösen der Busstruktur

Die Integration verschiedener Signale in Signalbusse in MATLAB/Simulink dient der Übersichtlichkeit großer Modelle. In manchen Fällen, z. B. bei einer besonders hohen Schachtelungstiefe von Signalbussen, kann jedoch auch der umgekehrte Weg von Nutzen sein, um auf einen Blick sämtliche Signalflüsse im Modell zu visualisieren. Dazu wurde eine Analyse implementiert, die dazu rekursiv jeden Signalbus durch die atomaren Signale ersetzt, die er transportiert. Jede Linie, die den Bus transportiert, wird dabei durch so viele Linien ersetzt wie der Bus atomare Signale beinhaltet. Dazu ist es folglich ggf. nötig, Blöcke zu vervielfachen, die nur mit einer Linie verbunden sein dürfen oder Blöcken weitere Ports hinzuzufügen. So muss beispielsweise Subsystemen, deren ein- bzw. ausgehende Linien Signalbusse transportieren, die entsprechende Menge an Ports und damit auch an Schnittstellenblöcken hinzugefügt und die innere Struktur an die somit geänderte Schnittstelle zur Außenwelt angepasst werden.



(a) Original

(b) Auflösen von Subsystem oberster Ebene

Abbildung 7.2: Auflösung der Subsystemhierarchie bis Tiefe 1

7.1.3 Auflösen der Subsystemhierarchie

Ein weiteres Beispiel für die Visualisierung der Zusammenhänge von Modellteilen betrifft den rekursiven Aufbau eines Simulink-Modells durch geschachtelte Subsysteme. Daher wurde eine Sicht generiert, die die Subsysteme auflöst, d. h. den Inhalt rekursiv eine Hierarchieebene nach oben verschiebt und die Schnittstellenblöcke dabei eliminiert. Hierbei profitiert man insbesondere von der Verknüpfung der Schnittstellenblöcke mit den Subsystemports bzw. den virtuellen Quellen und Zielen von Linien, die während der Modellaufbereitungsphase des jeweiligen Ansatzes erstellt wurden (vgl. Abschnitte 5.4.1 und 6.3.1), da die Verbindungen zwischen der Außenwelt und der inneren Struktur eines Subsystems so besonders komfortabel abgefragt werden können. Diese Analyse wurde mit Hilfe beider Ansätze umgesetzt, jedoch für den datenbankorientierten Ansatz erweitert. Hier kann zusätzlich eingestellt werden, bis zu welcher Hierarchietiefe die Auflösung erfolgen soll. Dadurch können Analysen auf denjenigen Teil des Modells beschränkt werden, der im jeweiligen Fall von Interesse ist. Tiefer liegende Subsysteme werden dann inklusive ihres Inhaltes unverändert in die Ergebnissicht übernommen, wodurch ein Effizienzgewinn zu erwarten ist. Die modellbasierte Analyse löst hingegen stets die gesamte Subsystemhierarchie auf. Abbildung 7.2 zeigt ein Beispiel für die Auflösung der Subsystemhierarchie mit einer Hierarchietiefe von 1. Abbildung 7.2a zeigt das Modell, welches analysiert wird[1]. Abbildung 7.2b zeigt das Resultat.

7.1.4 Schnittstellenidentifikation von Subsystemen

Die Schnittstellensicht visualisiert, welche Signale zwischen zwei beliebigen, im Simulink-Modell auszuwählenden Subsystemen A (Quelle) und B (Ziel) ausgetauscht werden. Dazu werden nur diese beiden Subsysteme aus dem Modell geladen, die mit den Ausgangs- bzw. Eingangsports verknüpften Signale ausgelesen, deren Schnittmenge gebildet und pro Signal eine Linie zwischen den Subsystemen gezogen. Diese Sicht ist aufgrund der Modellaufbereitungsphase für beide Ansätze besonders einfach zu implementieren,

[1] Das Subsystem wurde hier zu Illustrationszwecken mit einem Bild seines „Innenlebens“ versehen. In der Realität ist eine solche Vorschau auf den inneren Aufbau nicht möglich.

da Signale, Linien und Ports in der Aufbereitungsphase direkt miteinander verknüpft wurden. Ohne die Aufbereitungsphase hingegen hätten dafür ggf. diese Signale erst durch Rückverfolgung identifiziert werden müssen. Die ausgetauschten Signale manuell zu identifizieren, ist ebenfalls aufwändig, da durch ihre Schachtelung in Signalbussen und die Verwendung von Sprungblöcken Signale zum Teil über weite Strecken und über Subsystemhierarchieebenen hinweg unverändert transportiert werden können.

7.1.5 Markieren auskonfigurierter Funktionalität

Da das untersuchte Simulink-Modell ein 150%-Modell einer Produktlinie darstellt, werden bei der Variantenableitung über Parameter aus einer Konfigurationsdatei bestimmte Modellteile deaktiviert (vgl. Abschnitt 2.3.3). Dadurch kann es passieren, dass ein in einer Variante aktiviertes Subsystem ein Signal eines anderen Subsystems verarbeitet, welches aber selbst nicht in dieser Variante aktiviert ist. In diesem Fall ist das Signal undefiniert, wodurch unerwartetes Verhalten des Gesamtsystems verursacht werden kann. Solche Konfigurationsfehler müssen daher vom Ingenieur geprüft werden, wenn er eine Variante ableiten möchte. Um diese Aufgabe zu erleichtern, lassen sich nun die auskonfigurierten Teile im Modell automatisiert rot markieren, wodurch direkt ersichtlich ist, welche Teile aktiviert sind. Diese Aufgabe könnte auch ein MATLAB-Skript lösen. Durch den Einbezug solcher Konfigurationsinformationen in die Analyserahmenwerke lässt sich deren Analysepotenzial jedoch um Variabilitätsaspekte erweitern und somit das Variantenmanagement unterstützen. Da das Variantenmanagement idealerweise aber auf Merkmalmodellen anstelle von Konfigurationsdateien basiert, wurde im Falle des datenbankorientierten Ansatzes anstelle der Konfigurationsdatei ein konfiguriertes Merkmalmodell als Variantenbeschreibung verwendet. Dabei wird davon ausgegangen, dass die Merkmale des Merkmalmodells in der Datenbank mit allen Artefaktelementen verknüpft sind, auf die sie sich beziehen. Dies ist wichtig, um ein durchgängiges Variantenmanagement zu ermöglichen, da es dadurch ein Modell gibt, welches alle Artefakte auf funktionaler Ebene miteinander verknüpft. Da eine Variante als Teilmenge von Merkmalen aufgefasst werden kann, gestaltet sich diese Analyse extrem komfortabel über die Abfrage von Subsystemen, die mit keinem der angegebenen Merkmale verknüpft sind. Deren SIDs können dann einfach an ein MATLAB-Skript übergeben werden, das diese Subsysteme rot markiert.Für diese Fallstudie wurden nur sogenannte *Module* berücksichtigt, d. h. Subsysteme, die ein Merkmal implementieren[1].

7.1.6 Konsistenzuntersuchung

Das Verwaltungs- und Analysewerkzeug aus Abbildung 6.7 kann den Anwender bei der Artefaktintegration und im Varianten- und Konsisistenzmanagement unterstützen. Es ist über Kontextmenüs gut in MATLAB/Simulink integriert, wodurch eine komfortable Annotation von Subsystemen mit Merkmalen aus der Datenbank ermöglicht wird. Auch

[1] Es existieren auch Subsysteme, die als Hilfsfunktionen dienen und daher nicht explizit ein Merkmal umsetzen.

können Subsysteme direkt über Kontextmenüs in MATLAB/Simulink in Cliquen integriert und so mit anderen Artefakten verknüpft werden. Voraussetzung dafür ist jedoch, dass die Artefakte, aus denen die Elemente stammen sollen, vorher in die Datenbank importiert wurden. Dies ist im Falle des Simulink-Modells direkt über das entsprechende Import-Skript möglich, für andere Artefakte ist hingegen das externe Verwaltungs- und Analysewerkzeug zu verwenden, wobei die Artefaktintegration hier über einen gemeinsamen Importdialog ebenfalls sehr benutzerfreundlich umgesetzt wurde (vgl. Abbildung 6.7). Sind die Merkmalverknüpfungen durchgeführt und Cliquen erstellt, so lassen sich per Mausklick alle Analysen auf Merkmalkonsistenz untersuchen, wobei die Merkmalbasis konfigurierbar und somit leicht auf einen möglicherweise unterschiedlichen Entwicklungsprozess in Unternehmen anpassbar ist. Auch werden dem Anwender durch sogenannte *QuickFix*-Vorschläge Korrekturmaßnahmen für die Cliquenkorrektur und Merkmalverknüpfung angeboten. Verbleibende Merkmalinkonsistenzen sind damit Indizien für Artefaktinkonsistenz.

Hinderlich für den Praxiseinsatz ist hingegen, dass Cliquen und Merkmalverknüpfungen manuell erstellt werden müssen. Da in der Praxis in der Regel bereits Artefakte existieren, die einem längeren Entwicklungsprozess unterliegen, muss dieser Aufwand folglich *rückwirkend* für den Status Quo erfolgen, um erfolgreiche Konsistenzprüfungen durchführen zu können. Er ist also nicht auf *zukünftige* Änderungen beschränkt. Durch kontinuierliche Anwendung von Anfang der Entwicklung an dürfte der Aufwand sich jedoch aufgrund der dann zuverlässigeren Konsistenzprüfungen rentieren. Schließlich kann das Werkzeug semantische Inkonsistenzen zwischen Artefakten nicht direkt identifizieren, sondern nur indirekt von Merkmalinkonsistenzen auf *potenzielle* Artefaktinkonsistenzen schließen.

7.2 Anwendbarkeit

Um die Anwendbarkeit der Ansätze in der Praxis zu beurteilen, wurden sie von den Entwicklern am Lehrstuhl auf Grundlage der während der Fallstudien erworbenen Erfahrung und teilweise durch den Industriepartner bewertet. Die Bewertungskriterien erstrecken sich auf die Effizienz und Skalierbarkeit des Transfers von Simulink-Modellen in die jeweiligen Rahmenwerke, ihre praktischen Handhabung und auf ihre Erweiterbarkeit und Generalisierbarkeit. Die Ergebnisse werden in den folgenden Abschnitten erläutert.

7.2.1 Effizienz und Skalierbarkeit

Von besonderer Bedeutung hinsichtlich der Effizienz und Skalierbarkeit ist der initiale Import des Simulink-Modells in das gemeinsame Datenmodell inklusive dessen struktureller Aufbereitung für spätere Analysen gemäß den Abschnitten 5.4.1 und 6.3.1, da sich später Abhängigkeiten innerhalb des Simulink-Modells leichter identifizieren lassen und auch große Simulink-Modelle in das Datenmodell importiert werden müssen.

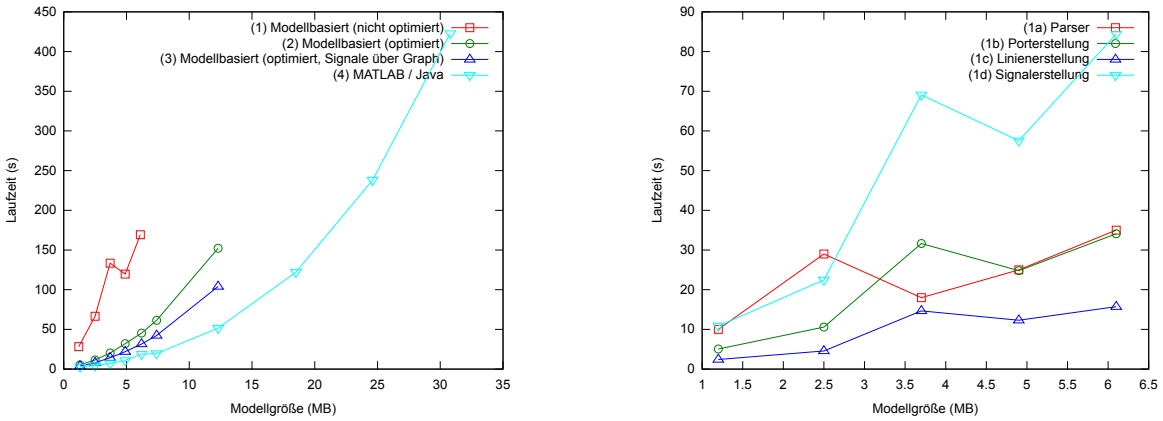
Daher wurde die Effizienz des Imports eines Simulink-Modells abhängig von der Größe des Modells für die beiden Ansätze untersucht. Dazu wurde ein beispielhaftes,

	Simulink-Modell									
	1	2	3	4	5	6	7	8	9	10
Charakteristika der importierten Simulink-Modelle										
Größe (MB)	1,3	2,5	3,7	4,9	6,2	7,4	12,3	18,5	24,6	30,8
# Blöcke	991	1.981	2.971	3.961	4.951	5.941	9.901	14.851	19.801	24.751
# Subsysteme	226	452	678	904	1.130	1.356	2.260	3.390	4.520	5.650
Laufzeit in Sekunden										
(1)	28,32	66,58	133,41	119,65	169,23	-	-	-	-	-
(1a)	10	29	18	25	35	-	-	-	-	-
(1b)	5,05	10,58	31,63	24,81	34,09	-	-	-	-	-
(1c)	2,42	4,56	14,67	12,31	15,72	-	-	-	-	-
(1d)	10,85	22,44	69,11	57,53	84,42	-	-	-	-	-
(2)	5,29	11,54	20,30	32,02	45,38	61.45	152,20	-	-	-
(3)	3,80	8,08	14,29	21,93	31,60	42,33	104.00	-	-	-
(4)	2,82	4,60	7,11	10,82	18,84	19,67	51,62	122,03	238,02	422,71

Tabelle 7.1: Charakteristika und Zeiten für die Erstellung des EMF-Modells (nicht optimierter, modellbasierter Ansatz aus Kapitel 5) bzw. des in der Datenbank zu persistierenden Java-Modells (datenbankorientierter Ansatz aus Kapitel 6) für ein Simulink-Modell mit Hierarchietiefe 12; (1) Modellbasiert (nicht optimiert), (1a) Parser, (1b) Porterstellung, (1c) Linienerstellung, (1d) Signalerstellung, (2) Modellbasiert (optimiert), (3) Modellbasiert (optimiert, Signale über Graph), (4) MATLAB/Java

aber kein real verwendetes Simulink-Modell verwendet, welches vom Industriepartner zur Verfügung gestellt wurde und dem realen Modell in Bezug auf die Struktur (z. B. verwendete Blocktypen und Variabilitätsmuster, Aktivierungsstrukturen von Modulen, Schachtelungstiefe von Subsystemen) ähnlich ist. Um die Skalierbarkeit der Ansätze zu evaluieren, wurde dieses Modell durch Vervielfältigung der enthaltenen Blöcke künstlich sukzessive vergrößert. Dadurch, dass diese Vervielfältigung innerhalb des Modells durch mehrfaches Hinzufügen des Modellinhaltes auf derselben Hierarchieebene erfolgte, konnte sichergestellt werden, dass sich dabei nur die Größe des Modells, nicht aber die Hierarchietiefe oder die Anzahl der Signale ändert. Der obere Teil von Tabelle 7.1 stellt die wesentlichen Charakteristika dieser so entstandenen Modelle dar. Da Subsysteme auch Blöcke darstellen, ist auch ihre Anzahl in der Anzahl der Blöcke enthalten. Der untere Teil beschreibt die Laufzeiten, um aus dem jeweiligen Simulink-Modell ein analysierbares Modell für den jeweiligen Ansatz zu generieren. Im Falle des datenbankorientierten Ansatzes endet die Messung mit dem Vorliegen des vollständigen Java-Modells, welches im nächsten Schritt in der Datenbank zu persistieren ist, da an dieser Stelle bereits ein mit Methoden der Java-Bibliothek analysierbares Modell vorliegt und somit derselbe Zustand erreicht ist wie mit dem Vorliegen des EMF-Repräsentanten beim modellbasierten Ansatz.

Zunächst einmal sind die Zeilen (1) und (4) von Interesse, da diese die beiden (nicht optimierten) Ansätze der Kapitel 5 und 6 darstellen, die in rot (Graph mit Quadraten) bzw. hellblau (Graph mit nach unten ausgerichteten Dreiecken) in das Diagramm in Abbildung 7.3a eingezeichnet sind. Dabei offenbart sich ein deutlicher Skalierbarkeits- und Effizienzvorteil für das MATLAB-Java-Skript des datenbankorientierten Ansatzes



(a) Gesamtlaufzeiten

(b) Einzelschritte (modellbasiert)

Abbildung 7.3: Laufzeiten der untersuchten Ansätze aus Tabelle 7.1

aufgrund des langsameren Laufzeitanstiegs. Als Gründe für die Unterschiede zwischen den Ansätzen lassen sich insbesondere zwei Faktoren ausfindig machen. Der erste Faktor ist konzeptbedingt. So wird beim modellbasierten Ansatz das gesamte Simulink-Modell geparst bzw. transformiert (vgl. Abschnitt 5.4.1), während das MATLAB-Java-Skript nur diejenigen Informationen des Modells berücksichtigt, die für die hier betrachteten Analysen relevant sind. Zudem werden die Modelle mit jedem Transformationsschritt aufgrund hinzukommender Entitäten (z. B. Ports, Signale) größer. Verschärft wird das Phänomen außerdem durch das textintensive XMI-Datenformat des EMF. Bedingt durch die großen Modelle scheitert der Ansatz auch frühzeitig an Speicherproblemen. Ein weiterer Faktor offenbart sich bei Betrachtung der einzelnen Schritte des modellbasierten Ansatzes (vgl. Abbildung 7.3b bzw. Zeilen (1a)-(1d) in Tabelle 7.1), die sich zu dessen Gesamtlaufzeit summieren. Hier hat allein die Signalerstellung und -verknüpfung mit einem Anteil von 33,7%-51,8% wesentlichen Einfluss auf die Gesamtlaufzeit. Dies deutet darauf hin, dass sich auch die Modelltransformationssprache ATL nicht für die Verfolgung komplexer Pfade in Modellen eignet, die gemäß Amelunxen et al. auch für rein deklarative Graphtransformationen ein Problem darstellt [2]. Keine Erklärung konnte für den Ausreißer der Laufzeit für Simulink-Modell 3 beim nicht optimierten, modellbasierten Ansatz (1) gefunden werden, der systematisch sowohl in verschiedenen Messreihen als auch auf verschiedenen Rechnern zutage trat.

Diese Erkenntnisse führen zu den im folgenden Abschnitt vorgestellten Optimierungsmöglichkeiten für den modellbasierten Ansatz.

Optimierungsansätze Abbildung 7.4 gibt einen Überblick über die beiden Analyseansätze dieser Dissertation sowie zwei in Abschnitt 5.7 angesprochene Optimierungsmöglichkeiten für das modellbasierte Vorgehen. Beide Optimierungsansätze arbeiten mit reduzierten EMF-Modellen, d. h. von Beginn an werden diejenigen Informationen, die für die hier betrachteten Analysen zunächst irrelevant sind, wie auch beim datenbankorientierten Vorgehen, nicht in die EMF-Modelle übernommen. Ein Ansatz ((2)

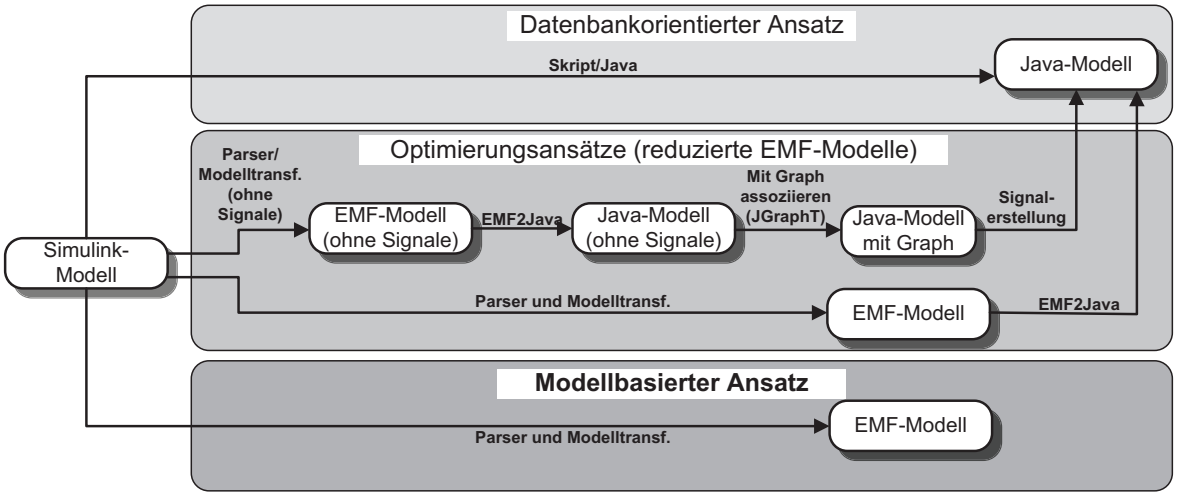


Abbildung 7.4: Optimierungen für den modellbasierten Ansatz

in Tabelle 7.1) verfährt ansonsten konzeptionell wie der ursprüngliche, modellbasierte Ansatz[1]. Folglich könnten Analysen auch hier durch Transformationen des dabei erstellten EMF-Modells umgesetzt werden. Da diesem Modell jedoch Informationen fehlen, um, wie beim ursprünglichen Ansatz, ein Analyseergebnis in MATLAB/Simulink darzustellen, wird aus diesem Modell ein Java-Modell im Sinne des datenbankorientierten Ansatzes erzeugt, sodass dessen Analysen und Visualisierungsmöglichkeiten verwendet werden können.

Die zweite Optimierungsmöglichkeit ((3) in Tabelle 7.1) besteht darin, die Signalerstellung über Graphalgorithmen des Java-Rahmenwerks JGraphT zu erstellen. Nach dem Parsen und Aufbereiten des Simulink-Modells liegt hier also zunächst ein EMF-Modell vor, welches noch keine Signale enthält. Dieses wird nun in ein Java-Modell umgewandelt, für das anschließend ein Graph erzeugt wird, der mit dem Java-Modell verbunden ist. Der assoziierte Graph kann nun verwendet werden, um mittels Traversierungsalgorithmen Signale zu identifizieren, die über die Linien oder Signalbusse des Modells transportiert werden, und das Java-Modell entsprechend anzureichern. Das Resultat ist wiederum das Java-Modell des datenbankorientierten Ansatzes.

Die Gesamtlaufzeiten der optimierten Ansätze sind ebenfalls in das Diagramm aus Abbildung 7.3a eingezeichnet. Der modellbasierte Ansatz (1) konnte durch die Reduktion der Modellgröße (2) und Ausgliederung der Signalerstellung und -verknüpfung (3) im Median um 86,6% gegenüber dem ursprünglichen Ansatz verbessert werden. Jedoch konnte auch dieser Ansatz nicht mit dem MATLAB-Java-Skript (4) des datenbankorientierten Ansatzes mithalten. Somit eignet sich aus Effizienz- und Skalierbarkeitsgründen dennoch der datenbankorientierte Ansatz am besten.

[1] Der Ansatz wurde völlig neu entwickelt, stellt also keine Überarbeitung des ursprünglichen Ansatzes dar.

7.2.2 Praktische Handhabung

Zur Bewertung der Handhabung der Ansätze in der Praxis wurden diese hinsichtlich ihrer Bedienbarkeit, Fehleranfälligkeit und Integrationsmöglichkeit in MATLAB/Simulink bewertet.

Modellbasierter Ansatz Neben der geringen Skalierbarkeit erweist sich auch die Handhabung des modellbasierten Ansatzes in der vorgestellten Form insbesondere aus industrieller Sicht als zu aufwändig, um in der implementierten Form praktisch gewinnbringend eingesetzt werden zu können. Auch ist teilweise Spezialwissen über verschiedene Rahmenwerke (z. B. Xtext, EMF, GMF, ATL) sowie die integrierte Entwicklungsumgebung Eclipse erforderlich, insbesondere dann, wenn der Anwender selbst Analysen damit implementieren möchte. Dieses Spezialwissen gehört im Umfeld der Entwicklung eingebetteter Software mit MATLAB/Simulink normalerweise nicht zum Grundwissen von Ingenieuren, die das Analyserahmenwerk einsetzen würden. Auch den meisten Softwareentwicklern dürften diese Rahmenwerke häufig unbekannt sein, sodass eine gründliche Einarbeitung in die Materie nötig ist, bevor das Konzept produktiv angewendet werden kann. Um den Entwicklungsprozess effizient zu unterstützen, sollten daher der Konfigurationsaufwand der Modelltransformationen in Eclipse reduziert oder die Integration der Analysen in vorhandene Werkzeuge ermöglicht werden. Der Konfigurationsaufwand betrifft beispielsweise Einstellungen von Pfaden zum Simulink-Modell, EMF-Modellen und Metamodellen sowie die Angabe von Laufzeitparametern, Transformationsreihenfolge und Zwischenergebnissen in Eclipse. Durch die Verwendung von Skripten für die Zusammenschaltung von Modelltransformationen (z. B. Ant [6]) lässt sich der Anwender zwar leicht von diesem Aufwand entlasten, sofern man sich auf die vorimplementierten Analysen beschränkt. Da man aber den Anwender auch in die Lage versetzen möchte, selbst über die Zusammenschaltung von Modelltransformationen Analysen zu erstellen, helfen vorimplementierte, statisch konfigurierte Skripte nur begrenzt weiter.

Der modellbasierte Ansatz lässt sich, wie in Abschnitt 5.6 beschrieben, durch die Interaktion mit einem Transformationsserver in MATLAB/Simulink integrieren. Für neu implementierte Analysen hingegen muss der Benutzer in jedem Fall selbst entsprechende Erweiterungen am Transformationsserver bzw. den JAR-Archiven und an den Kontextmenüs in MATLAB/Simulink um die neu erstellten Analysen vornehmen.

Eine weitere Schwierigkeit ergibt sich beim modellbasierten Ansatz aus der Verwendung des Parsers für die Umwandlung eines Simulink-Modells in ein EMF-Modell bzw. des Generators für die Visualisierung eines Analyseergebnisses in MATLAB/Simulink. Dadurch, dass für das Auslesen und Generieren keine Funktionen von MATLAB/Simulink verwendet werden, kann weder validiert werden, ob eine generierte MDL-Datei korrekt ist, noch kann garantiert werden, dass der Parser für jedes beliebige Simulink-Modell lauffähig ist. Bereits leichte Unterschiede in der Reihenfolge von Blockparametern führten beispielsweise bei Tests zu nicht darstellbaren Simulink-Modellen. Grundsätzlich kann es, z. B. verursacht durch verschiedene MATLAB-Versionen, zu Änderungen der MDL-Datei kommen, sodass der Parser angepasst werden muss.

Von den vorgestellten Fallstudien konnten die Signalverfolgung (Abschnitt 7.1.1), die Auflösung der Busstruktur (Abschnitt 7.1.2), die Schnittstellenidentifikation (Abschnitt 7.1.4) sowie die Markierung auskonfigurierter Funktionalität (Abschnitt 7.1.5) voll in MATLAB/Simulink integriert werden, d. h. das Initiieren der Analysen sowie die Visualisierung sind aus MATLAB/Simulink möglich.

Datenbankorientierter Ansatz Der datenbankorientierte Ansatz ist in verschiedener Hinsicht besser zu handhaben.

Zunächst einmal ist er mit Java in einer weit verbreiteten, ausgereiften Programmiersprache implementiert, wodurch zumindest Entwickler mit softwaretechnischem Hintergrundwissen in der Lage sein sollten, selbst Analysen damit zu implementieren. Für die erstellten Analyse- und Visualisierungsbibliotheken ist eine Einarbeitung nötig, nicht jedoch für das grundlegende objektorientierte Programmierparadigma. Außerdem existiert im Gegensatz zum modellbasierten Ansatz ein großes und stetig wachsendes Repertoire an Standard-Java-Funktionsbibliotheken, die wiederverwendet werden können, wodurch die Effizienz voraussichtlich erhöht und die Fehleranfälligkeit reduziert werden kann.

Gegenüber dem ursprünglichen, modellbasierten Ansatz aus Kapitel 5 weist der datenbankorientierte Ansatz zudem den Vorteil auf, dass dieser sich besser in MATLAB/Simulink integrieren lässt und daher keine unnötigen Informationen „mitschleppen“ muss, die dafür nach Bedarf ausgelesen werden können. Dieser Aspekt wird beispielsweise für die Visualisierung von Analyseergebnissen durch die Übernahme von Positionierungsinformationen aus dem analysierten Modell ausgenutzt. Außerdem sinkt die Fehleranfälligkeit, da die Simulink-Modelle durch Verwendung von MATLAB-Funktionalität weder geparst (beim Import) noch generiert (bei der Visualisierung) werden müssen, wodurch sichergestellt ist, dass das ursprüngliche Modell eingelesen und das resultierende Modell dargestellt werden kann.

Durch die gute Integrierbarkeit von Java-Funktionalität und relationalen Datenbanken in MATLAB/Simulink ist außerdem kein Werkzeugwechsel für Analysen von Simulink-Modellen erforderlich. Zudem können Anwender Analysen in Java implementieren oder direkt in MATLAB/Simulink[1]. Je nach Komplexitätsgrad einer Analyse bietet sich erstere oder letztere Alternative an. Die Implementierung in Java dient insbesondere der Realisierung komplexer Analysen, während sich die Direktabfrage per SQL aus MATLAB/Simulink für einfache Informationsabfragen anbietet, z. B. für das Abfragen von Subsystemen, die mit bestimmten Annotationen versehen sind.

Als nachteilig erweist sich die Tatsache, dass der Ansatz nicht in andere Werkzeuge integriert wurde. So ist die Annotation von Anforderungen beispielsweise nur über das Verwaltungs- und Analysewerkzeug möglich (vgl. Abschnitt 6.5.2).

[1] Im Rahmen dieser Dissertation wurde stets der Java-Ansatz gewählt. Jedoch bietet MATLAB/Simulink einen Direktzugriff auf relationale Datenbanken über die Database Toolbox an.

7.2.3 Erweiterbarkeit, Änderbarkeit und Generalisierbarkeit

Unter Verwendung der implementierten Funktionsbibliothek für Modellanalysen kann der Java-versierte Anwender mithilfe des datenbankorientierten Ansatzes weitere Analysen implementieren, ohne sich um Spezialfälle kümmern zu müssen, da die Funktionen davon bereits abstrahieren. Dadurch lässt sich das Rahmenwerk nach Einschätzung der Entwickler auch gut um zusätzliche Analysefunktionalität erweitern.

Neue Java-Funktionalität lässt sich über eine Aktualisierung der JAR-Archive in MATLAB/Simulink integrieren. Änderungen an der Datenbank erfordern hingegen ggf. eine Modifikation der Datenbankzugriffskomponente. Aufgrund der Tatsache, dass die gesamte Interaktion mit der Datenbank über die Funktionen dieser Komponente erfolgt, lassen sich Änderungen jedoch auf diese beschränken und schlagen nicht notwendigerweise auf weitere Komponenten durch.

Die Ausweitung des Integrations- und Analysekonzepts auf andere Werkzeuge oder gar auf andere Entwicklungsparadigmen dürfte schwierig sein, wenn man umfangreiche Änderungen am Rahmenwerk vermeiden möchte. Verwendet man beispielsweise andere blockschaltbildorientierte Modellierungswerkzeuge anstelle von MATLAB/Simulink, so können zwar das Datenmodell und somit auch die Java-Funktionen weiter verwendet werden, jedoch muss der Import angepasst werden, da dieser aktuell über in ein MATLAB-Skript integrierte Java-Funktionen erfolgt. Würde man hingegen auch das Modellierungsparadigma ändern, so würde das Datenmodell und damit das Fundament des Ansatzes verändert und die Funktionalität des Rahmenwerks somit größtenteils unbrauchbar.

Dadurch, dass der ursprüngliche, modellbasierte Ansatz auf EMF-Modellen operiert, die alle Informationen aus dem Simulink-Modell beinhalten, können Analysen nach dem Import völlig unabhängig von den verwendeten Werkzeugen implementiert werden, während Analysen des datenbankorientierten Ansatzes evtl. auf Simulink-Funktionalität zurückgreifen müssen, um nicht importierte Daten nachzuladen. Insbesondere kann die Visualisierung in MATLAB/Simulink so ohne Zuhilfenahme des analysierten Originalmodells erfolgen.

7.3 Einschränkungen und Fazit

Insgesamt erweist sich der datenbankorientierte Ansatz für den Transfer eines Simulink-Modells in ein analysierbares Modell als der effizienteste der untersuchten Ansätze. Optimierungsmaßnahmen (vgl. Abschnitt 7.2.1) konnten die Effizienz des modellbasierten Ansatzes zwar deutlich erhöhen, jedoch wurde die Effizienz des datenbankorientierten Ansatzes nicht erreicht. Im nächsten Schritt bietet es sich daher an, den Zusammenhang zwischen Größe, Struktur und Speicherbelegung während der Transformationen von EMF-Modellen zu untersuchen, um weiteres Optimierungspotenzial zu identifizieren, z. B. wie sich die Subsystemhierarchietiefe eines Simulink-Modells bei gleich bleibender Gesamtgröße auf die Laufzeiten auswirkt.

Die Werkzeugintegration der Ansätze erfolgte in MATLAB/Simulink für das datenbankorientierte Konzept über den Aufruf zugehöriger Java-Funktionen innerhalb von MATLAB-Skripten, die ihrerseits in Kontextmenüs integriert wurden und in JAR-Archiven abgelegt sind. Die Integration des modellbasierten Ansatzes in MATLAB/-Simulink wurde über den Transformationsserver aus Abschnitt 5.6 realisiert, wodurch sich die Anwendung als weniger komfortabel erweist. Einschränkend muss jedoch darauf hingewiesen werden, dass nach neueren Erkenntnissen auch die Integration in MATLAB/Simulink durch entsprechende JAR-Archive möglich wäre. Wie komplex sich dies gestaltet, wurde nicht untersucht.

Die Anwendung der Ansätze erfordert grundsätzlich eine tiefgründige Einarbeitung in den Aufbau der erstellten Rahmenwerke sowie in die zugrunde liegenden Werkzeuge. Dennoch ist davon auszugehen, dass der datenbankorientierte Ansatz komfortabler anzuwenden ist. Dies ist insbesondere der Tatsache geschuldet, dass er mit Java in einer ausgereiften und weit verbreiteten Programmiersprache implementiert wurde. Demgegenüber sind für den modellbasierten Ansatz umfangreiche Kenntnisse von Modelltransformationen in ATL und dem EMF notwendig, die wesentlich weniger verbreitet und auch bei Weitem nicht so ausgereift sind wie die Programmierung mit Java. Insbesondere der hybride Charakter der ATL dürfte Anwendern die Einarbeitung in dieses Analyseparadigma erschweren. Um diesen Aspekt verlässlich zu evaluieren, ist jedoch eine längerfristige, industrielle Studie erforderlich.

Auch ist der datenbankorientierte Ansatz durch die Verwendung von MATLAB-Funktionalität anstelle von externen Parsern und Generatoren weniger fehleranfällig als der modellbasierte.

Des Weiteren bietet die dem datenbankorientierten Ansatz zugrunde liegende Datenbank die Möglichkeit, einfache Anfragen direkt aus MATLAB/Simulink über die Database Toolbox per SQL an die Datenbank zu richten, nachdem ein Modell einmal importiert wurde. Inwiefern die Integration in weitere Werkzeuge möglich ist, wurde nicht untersucht. Die Tatsache, dass mit MySQL eine gängige, relationale Datenbank verwendet wurde, legt jedoch nahe, dass dies möglich ist.

8 Zusammenfassung und Ausblick

8.1 Zusammenfassung

Die vorgestellte Arbeit zeigt Lösungsansätze für die in der aktuellen Forschung und Entwicklung häufig thematisierte Komplexitätsbeherrschung eingebetteter Software durch Artefaktintegration sowie automatisierte Artefakt- und Konsistenzanalysen auf. Sie sehen vor, Artefakte in ein gemeinsames Datenmodell abzubilden und sie dort miteinander zu verknüpfen, zu analysieren und einheitlich zu dokumentieren. Dabei kommen unterschiedliche Paradigmen zur Anwendung. Während der erste Ansatz das gemeinsame Datenmodell mit einem Modell des Eclipse Modeling Frameworks realisiert und Analysen mit Hilfe von Modelltransformationen implementiert, verwendet der zweite Ansatz eine relationale Datenbank als Integrationsbasis für Artefakte und ihre Informationsanreicherung und ist – abgesehen vom Import aus MATLAB/Simulink– vollständig in Java implementiert. Des Weiteren vergleicht die Arbeit die Ansätze hinsichtlich ihrer Handhabung in der Praxis. Dazu gehören die Effizienz und Skalierbarkeit auf große Simulink-Modelle, sowie ihre Integrationsmöglichkeit in den industriellen Entwicklungsprozess und ihre Generalisierbarkeit. Dazu wurden verschiedene Fallstudien definiert und mit den Ansätzen umgesetzt. Dabei wurde der zeitliche Aufwand für die Erstellung der Artefaktrepräsentanten in der jeweiligen Datenbasis evaluiert, da diesem eine besondere Bedeutung hinsichtlich der darauf aufbauenden konkreten Analysen und der leichteren benutzerdefinierten Analyseimplementierung zukommt. Des Weiteren wurde sowohl bewertet, wie werkzeugabhängig die Ansätze sind, als auch, wie gut sie sich in selbige integrieren lassen. Letzteres ist im industriellen Kontext von besonderer Bedeutung, da Arbeitsabläufe durch häufige Werkzeugwechsel gestört werden.

Das Ergebnis der Dissertation ist, dass sich grundsätzlich zwar beide Ansätze für die Artefaktintegration und -analyse eignen, sich für die praktische Handhabung und die Integration in den Arbeitsfluss aber aufgrund der besseren Effizienz, Skalierbarkeit und Integrationsmöglichkeit in MATLAB/Simulink sowie der geringeren Fehleranfälligkeit der datenbankorientierte Ansatz als leistungsfähiger erweist. Insbesondere durch die Implementierung in Java ist zu erwarten, dass sich dieser Ansatz deutlich leichter in der Praxis etabliert, da Java im Gegensatz zu Modelltransformationssprachen eine gängige und weit verbreitete Programmiersprache ist, wodurch der Einarbeitungsaufwand geringer einzuschätzen ist als die Einarbeitung in die Praxis der Modelltransformationen.

Auch wenn die Effizienz des modellbasierten Ansatzes durch verschiedene Maßnahmen verbessert werden konnte, so ist es dennoch nicht gelungen, ähnliche Effizienzwerte wie beim datenbankorientierten Ansatz zu erreichen.

8.2 Ausblick

Auch der datenbankorientierte Ansatz besitzt weiteres Optimierungspotenzial. Dazu ist zu prüfen, ob Import und Analysen von Simulink-Modellen durch Verwendung einer Graphdatenbank beschleunigt werden können. Da diese Art von Datenbanken für Abfragen auf Graphstrukturen optimiert sind, entfiele auch das objektrelationale Mapping und die damit einhergehende Erstellung eines Java-Modells. Zu berücksichtigen ist dabei allerdings, dass andere zu integrierende Artefakte und Annotationen nicht notwendigerweise eine Graphstruktur aufweisen. Weiterhin ist die Verwendung einer objektorientierten anstelle einer relationalen Datenbank zu prüfen. Auch in diesem Fall entfiele das objektrelationale Mapping. Beide Optionen könnten aber eventuell die Werkzeugintegration erschweren. So kann die Database Toolbox von MATLAB/Simulink beispielsweise aktuell nur mit relationalen Datenbanken kommunizieren. Der Zugriff auf die Datenbank über reine MATLAB-Mittel wäre dann nicht so leicht möglich. Schließlich ist zu prüfen, ob analog zum optimierten, modellbasierten Ansatz durch Auslagerung der Signalstrukturerstellung die Generierung des Java-Modells beschleunigt werden kann.

Aus praktischer Sicht besonders relevant ist die bisher unberücksichtigt gebliebene Problematik von Artefaktmodifikationen in den Originalwerkzeugen nach deren Import in die Datenbasis, da eine Synchronisationsfunktion zwischen Original und dessen Repräsentation noch fehlt. Dies erfordert einen erneuten Import des Originals, wodurch der Arbeitsfluss gestört wird. Um dies zu vermeiden, sollte daher eine Aktualisierungsfunktionalität in die Rahmenwerke integriert werden, die unter Zuhilfenahme von Vergleichsmethoden oder -werkzeugen Unterschiede zwischen Artefaktversionen auf deren Repräsentation im Datenmodell überträgt. Die mit einem modifizierten Artefaktelement verknüpften Elemente anderer Artefakte könnten dabei zugleich zur Überprüfung markiert werden (z. B. durch eine Annotation), wodurch ein Beitrag zur Konsistenzprüfung geleistet würde.

Aktuell decken die vorgestellten Ansätze keine Entwurfsartefakte ab, die im Entwicklungsprozess des Industriepartners auf Seiten des Zulieferers angesiedelt sind. Die Erweiterung des Ansatzes auf dessen Artefakte würde die Durchgängigkeit und Rückverfolgbarkeit verbessern und könnte den Gesamtprozess beschleunigen, da diese dann ebenfalls in Analysen berücksichtigt werden könnten.

Auch besitzen beide Ansätze Erweiterungspotenzial im Hinblick auf Projekt- und Prozesssteuerung. So ist denkbar, die Rahmenwerke um Methoden für die Aufwandsabschätzung für Änderungsanträge zu ergänzen. Dazu können die bereits vorhandenen Annotationen und Cliquen verwendet werden.

Da in der Praxis Domänenexperten an den Artefakten arbeiten, die nicht notwendigerweise auch Programmierkenntnisse haben, ist es zudem sinnvoll, eine spezielle, domänenspezifische Sprache für die Definition von Analysen zur Verfügung zu stellen. Damit definierte Analysen müssten dann in Java bzw. Modelltransformationen übersetzt werden, um in den vorgestellten Rahmenwerken ausgeführt zu werden.

Schließlich wird eine industrielle Studie erforderlich sein, um ein detaillierteres Bild der Praxistauglichkeit zu bekommen. Dabei sollten neben der Prozessintegration der

Ansätze auch die benutzerdefinierte Spezifikation von Analysen sowie die Nutzbarkeit von Annotationen für Artefaktverknüpfungen untersucht werden. Insbesondere für letzteren Aspekt ist eine längerfristige Studie erforderlich, da mit zunehmender Anzahl von Annotationen immer mehr potenzielle Zusammenhänge zwischen Artefakten entstehen und der Nutzen des Vorgehens sich damit nur mittel- bis langfristig auszahlen kann.

Literaturverzeichnis

[1] Amelunxen, C., Königs, A., Rötschke, T. und Schürr, A.: *MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations*. In: Rensink, A. und Warmer, J. (Hrsg.): *Model Driven Architecture – Foundations and Applications*, Bd. 4066 d. Reihe *Lecture Notes in Computer Science (LNCS)*, S. 361–375. Springer Verlag, Springer Verlag, 2006.

[2] Amelunxen, C., Legros, E., Schürr, A. und Stürmer, I.: *Checking and Enforcement of Modeling Guidelines with Graph Transformations*. In: Schürr, A., Nagl, M. und Zündorf, A. (Hrsg.): *Applications of Graph Transformations with Industrial Relevance*, Bd. 5088 d. Reihe *Lecture Notes in Computer Science*, S. 313–328. Springer Berlin / Heidelberg, 2008, ISBN 978-3-540-89019-5.

[3] Andries, M. und Engels, G.: *A Hybrid Query Language for an Extended Entity-Relationship Model*. Journal of Visual Languages and Computing, 7(3):321–352, 1996.

[4] Andries, M., Engels, G., Habel, A., Hoffmann, B., Kreowski, H. J., Kuske, S., Plump, D., Schürr, A. und Taentzer, G.: *Graph Transformation for Specification and Programming*. Science of Computer Programming, 34(1):1–54, 1999, ISSN 0167-6423.

[5] Anquetil, N., Kulesza, U., Mitschke, R., Moreira, A., Royer, J. C., Rummler, A. und Sousa, A.: *A model-driven traceability framework for software product lines*. Software and Systems Modeling, 9(4):427–451, 2010, ISSN 1619-1366.

[6] Apache Software Foundation: *Apache Ant*. Online, letzter Aufruf: Mai 2013. `http://ant.apache.org/`.

[7] Atego™: *Artisan Studio*. Online, letzter Aufruf: Mai 2013. `http://www.atego.com/products/artisan-studio/`.

[8] AUTOSAR: *AUTOSAR AUTomotive Open System ARchitecture*. Online, letzter Aufruf: Mai 2013. `http://www.autosar.org/`.

[9] Aytekin, O.: *LINQ: Theorie und Praxis für Einsteiger*. Programmer's choice. Addison-Wesley, 2008, ISBN 978-3827326164.

[10] Balzert, H.: *Software-Entwicklung*. Lehrbuch der Software-Technik. Spektrum Akademischer Verlag, 2. Aufl., 2001, ISBN 3-8274-0480-0.

[11] Balzert, H.: *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering.* Spektrum Akademischer Verlag GmbH, 3. Aufl., 2009, ISBN 9783827422477.

[12] Berg, K., Bishop, J. und Muthig, D.: *Tracing Software Product Line Variability: From Problem to Solution Space.* In: *Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries (SAICSIT '05)*, S. 182–191. South African Institute for Computer Scientists and Information Technologists, 2005, ISBN 1-59593-258-5.

[13] Beuche, D.: *Modeling and Building Software Product Lines with pure::variants.* In: *12th International Software Product Line Conference (SPLC '08)*, S. 358, September 2008.

[14] Biehl, M., DeJiu, C. und Törngren, M.: *Integrating Safety Analysis into the Model-based Development Toolchain of Automotive Embedded Systems.* In: *Proceedings of the ACM SIGPLAN/SIGBED 2010 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '10)*, S. 125–132, New York, NY, USA, 2010. ACM, ISBN 978-1-60558-953-4.

[15] Boehm, B. W.: *Guidelines for Verifying and Validating Software Requirements and Design Specifications.* In: Samet, P. A. (Hrsg.): *Euro IFIP '79*, S. 711–719. North Holland, 1979.

[16] Botterweck, G., Lee, K. und Thiel, S.: *Automating Product Derivation in Software Product Line Engineering.* In: *Proceedings of Software Engineering 2009 (SE '09)*, Kaiserslautern, Germany, 2009.

[17] Botterweck, G., Pleuss, A., Dhungana, D., Polzer, A. und Kowalewski, S.: *EvoFM: Feature-driven Planning of Product-line Evolution.* In: *ICSE Workshop on Product Line Approaches in Software Engineering (PLEASE '10)*, S. 24–31, New York, USA, 2010. ACM Press, ISBN 978-1-60558-968-8.

[18] Botterweck, G., Pleuss, A., Polzer, A. und Kowalewski, S.: *Towards Feature-driven Planning of Product-Line Evolution.* In: *First International Workshop on Feature-Oriented Software Development (FOSD '09)*, S. 109–116, New York, USA, 2009. ACM, ISBN 978-1-60558-567-3.

[19] Botterweck, G., Polzer, A. und Kowalewski, S.: *Interactive Configuration of Embedded Systems Product Lines.* In: *International Workshop on Model-driven Approaches in Product Line Engineering (MAPLE '09)*, Bd. 557, S. 29–35, August 2009.

[20] Briand, L. C., Labiche, Y. und O'Sullivan, L. L.: *Impact Analysis and Change Management of UML Models.* In: *International Conference on Software Maintenance (ICSM '03)*, S. 256–265, 2003.

[21] Broy, M., Feilkas, M., Herrmannsdoerfer, M., Merenda, S. und Ratiu, D.: *Seamless Model-Based Development: From Isolated Tools to Integrated Model Engineering Environments*. Proceedings of the IEEE, 98(4):526–545, April 2010, ISSN 0018-9219.

[22] Bryllok, C.: *Datenbankgestütztes Variantenmanagement in der modellbasierten Software-Entwicklung*. Diplomarbeit, RWTH Aachen University, 2013. unveröffentlicht.

[23] Böckle, G., Knauber, P., Pohl, K. und Schmid, K.: *Software-Produktlinien: Methoden, Einführung und Praxis*. dpunkt.verlag, 2004, ISBN 3-89864-257-7.

[24] Cabot, J. und Teniente, E.: *Incremental integrity checking of UML/OCL conceptual schemas*. Journal of Systems and Software, 82(9):1459–1478, September 2009, ISSN 0164-1212.

[25] Clements, P. C. und Northrop, L. M.: *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley, 2002, ISBN 978-0201703320.

[26] Cmyrev, A., Noerenberg, R., Hopp, D. und Reissing, R.: *Consistency Checking of Feature Mapping Between Requirements and Test Artefacts*. In: Stjepandić, J., Rock, G. und Bil, C. (Hrsg.): *Concurrent Engineering Approaches for Sustainable Product Development in a Multi-Disciplinary Environment*, S. 121–132. Springer London, 2013, ISBN 978-1-4471-4425-0.

[27] Czarnecki, K. und Antkiewicz, M.: *Mapping Features to Models: A Template Approach Based on Superimposed Variants*. In: *ACM SIGSOFT/SIGPLAN International Conference on Generative Programming and Component Engineering (GPCE '05)*, S. 422–437. Springer, 2005.

[28] Czarnecki, K. und Eisenecker, U.: *Generative Programming: Methods, Tools, and Applications*. Addison Wesley, 2000, ISBN 978-0201309775.

[29] Czarnecki, K. und Helsen, S.: *Classification of Model Transformation Approaches*. In: *OOPSLA '03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[30] Dhungana, D., Grünbacher, P. und Rabiser, R.: *DecisionKing: A Flexible and Extensible Tool for Integrated Variability Modeling*. In: *1st International Workshop on Variability Modelling of Software-Intensive Systems*, S. 119–128, 2007.

[31] dSpace: *Target Link*. Online, letzter Aufruf: Mai 2013. `http://www.dspace.com/en/ltd/home/products/sw/pcgs/targetli.cfm`.

[32] Duhr, Y.: *Verknüpfung von Artefakten der Entwicklungsphasen in der modellbasierten Software-Produktlinienentwicklung*. Diplomarbeit, RWTH Aachen University / Daimler AG, 2010. unveröffentlicht.

[33] Dziobek, C., Loew, J., Przystas, W. und Weiland, J.: *Von Vielfalt und Variabilität.* Elektronik Automotive, 2:33–37, 2008.

[34] Dziobek, C., Ringler, T. und Wohlgemuth, F.: *Herausforderungen bei der modellbasierten Entwicklung verteilter Fahrzeugfunktionen in einer verteilten Entwicklungsorganisation.* In: *Modellbasierte Entwicklung eingebetteter Systeme VIII (MBEES '12)*, S. 1–10, 2012.

[35] Dziobek, C. und Weiland, J.: *Variantenmodellierung und -konfiguration eingebetteter automotive Software mit Simulink.* In: *Modellbasierte Entwicklung eingebetteter Systeme V (MBEES '09)*, S. 36–45, 2009.

[36] Eclipse Foundation: *ATL – Atlas Transformation Language.* Online, letzter Aufruf: Mai 2013. `http://www.eclipse.org/atl/`.

[37] Eclipse Foundation: *Eclipse.* Online, letzter Aufruf: Mai 2013. `http://www.eclipse.org/`.

[38] Eclipse Foundation: *Eclipse Modeling Project.* Online, letzter Aufruf: Mai 2013. `http://www.eclipse.org/modeling/`.

[39] Eclipse Foundation: *QVT Operational.* Online, letzter Aufruf: Mai 2013. `http://projects.eclipse.org/projects/modeling.mmt.qvt-oml`.

[40] Eclipse Foundation: *SWT – The Standard Widget Toolkit.* Online, letzter Aufruf: Mai 2013. `http://www.eclipse.org/swt/`.

[41] Esterel Technologies SA: *SCADE Suite.* Online, letzter Aufruf: Mai 2013. `http://www.esterel-technologies.com/products/scade-suite/`.

[42] Falkowski, K.: *Modelltransformationsansätze im Kontext Modellgetriebener Softwareentwicklung.* Diplomarbeit, Universität Koblenz-Landau, 2005. Online, letzter Aufruf: Juni 2013. `http://www.uni-koblenz.de/~ist/documents/falkowski2005da.pdf`.

[43] Farkas, T.: *Regelbasierte Konformitätsprüfung kollaborativer Artefakte.* Dissertation, Technische Universität Berlin, Fraunhofer Gesellschaft, 2011, ISBN 978-3-8396-0266-9.

[44] Farkas, T., Leicher, A., Röbig, H., Born, M., Klein, T. und Zander-Nowicka, J.: *Werkzeugübergreifende Konsistenzsicherung von Artefakten bei der Entwicklung softwarebasierter Systeme im Automobil.* In: *4. Workshop Automotive Software Engineering*, Dresden, Germany, 2006.

[45] Farkas, T. und Röbig, H.: *Automatisierte, werkzeugübergreifende Richtlinienprüfung zur Unterstützung des Automotive-Entwicklungsprozesses.* In: *Modellbasierte Entwicklung eingebetteter Systeme III (MBEES '07)*, S. 61–73, Jan. 2007.

[46] Fraunhofer-Institut für Offene Kommunikationssysteme FOKUS: *MESA – Metamodellierung zur Automatisierung.* Online, letzter Aufruf: Juni 2013. http://www.fokus.fraunhofer.de/de/motion/projekte/projekt_archiv/2007/mesa/index.html.

[47] Giese, H., Meyer, M. und Wagner, R.: *A prototype for guideline checking and model transformation in Matlab/Simulink.* Proceedings of the 4th International Fujaba Days 2006, S. 56–60, 2006.

[48] Gill, A.: *Introduction to the theory of finite-state machines.* McGraw-Hill Electronic Sciences Series. McGraw-Hill, 1962.

[49] Gleis, R.: *Datenbankgestützte Signalflussanalyse in Simulink-Modellen.* Bachelorarbeit, RWTH Aachen University, 2012. unveröffentlicht.

[50] Groher, I. und Völter, M.: *Expressing Feature-Based Variability in Structural Models.* In: *Workshop on Managing Variability for Software Product Lines*, 2007.

[51] Heckel, R.: *Graph Transformation in a Nutshell.* Electronic Notes in Theoretical Computer Science, 148(1):187–198, 2006, ISSN 1571-0661.

[52] Holtmann, J., Meyer, J. und Meyer, M.: *A Seamless Model-Based Development Process for Automotive Systems.* In: *Software Engineering 2011 – Workshopband (inkl. Doktorandensymposium)*, 2011.

[53] IBM-Corporation: *IBM Jazz.* Online, letzter Aufruf: Mai 2013. http://www-01.ibm.com/software/de/rational/jazz/.

[54] IBM-Corporation: *IBM Rational DOORS.* Online, letzter Aufruf: Mai 2013. http://www-01.ibm.com/software/awdtools/doors/.

[55] IBM-Corporation: *IBM Rhapsody.* Online, letzter Aufruf: Mai 2013. http://www-142.ibm.com/software/products/us/en/ratirhapfami/.

[56] Institute of Mathematics and Computer Science University of Latvia: *MOLA Tool Description.* Online, letzter Aufruf: Mai 2013. http://mola.mii.lu.lv/tool_description.html.

[57] International Organization for Standardization: *ISO 26262.* Online, letzter Aufruf: Mai 2013. http://www.iso.org/iso/home/news_index/news_archive/news.htm?refid=Ref1499.

[58] Jacobson, I., Griss, M. und Jonsson, P.: *Software Reuse: Architecture Process and Organization for Business Success.* ACM Press books. 1997, ISBN 978-0201924763.

[59] Jahnke, J. H., Schäfer, W., Wadsack, J. P. und Zündorf, A.: *Supporting iterations in exploratory database reengineering processes.* Science of Computer Programming, 45(2–3):99–136, 2002, ISSN 0167-6423.

[60] JBoss Community: *Hibernate*. Online, letzter Aufruf: Mai 2013. `http://www.hibernate.org/`.

[61] JGraph Ltd.: *jgraph*. Online, letzter Aufruf: Mai 2013. `http://www.jgraph.com/`.

[62] Joshi, A. und Heimdahl, M. P. E.: *Model-Based Safety Analysis of Simulink Models Using SCADE Design Verifier*. In: *Proceedings of the 24th international conference on Computer Safety, Reliability, and Security*, Bd. 3688, S. 122–135, 2005.

[63] Jouault, F., Allilaire, F., Bézivin, J. und Kurtev, I.: *ATL: A Model Transformation Tool*. Science of Computer Programming, 72:31–39, 2008, ISSN 01676423.

[64] Jouault, F. und Kurtev, I.: *Transforming Models with ATL*. In: Bruel, J. M. (Hrsg.): *Satellite Events at the MoDELS 2005 Conference*, Bd. 3844 d. Reihe *Lecture Notes in Computer Science*, S. 128–138. Springer Berlin / Heidelberg, 2006, ISBN 978-3-540-31780-7.

[65] Kalnins, A., Barzdins, J. und Celms, E.: *Model Transformation Language MOLA*. In: Aßmann, U., Aksit, M. und Rensink, A. (Hrsg.): *Model Driven Architecture*, Bd. 3599 d. Reihe *Lecture Notes in Computer Science*, S. 62–76. Springer Berlin Heidelberg, 2005, ISBN 978-3-540-28240-2.

[66] Kalnins, A., Celms, E. und Sostaks, A.: *Simple and Efficient Implementation of Pattern Matching in MOLA Tool*. In: *7th International Baltic Conference on Databases and Information Systems, 2006*, S. 159–167, 2006.

[67] Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E. und Peterson, A. S.: *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Techn. Ber., Carnegie-Mellon University, Software Engineering Institute, November 1990.

[68] Kiesel, N., Schürr, A. und Westfechtel, B.: *GRAS, a Graph-Oriented Database System for (Software) Engineering Applications*. In: *Proceeding of the Sixth International Workshop on Computer-Aided Software Engineering (CASE '93)*, S. 272–286, 1993.

[69] Kindler, E. und Wagner, R.: *Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios*. Techn. Ber. tr-ri-07-284, University of Paderborn, 2007.

[70] Kolovos, D. S., Paige, R. F. und Polack, F. A. C.: *On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages*. In: Abrial, J. R. und Glässer, U. (Hrsg.): *Rigorous Methods for Software Construction and Analysis*, Bd. 5115 d. Reihe *Lecture Notes in Computer Science*, S. 204–218. Springer Berlin / Heidelberg, 2009, ISBN 978-3-642-11446-5.

[71] Kolovos, Dimitris and Rose, Louis and García-Domínguez, Antonio and Paige, Richard: *The Epsilon Book*. Online, letzter Aufruf: Mai 2013. `http://www.eclipse.org/epsilon/doc/book/`.

[72] Kubica, S.: *Variantenmanagement modellbasierter Funktionssoftware mit Software-Produktlinien*. Dissertation, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2007.

[73] Körtgen, A. T.: *New Strategies to Resolve Inconsistencies between Models of Decoupled Tools*. In: Egyed, A., Lopez-Herrejon, R. E., Nuseibeh, B., Botterweck, G. und Hu, Z. (Hrsg.): *3rd Workshop on Living with Inconsistencies in Software Development*, Bd. 661, S. 21–31, September 2010.

[74] Küster, J. M. und Engels, G.: *Consistency Management Within Model-Based Object-Oriented Development of Components*. In: Boer, F. S., Bonsangue, M., Graf, S. und Roever, W. P. (Hrsg.): *Formal Methods for Components and Objects*, Bd. 3188 d. Reihe *Lecture Notes in Computer Science*, S. 157–176. Springer Berlin / Heidelberg, 2004, ISBN 978-3-540-22942-1.

[75] Lauenroth, K. und Pohl, K.: *Towards Automated Consistency Checks of Product Line Requirements Specifications*. In: *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*, S. 373–376, New York, USA, 2007. ACM, ISBN 978-1-59593-882-4.

[76] Lauenroth, K. und Pohl, K.: *Dynamic Consistency Checking of Domain Requirements in Product Line Engineering*. In: *International Requirements Engineering, 2008. RE '08. 16th IEEE*, S. 193–202, September 2008.

[77] Legros, E., Schäfer, W., Schürr, A. und Stürmer, I.: *MATE - A Model Analysis and Transformation Environment for MATLAB Simulink*. In: Giese, H., Karsai, G., Lee, E., Rumpe, B. und Schätz, B. (Hrsg.): *Model-Based Engineering of Embedded Real-Time Systems*, Bd. 6100 d. Reihe *Lecture Notes in Computer Science*, S. 323–328. Springer Berlin / Heidelberg, 2011, ISBN 978-3-642-16276-3.

[78] Lehrstuhl für Software Engineering – RWTH Aachen: *MontiCore*. Online, letzter Aufruf: Mai 2013. `http://www.monticore.org/`.

[79] Lero, The Irish Software Engineering Research Centre: *S2T2 Configurator*. Online, letzter Aufruf: Mai 2013. `http://download.lero.ie/spl/s2t2/`.

[80] Mader, R., Armengaud, E., Leitner, A., Kreiner, C., Bourrouilh, Q., Grießnig, G., Steger, C. und Weiß, R.: *Computer-Aided PHA, FTA and FMEA for Automotive Embedded Systems*. In: Flammini, F., Bologna, S. und Vittorini, V. (Hrsg.): *Computer Safety, Reliability, and Security*, Bd. 6894 d. Reihe *Lecture Notes in Computer Science*, S. 113–127. Springer Berlin / Heidelberg, 2011, ISBN 978-3-642-24269-4.

[81] Malgouyres, H. und Motet, G.: *A UML Model Consistency Verification Approach Based on Meta-modeling Formalization.* In: Haddad, H. (Hrsg.): *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC '06)*, S. 1804–1809. ACM, April 2006, ISBN 1-59593-108-2.

[82] McGregor, J. D., Northrop, L. M., Jarrad, S. und Pohl, K.: *Initiating Software Product Lines.* Software, IEEE, 19(4):24–27, 2002, ISSN 0740-7459.

[83] Mens, T. und Demeyer, S.: *Software Evolution.* Springer, 2008, ISBN 978-3540764397.

[84] Mens, T., Straeten, R. und D'Hondt, M.: *Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis.* In: Nierstrasz, O., Whittle, J., Harel, D. und Reggio, G. (Hrsg.): *Model Driven Engineering Languages and Systems*, Bd. 4199 d. Reihe *Lecture Notes in Computer Science*, S. 200–214. Springer Berlin / Heidelberg, 2006, ISBN 978-3-540-45772-5.

[85] Merschen, D., Duhr, Y., Hedenetz, B. und Kowalewski, S.: *Evolutionsunterstützung für komplexe, modellbasierte Software-Produktlinien.* In: *Embedded-Software-Engineering-Kongress (ESE)*, S. 65–71. ELEKTRONIKPRAXIS und MicroConsult, 2012, ISBN 978-3-8343-2407-8.

[86] Merschen, D., Duhr, Y., Ringler, T., Hedenetz, B. und Kowalewski, S.: *Model-Based Analysis of Design Artefacts Applying an Annotation Concept.* In: Jähnichen, S., Küpper, A. und Albayrak, S. (Hrsg.): *Software Engineering 2012 (SE '12)*, S. 169–180. Gesellschaft für Informatik e.V. (GI), 2012, ISBN 978-3-88579-292-5.

[87] Merschen, D., Gleis, R., Pott, J. und Kowalewski, S.: *Analysis of Simulink Models Using Databases and Model Transformations.* In: Machando, R. J., Maciel, R. S. P., Rubin, J. und Botterweck, G. (Hrsg.): *8th International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MoMPES '12)*, Lecture Notes in Computer Science, S. 69–84. Springer Berlin / Heidelberg, 2013, ISBN 978-3-642-38208-6.

[88] Merschen, D., Polzer, A., Botterweck, G. und Kowalewski, S.: *Experiences of Applying Model-Based Analysis to Support the Development of Automotive Software Product Lines.* In: *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, S. 141–150. ACM Press, 2011, ISBN 978-1-4503-0570-9.

[89] Merschen, D., Pott, J. und Kowalewski, S.: *Integration and Analysis of Design Artefacts in Embedded Software Development.* In: *The 1st IEEE International Workshop on Tools in Embedded Systems Design Process (TiP '12)*, S. 503–508. IEEE, 2012.

[90] Michailidis, A., Spieth, U., Hedenetz, B., Ringler, T. und Kowalewski, S.: *Test Front Loading in Early Stages of Automotive Software Development Based on*

AUTOSAR. In: *Design, Automation and Test in Europe (DATE) Conference*, S. 435–440. European Design and Automation Association, 2010.

[91] Microsoft Corporation: *.NET Framework*. Online, letzter Aufruf: Mai 2013. `http://msdn.microsoft.com/de-de/vstudio/aa496123`.

[92] MIRA Ltd.: *MISRAMotor Industry Software Reliability Association*. Online, letzter Aufruf: Mai 2013. `http://www.misra.org.uk/`.

[93] Model Engineering Solutions GmbH: *MXAM*. Online, letzter Aufruf: Mai 2013. `http://www.model-engineers.com/de/model-examiner.html`.

[94] Mougenot, A., Blanc, X. und Gervais, M. P.: *Inconsistency Detection in Distributed Model Driven Software Engineering Environments*. In: Egyed, A., Lopez-Herrejon, R. E., Nuseibeh, B., Botterweck, G. und Hu, Z. (Hrsg.): *3rd Workshop on Living with Inconsistencies in Software Development*, Bd. 661, S. 2–7, September 2010.

[95] Mulholland, P., Zdrahal, Z., Valášek, M., Sainter, P., Koss, M. und Trejtnar, L.: *Supporting the sharing and reuse of modelling and simulation design knowledge*. In: *9th International Conference of Concurrent Enterprising (ICE '03)*, Juni 2003.

[96] Müller, M., Hörmann, K., Dittmann, L. und Zimmer, J.: *Automotive SPICE in der Praxis: Interpretationshilfe für Anwender und Assessoren*. dpunkt.verlag, 2007, ISBN 978-3898644693.

[97] Nagl, M.: *Graph-Grammatiken: Theorie, Anwendungen, Implementierung*. Vieweg, 1979, ISBN 978-3-528-03338-5.

[98] Nentwich, C., Capra, L., Emmerich, W. und Finkelstein, A.: *xlinkit: A Consistency Checking and Smart Link Generation Service*. ACM Transactions on Internet Technology (TOIT), 2(2):151–185, Mai 2002, ISSN 1533-5399.

[99] Nolte, S.: *QVT- Relations Language: Modellierung mit der Query Views Transformation*. Springer, 2009, ISBN 978-3540921707.

[100] Nöhrer, A. und Egyed, A.: *Utilizing the Relationships Between Inconsistencies for more Effective Inconsistency Resolution*. In: Egyed, A., Lopez-Herrejon, R. E., Nuseibeh, B., Botterweck, G. und Hu, Z. (Hrsg.): *3rd Workshop on Living with Inconsistencies in Software Development*, Bd. 661, S. 39–43, September 2010.

[101] Object Management Group: *Object Constraint Language (OCL)*. Online, letzter Aufruf: Mai 2013. `http://www.omg.org/spec/OCL/`.

[102] Object Management Group: *OMG*. Online, letzter Aufruf: Mai 2013. `http://www.omg.org/`.

[103] Object Management Group: *OMG's Meta Object Facility (MOF)*. Online, letzter Aufruf: Mai 2013. http://www.omg.org/mof/.

[104] Object Management Group: *Requirements Interchange Format (ReqIF)*. Online, letzter Aufruf: Mai 2013. http://www.omg.org/spec/ReqIF/.

[105] Object Management Group: *Systems Modeling Language*. Online, letzter Aufruf: Mai 2013. http://www.omgsysml.org/.

[106] Object Management Group: *Unified Modeling Language (UML)*. Online, letzter Aufruf: Mai 2013. http://www.uml.org/.

[107] Oracle Corporation: *Java Persistence API*. Online, letzter Aufruf: Mai 2013. http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html.

[108] Oracle Corporation: *Java SE Technologies - Database*. Online, letzter Aufruf: Mai 2013. http://www.oracle.com/technetwork/java/javase/jdbc/index.html.

[109] Oracle Corporation: *MySQL*. Online, letzter Aufruf: Mai 2013. http://dev.mysql.com/.

[110] Parnas, D. L.: *On the Design and Development of Program Families*. IEEE Transactions on Software Engineering, SE-2(1):1–9, März 1976, ISSN 0098-5589.

[111] Patzina, S. und Patzina, L.: *A Case Study Based Comparison of ATL and SDM*. In: Schürr, A., Varró, D. und Varró, G. (Hrsg.): *Applications of Graph Transformations with Industrial Relevance*, Bd. 7233 d. Reihe *Lecture Notes in Computer Science*, S. 210–221. Springer Berlin / Heidelberg, 2012, ISBN 978-3-642-34175-5.

[112] Petri, C. A.: *Kommunikation mit Automaten*. Dissertation, Rheinisch-Westfälisches Institut für Instrumentelle Mathematik an der Universität Bonn, 1962.

[113] PikeTec GmbH: *TPT - Test eingebetteter System*. Online, letzter Aufruf: Mai 2013. http://www.piketec.com/products/tpt.php.

[114] Pohl, K., Böckle, G. und Linden, F. van der: *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer-Verlag New York, Inc., 2005, ISBN 978-3540243724.

[115] Polzer, A., Botterweck, G., Wangerin, I. und Kowalewski, S.: *Variabilität im modellbasierten Engineering von eingebetteten Systemen*. In: *7. Workshop Automotive Software Engineering*, Bd. P-154 d. Reihe *Lecture Notes in Informatics (LNI)*, S. 2702–2719. Gesellschaft für Informatik (GI), 2009.

[116] Polzer, A., Kowalewski, S. und Botterweck, G.: *Applying Software Product Line Techniques in Model-Based Embedded Systems Engineering.* In: *Model-Based Methodologies for Pervasive and Embedded Software (MoMPES '09)*, S. 2–10, Mai 2009.

[117] Polzer, A., Merschen, D., Botterweck, G., Pleuss, A., Thomas, J., Hedenetz, B. und Kowalewski, S.: *Managing Complexity and Variability of a Model-based Embedded Software Product Line.* Innovations in Systems and Software Engineering (ISSE), 8(1):35–49, 2011, ISSN 1614-5054.

[118] Polzer, A., Merschen, D., Thomas, J., Hedenetz, B., Botterweck, G. und Kowalewski, S.: *View-Supported Rollout and Evolution of Model-Based ECU Applications.* In: Botterweck, G., Fernandes, J. a. M. und Lamb, L. (Hrsg.): *7th International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MoMPES '10)*, S. 37–44. ACM, September 2010, ISBN 978-1-4503-0123-7.

[119] Pott, J. D.: *Ein Annotationskonzept für die modellbasierte Analyse von Prozessartefakten.* Diplomarbeit, RWTH Aachen University, 2012. unveröffentlicht.

[120] Poulin, J. S.: *Measuring software reuse: principles, practices, and economic models.* Addison-Wesley, 1997, ISBN 978-0201634136.

[121] Pratt, T. W.: *Pair Grammars, Graph Languages and String-to-Graph Translations.* Journal of Computer and System Sciences, 5(6):560–595, Dezember 1971, ISSN 0022-0000.

[122] pure-systems GmbH: *pure::variants.* Online, letzter Aufruf: Mai 2013. `http://www.pure-systems.com/Variant_Management.49+M54a708de802.0.html`.

[123] Reder, A.: *Inconsistency Management Framework for Model-Based Development.* In: *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*, S. 1098–1101. ACM, 2011, ISBN 978-1-4503-0445-0.

[124] Reifer, D. J.: *Practical Software Reuse.* Wiley Series in Software Engineering Practice Series. John Wiley & Sons, 1997, ISBN 978-0471578536.

[125] Ricardo Inc.: *MINT.* Online, letzter Aufruf: Mai 2013. `http://www.mathworks.de/products/connections/product_detail/product_35611.html`.

[126] Riebisch, M.: *Supporting Evolutionary Development by Feature Models and Traceability Links.* In: *Proceedings of the 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, S. 370–377, 2004.

[127] Rumpe, B.: *Modellierung mit UML.* Xpert.press. Springer Berlin, 2. Aufl., 2011, ISBN 978-3-642-22412-6.

[128] Rumpe, B.: *Agile Modellierung mit UML.* Xpert.press. Springer Berlin, 2. Aufl., 2012, ISBN 3-540-20905-0.

[129] Rupp, C., Queins, S. und Zengler, B.: *UML 2 glasklar: Praxiswissen für die UML-Modellierung*. Hanser, 3. Aufl., 2007, ISBN 978-3446411180.

[130] Schaefer, I., Bettini, L., Damiani, F. und Tanzarella, N.: *Delta-Oriented Programming of Software Product Lines*. In: *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond (SPLC '10)*, S. 77–91, Berlin / Heidelberg, 2010. Springer-Verlag, ISBN 3-642-15578-2, 978-3-642-15578-9.

[131] Schmerler, S., Ringler, T., Hedenetz, B., Grüner, U., Wohlgemuth, F., Dziobek, C. und Lohrmann, P.: *Mit AUTOSAR zu einem integrierten und durchgängigen Entwicklungsprozess*. In: VDI (Hrsg.): *15. Internationaler Kongress mit Fachausstellung*, Baden Baden, 2011.

[132] Schäuffele, J. und Zurawka, T.: *Automotive Software Engineering: Grundlagen, Prozesse, Methoden und Werkzeuge effizient einsetzen*. Atz/Mtz-Fachbuch. Vieweg+Teubner Verlag, 4. Aufl., 2010, ISBN 978-3-8348-0364-1.

[133] Schürr, A.: *Introduction to PROGRESS, an Attribute Graph Grammar Based Specification Language*. In: Nagl, M. (Hrsg.): *Graph-Theoretic Concepts in Computer Science*, Bd. 411 d. Reihe *Lecture Notes in Computer Science*, S. 151–165. Springer Berlin / Heidelberg, 1990, ISBN 978-3-540-52292-8.

[134] Schürr, A.: *Specification of Graph Translators with Triple Graph Grammars*. In: Mayr, E. W., Schmidt, G. und Tinhofer, G. (Hrsg.): *Graph-Theoretic Concepts in Computer Science*, Bd. 903 d. Reihe *Lecture Notes in Computer Science*, S. 151–163. Springer Berlin / Heidelberg, 1995, ISBN 978-3-540-59071-2.

[135] Schürr, A., Winter, A. und Zündorf, A.: *The PROGRES Approach: Language and Environment*. In: Ehrig, H., Engels, G., Kreowski, H. J. und Rozenberg, G. (Hrsg.): *Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools*, Bd. 3, S. 487–550. World Scientific, 1999.

[136] Siy, H. und Mockus, A.: *Measuring Domain Engineering Effects on Software Change Cost*. In: *Sixth International Software Metrics Symposium*, S. 304–311, 1999.

[137] Sommerville, I.: *Software Engineering*. International Computer Science Series. Addison-Wesley, 2007, ISBN 978-0321313799.

[138] SourceForge: *Open CSV*. Online, letzter Aufruf: Mai 2013. `http://opencsv.sourceforge.net/`.

[139] Stachowiak, H.: *Allgemeine Modelltheorie*. Springer Verlag, University of California, 1973, ISBN 3211811060.

[140] Stahl, T., Völter, M., Efftinge, S. und Haase, A.: *Modellgetriebene Softwareentwicklung*. dpunkt.verlag, Heidelberg, 2. Aufl., 2007, ISBN 3-89864-448-8.

[141] Steinbauer, P., Valášek, M., Šika, Z. und Zdrahal, Z.: *Knowledge supported design and reuse of simulation models*. MATLAB, 2001.

[142] Stürmer, I., Dziobek, C. und Pohlheim, H.: *Modeling Guidelines and Model Analysis Tools in Embedded Automotive Software Development*. In: *Modellbasierte Entwicklung eingebetteter Systeme IV (MBEES '08)*, S. 28–39, 2008.

[143] Stürmer, I., Dörr, H., Giese, H., Kelter, U., Schürr, A. und Zündorf, A.: *Das MATE Projekt - visuelle Spezifikation von MATLAB Simulink/Stateflow Analysen und Transformationen*. In: Conrad, M., Giese, H., Rumpe, B. und Schätz, B. (Hrsg.): *Modellbasierte Entwicklung eingebetteter Systeme III (MBEES '07)*, Bd. 2007-1 d. Reihe *Informatik-Bericht*, S. 83–94. TU Braunschweig, Institut für Software Systems Engineering, Januar 2007.

[144] Taentzer, G.: *AGG: A Graph Transformation Environment for Modeling and Validation of Software*. In: Pfaltz, J. L., Nagl, M. und Böhlen, B. (Hrsg.): *Applications of Graph Transformations with Industrial Relevance*, Bd. 3062 d. Reihe *Lecture Notes in Computer Science*, S. 446–453. Springer Berlin / Heidelberg, 2004, ISBN 978-3-540-22120-3.

[145] The MathWorks, Inc.: *Automatic Code Generation - Simulink Coder*. Online, letzter Aufruf: Mai 2013. `http://www.mathworks.de/products/simulink-coder/`.

[146] The MathWorks, Inc.: *Database Toolbox*. Online, letzter Aufruf: Mai 2013. `http://www.mathworks.de/products/database/`.

[147] The MathWorks, Inc.: *Function Reference (MATLAB)*. Online, letzter Aufruf: Mai 2013. `http://www.mathworks.com/help/techdoc/ref/f16-6011.html`.

[148] The MathWorks, Inc.: *MathWorks Automotive Advisory Board (MAAB) – Automotive Industry Standards – MathWorks Deutschland*. Online, letzter Aufruf: Mai 2013. `http://www.mathworks.de/automotive/standards/maab.html`.

[149] The MathWorks, Inc.: *Simulink – Simulation and Model-Based Design*. Online, letzter Aufruf: Mai 2013. `http://www.mathworks.com/products/simulink/`.

[150] Thomas, J., Dziobek, C. und Hedenetz, B.: *Variability Management in the AUTOSAR-based Development of Applications for in-Vehicle Systems*. In: *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems (VaMoS '11)*, S. 137–140, New York, NY, USA, 2011. ACM, ISBN 978-1-4503-0570-9.

[151] Tip, F.: *A Survey of Program Slicing Techniques*. Techn. Ber., CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, 1994.

[152] Tracz, W.: *Confessions of a Used Program Salesman: Institutionalizing Software Reuse*. Addison-Wesley, 1995, ISBN 978-0-201-63369-6.

[153] TU Darmstadt: *MOFLON*. Online, letzter Aufruf: Mai 2013. `http://www.moflon.org/`.

[154] TU München: *ConQAT*. Online, letzter Aufruf: Mai 2013. `http://www.conqat.org/`.

[155] Universität Paderborn: *TGG-Interpreter*. Online, letzter Aufruf: Mai 2013. `http://www.cs.uni-paderborn.de/fachgebiete/fachgebiet-softwaretechnik/forschung/projekte/tgg-interpreter.html`.

[156] University of Paderborn: *The Fujaba Project*. Online, letzter Aufruf: Mai 2013. `http://www.fujaba.de`.

[157] Universität Siegen, Daimler AG, Universität Paderborn, TU Darmstadt, Universität Kassel, Model Engineering Solutions GmbH: *MATE*. Online, letzter Aufruf: Mai 2013. `http://pi.informatik.uni-siegen.de/Projekte/sidiff/simulinkMATE.php`.

[158] Varró, G., Friedl, K. und Varró, D.: *Graph Transformation in Relational Databases*. Electronic Notes in Theoretical Computer Science, 127(1):167–180, 2005, ISSN 1571-0661.

[159] Weiland, J. und Richter, E.: *Konfigurationsmanagement variantenreicher Simulink-Modelle*. In: Cremers, A. B., Manthey, R., Martini, P. und Steinhage, V. (Hrsg.): *GI Jahrestagung*, Bd. 68 d. Reihe *LNI*, S. 176–180, Bonn, 2005. Gesellschaft für Informatik e.V. (GI), ISBN 3-88579-397-0.

[160] Weisemöller, I., Klar, F. und Schürr, A.: *Development of Tool Extensions with MOFLON*. In: Giese, H., Karsai, G., Lee, E., Rumpe, B. und Schätz, B. (Hrsg.): *Model-Based Engineering of Embedded Real-Time Systems*, Bd. 6100 d. Reihe *Lecture Notes in Computer Science*, S. 337–343. Springer Berlin / Heidelberg, 2011, ISBN 978-3-642-16276-3.

[161] Weiss, D. M. und Lai, C. T. R.: *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999, ISBN 978-0-201-69438-3.

[162] Wiechowski, N.: *Konsistenzmanagement in der modellbasierten Software-Produktlinien-Entwicklung*. Diplomarbeit, RWTH Aachen University, 2013. unveröffentlicht.

[163] Wiechowski, N., Gerlitz, T., Merschen, D. und Kowalewski, S.: *Ein Ansatz zum merkmalsbasierten Konsistenzmanagement in der Produktlinienentwicklung*. In: Horbach, M. (Hrsg.): *INFORMATIK 2013 - Informatik angepasst an Mensch, Organisation und Umwelt*, Bd. P-220 d. Reihe *Lecture Notes in Informatics (LNI)*, S. 2502–2516. Köllen Druck+Verlag GmbH, Bonn, 2013, ISBN 978-3-88579-614-5.

[164] Wolf, T. und Dutoit, A. H.: *Supporting Traceability in Distributed Software Development Projects*. International Workshop on Distributed Software Development (DiSD '05), S. 111–124, 2005.

[165] Zdrahal, Z., Mulholland, P., Valášek, M., Sainter, P., Koss, M. und Trejtnar, L.: *A Toolkit and Methodology to Support the Collaborative Development and Reuse of Engineering Models*. In: *Database and Expert Systems Applications (DEXA '03)*, S. 856–865, September 2003, ISBN 3-540-40806-1.

Abkürzungsverzeichnis

API Programmierschnittstelle (engl. Application Programming Interface)

AST Abstrakter Syntaxbaum (engl. Abstract Syntax Tree)

ATL ATLAS Transformation Language [36]

AUTOSAR Autmotive Open System Architecture [8]

CR Änderungsantrag (engl.: Change Request)

CSV kommagetrennte Werte (engl. Comma-Separated-Values)

ECU Steuergerät (engl. Electronic Control Unit)

EMF Eclipse Modeling Framework [38]

EVL Eclipse Validation Language [70, 71]

FODA Feature-Oriented Domain Analysis [67]

GMF Graphical Modeling Framework [38]

GMP Graphical Modeling Project [38]

JPA Java Persistence API [107]

LOP Liste offener Punkte

MAAB MATLAB Automotive Advisory Board [148]

MOF Meta Object Facility [103]

OCL Object Constraint Language [101]

OEM Original Equipment Manufacturer

OMG Object Management Group [102]

QVT Query / Views / Transformations [39]

ReqIF Requirements Interchange Format [104]

SQL Structured (Standard) Query Language

SWT Standard Widget Toolkit [40]

SysML Systems Modeling Language [105]

TGG Tripel-Graph-Grammatik [134]

TPT Time Partition Testing [113]

UML Unified Modelling Language [106]

XMI XML Metadata Interchange

XML Extensible Markup Language

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:

`http://aib.informatik.rwth-aachen.de/`

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: `biblio@informatik.rwth-aachen.de`

2010-01 *	Fachgruppe Informatik: Jahresbericht 2010
2010-02	Daniel Neider, Christof Löding: Learning Visibly One-Counter Automata in Polynomial Time
2010-03	Holger Krahn: MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering
2010-04	René Wörzberger: Management dynamischer Geschäftsprozesse auf Basis statischer Prozessmanagementsysteme
2010-05	Daniel Retkowitz: Softwareunterstützung für adaptive eHome-Systeme
2010-06	Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Computing maximum reachability probabilities in Markovian timed automata
2010-07	George B. Mertzios: A New Intersection Model for Multitolerance Graphs, Hierarchy, and Efficient Algorithms
2010-08	Carsten Otto, Marc Brockschmidt, Christian von Essen, Jürgen Giesl: Automated Termination Analysis of Java Bytecode by Term Rewriting
2010-09	George B. Mertzios, Shmuel Zaks: The Structure of the Intersection of Tolerance and Cocomparability Graphs
2010-10	Peter Schneider-Kamp, Jürgen Giesl, Thomas Ströder, Alexander Serebrenik, René Thiemann: Automated Termination Analysis for Logic Programs with Cut
2010-11	Martin Zimmermann: Parametric LTL Games
2010-12	Thomas Ströder, Peter Schneider-Kamp, Jürgen Giesl: Dependency Triples for Improving Termination Analysis of Logic Programs with Cut
2010-13	Ashraf Armoush: Design Patterns for Safety-Critical Embedded Systems
2010-14	Michael Codish, Carsten Fuhs, Jürgen Giesl, Peter Schneider-Kamp: Lazy Abstraction for Size-Change Termination
2010-15	Marc Brockschmidt, Carsten Otto, Christian von Essen, Jürgen Giesl: Termination Graphs for Java Bytecode
2010-16	Christian Berger: Automating Acceptance Tests for Sensor- and Actuator-based Systems on the Example of Autonomous Vehicles
2010-17	Hans Grönniger: Systemmodell-basierte Definition objektbasierter Modellierungssprachen mit semantischen Variationspunkten

2010-18 Ibrahim Armaç: Personalisierte eHomes: Mobilität, Privatsphäre und Sicherheit

2010-19 Felix Reidl: Experimental Evaluation of an Independent Set Algorithm

2010-20 Wladimir Fridman, Christof Löding, Martin Zimmermann: Degrees of Lookahead in Context-free Infinite Games

2011-01 * Fachgruppe Informatik: Jahresbericht 2011

2011-02 Marc Brockschmidt, Carsten Otto, Jürgen Giesl: Modular Termination Proofs of Recursive Java Bytecode Programs by Term Rewriting

2011-03 Lars Noschinski, Fabian Emmes, Jürgen Giesl: A Dependency Pair Framework for Innermost Complexity Analysis of Term Rewrite Systems

2011-04 Christina Jansen, Jonathan Heinen, Joost-Pieter Katoen, Thomas Noll: A Local Greibach Normal Form for Hyperedge Replacement Grammars

2011-06 Johannes Lotz, Klaus Leppkes, and Uwe Naumann: dco/c++ - Derivative Code by Overloading in C++

2011-07 Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe: An Operational Semantics for Activity Diagrams using SMV

2011-08 Thomas Ströder, Fabian Emmes, Peter Schneider-Kamp, Jürgen Giesl, Carsten Fuhs: A Linear Operational Semantics for Termination and Complexity Analysis of ISO Prolog

2011-09 Markus Beckers, Johannes Lotz, Viktor Mosenkis, Uwe Naumann (Editors): Fifth SIAM Workshop on Combinatorial Scientific Computing

2011-10 Markus Beckers, Viktor Mosenkis, Michael Maier, Uwe Naumann: Adjoint Subgradient Calculation for McCormick Relaxations

2011-11 Nils Jansen, Erika Ábrahám, Jens Katelaan, Ralf Wimmer, Joost-Pieter Katoen, Bernd Becker: Hierarchical Counterexamples for Discrete-Time Markov Chains

2011-12 Ingo Felscher, Wolfgang Thomas: On Compositional Failure Detection in Structured Transition Systems

2011-13 Michael Förster, Uwe Naumann, Jean Utke: Toward Adjoint OpenMP

2011-14 Daniel Neider, Roman Rabinovich, Martin Zimmermann: Solving Muller Games via Safety Games

2011-16 Niloofar Safiran, Uwe Naumann: Toward Adjoint OpenFOAM

2011-17 Carsten Fuhs: SAT Encodings: From Constraint-Based Termination Analysis to Circuit Synthesis

2011-18 Kamal Barakat: Introducing Timers to pi-Calculus

2011-19 Marc Brockschmidt, Thomas Ströder, Carsten Otto, Jürgen Giesl: Automated Detection of Non-Termination and NullPointerExceptions for Java Bytecode

2011-24 Callum Corbett, Uwe Naumann, Alexander Mitsos: Demonstration of a Branch-and-Bound Algorithm for Global Optimization using McCormick Relaxations

2011-25 Callum Corbett, Michael Maier, Markus Beckers, Uwe Naumann, Amin Ghobeity, Alexander Mitsos: Compiler-Generated Subgradient Code for McCormick Relaxations

2011-26 Hongfei Fu: The Complexity of Deciding a Behavioural Pseudometric on Probabilistic Automata

2012-01 Fachgruppe Informatik: Annual Report 2012

2012-02 Thomas Heer: Controlling Development Processes

2012-03 Arne Haber, Jan Oliver Ringert, Bernhard Rumpe: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems

2012-04 Marcus Gelderie: Strategy Machines and their Complexity

2012-05 Thomas Ströder, Fabian Emmes, Jürgen Giesl, Peter Schneider-Kamp, and Carsten Fuhs: Automated Complexity Analysis for Prolog by Term Rewriting

2012-06 Marc Brockschmidt, Richard Musiol, Carsten Otto, Jürgen Giesl: Automated Termination Proofs for Java Programs with Cyclic Data

2012-07 André Egners, Björn Marschollek, and Ulrike Meyer: Hackers in Your Pocket: A Survey of Smartphone Security Across Platforms

2012-08 Hongfei Fu: Computing Game Metrics on Markov Decision Processes

2012-09 Dennis Guck, Tingting Han, Joost-Pieter Katoen, and Martin R. Neuhäußer: Quantitative Timed Analysis of Interactive Markov Chains

2012-10 Uwe Naumann and Johannes Lotz: Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Direct Solvers for Systems of Linear Equations

2012-12 Jürgen Giesl, Thomas Ströder, Peter Schneider-Kamp, Fabian Emmes, and Carsten Fuhs: Symbolic Evaluation Graphs and Term Rewriting — A General Methodology for Analyzing Logic Programs

2012-15 Uwe Naumann, Johannes Lotz, Klaus Leppkes, and Markus Towara: Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Solvers for Systems of Nonlinear Equations

2012-16 Georg Neugebauer and Ulrike Meyer: SMC-MuSe: A Framework for Secure Multi-Party Computation on MultiSets

2012-17 Viet Yen Nguyen: Trustworthy Spacecraft Design Using Formal Methods

2013-01 * Fachgruppe Informatik: Annual Report 2013

2013-02 Michael Reke: Modellbasierte Entwicklung automobiler Steuerungssysteme in Klein- und mittelständischen Unternehmen

2013-03 Markus Towara and Uwe Naumann: A Discrete Adjoint Model for OpenFOAM

2013-04 Max Sagebaum, Nicolas R. Gauger, Uwe Naumann, Johannes Lotz, and Klaus Leppkes: Algorithmic Differentiation of a Complex C++ Code with Underlying Libraries

2013-05 Andreas Rausch and Marc Sihling: Software & Systems Engineering Essentials 2013

2013-06 Marc Brockschmidt, Byron Cook, and Carsten Fuhs: Better termination proving through cooperation

2013-07 André Stollenwerk: Ein modellbasiertes Sicherheitskonzept für die extrakorporale Lungenunterstützung

2013-08 Sebastian Junges, Ulrich Loup, Florian Corzilius and Erika Ábrahám: On Gröbner Bases in the Context of Satisfiability-Modulo-Theories Solving over the Real Numbers

2013-10 Joost-Pieter Katoen, Thomas Noll, Thomas Santen, Dirk Seifert, and Hao Wu: Performance Analysis of Computing Servers using Stochastic Petri Nets and Markov Automata

2013-12 Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl: Alternating Runtime and Size Complexity Analysis of Integer Programs

2013-13 Michael Eggert, Roger Häußling, Martin Henze, Lars Hermerschmidt, René Hummen, Daniel Kerpen, Antonio Navarro Pérez, Bernhard Rumpe, Dirk Thißen, and Klaus Wehrle: SensorCloud: Towards the Interdisciplinary Development of a Trustworthy Platform for Globally Interconnected Sensors and Actuators

2013-19 Florian Schmidt, David Orlea, and Klaus Wehrle: Support for error tolerance in the Real-Time Transport Protocol

* These reports are only available as a printed version.
Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.