RWTH Aachen

Department of Computer Science
Technical Report

Ansatz zur variantenreichen und modellbasierten Entwicklung von eingebetteten Systemen unter Berücksichtigung regelungs- und softwaretechnischer Anforderungen

Andreas Polzer

ISSN 0935–3232 · Aachener Informatik-Berichte · AIB-2015-13

RWTH Aachen · Department of Computer Science · December 2015

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/

Ansatz zur variantenreichen und modellbasierten Entwicklung von eingebetteten Systemen unter Berücksichtigung regelungs- und softwaretechnischer Anforderungen

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
Andreas Polzer
aus Limburg an der Lahn

Berichter: Universitätsprofessor Dr.-Ing. Stefan Kowalewski
Universitätsprofessor Dr. rer. nat. Horst Lichter

Tag der mündlichen Prüfung: 11. Dezember 2014

Andreas Polzer
Lehrstuhl Informatik 11
polzer@embedded.rwth-aachen.de

Aachener Informatik Bericht AIB-2015-13

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

ISSN 0935-3232

Kurzfassung

Die modellbasierte Entwicklung ist ein verbreiteter Ansatz zur Handhabung steigender Komplexität von eingebetteten Systemen. Diese entsteht aufgrund vieler Varianten und eines großen Funktionsumfangs, die während der Entwicklung realisiert werden. Die eingesetzten Modellierungswerkzeuge sind nicht für die variantenreiche und domänenübergreifende Entwicklung von eingebetteten Systemen ausgelegt. Infolge dessen ist die Komplexität der Modelle trotz des Abstraktionsniveaus sehr hoch, da Techniken wie Sichten oder Produktlinienansätze nicht ausreichend unterstützt werden. Zusätzlich sind die Werkzeuge nur unzureichend miteinander verknüpft. Dies hat z. B. eine schlechte Nachverfolgbarkeit von Designentscheidungen zur Folge.

Ein weiteres Problem bei der Entwicklung eingebetteter Systeme ist die gemeinsame Entwicklung über verschiedene Domänen hinweg. Diese Entwicklungsdomänen wie beispielsweise Maschinenbau, Elektrotechnik und Informatik haben teilweise konträre Anforderungen und unterschiedliche Sprachen, in welchen die Lösungen für die spezifischen Probleme formuliert werden. Eine Entwicklung von eingebetteten Systemen erfordert daher einen integrieren Ansatz aller beteiligten Domänen.

In dieser Arbeit wird ein Konzept vorgestellt, das die Bedürfnisse und Anforderungen der verschiedenen Entwicklungsdomänen eines eingebetteten Systems berücksichtigt und die durchgängige Entwicklung einer Familie von eingebetteten Systemen unterstützt. Die eingesetzten Konzepte betrachten dabei die gesamte Entwicklung von eingebetteten Systemen. Der Schwerpunkt liegt auf der Weiterentwicklung von bestehenden Systemen.

Der hier entwickelte Ansatz richtet sich nach regelungstechnischen und softwaretechnischen Anforderungen. Zusätzlich wird einerseits ein hohes Maß an Wiederverwendbarkeit und andererseits eine hohe Flexibilität bei der Modellierung sichergestellt. Diese Aspekte sind im Hinblick auf Sensoren und Aktuatoren besonders wichtig, da diese bei eingebetteten Systemen einen großen Einfluß auf die Ergebnisse der Anwendungen haben.

Neben dem Einfluß der Hardware spielen auch die Anforderungen der Produktlinienentwicklung eine Rolle. Hierzu wird ein Konzept vorgestellt, das den Produktlinienansatz aus der klassischen Softwareentwicklung an die modellbasierte eingebettete Welt anpasst und den Entwickler bei der Entwicklung durch Sichten und Analysen unterstützt.

Abstract

Nowadays, model-based development is a common method to deal with the complexity and wide rages of functionality in embedded systems. Especially in automotive domain MATLAB/Simulink is often used to develop controller software for embedded systems. However the usually used tools are originally not designed to model a product-line of embedded systems applications. Therefore there is a high complexity of the designed models. Additionally different tools are used for different design artefacts like requirements and implementation model. These tools are not connected to each other's which causes inconsistencies.

Different domains like electrical engineers, control engineers and computer scientist are involved in designing embedded systems. The different domains use different domain specific languages and have different requirements.

This thesis introduces an approach considering requirements of different domains designing embedded systems. The main focus is the further model-based development of existing embedded applications, in particular the cooperation of control engineering and computer science. Both domains have partly different requirements. E. g. in computer science it is important to reuse software. However in control engineering there is a need for great flexibility in terms of external interface (actuators and sensors) and the according software parts since the actuators and sensor have critical influence on the performance of the controller. Hence both domains have conflicting requirements.

In the second part of this thesis an approach on software product-line is adopted to the model-based development process. Model-based transformations and analysis are used to support the development. The transformations use all design artefacts to achieve consistent models with respect derived variants. The result of the analysis are views. This views are implemented using standardized transformations provided by a framework. Therefore new analysis views can be created combining and adopting these transformations.

Danksagung

Diese Dissertation ist erst durch Hilfe und Unterstützung möglich gewesen. Daher danke ich als erstes Professor Dr.-Ing. Stefan Kowalewski, der mir die Möglichkeit zum Verfassen dieser Arbeit gegeben hat und mich bei der Forschung mit wichtigen Ratschlägen unterstützte. Ich danke auch Professor Dr. rer. net. Horst Lichter für seine Bereitschaft ein Gutachten meiner Dissertation zu verfassen.

Mein großer Dank gilt auch allen Kollegen des Lehrstuhls für Informatik 11, die mit einer guten Atmosphäre und aufrichtiger Kritik die Erstellung dieser Arbeit unterstützt und manchmal durch Kickerspiele auch für die nötige Zerstreuung gesorgt haben. Für eine gute Zusammenarbeit in Projekten und für die Diskussionen danke ich besonders Jacob Palczynski und Daniel Merschen.

Ich danke auch den Studenten, Afshan Aman, Philipp Fischer, Demitrius Haak, Sven Hendriks, Bo-Min Kang, Stefan Kockelkoren, Christian Steffens, Michel Schmitz und Iris Wangerin für die gute Zusammenarbeit sowie dem Carolo-Cup Team mit Yves Duhr, Philipp Fischer, Julian Krenge und Hugues Tchouankem für viele spannende Momente.

Die wichtigen Erkenntnisse im Projekt ZAMOMO wären ohne die Förderung des Bundesministeriums für Bildung und Forschung (BMBF) und die Zusammenarbeit mit Herrn Michael Reke nicht möglich gewesen. Für die gute und vertrauensvolle Zusammenarbeit bei der Forschung auf dem Gebiet der Variabilität danke ich Dr. Goetz Botterweck und den Ansprechpartern bei der Daimler AG Jacques Thomas und Dr. Bernd Hedenetz.

Zuletzt danke ich meiner Familie für die stete Unterstützung und meiner Frau Susanne für die schönen Jahre trotz einiger schwieriger Momente. Ihr will ich diese Arbeit widmen.

Andreas Polzer

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1.

Einleitung

Eingebettete Systeme durchdringen unser tägliches Leben in vielen Bereichen. Jeder kommt in der heutigen Zeit mehrmals täglich bewusst oder unbewusst mit eingebetteten Systemen in Berührung. Wir nutzen beispielsweise täglich die Dienste von mobilen Endgeräten, um Informationen zu verteilen oder abzurufen. Der Einfluss eingebetteter Geräte reicht von den Bereichen der Hausautomatisierung bis zu Fahrzeugen. Eine gesteuerte Anwendung ist z. B. die Temperaturregelung im Haus.

Zusätzlich zu den Funktionen, die der Anwender wahrnimmt, übernehmen eingebettete Systeme eine Vielzahl von Aufgaben im Alltag, die den Kompfort erhöhen, den Ressourcenverbrauch optimieren oder für unsere Sicherheit verantwortlich sind.

Eine weitere wichtige Anwendungsdomäne von eingebetteten Systemen sind automotive Applikationen. Die Hersteller von Automobilen gehen davon aus, dass bei neuen Fahrzeugen die meisten neuen Funktionen durch Software in eingebetteten Systemen realisiert werden. In allen Bereichen des Automobils werden eingebettete Systeme eingesetzt, um die Sicherheit und den Komfort zu erhöhen, aber auch den Ressourcenverbrauch zu senken.

Ein wichtiger Bereich für automotive Funktionen sind Fahrerassistenzsysteme. Mit diesen Systemen ist es möglich die Sicherheit zu erhöhen, indem wo möglich Fehler der Fahrer ausgeglichen oder zumindest die Auswirkungen eines Fehlers gemildert werden. Ein Beispiel für solche Systeme ist der Bremsassistent. Dieser erkennt, ob es sich bei einem Bremsmanöver um eine Notbremsung handelt und erhöht die Bremskraft, um eine optimale Verzögerung zu erreichen. Zusätzlich wird von einem zweiten Assistenten die Bremskraft am Rad geregelt, um die Bremsung in einem Bereich zwischen Gleit- und Reibreibung zu halten und somit den Bremsweg zu verkürzen. Noch komplexere Aufgaben, wie die Spurführung des Fahrzeugs werden teilweise schon von eingebetteten Systemen übernommen. Diese Systeme versuchen beispielsweise das Unter- oder Übersteuern zu verhindern. Dazu werden ständig Drehbeschleunigung, Geschwindigkeit und Lenkeinschlag gemessen. Falls nötig, werden einzelne Räder abgebremst und dadurch das Ausbrechen des Fahrzeugs verhindert.

Im Bereich des Fahrzeugkomforts werden Systeme eingesetzt, welche Regelungsaufgaben oder andere Funktionalitäten zur Verfügung stellen. Beispiele hierfür sind die

Temerpaturregelung im Fahrzeuginneren aber auch eine intelligente Innenraumbeleuchtung, die dem Benutzer einen komfortablen und sicheren Zugang zum Fahrzeug ermöglicht.

Neben Aspekten der Sicherheit und des Komforts werden eingebettete Systeme auch eingesetzt, um gesetzliche Vorgaben zu erfüllen. Im Bereich des Antriebs sind Regelungen der Einspritzmengen und Zündzeitpunkte notwendig, um gesetzliche Abgasgrenzwerte einzuhalten. Katalysatoren können ohne Motorsteuerungen nicht betrieben werden. Zusätzliche Systeme zur Verbrauchssenkung (z. B. Stop-Start-Systeme) sind in der Lage, den Kraftstoffverbrauch zu senken.

Auch vermeintlich einfache Anforderungen, wie z. B. die Startfähigkeit eines Autos über einen Zeitraum von 6 Wochen sicherzustellen, können heute oftmals nur über Überwachungen und entsprechende Regelung sichergestellt werden. Diese Regelung überwacht den Ladezustand der Batterie und stellt sicher, dass insgesamt im System nicht mehr Energie gebraucht als produziert wird. Tritt der Fall auf, dass mehr elektrische Energie gebraucht als erzeugt wird, werden verschiedene Maßnahmen ergiffen, z. B. die Erhöhung der Leerlaufdrehzahl oder Abschaltung von Verbrauchern, um ein Entladen der Batterie zu verhindern.

Die Funktionalität solcher Regelungsysteme wird durch den Benutzer häufig nicht wahrgenommen, hat aber vielfältigen Einfluss unter anderem auf den Kraftstoffverbrauch des Autos oder auf die Startfähigkeit. Eine wichtige Eigenschaft eingebetteter Systeme ist die Vernetzung mit anderen Systemen. Der Informationsaustausch erfolgt mit weiteren Systemen, um beispielsweise den Energieverbrauch festzustellen. Die Informationen werden dann genutzt, um durch Abschaltung von Verbrauchern den Energieverbrauch zu beeinflussen. Eingebettete Systeme durchdringen sichtbar und unsichtbar das alltägliche Leben und erhöhen so die Sicherheit und den Komfort bei alltäglichen Aufgaben.

1.1. Motivation

Die Anforderungen an eingebettete Software steigen stetig. Die Funktionalität und Qualität der Software bestimmen in immer stärkem Maße die Eigenschaften des Gesamtsystems. Zusätzlich werden immer geringere Fehlerraten und schnellere Entwicklungszyklen gefordert. In diesen Entwicklungszyklen müssen Funktionalität und Vernetzung immer stärker beachtet werden. Die Kosten der Software machen dadurch einen immer größeren Anteil an den Gesamtentwicklungskosten aus. Ein wichtiger Ansatz zur Beherrschung der steigenden Komplexität ist die modellbasierte Entwicklung.

Die modellbasierte Entwicklung ermöglicht Domänenexperten Funktionalität und Zusammenhänge in ihnen bekannten Sprachen auszudrücken. Im Bereich der Regelungsentwicklung gibt es das Werkzeug Matlab / Simulink (MaSi), das mit einer

wirkplanähnlichen Darstellung und einer Simulation eine einfache Implementierung von Reglern ermöglicht. Doch auch dort müssen Anforderungen aus der Informatik berücksichtigt werden.

Eine der wichtigsten Anforderungen ist die gute Wiederverwendbarkeit von existierender Software. Sie ist ein wichtiger Faktor, um Komplexität, Fehlerraten und Kosten von Software zu reduzieren. Der Produktlinienansatz ist ein Verfahren, das eine systematische Wiederverwendung von Software zum Ziel hat. In der Literatur gibt es Untersuchungen zur Integration eines Produktlinienansatzes in die modellbasierte Entwicklung [46]. Es zeigt sich jedoch, dass die Verfahren in der Praxis nicht gut einzusetzen sind, da weitere Abhängigkeiten die Komplexität so weit erhöhen, dass verfügbare Werkzeuge nicht einfach genutzt werden können.

Zusätzlich widersprechen sich Anforderungen aus den verschiedenen Domänen. Ein Beispiel hierfür ist die Anforderung an eine minimale Einschwingzeit aus der Regelungstechnik. Diese kann durch eine hochgenaue und schnelle Regelung ermöglicht werden, die einen hohen Bedarf an Rechenzeit hat. Dies widerspricht aber der Anforderung der Informatik an eingebettete Systeme, möglichst wenig Hardwareressourcen zu benötigen. In solchen Fällen ist es notwendig, die Berücksichtigung beider Anforderungen schon im modellbasierten Entwicklungsprozess abzubilden. Im Besonderen gilt dieser Zusammenhang für Sensoren und Aktuatoren, die einen großen Einfluss auf die realisierte Funktionalität haben.

Zentrales Element des Produktlinienansatzes ist es, die Softwarearchitektur so zu wählen, dass es sowohl einen gemeinsamen Kern an Funktionalität gibt, als auch definierte Schnittstellen, an denen Änderungen und Anpassungen vorgenommen werden können. Es existieren Arbeiten, die diesen Ansatz in die modellbasierte Entwicklung übertragen [46]. Mit Hilfe einer Konfiguration kann ein Produkt abgeleitet werden, welches die gewünschten Eigenschaften besitzt. Um einen solchen Ansatz zu implementieren ist es notwendig, den variablen und festen Teil der Funktionalität genau festzulegen. Dies ist speziell in kleineren Firmen kaum möglich. In diesen Firmen werden eingebetteten Systeme im Rahmen von Projekten entwickelt, welche auf den Kunden zugeschnitten sind. Es ist für diese Firmen meist nicht möglich, die Aufträge passend zur eigenen Produktfamilie auszusuchen. Ausschlaggebend für die Annahme eines Auftrags sind vorhandene Kompetenzen und Kapazitäten. Trotzdem ist es notwendig, in Projekten erzielte Ergebnisse wiederzuwerwenden.

Auch hier wird die modellbasierte Entwicklung eingesetzt. Die Modellbasierung ermöglicht es dabei, spezielle Sprachen aus verschiedenen Anwendungsdomänen zu verwenden. Neben der Wiederverwendung von Wissen, welches in Projekten erzeugt wurde, ist auch die Wiederverwendung innerhalb eines dokumentierten Prozesses wichtig. Innerhalb dieses Prozesses müssen verschiedene Prozessartefakte berücksichtigt werden.

1.2. Zielsetzung und Beitrag

Ziel dieser Arbeit ist es, bereits entwickelte Kompontenten in einem modellbasierten Prozess wiederzuverwenden und somit eine möglichst effiziente Entwicklung von eingebetteten Systemen zu ermöglichen. Diese Wiederverwendung soll alle Abstraktionsebenen des Implementationsmodells umfassen. Für eingebettete Systeme, bedeutet dies eine hardwarenahe Betrachtung der modellbasierten Entwicklung aber auch eine abstrakte Betrachtung auf Ebene der Modelle.

Dazu sollen Konzepte erarbeitet und umgesetzt werden, welche ein hohes Maß an Wiederverwendung ermöglichen. Hierzu wurden eine Vielzahl von Anwendungsfällen identifiziert. Diese werden in einer Analyse aus Sicht von verschiedenen Domänen und Anwendungsbereichen untersucht. Die Anforderungen an den Entwicklungsprozess werden hauptsächlich durch einen hohen Grad an Wiederverwendbarkeit bestimmt. Die Analyse der Anforderungen wird durch Definition und Bewertung von Anwendungsfällen durchgeführt. Die Analyse der Anwendungsfälle ergibt funktionale und nicht-funktionale Anforderungen, welche priorisiert und bewertet als Grundlage für die entwickleten Konzepte dienen. Die entwickelten Konzepte ermöglichen einen hohen Grad an Wiederverwendung durch Einführung eines Mechanismus, welcher hardwarenahe Zugriffe abstrahiert. Dabei werden die Anforderungen von kleinen bzw. mittelständischen Unternehmen, die eine große Flexibilität der implementierten Funktionalität auf der einen Seite und ein hohes Maß an Wiederverwendbarkeit auf der anderen Seite benötigen, beachtet.

Neben der Unterstützung bei der Implementierung der Schnittstellen wird im zweiten Teil dieser Arbeit ein Konzept zur Einführung von Produktlinienansätzen auf der Modellebene beschrieben. Dieses Konzept integriert alle am Entwicklungsprozess beteiligten Artefakte. Die Verwaltung aller Artefakte erfolgt zentral über ein Feature-Modell (FM). Dabei wird der Modellierer bei Analyse und Einrichtung des Produktlinienansatzes durch Modelltransformationen unterstützt. Diese sind mithilfe eines in dieser Arbeit entwickelten Rahmenwerks implementiert, das eine schnelle Umsetzung von Modelltransformationen erlaubt.

1.3. Umfeld

Die in dieser Arbeit vorgestellten Konzepte wurden im Rahmen zweier Projekte erarbeitet. Diese wurden nacheinander mit praktischer Begleitung von Industrieunternehmen durchgeführt. Ziel des ersten Projekts ZAMOMO (Zusammenarbeit modellbasierter Regelungsentwicklung mit modellbasierter Softwareentwicklung) war es, die Zusammenarbeit verschiedener Domänenexperten bei der Entwicklung eingebetteter Systeme zu verbessern. Die Projektpartner kamen aus den verschiedenen Domänen der Regelungsentwicklung (IRT - Institut für Regelungsentwicklung),

der Modellierung (AVR Detuschland GmbH), der eingebetteten Systementwicklung (VEMAC - Engieerdienstleister), der Anforderungsmodellierung (FIT - Fraunhovergesellschaft für Informationstechnologie) und der Informatik (Lehrstuhl für Informatik 11 - Software für eingebettete Systeme).

Das Arbeitspaket umfasste die Entwicklung einer einheitlichen Entwicklungsmethodik, welche die Anforderungen der Regelungstechnik, die Anforderungen eines kleinen Unternehmens und die Anforderungen der Informatik beachtet. Das entwickelte Konzept ist in Zusammenarbeit mit der VEMAC entstanden und wird dort eingesetzt.

Im zweiten Projekt wurden Konzepte der modellbasierten Entwicklung auf Modelleebene betrachtet. Die Anforderungen wurden in diesem Projekt durch die Firma Daimler formuliert, welche bei der Anwendungsentwicklung eine bessere Wiederverwendung anstrebt. In diesem Bereich wurden durch eine Analyse des Entwicklungsprozesses Anforderungen identifiziert und bei typischen Anwendungsfällen entsprechene Werkzeuge entwickelt, die die Anforderungen setzen. Der Modellierer wird durch Sichten und Modelltransformationen unterstützt.

Das Ergebnis umfasst eine neue Art der Modellierung, welche es ermöglicht, die einzelnen Teile des Modells wiederverwendbar zu implementieren. Neben den Modellierungsansätzen wurden neue Konzepte zur variantenreichen Entwicklung der zentalen Prozesselemente erarbeitet. Die Konzepte wurden mit Hilfe des Eclipse Modeling Frameworks umgesetzt und erprobt. Die Grundlage dieser Entwicklung sind Metamodelle und Transformationen, welche ein Analyse-Rahmenwerk bilden. Die Ergebnisse dieser Arbeit wurde von Modellierern bei Daimler evaluiert.

1.4. Aufbau der Arbeit

Die vorliegende Arbeit gliedert sich in drei Hauptteile. Der erste Teil umfasst die Grundlagen für eingebettete Systeme deren Entwicklungsprozesse und spezielle Anforderungen. Ein weiteres Basiskonzept ist die Produktlinienentwicklung und deren Ausprägung in der Softwareentwicklung. Der letzte Teil der Grundlagen beschäftigt sich mit Ansätzen der modellbasierten Entwicklung im Bereich der eingebetteten Systeme.

Im zweiten Teil dieser Arbeit wird auf die variantenreiche Entwicklung eingebetteter System in Bezug auf Hardware eingegangen. Nach einer Analyse der Anforderungen, die regelungstechnische und softwaretechnische Aspekte umfasst, wird ein Konzept zur Abstraktion von Hardwareschnittstellen und dessen Implementierung vorgestellt. Anschließend wird ein Konzept für die Analyse von Implementierungsmodellen gezeigt. Beginnend mit den Anforderungen an den Entwicklungsprozess und die Entwicklungsartefakte, zeigt das Konzept Unterstützungsmöglichkeiten in der variantenreichen modellbasierten Entwicklung. Das Ziel ist es, durch einen

modellbasierten Entwicklungsprozess, einen hohen Grad an Wiederverwendung von verschiedenen Artefakten zu erreichen.

Im letzten Teil dieser Arbeit werden verschiedene Fallstudien vorgestellt, welche die Anwendung der entwickelten Konzepte evaluieren. Dazu wurde ein Assistenzsystem (Einparkassistent) auf einem automotiven Versuchsträger implementiert. Die erarbeiteten Konzepte zur Unterstützung der variantenreichen, modellbasierten Entwicklung wurden gemeinsam mit Entwicklern der Firma Daimler evaluiert. Dazu wurde eine Analyse der Einsatzmöglichkeiten von modellbasierten Verfahren mittels des Eclipse Modeling Framework (EMF)s durchgeführt. Abschließend wird diese Arbeit zusammengefasst und mit einem Ausblick abgeschlossen.

Teil I.

Grundlagen

Kapitel 2.

Eingebettete Systeme

Ein eingebettetes System ist laut [29] ein System, welches in ein größeres, umfassendes Produkt eingebettet ist und normalerweise nicht direkt sichtbar für den Benutzer ist.

Solche Systeme übernehmen üblicherweise Steuerungs- und Regelungsaufgaben und können durch zwei große Anwendungsbereiche getrennt werden:

1. Einsatz als Produktionssystem
2. Einsatz als Automationssystem

Typische Anwendungen für eingebettete Systeme als Automatisierungssysteme sind in Kraftfahrzeugen, Flugzeugen oder Krankenassistenzsystemen zu finden. Anwendungen im Bereich der Produktionssysteme sind die Steuerung oder Regelung von Fabriken, die Regelung von chemischen Prozessen oder die Kontrolle von Warenströmen.

Die beiden Bereiche unterscheiden sich vor allem durch die Massenproduktion der Automationssysteme im Gegensatz zu den spezifischen Systemen der Produktionssysteme. Die Einsatzbereiche haben großen Einfluss auf die Anforderungen im Bezug der Ressourcen, Wartung und Betrieb. Daher unterscheiden sich die eingesetzten Technologien und Programmiersprachen stark. Während die Programmiersprachen im Bereich der Produktionssysteme standardisiert sind und von eigentlich allen Herstellern unterstützt werden, werden Automatisierungssysteme mit stark unterschiedlichen Programmiersprachen wie C, Assembler oder modellbasierten Verfahren entwickelt. Trotz dieser Unterschiede gibt es wichtige Anforderungen, die von eingebetteten Systemen beider Bereiche eingehalten werden müssen.

2.1. Anforderungen eingebetteter Systeme

Ein wichtiger Teil der Anforderungen eingebetteter Systeme hat nichts mit der Funktion des Systems zu tun. Diese Anforderungen nennt man entsprechend nichtfunktionale Anforderung (NFA). Die NFA sind Qualitäten, die durch den Entwicklungsprozess eingehalten werden.

Für eingebettete Systeme ist *Zuverlässigkeit* eine wichtige Qualität. Das Fehlen von Zuverlässigkeit kann z. B. bei Produktionssystemen großen wirtschaftlichen Schaden verursachen. Dies gilt auch für Automatisierungssysteme, deren große Anzahl und eingeschränkte Wartbarkeit eine Korrektur von Fehlern oftmals teuer macht.

Fehler in eingebetteten Systemen müssen vor allem beim sicherheitsrelevanten Einsatz betrachtet werden. Daher sind Sicherheitsbetrachtungen und -mechanismen bei der Entwicklung von sicherheitskritischen Systemen von großer Wichtigkeit. Beispiele solcher Systeme finden sich in der chemischen Industrie. Diese setzt sicherheitskritische eingebettete Systeme ein, um gefährliche Prozesse zu steuern. Im Bereich der Automatisierungssysteme werden sicherheitskritische Funktionen z. B. Fahrerassistenzsysteme im Kraftfahrzeug umgesetzt.

Ein wichtiger Teil der Anwendungen von eingebetteten Systemen entfällt auf Steuerungs- und Regelungsaufgaben. Bei diesen Anwendungen ist die *Einhaltung von Reaktionszeiten* ein wichtiges Kriterium, um Zuverlässigkeit, Sicherheit und Stabilität zu erreichen. Die Reaktionszeiten werden sowohl als harte Echtzeit- als auch als weiche Echtzeitanforderung an eingebettete Systeme bzw. den Regler gestellt.

Zusätzlich müssen eingebettete Systeme Anforderungen an Ressourcen wie Kosten, Energieverbrauch beachten. Diese stehen oftmals den Anforderungen wie Sicherheit oder Zuverlässigkeit entgegen. Beim Design eingebetteter Systeme ist daher eine Analyse der funktionalen als auch NFA und eine entsprechende Umsetzung wichtig. Ein strukturiertes Vorgehen zur Durchführung von Analysen, dem Ableiten von Designentscheidungen und deren Umsetzung bieten Entwicklungsprozesse und Entwicklungsmethodiken. In der vorliegenden Arbeit wird der modellbasierte Ansatz verwendet, um die funktionalen und NFA umzusetzen.

Kapitel 3.

Modellbasierte Entwicklung

Die modellbasierte Entwicklung basiert auf dem Prinzip, Informationen über das zu entwickelnde System in abstrakten Vorbildern (Modellen) zu beschreiben, diese in Beziehung zu setzen und weiter zu verfeinern. Ziel der Abstraktion ist es, nur solche Informationen zur Verfügung zu stellen, die zur Entwicklung notwendig sind. Die angesprochene Verfeinerung der Modelle dient dazu, eine angepasste Abstraktion zu erreichen. Dabei ergänzt man die Modelle mit Informationen. Dies geschieht entweder direkt im Modell oder in Form von weiteren Modellen, die eingebunden werden.

Die Literatur setzt die modellbasierte Entwicklung mit der modellgetriebenen Entwicklung gleich. Die beiden Begriffe unterscheiden sich leicht. Wenn Modelle der zentrale Bestandteil des Entwicklungsprozesses sind, spricht man von der modellbasierten Entwicklung. Von modellgetriebener Entwicklung spricht man, wenn Modelle die treibende Kraft sind. Dies führt nicht zu einer klaren Unterscheidung. Aus diesem Grund werden in dieser Arbeit die beiden Begriffe als Synonyme benutzt. Ein Modell ist nach [42] durch drei zentrale Merkmale gekennzeichnet ist:

1. **Abbildung:** Ein Modell ist ein Abbild von etwas, also eine Repräsentation natürlicher oder künstlicher Originale. Diese Repräsentation kann ein Modell eines Modells sein.

2. **Verkürzung:** Ein Modell zeigt nicht alle Eigenschaften des repräsentierten Originals. Es hat zum Ziel, nur die für den Zweck relevanten Informationen darzustellen.

3. **Pragmatismus:** Ein Modell ist dem repräsentierten Original nicht automatisch zugeordnet. Die Verknüpfung zwischen Original und Modell erfolgt durch den Modellierer.

Dies bedeutet, „Modelle helfen uns, große und komplexe Probleme zu verstehen und zu analysieren“ [26]. Dazu nutzen die Modelle Abstraktionen aus, welche eine Reduktion bestimmter Merkmale vornehmen und auf Merkmale fokusiert, die für den Anwendungsbereich wesentlich sind. Neben der Abstraktion wird die Darstellung von Modellen genutzt, um ein besseres Problemverständnis zu schaffen.

In der modellbasierten Entwicklung werden auf formalen Sprachen basierende, formale Modelle verwendet. Die formalen Sprachen und Modelle zeichnen sich durch eine eindeutige Definition von Syntax und Semantik aus. Diese Eindeutigkeit schließt Nichtdeterminismus aus und ermöglicht es, dass alle Modellierer zur gleichen Deutung der Modelle kommen.

Einen Weg zur Festlegung der Semantik von Modellen ist die Definition mittels Metamodellen. Ein Metamodell legt die Spezifikationsprache von Modellen fest. Dabei eignen Metamodelle sich gut für die Beschreibung der Semantik, da für den Modellierer das gleiche Konzept (ein Modell) eingesetzt wird. Hierzu müssen Modellierer die Semantik der Metamodelle erfassen, um anschließend die Semantik unterschiedlicher Modelle zu verstehen. Eine Modellierungssprache, die die Semantik und Syntax definiert und in dieser Arbeit verwendet wird ist die Unified Modeling Language (UML) [33].

Die Grundlage der UML wird definiert durch die Meta Object Facility (MOF) [32]. Die MOF definiert eine Sprache für Metamodelle. Diese besteht aus einer Menge von Elementen (bzw. deren Semantik und Syntax), welche zur Modellierung von Metamodellen verwendet werden.

Die Syntax und Semantik sind hierarchisch in vier Metaebenen spezifiziert, wobei jedes Modell einer Metaebene eine Instanz des jeweils abstrakteren Modells ist. Zur abstraktesten Metaebene gibt es kein Metamodell. Die dort verwendete Syntax und Semantik wird beschrieben. Die Metaebenen sind wie folgt definiert:

- Metaebene 3: Meta-Metamodell. Auf dieser Ebene werden die Syntax und die Semantik von Metamodellen definiert.
- Metaebene 2: Metamodelle. Auf dieser Ebene werden die Metamodelle mit Hilfe von Instanzen der in M3 definierten Elemente beschrieben. Hier werden die Syntax und Semantik der entwicklungsspezifischen Modelle festgelegt.
- Metaebene 1: Innerhalb dieser Ebene werden die Modelle definiert. Diese entsprechen den Metamodellen aus M2 und sind Abbilder der realen Systeme und entsprechen dem Modellbegriff.
- Metaebene 0: Auf dieser Ebene werden die zu modellierten Systeme angesiedelt. Aus dieser Ebene hat keine Abstraktion bezüglich des Modells aus M1 stattgefunden.

Neben der Abstraktion bietet die modellbasierte Entwicklung die Möglichkeit, die Darstellung an verschiedene Domänen anzupassen. Eine solche Anpassung wird domänenspezifische Sprache (DSL) genannt und für eine Anwendungsdomäne entworfen und implementiert. Der Begriff Anwendungsdomäne bezeichnet ein definiertes, abgrenzbares Gebiet, z. B. Regelungstechnik oder Softwaretechnik. Domänen sind

nicht überschneidungsfrei und eindeutig. Dies hat zur Folge, dass gleiche Begriffe in verschiedenen Domänen unterschiedliche Bedeutung und Verwendung haben können.

DSLs werden eingesetzt, um Entwicklern einer Domäne eine Sprache zur Verfügung zu stellen, deren Abstraktionsniveau, Notation und Semantik möglichst der in der Domäne gebräuchlichen Sprache entsprechen. Dadurch kann ein Domäneningenieur die bekannten Konzepte der Domäne benutzen und so eine Problembeschreibung und Problemlösung beschreiben.

Die modellbasierte Entwicklung bietet mit dem vorgestellten Metamodellansatz eine Möglichkeit, Modelle für DSLs entwickeln zu können und somit die Entwicklung in verschiedenen Domänen zu erleichtern. Diese domänenspezifischen Modelle können im Allgemeinen für einen Domänenexperten kompakter, besser lesbar und verständlicher sein. Die Metamodelle legen die Modellelemente und deren Syntax und Semantik entsprechend der Domäne fest. Durch die gemeinsame Basis der Metamodelle ist eine weitere Verarbeitung und auch Kombination von verschiedenen Modellen möglich. Im Folgenden werden DSLs und modellbasierte Entwicklung der in dieser Arbeit betrachteten Domänen vorgestellt.

3.1. Modellbasierte Entwicklung

Es gibt eine große Anzahl von Ansätzen zur modellbasierten Softwareentwicklung. Die verschiedenen Ansätze definieren die Modelle, deren Beziehungen und meist auch einen Entwicklungsprozess. Im Folgenden werden die in dieser Arbeit verwendeten Rahmenwerke vorgestellt, die es ermöglichen, eigene Modelle nach dem obigen Ansatz zu definieren.

3.1.1. Eclipse Modeling Framework

Das EMF ist ein in Eclipse integriertes Rahmenwerk für die modellbasierte Entwicklung und Quelltextgenerierung. Das Rahmenwerk basiert auf Metamodellen, die im XMI-Standard [1] der Object Management Group (OMG) abgelegt werden. Die Definition des Metamodellformats (Metaebene 3) bildet den Kern des Rahmenwerks und ist die Grundlage für alle weiteren Werkzeuge.

Die Metamodelle werden durch ein Metametamodell spezifiziert (*Ecore*-Format). Das Dateiformat definiert *Klassen*, *Attribute* und *Referenzen*. Diese Entitäten können dann genutzt werden um Klassen zu definieren, diesen Attribute zuzuordnen und Beziehungen zwischen Klassen durch Referenzen auszudrücken. Zusätzlich wird eine große Zahl von Parametern festgelegt, welche Eigenschaften wie Namen, Kardinalitäten und Ordnungseigenschaften für Entitäten der Metamodelle festlegen. Durch integrierte Werkzeuge ist es möglich, die Metamodelle zu verwalten, zu visualisieren und zu transformieren.

Die Transformation der Metamodelle stellt die Quelltextgenerierung aus den Metamodellen dar. Für die Transformation werden weitere Informationen und Parameter in einem *Generator*-Modell festgelegt. Mittels Quelltextgenerierung werden Quelltexte für die Repräsentation des Modells (Modellcode), für den Zugriff aufs Modell (Editcode), Quelltext für einen Modelleditor und Testfälle generiert.

Der Modellcode enthält die Konstruktoren zur Verwaltung für alle Klassen und deren Attribute des Metamodells. Die Implementierung der einzelnen Methoden kann durch Parameter gesteuert werden. Zusätzlich gibt es Mechanismen, um eigene Implementierungen im generierten Quelltext einzubringen. Diese Implementierungen können bei einer Weiterentwicklung unverändert übernommen werden. Notwendige Anpassungen müssen aber bei Änderungen am Metamodell manuell gepflegt werden.

Auf Grundlage des EMF gibt es weitere Rahmenwerke, die verschiedene Spezialisierungen haben. In dieser Arbeit werden das Graphical Modeling Framework (GMF) und das Connected Data Object (CDO) Modell Repository benutzt und im Anschluss vorgestellt. Das GMF bietet die Möglichkeit, graphische Editoren modellbasiert zu entwickeln. CDO bietet die Möglichkeit, Modelle in Datenbanken zu verwalten.

Graphical Modeling Framework (GMF)

Für den Entwurf eines graphischen Editors für ein EMF-Modell mit dem GMF werden verschiedene Modelle entworfen, welche die Eigenschaften des Editors beschreiben. In der Abbildung 3.1 werden die beteiligten Modelle gezeigt und in Beziehung gesetzt.

Um die Eigenschaften des graphischen Editors zu beschreiben, werden das Metamodell der darzustellenden Modelle, ein Graphikmodell und ein Werkzeugmodell verwendet. Das Metamodell legt die Struktur und die Bedeutung der dargestellten Entitäten fest. Das Graphikmodell beschreibt die Eigenschaften der graphischen Darstellung (z. B. Farbe, Form, usw.). Das Werkzeugmodell beschreibt eine Werkzeugleiste am Rand des Editors, die für die Erzeugung neuer Entitäten genutzt wird.

Durch ein Verknüpfungsmodell werden die drei Modelle in Beziehung gesetzt. Dazu werden den Entitäten aus dem Metamodell jeweils eine graphische Repräsentation und ein Werkzeug aus dem Werkzeugmodell zugeordnet.

Die Modelle sind anschließend die Eingabe für die Quelltextgenerierung des graphischen Editors. Dazu wird ein Generatormodell erzeugt. In diesem Modell sind Eigenschaften wie Dateipfade und Dateierweiterungen für die Erzeugung des Quelltextes und die beteiligten Modelle (Meta-, Graphik-, Werkzeug- und Verknüpfungsmodel) festgelegt. Der Modellierer hat die Möglichkeit, bei den Generierungsschritten einzugreifen und die Eigenschaften, die durch Standardwerte belegt werden, zu ändern.

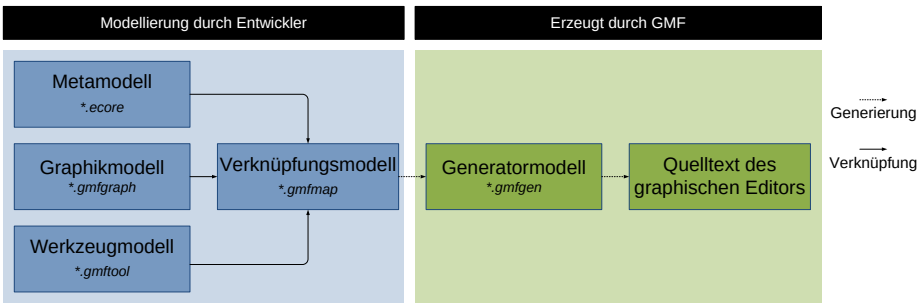

Abbildung 3.1.: Komponenten der Modellierung eines graphischen Editors

Connected Data Object Modell Repository (CDO)

Das CDO Modell Repository ist ein Rahmenwerk, das die Verwaltung der Modelle in Datenbanken ermöglicht. Die Speicherung der Modelle in Datenbanken ermöglicht zum einen den Umgang mit großen Modellen, zum anderen auch ein gleichzeitiges Bearbeiten von mehreren Modellierern an einen Modell.

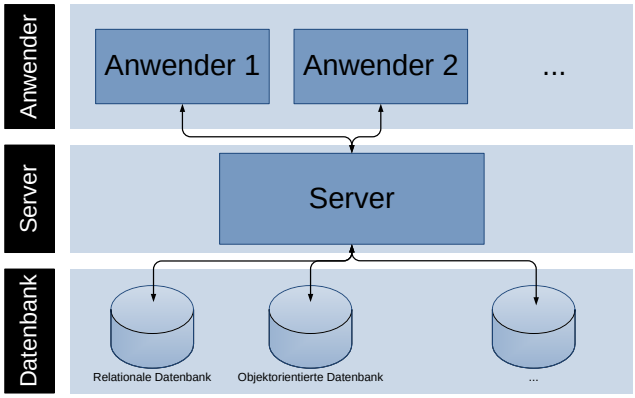

Abbildung 3.2.: Zugriff auf Modelle über das CDO Rahmenwerk

In der Abbildung 3.2 ist der Zugriff auf die Modelle über das CDO Rahmenwerk abgebildet. Für Anwender (Modellierer) ist der Zugriff auf Modelle über eingerichtete Editoren und Projekte möglich. Der Zugriff erfolgt über den URI (Unified Resolution Identifier) Mechanismus. Dieser Mechanismus wird von einen zentralen CDO Server, welcher die Anfragen an verschiedene Datenbanken weiterleiten kann, ausgewertet. Somit ist die Verbindung zu den eigentlichen Datenbanken für die Anwender nicht sichtbar.

Beim gleichzeitigen Zugriff auf ein Modell werden Datenbankmechanismen genutzt, um das Modell in der Datenbank konsistent zu halten. Die Schreibzugriffe werden

nur exklusiv ausgeführt und können fehlschlagen, wenn auf einer nicht aktuellen Version gearbeitet wurde. In diesem Fall muss der Anwender den Konflikt manuell lösen. Der lesende Zugriff ist gleichzeitig möglich. Zusätzlich besteht die Möglichkeit, die Modelle verteilt über mehrere Datenbanken zu speichern um so besseren Zugriff auf große Modelle zu gewährleisten.

3.2. Modellbasierter Entwicklungsprozess von Reglersystemen

Wie schon erwähnt werden viele komplexe Regelungen mit Software in eingebetteten Systemen umgesetzt. Diese Systeme haben sehr unterschiedliche Einsatzszenarien, z. B. in Haushaltsgeräten (Waschmaschine oder Toaster), Transportsystemen (Flugzeugen) oder auch Produktionsanlagen. Weite Verbreitung finden die softwarebasierten Regelungen im Automobil. Die Regelungssysteme erfüllen dort Aufgaben in verschiedenen Bereichen, wie Sicherheit, Verbrauchsreduktion und Komfortfunktionen.

Zur Entwicklung solcher Systeme ist in der Regelungstechnik ein modellbasierter Ansatz üblich. Die dort verwendeten Modelle werden hauptsächlich genutzt, um den Regler und das zu regelnde System zu beschreiben und beteiligte Größen in Beziehung zu setzen. Während eines Entwicklungsprozesses verfeinert man die Modelle weiter. Dazu werden Informationen über verschiedene Aspekte der Reglerentwicklung in den Modellen hinterlegt. Ziel dieser Entwicklungsprozesse ist es, eine hohe Wiederverwendung der verschiedenen Modelle zu erreichen. Ein Ansatz ist die Entwicklung eines Softwarereglers durch den Einsatz von Rapid-Control-Prototyping (RCP).

Der RCP-Entwicklungsprozess ist eine der Grundlagen dieser Arbeit und wird deshalb im Folgenden näher beschrieben. Er kann in drei Phasen aufgeteilt werden. Diese Phasen sind in der Abbildung 3.3 dargestellt. In der ersten Phase wird ein Modell des Reglers und der zu regelnden Strecke entworfen. Ein übliches Modellierungswerkzeug ist MaSi. Die DSL, die in diesem Modellierungswerkzeug verwendet wird, lehnt sich an einen Wirkplan an. Dieser wird in der Regelungstechnik dazu verwendet, Einflussgrößen in Beziehung zu setzen.

Ein wichtiger Bestandteil des Entwicklungsprozesses ist die Simulation der Modelle. Die Simulation ermöglicht in der Regelungstechnik, wichtige Informationen über die Eigenschaften des zu regelnden Systems zu bestimmen. In der MaSi Umgebung ist eine entsprechende Simulation der dort erstellten Modelle möglich. Um eine Simulation zu ermöglichen, müssen alle Komponenten des gesamten Systems modelliert werden. Dieses System besteht aus dem zu regelndem System (Strecke), dem Reglersystem (Verhalten des Reglers) und der Verknüpfung der beiden über Sensoren und Aktuatoren.

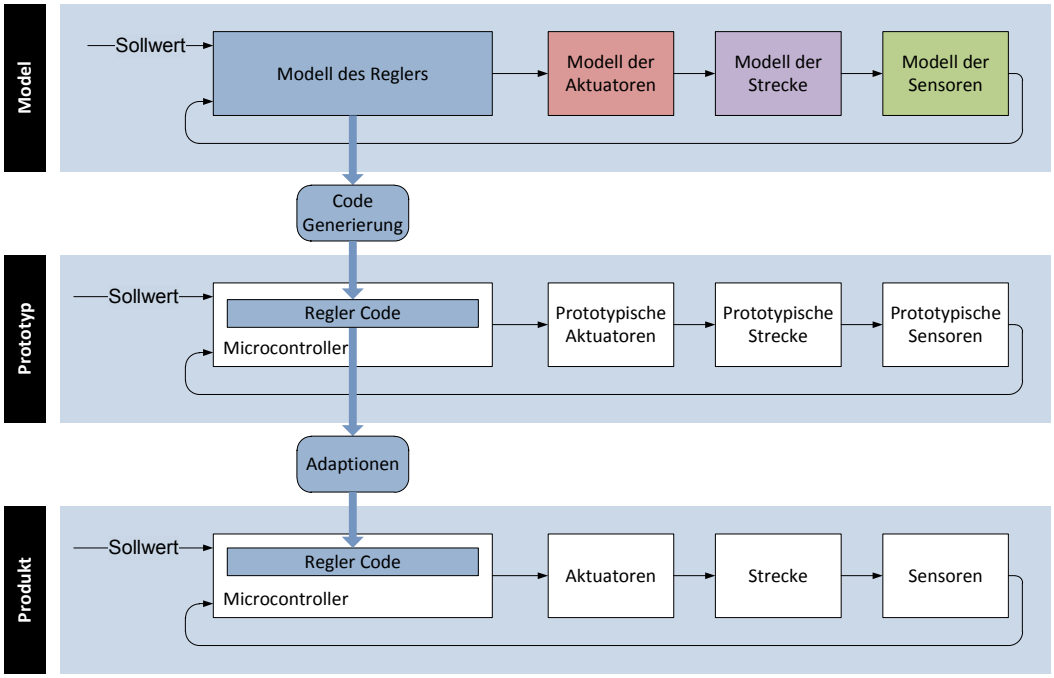

Abbildung 3.3.: Modellbasierte Entwicklung von Regelungssystemen

Während der Simulationen wird das Verhalten der einzelnen Komponenten bestimmt und in mathematischen Konstrukten im Modell hinterlegt. Durch Vergleich mit dem realen System werden die Simulationsergebnisse evaluiert und verbessert. Diese Verbesserung hat das Ziel, die Regelung bezüglich bestimmter Eigenschaften (z. B. Einschwing- oder Totzeit) zu verbessern.

Wenn der modellierte Regler die regelungstechnischen Anforderungen erfüllt, wird in die zweite Entwicklungsphase übergegangen. In dieser Prototypenphase wird das Modell des Reglers in eine Implementierung überführt und an einem Prototyp getestet. Die Überführung des Reglermodells erfolgt meist durch automatische Quelltextgenerierung. Die so erzeugte Implementierung wird dann mit Hilfe von spezieller Hardware (einem Rapid-Control-Prototyping System (RCP-System)) ausgeführt.

Ein RCP-System zeichnet sich dadurch aus, dass die verwendete Hardwarebasis für eine Vielzahl von Regelungsaufgaben geeignet ist. Dies bedingt eine Schnittstelle, die in der Lage ist, Sensoren und Aktoren zu lesen beziehungsweise zu bedienen. Die große Flexibilität bei den Regelungsaufgaben sowie die automatische Generierung von lauffähigen Quelltexten bedingen, dass die Ressourcen der Hardware größer sind als die Ressourcen in dem späteren eingebetteten System.

In der zweiten Phase werden die Komponenten Sensoren, Aktuatoren und Strecke

durch reale Hardware ersetzt. In diesem Schritt kann nun das simulierte Verhalten aus der ersten Phase nachvollzogen und überprüft werden. Das Ziel der Tests ist es, die Eigenschaften der Strecke im Bezug auf Zeitverhalten wie Einschwing- oder Totzeiten zu überprüfen. Während der Tests sucht man Parameter für den Regler, die möglichst gut das gewünschte Verhalten des gesamten Systems zeigen.

In der dritten Phase, die im Bild 3.3 zu sehen ist, wird die endgültige Hardware getestet. Die Funktionalität wurde nun auf die Zielhardware optimiert und integriert. Die Aufgabe von Applikateuren ist es, die Parameter des Reglers so einzustellen, dass die besten Reglergebnisse erzielt werden. Der so entwickelte Regler besteht aus der Hardware, dem Regler und den Parametern des Reglers.

3.3. Simulation

Ein wichtiger Aspekt der modellbasierten Entwicklung ist die Möglichkeit der Simulation der Modelle. Die Simulation ist nach [3] die Imitation eines Prozesses oder Systems der realen Welt. Die Simulation generiert eine künstliche Vergangenheit und benutzt diese, um Schlußfolgerungen über die Ausführungscharakteristiken des repräsentierten realen Systems zu ziehen. Sie wird außerdem genutzt, um das Verhalten eines Systems zu beschreiben und zu analysieren. Mit Hilfe der Simulation werden „Was passiert wenn?“-Fragen beantwortet und der Entwurf eines realen Systems unterstützt. Neben realen werden auch konzeptuelle Prozesse und Systeme simuliert.

Die Simulation ist in vielen Domänen ein Hilfsmittel, um den Entwurf eines Systems zu unterstützen und die Funktionalität frühzeitig im Entwicklungsprozess zu verstehen. Grundlage der Simulation sind Modelle, deren Semantik interpretiert wird. Im Folgenden wird der Einsatz der Simulation in der Regelungstechnik und der modellbasierten Softwareentwicklung erläutert. Die Simulation ist in beiden Disziplinen ein zentrales Element des Entwicklungsprozesses.

Der Entwurf von Reglern ist eng mit der Simulation des geschlossenen Regelkreises verknüpft. Dieser enthält neben dem Regler auch das zu regelnde System (Strecke). Eine wichtige Aufgabe beim Reglerentwurf ist es, ein Verständnis über die Charakteristiken der Strecke zu erhalten. Durch Simulationstechniken wird das Verhalten der realen Strecke mit dem der simulierten verglichen. Wenn die wesentlichen Eigenschaften der Strecke abgebildet werden können, beinhaltet das Modell wichtige Informationen, die für den Entwurf des Reglers verwendet werden. Zusätzlich bietet die Simulation die Möglichkeit, Vorgänge, die in der realen Welt nicht beobachtbar (messbar) sind, abzubilden. Die Modellbildung und Simulation sind in der Regelungstechnik ein zentraler Entwurfsbestandteil.

In der Softwaretechnik ist die Simulation ein Verfahren, um neben dem Testen ein frühes Verständnis von der Funktionalität zu erhalten und somit Fehler zu finden.

Fehler früh zu finden, ist ein wichtiges Ziel in der Softwareentwicklung, da dann die Kosten der Beseitigung von Fehler geringer sind [31]. Die Simulation wird deshalb während des Softwareentwicklungsprozesses in verschiedenen Phasen eingesetzt. Ein Ansatz ist, schon während der Anforderungserfassung eine Simulation zu benutzen. Ein Beispiel hierfür ist die Simulation von Anwendungsfällen (Use Cases) wie sie in [20] beschrieben wird.

Neben der Simulation in frühen Phasen des Entwicklungsprozesses, wie der Anforderungserfassung, wird auch während der Implementierung simuliert. Bei der modellbasierten Entwicklung sind Simulationsumgebungen meist direkt in das Modellierungswerkzeug integriert. Dies ermöglicht eine einfache Bedienung. Ein Beispiel für eine integrierte Modellierungs- und Simulationsumgebung ist MaSi.

Kapitel 4.

Variantenreiche Entwicklung

Seit den 1970er Jahren werden Ansätze der Softwareentwicklung diskutiert, eine gemeinsame Entwicklung von gleichartigen Systemen vorzunehmen [36]. Diese gemeinsame Entwicklung wird als vorteilhaft angesehen, da zwar zu Beginn mehr Aufwand für die Erstellung der gemeinsamen Architektur aufgewendet wird aber anschließend die Ableitung der konkreten Produkte einfacher ist. In [37] und [9] werden entsprechende Methoden und Techniken vorgestellt, die eine systematische Vorgehensweise beschreiben.

Die vorgestellten Methoden bilden die Grundlage der in dieser Arbeit vorgestellten Ansätze. Die Anwendung der Methoden ist aber im industriellen Umfang nicht trivial. Dies gilt vor allem, wenn die Komplexität der Applikationen steigt. Zusätzlich ist die Integration von Entwicklungsansätzen wie der modellbasierten Entwicklung durch Randbedingungen nicht einfach. Das gilt auch für definierte Prozesse, die für große Organisationen (welche einen hohen Bedarf an Produktlinien haben) elementar sind.

4.1. Produktlinienansatz in der Softwaretechnik

Bei der Produktlinienentwicklung entstehen viele ähnliche Varianten eines Produkts. Diese besitzen eine gemeinsame zentrale Kernarchitektur, welche durch variantenspezifische Teilfunktionalitäten erweitert wird. Eine Möglichkeit bei der Produktlinienentwicklung ist, jede Variante eigenständig zu entwickeln und zu implementieren. Offensichtlich verursacht diese Methode aber einen hohen Grad an Redundanz, da auch die Gemeinsamkeiten der Varianten für jede Variante eigenständig erarbeitet werden müssen. Es ist nahe liegend, diesen Arbeitsaufwand zu vermeiden, indem die gemeinsame Architektur als Grundlage für alle daraus entstehenden Varianten entwickelt wird. Jedoch ist dies nicht einfach zu realisieren. Erste Erfahrungen in den 90er Jahren zeigten, dass die Kosten beim Versuch der Ausnutzung von Gemeinsamkeiten einer Produktlinie in ihrer Entwicklung ohne angemessene Planung deutlich höher sein können als die Kosten für die eigenständige Entwicklung jeder Variante [37].

Der Produktlinienansatz ist ein Konzept, welches eine systematische Wiederverwendbarkeit der gemeinsamen Softwarefragmente ermöglicht. Im Folgenden wird

ein Produktlinien-Framework vorgestellt, welche die Idee des Produktlinienansatzes verfolgt.

4.1.1. Produktlinien-Framework nach Pohl, Böckle, van der Linden

In diesem Abschnitt wird ein Produktlinien-Framework vorgestellt, welches auf den Ergebnissen der drei Forschungsprojekte ESAPS, CAFÉ und FAMILIES basiert. Die Ergebnisse wurden u. a. in Kapitel 2 des Buchs Software Product Line Engineering von Klaus Pohl, Günter Böckle und Frank van der Linden publiziert [37]. Im Framework wird zwischen Domain-Engineering und Application-Engineering unterschieden.

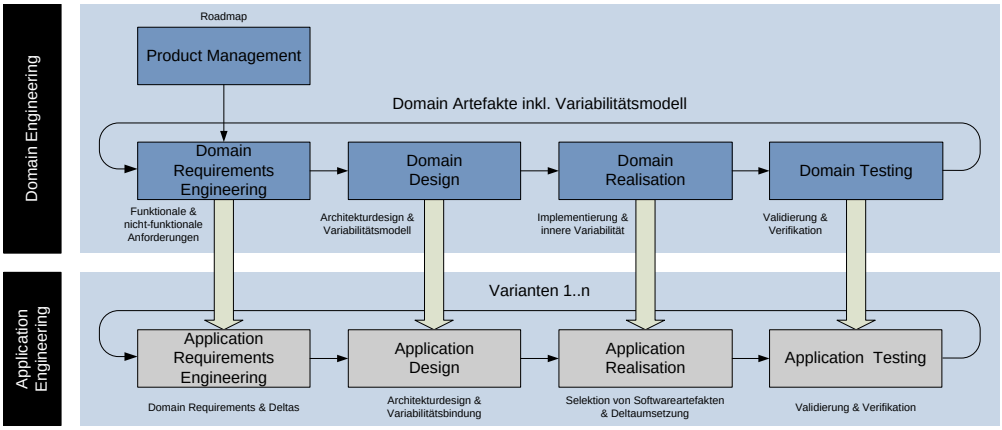

Abbildung 4.1.: Produktlinien-Framework nach Pohl, Böckle, van der Linden

Die Abbildung 4.1 gibt einen Überblick über die Phasen des Produktlinien-Frameworks. In der oberen Hälfte der Grafik befinden sich die Phasen des Domain-Engineering, in der unteren Hälfte die Phasen des Application-Engineering. Die Pfeile skizzieren den Informationsfluss im Framework, wobei die Pfeile vom Testing zum Requirements-Engineering andeuten, dass die Phasen in der Regel einen Zyklus bilden und nach den Tests weitere Anpassungen folgen können.

Domain-Engineering

Domain-Engineering beschreibt den Prozess, in dem die gemeinsamen Aspekte einer Produktlinie und die variablen Funktionalitäten für die Variantenspezifikationen definiert und entwickelt werden. Er umfasst Anforderungsspezifikation, Architekturdesign, Implementierung und Test des Domänenmodells. Diese Schritte werden innerhalb des Domain-Engineering durch die Teil- oder Subprozesse

Product-Management, Domain-Requirements-Engineering, Domain-Design, Domain-Realisation und Domain-Testing realisiert, welche im Folgenden vorgestellt werden.

Das Product-Management befasst sich mit den ökonomischen Aspekten einer Produktlinie und der Marktstrategie des entwickelnden Unternehmens. Die für die Produktlinie verantwortliche Person extrahiert in dieser Phase die geforderten Funktionalitäten und dokumentiert diese in einer Roadmap. Die Roadmap definiert die zukünftigen Produkte, deren Features sowie einen Zeitplan. Features bezeichnen hier die für den Benutzer sichtbaren Funktionen der Produktlinie.

Beim Domain-Requirements-Engineering werden aus der Roadmap die funktionalen sowie nicht-funktionalen Anforderungen an die Produktlinie bestimmt. Die funktionalen Anforderungen legen fest, mit welchen Funktionalitäten die in der Roadmap angegebenen Features umgesetzt werden. Die nicht-funktionalen Anforderungen dagegen bestimmen, mit welcher Qualität (z. B. Effizienz der Software) und unter welchen Bedingungen (z. B. Plattformunabhängigkeit) die geforderten Funktionalitäten zu erbringen sind. Außerdem wird das Variabilitätsmodell definiert, welches die gemeinsamen und variablen Anforderungen identifiziert.

Der Subprozess Domain-Design modelliert aus den Anforderungen und dem Variabilitätsmodell die Kernarchitektur, welche als Grundgerüst für alle später entstehenden Varianten der Produktlinie dient. Diese Modellierung muss sehr sorgfältig erfolgen, da sich Mängel der Architektur auf die gesamte Produktlinie auswirken. Zusätzlich erweitert das Domain-Design das Variabilitätsmodell um sogenannte interne Variabilität. Interne Variabilität entsteht z. B., wenn variable Anforderungen verschiedene technische Realisierungen erfordern.

Die Domain-Realisation beschäftigt sich mit der Umsetzung der vom Domain-Design gelieferten Kernarchitektur. Dabei wird das Modell der Kernarchitektur verfeinert; auf Basis dieses verfeinerten Modells wird dies dann implementiert. Das Ergebnis ist das detaillierte Design und die wiederverwendbaren Softwareartefakte der Kernarchitektur.

Als letzter Teilprozess im Domain-Engineering befasst sich das Domain-Testing mit der Validierung und Verifikation der Kernarchitektur. Die verschiedenen Softwareartefakte werden dabei gegen ihre Anforderungen getestet. Um den Aufwand zu verringern, werden dabei wiederverwendbare Testfälle entwickelt. Auf Basis der hier ermittelten Testergebnisse kann der Domain-Engineering-Prozess mit Ausnahme des Product-Managements weitere Male durchlaufen werden, um Anforderungen, Kerndesign und Implementierungen so anzupassen, dass vorhandene Fehler beseitigt werden.

Application-Engineering

Das Application-Engineering hat die Ableitung konkreter Varianten der Produktlinie unter Verwendung der vom Domain-Engineering gelieferten Ergebnisse zum

Ziel. Das Ziel ist eine möglichst hohe Wiederverwendbarkeit der Kernarchitekturkomponenten bei der Variantenentwicklung und damit die Ausnutzung von gemeinsamen und variablen Anforderungen der Produktlinie. Außerdem werden die Applikations-Artefakte, z. B. Applikationsanforderungen, Architektur, Komponenten und Tests, dokumentiert und mit den zugehörigen Domain-Artefakten verknüpft. Das Application-Engineering besteht aus den vier Subprozessen Application-Requirements-Engineering, Application-Design, Application-Realisation und Application-Testing, welche im Folgenden näher erläutert werden.

Das Application-Requirements-Engineering befasst sich mit der Anforderungsspezifikation einer Variante. Die Anforderungen werden auf Basis der aus der Roadmap extrahierten Hauptfeatures spezifiziert. Zusätzlich können Anforderungen hinzukommen, welche beim Domain-Requirements-Engineering noch nicht berücksichtigt wurden, z. B. spezielle Kundenwünsche an die Variante. Außerdem erfolgt beim Application-Requirements-Engineering die Bindung der Variabilität durch Auswahl bestimmter Varianten für die Variationspunkte. Die Anforderungsspezifikation hat signifikante Bedeutung für die Anzahl der wiederverwendbaren Komponenten der Kernarchitektur. Deswegen ist es eine Hauptaufgabe des Application-Requirements-Engineering so genannte *Deltas* ausfindig zu machen. Deltas bezeichnen Anforderungen, welche nicht durch Komponenten der Kernarchitektur realisiert werden können. Deltas verursachen beim Application-Engineering den größten Aufwand, da für diese die benötigten Artefakte noch entwickelt werden müssen.

Der Teilprozess Application-Design ist für die Modellierung der Variantenarchitektur zuständig. Auf Basis der Anforderungsspezifikation der Variante werden aus der Kernarchitektur die benötigten Komponenten ausgewählt, konfiguriert und an die Variante angepasst.

Analog zur Domain-Realisation setzt die Application-Realisation die Variantenarchitektur um. Dazu werden aus den in der Domain-Realisation erzeugten Softwareartefakten die für die Variante benötigten ausgewählt. Des Weiteren werden hier die Softwareartefakte für die Deltas entwickelt. Liegen alle Softwareartefakte vor, können diese miteinander kombiniert werden, um die gewünschte Variante zu erhalten.

Die in der Application-Realisation konstruierte Variante wird im Subprozess Application-Testing validiert und verifiziert. Auf Grundlage der Variantenartefakte (Anforderungen, Architektur, Implementierung) und der im Domain-Testing entwickelten, wiederverwendbaren Testfälle werden variantenspezifische Testfälle entwickelt. Wurde mit diesen die Variante gegen ihre Anforderungen getestet, werden die Testergebnisse in Testberichten erfasst. Zusätzlich können Problemberichte erstellt werden, welche die in der Variante entdeckten Fehler dokumentieren. Wie auch beim Domain-Engineering kann die Application-Engineering-Kette mehrmals durchlaufen werden, falls beim Application-Testing Fehler entdeckt werden.

4.2. Produktlinienansatz in der modellbasierten Entwicklung

Variabilität spielt im Allgemeinen in allen Produktlinien-Entwicklungsphasen eine Rolle:

- Anforderungen (unterschiedliche Szenarien),
- Architektur (optionale / alternative Komponenten)
- Implementierung (Realisierung der Variabilität auf Implementierungsebene).

Für jede Entwicklungsphase existieren unterschiedliche Modellierungsarten für Variabilität. Variabilität lässt sich unabhängig von der Entwicklungsphase mit dem Allgemeinen Variabilitätsmodell (AVM) modellieren. In dem AVM wird die allgemeine Beziehung zwischen Variationspunkt und Variante modelliert unter Einbeziehung eventueller Abhängigkeiten. Eine Variante gehört damit zu mindestens einem Variationspunkt. Außerdem können sowohl Varianten als auch Variationspunkte voneinander abhängen. So könnte zum Beispiel im automobilen Sektor eine bestimmte Softwarevariante von einer bestimmten Variante eines Steuergerätes abhängen (Variantenabhängigkeit).

Darüber hinaus existiert noch ein weiteres Modell, das Orthogonale Variabilitätsmodell, das Variabilität separat von Gemeinsamkeiten betrachtet und alle Entwicklungsstufen einbezieht. Im Folgenden werden für alle Phasen des AVM Modelle vorgestellt.

Auf der Anforderungsebene bieten sich Features, d. h. bestimmte Charakteristika, an, um Variabilitäten zwischen den Produkten zu modellieren. Sie behandeln also die Frage, was variabel ist bzw. wo die Variationspunkte liegen. Dazu unterscheidet man z. B. bei der Feature-Oriented-Domain-Analysis (FODA) drei Arten von Features [17]:

- obligatorische Features,
- optionale Features,
- alternative Features.

Obligatorische Features sind dabei Features, die in jedem Produkt der Produktlinie verfügbar sein müssen. Optionale Features können in einem Produkt vorhanden sein, müssen es aber nicht. Die alternativen Features stehen in Kontravalenz zueinander. Von ihnen muss genau eines ausgewählt werden. Diese Einteilung von Features ermöglicht es, die Variabilität in den Anforderungen an die unterschiedlichen Produkte einer Produktlinie auf einfache Weise in einem Feature-Baum darzustellen.

Jeder Knoten dieses Baumes stellt ein bestimmtes Feature dar, das aus seinen Kind-Features zusammengesetzt ist. Ist ein Knoten mit einem ausgefüllten Kreis gekennzeichnet, so bedeutet dies, dass das entsprechende Feature obligatorisch ist wenn das Feature des Vaterknotens in einem System vorhanden ist. Ist der Kreis nicht ausgefüllt, so ist das jeweilige Feature optional. Alternative Features werden durch einen Bogen zwischen den Ästen dargestellt. Neben dieser Kategorisierung existieren zusätzlich Restriktionen. Ein Feature kann ein anderes Feature ausschließen (exclude) oder ein anderes Feature benötigen (require). Zum Beispiel könnte es sein, dass ein bestimmter Motor nur mit Automatikgetriebe geliefert werden soll.

Auch im Hinblick auf die Beziehung zwischen Automotive-Software und zugehöriger Hardware oder Sensoren ergeben sich häufig Restriktionen. So kann zum Beispiel bestimmte Hardware bestimmte Sensoren benötigen, wodurch auch die Ansteuerung selbiger durch die Software unterschiedlich erfolgen muss. Diese Restriktionen bilden zusammen mit dem Feature-Baum das Feature-Modell.

Schließlich ist auch die Angabe von Kardinalitäten wie [11] sie beschreibt möglich, um zu spezifizieren, wie viele Features in der jeweiligen Beziehung zum Vaterknoten stehen. Optionale Features könnten somit durch [0,1] und alternative Features mit [1,n] markiert werden. Außerdem lassen sich beliebige weitere Kardinalitäten spezifizieren. Können von einem Feature zum Beispiel zwischen einem und fünf Einheiten ausgewählt werden, so würde man die Verzweigung mit [1,5] markieren.

In der Praxis hat sich gezeigt, dass die Unterteilung von Features in die drei Kategorien nicht ausreicht. Außerdem können Restriktionen durch den Feature-Baum nicht ausgedrückt werden. Dieses Problem hat zu Versuchen geführt, die Menge der Feature-Kategorien zu vergrößern.

Eine Erweiterung wird in [45] vorgestellt. Hier wird die zusätzliche Kategorie *Oder* eingeführt, die aussagt, dass von der so modellierten Featuremenge beliebig viele Features in einem System der Produktlinie vorhanden sein können. Im Baum werden diese Features durch einen Bogen zwischen den Ästen dargestellt, wobei die Fläche innerhalb des Bogens bis zum Ausgangspunkt der Äste ausgefüllt ist. Die zugehörige implizite Kardinalität ist [0,n].

Des Weiteren wird auch eine Modellierungstechnik für Abhängigkeiten vorgestellt. Abhängigkeiten werden durch Pfeile zwischen den betroffenen Features modelliert. Impliziert die Auswahl eines Features A die Auswahl eines anderen Features B, so wird dies mit einem Pfeil von A nach B modelliert. Schließen sich zwei Features gegenseitig aus, so sind sie durch einen Doppelpfeil miteinander verbunden. Eine Zusammenstellung der Modellelemente ist in Table 4.1

Tabelle 4.1.: Elemente eines erweiterten Feature-Baumes aus [17] und [45]

	Typ	**Semantik**	**Charakter**	**Notation**
Domänenbeziehung	Obligatorisch	Wenn das Vater-Feature ausgewählt wird, so muss das Kind-Feature ebenfalls ausgewählt werden.		
	Optional	Wenn das Vater-Feature ausgewählt wird, so kann das Kind-Feature wahlweise ausgewählt werden.		
	Alternativ	Wenn das Vater-Feature ausgewählt wird, so muss genau ein Kind-Feature aus der Menge alternative Kind-Features ausgewählt werden.	Impliziter wechselseitiger Ausschluss zwischen alternativen Kind-Features	
	Oder	Wenn das Vater-Feature ausgewählt wird, so muss mindestens ein Kind-Feature ausgewählt werden.		
Abhängigkeit	Implikation	Die Auswahl eines Features impliziert die Auswahl eines anderen Features.	Transitiv	
	Ausschluss	Zwei Features schließen sich wechselseitig aus und können nicht zusammen Bestandteil einer Produktdefinition sein.	Symmetrisch	

Teil II.

Methodische Konzepte

Kapitel 5.

Konzept einer modellbasierten Reglerentwicklung mit variablen Schnittstellen

Die modellbasierte Entwicklung von softwarebasierten Reglern bzw. eingebetteten Systemen ist eng mit der Integration von Schnittstellen verknüpft. Die Schnittstellen dieser Systeme sind Sensoren und Aktoren, welche die einbettende Umgebung erfassen und beeinflussen. Die Eigenschaften dieser Schnittstelle haben in vielen Fällen einen entscheidenen Einfluss auf die Systeme. Dies ist ein wichtiger Aspekt bei der Entwicklung von Reglern, der in der Regelungstechnik betrachtet wird.

Die Sensoren und Aktoren werden über verschiedee Hardwareschnittstellen integriert. Diese können in digitale und analoge Schnittstellen unterschieden werden. Für die analogen Schnittstellen gilt, dass die Übersetzung in einen digitalen Wert innerhalb des eingebetteten Systems erfolgt. Das eingebettete System muss entsprechende Ressourcen bereitstellen, um eine Digitalisierung vorzunehmen. Zusätzlich ist auch eine Aufbereitung für digitale Signale notwendig. Dies gilt für Sensoren oder Aktuatoren, die die Information in Form von einfachen Bitfolgen oder mittels Protokollen wie z. B. Busstandards bereitstellen. Die elektrische Aufbereitung bzw. die Ausgabe der Signale wird von Elektrotechnikern entwickelt. Deren Anforderung ist es, eine möglichst günstig zu realisierende und universelle Schaltung zu entwickeln. Dies widerspricht meist den Anforderungen aus Sicht der Informatik und Regelungstechnik an die Systemschnittstelle. Aus der Regelungstechnik kommen Echtzeitforderungen an die Bereitstellung von Informationen. Diese müssen dann durch Software erfüllt werden, die dann wiederum Anforderungen an die zur Verfügung stehende Hardware stellt. Insgesamt arbeiten an der Erstellung eines eingebetteten Reglersystems verschiedene Domänen zusammen, die teilweise widersprechende Anforderungen erfüllen müssen. Diese Anforderungen treffen im Bereich der Schnittstellen aufeinander.

In diesem Kapitel wird ein Ansatz vorgestellt, der die Anforderungen der verschiedenen Domänen bei der Reglerentwicklung beachtet. Dazu wird mittels einer Hardwareabstraktionsschicht eine vollständige Abstraktion der Sensoren und Aktuatoren

zur Modellebene vorgenommen. Um eine gemeinsame Basis für die unterschiedlichen Sensoren und Aktuatoren zu schaffen, werden die Informationen basierend auf physikalischen Größen zur Verfügung gestellt. Die dazu notwendigen Komponenten sind simulierbar und ohne Modelländerungen austauschbar und erfüllen so wichtige Anforderungen aus der Regelungstechnik. Die Komponenten sind wiederverwendbar und können den Schutz des Knowhows aus Sicht der Informatik bieten. Aus Sicht der Elektrotechnik dient ein hierarchischer Aufbau der Architektur dazu, dass unterschiedliche Umsetzungen der Hardwareschnittstelle im System schnell und zuverlässig benutzt werden können.

Um die Anforderungen der verschiedenen Domänen zu erfassen, erfolgt in diesem Kapitel zuerst eine Erfassung und Analyse der Anforderungen. Ausgehend von dieser Analyse wird das Konzept und die Umsetzung einer variablen Schnittstelle für die modellbasierte Entwicklung mit einem Rapid-Control-Prototyping System vorgestellt.

5.1. Regelungstechnische Anforderungen

Die Anforderungen der Regelungstechnik können bei der softwarebasierten Reglerentwicklung in funktionale und nicht-funktionale Anforderungen eingeteilt werden. Die funktionalen oder auch technischen Anforderungen des Reglers lassen sich laut [28] sich in vier Gruppen unterteilen:

1. Stabilitätsforderungen
2. Störgrößenkompensation und Sollwertfolge
3. Dynamisches Verhalten
4. Robustheit

Diese klassischen Anforderungen an einen Regler werden im Folgenden kurz erläutert. Die Stabilitätsforderung wird an einen Regelkreis gestellt. Dies bedeutet, dass ein geschlossener Regelkreis (bestehend aus Regler und zu regelndem System) innerhalb der Spezifikation kein aufschwingendes Verhalten haben darf und letztendlich wieder in eine Ruhelage kommt.

Die Störgrößenkompensation und Sollwertfolge beschreibt die Eigenschaft, dass die Regelgröße (das Ergebnis der Regelung) der vorgegeben Führungsgröße (die Vorgabe für den Regler) folgt. Dazu muss der Regler die Regelgröße so beeinflussen, dass die Abweichung zwischen Führungsgröße und Regelgröße kleiner als ein festgelegter maximaler Wert ist.

Das Einstellen der Führungsgröße muss zusätzlich einem dynamischen Verhalten entsprechen. Diese Vorgaben beziehen sich auf die Überschwingung (maximale

Differenz zwischen Regelungs- und Führungsgröße), die maximale Einschwingzeit und die Anstiegszeit.

Die Robustheit fasst diese vorher genannten Anforderungen zusammen und verlangt zusätzlich, dass diese Kriterien auch unter der Voraussetzung von einer unsicheren Strecke erreicht werden. Dies bedeutet, dass der Regler die Kriterien trotz leichter Änderungen der Strecke, wie sie während der Lebenszeit des Reglers in der Realität vorkommen, erreicht.

Neben diesen funktionalen Anforderungen müssen auch nicht-funktionale Anforderungen in der Regelungstechnik beachtet werden. Diese Anforderungen werden aber meistens nicht im Entwicklungsprozess der Regelungstechnik berücksichtigt. Um diese zu bestimmen, wurde im Rahmen dieser Arbeit mit Regelungstechnikern eine Liste von relevanten nicht-funktionalen Anforderungen erarbeitet. Als Grundlage diente eine Aufstellung von nicht-funktionalen Anforderungen aus der Softwaretechnik. Das Ergebnis ist eine Aufstellung nicht-funktionaler Anforderungen, die bei der Entwicklung von Reglern (insbesondere modellbasierten Reglern) wichtig sind. Die nicht-funktionalen Anforderungen lassen sich in drei Kategorien einteilen. Diese Anforderungen beziehen sich auf:

- das zu entwickelnde Produkt (Regler),
- den Prozess der Reglerentwicklung und
- externe Anforderungen.

Die *Produktanforderungen* für einen softwarebasierten Regler sind ähnlich zu Anforderungen von eingebetteten Systemen. Allerdings sind einige Begriffe in einer anderen Bedeutung definiert. Eine wichtige Anforderung ist die *Bedienbarkeit* eines Reglers. In der Regelungstechnik wird diese mit der Applizierbarkeit gleich gesetzt und bezieht sich nicht auf den Endanwender des Reglers, sondern auf den Applikateur, der die optimalen Parameter des Reglers am Ende des Entwicklungsprozesses, meist durch Prüfstandsversuche bestimmt. In dieser Entwicklungsphase werden die Parameter der Regler so eingestellt, dass der Regler das gewünschte Verhalten bezüglich der funktionalen Anforderungen zeigt. Die Anzahl der Parameter, eine Dokumentation der Funktionsweise und Empfehlungen für die Einstellungen, sind wichtige Maßnahmen, um eine gute Bedienbarkeit / Applizierbarkeit zu erreichen.

Daneben ist die *Verlässlichkeit* ein wichtiger Oberbegriff, welcher die Sicherheit, Zuverlässigkeit und Verfügbarkeit zusammenfasst. Diese Qualitäten werden vor allem bei sicherheitskritischen Systemen betrachtet. Um eine hohe Verlässlichkeit aus Sicht der Regelungstechnik zu erreichen, konzentriert sich die Regelungstechnik darauf, dass der Regler dem vorgegebenen Referenzsignal immer folgt.

Eine weitere wichtige Qualität, die *Effizienz*, unterscheidet sich in der Software- und Regelungstechnik. In der Softwaretechnik umfasst der Begriff die benötigten

Ressourcen im Bezug auf Rechenzeit, benötigten Speicher und Energieverbrauch der Hardware. Die Effizienz der Regelungstechnik beinhaltet zusätzlich noch den Energieverbrauch für die Regelung. Eine hohe Effizienz lässt sich beispielsweise durch eine Begrenzung der Stellgröße erreichen, dies hat aber meist negativen Einfluss auf die funktionalen Anforderungen.

Ein zweiter Bereich nicht-funktionale Anforderungen der Regelungstechnik ist in der Softwaretechnik dem *Entwicklungsprozess* zugeordnet. Die Analyse der Anforderungen ergab, dass sich die Anforderungen von softwaretechnischen Anforderungen an einen Entwicklungsprozess unterscheiden. Die wichtigsten Qualitäten für den Reglerentwicklungsprozess sind die Skalierbarkeit, die Portabilität und die Adaptierbarkeit. Die Skalierbarkeit bezieht sich auf die vorhandenen Hardwareresourcen. Dies bedeutet, dass mehr Hardwareresourcen den Regelfehler verringern können, da eine besser und meist aufwändigere Regelung verwendet werden kann.

Die Portabilität des entwickelten Reglers ist eine weitere wichtige Anforderung an den Regler. Der Regler muss im Bezug auf die verwendete Hardware (insbesondere Aktuatoren und Sensoren) wiederverwendbar sein. Die Auswahl oder Anpassung der Sensoren und Aktuatoren hat entscheidenden Einfluss auf die technischen Anforderungen und ist damit ein Teil des Entwicklungsprozesses.

Um einen adaptierbaren Regler zu entwickeln, müssen im Regler Schnittstellen geschaffen werden, die es ermöglichen den Regler anzupassen. Diese Parameter müssen je nach Einsatzszenarien verschiedene Eigenschaften des Reglers beeinflussen. Zusätzliche *externe Anforderungen*, wie zum Beispiel gesetzliche Vorgaben, spielen bei der Reglerentwicklung eine wichtige Rolle. Diese Vorgaben sind in vielen Fällen ein wichtiger Faktor für den Einsatz von Reglern. Diese hängen stark vom Einsatzszenario ab und können nicht verallgemeinert werden. Im Fall der vorliegenden Arbeit wurden modellbasierte Regler für Verbrennungsmotoren entwickelt, um Abgasvorschriften einzuhalten. Diese Vorschriften beinhalten neben Grenzwerten auch Vorschriften über Protokollierungen und Definitionen von Schnittstellen.

5.2. Softwaretechnische Anforderungen

Neben den regelungstechnischen Anforderungen spielen bei der softwarebasierten Reglerentwicklung auch softwaretechnische Anforderungen eine wichtige Rolle. Die Anforderungen hängen stark von Einsatzszenario und der Situation ab, in der der Regler eingesetzt werden soll. In diesem Fall wurden die Anforderungen in einem Unternehmen bestimmt, welches ein RCP-System für die Reglerentwicklung vermarkten will. Die Anforderungen wurden in Workshops und Befragungen von Mitarbeitern szenarienbasiert dokumentiert und anschließend analysiert. Die Anforderungen lassen sich in die gesetzten Randbedingungen, die technischen Anforderungen und nicht-funktionale Anforderungen gliedern.

5.2.1. Randbedingungen

Die gesetzten Randbedingungen folgen aus Designentscheidungen, welche die Hardware betreffen. Zusätzliche Randbedingungen stammen aus betriebswirtschaftlichen Entscheidungen. Die Hardware, die Grundlage des Projektes ist, ist eine RCP Plattform, die mit speziellen Hardwareschnittstellen ausgestattet wurde. Diese Schnittstellen sind auf die Motorenregelung zugeschnitten, sollen aber möglichst flexibel einsetzbar sein.

Aus betriebswirtschaftlichen Gründen muss die neu zu entwickelnde Architektur das Modellierungswerkzeug Matlab / Simulink (MaSi) unterstützen. Diese Anforderung ergibt sich aus den Wünschen der Anwender. Die wichtigsten Anwender sind Ingenieure aus dem Maschinenbau, der Regelungstechnik und Elektrotechnik. Diese fordern den Umgang mit MaSi. Zusätzlich muss die Quelltextgenerierung aus Modellen in MaSi mit dem Realtime Workshop erfolgen.

Auf der Hardwarebasis muss ein spezielles Betriebsystem verwendet werden, um den generierten Quelltext auszuführen. Zudem muss der generierte Quelltext eine Möglichkeit zur Kalibrierung nach dem ASAM MCD Standard unterstützen.

5.2.2. Anforderungserfassung

Die funktionalen Anforderungen lassen sich aus den Anforderungen an ein RCP-System ableiten. Die Hauptaufgabe eines RCP-Systems ist es, eine durchgängige Entwicklung zu ermöglichen. Dazu ist es notwendig, ein Modellierungswerkzeug zur Verfügung zu stellen, welches eine modellbasierte Reglerentwicklung ermöglicht. Diese Entwicklungsumgebung wird durch die Randbedingungen bereits festgelegt. Zusätzlich ist es notwendig, dass die modellierte Funktionalität automatisch in ausführbaren Quelltext umgewandelt werden kann. Ohne weitere Änderungen soll der generierte Quelltext auf der Zielplattform ausführbar sein.

Die Eigenschaften der Entwicklungsumgebung MaSi müssen erhalten bleiben. Dazu zählt die Fähigkeit, einen Regler oder andere Software zu modellieren und zu simulieren. Die in MaSi verfügbaren Erweiterungen (z. B. Zustandsmodellierung mit Stateflow) müssen weiter nutzbar bleiben.

Zusätzlich muss die Einbindung von Sensoren und Aktuatoren auf Modellebene möglich sein, sodass ohne weitere Eingriffe der Entwickler direkt ein lauffähiges Programm generiert werden kann. Die Erfassung der nicht-funktionalen Anforderungen erfolgt durch das Aufstellen und Bewerten von Szenarien. Die Szenarien können in die folgenden Bereiche eingeteilt werden:

- Szenarien zur **Fehlerbehandlung**
- Szenarien zu **allgemeinen Eingenschaften und Quelltextgenerierung des Rapid-Control-Prototyping Systems**

- Szenarien zur **Parametrierung**
- Szenarien zur **Modellierung**
- Szenarien zum **Entwicklungsprozess des Rapid-Control-Prototyping Systems**

Die Szenarien werden in den folgenden Kapiteln dargestellt. Zur Darstellung der Szenarien werden drei Einträge verwendet. Der ersten Eintrag beschreibt das Szenario. Eine Erläuterung und Vertiefung wird im Kommentar gegeben. Eine Bewertung findet im letzten Eintrag statt. Die Bewertung der Szenarien erfolgt durch einen Entwickler und kann in den drei Abstufungen

- L – Low
- M – Medium
- H – High

abgegeben werden. Der Entwickler bewertet im ersten Eintrag die Bedeutung des Szenarios und im zweiten die Schwierigkeit, dieses in der Architektur zu realisieren. Die Bewertung der Kriterien erfolgte in Anlehnung an das Architektur-Bewertungsverfahren Architecture Analysis Tradeoff Method (ATAM) [8, 23].

Fehlerbehandlung

Die Szenarien zur Fehlerbehandlung behandeln mögliche Fehlerfälle, die durch Entwickler bzw. durch einen Ausfall von Komponenten zu Stande kommen.

Szenario	**Kommentar**	**Bedeutung / Schwierigkeit**
F1 Ein Kunde möchte selbstständig auf Fehlerinformationen zugreifen können.	Die Architektur muss ein umfassendes Konzept zur Fehlerbehandlung anbieten. Dies muss Modellierungsfehler bis hin zu Hardwarefehlern, die die Treiberschicht meldet, berücksichtigen. Während der nachträglichen Kalibrierung sollte dieses Szenario wie bei S5 über die Integration der A2L Schnittstelle abgefangen werden.	H / H
Fortsetzung auf der nächsten Seite		

Szenario	Kommentar	Bedeutung / Schwierigkeit
F2 Das Prototypingsystem fällt bei einer Probefahrt auf Grund von SW Problemen aus. Das System geht in einen vordefinierten Zustand.	Wichtig ist, dass der Ausfall dokumentiert wird. Ein definierter Fehlerzustand muss erreicht werden. Die Definition und Reaktion auf einen Fehlerzustand erfolgt jedoch kundenseitig.	M / M
F3 Bei Ausfall einer Teilfunktion läuft das System mit einem geringeren Funktionsumfang weiter.	Entscheidend ist hierbei, dass der Kunde selbst entscheiden kann, wie sich das System im Fehlerfall einer Teilkomponente verhalten soll.	M / H
F4 Ein Ausfall der Lambdasonde wird signalisiert und in den Fehlerspeicher geschrieben.	Wichtig. Vgl. Szenario S5 und S6.	M / M
F5 Im Fehlerfall werden Fehlerinformationen in einer On-Board-Diagnose Funktion abgespeichert. Diese muss den gesetzlichen Vorgaben entsprechen und vom Kunden über eine A2L Schnittstelle auszulesen sein.	Die Definition, welche Fehlerinformationen abgespeichert werden, erfolgt kundenseitig.	M / H

Eigenschaften des Rapid Control Prototyping Systems

Die Szenarien zu den Eigenschaften des RCP-Systems beschreiben die allgemeine Situation und die Generierung des Quelltextes aus dem Modell.

Szenario	Kommentar	Bedeutung / Schwierigkeit
R1 Bei Beendigung der Entwicklung können Debuginformationen automatisch und ohne Nebeneffekte entfernt werden.	Dieses Szenario ist wichtig für die Seriencodegenerierung. Diese ist aber kein wichtiges Ziel des Systems.	L / M

Fortsetzung auf der nächsten Seite

Szenario	**Kommentar**	**Bedeutung / Schwierigkeit**
R2 Fertig getestete Funktionssoftware muss beim Übergang zum Seriensteuergerät nicht mehr angepasst werden.	Aus der Software soll der Seriencode ableitbar sein. Die Optimierung der RCP Software muss nicht bis ins Letzte geschehen, da bei den Anwendungen nicht der optimale Code das Wichtigste ist, sondern dass die Software nicht mehr stark verändert werden muss.	L / M
R3 Ein Kunde möchte auf der Prototypplattform Module ohne Seiteneffekte hinzufügen und entfernen können.	Die Schnittstelle sollte klar definiert sein. Wichtig für den Kunden sind die Matlab Modul Schnittstellen. Die der C-Module sind wichtig für die VEMAC	H / M
R4 Die Kunden können Änderungen vornehmen, ohne Einsicht in Firmen Knowhow zu bekommen.	Um das in der Library enthaltene geistige Eigentum zu schützen, muss der Code weitestgehend geschützt werden. Des Weiteren sollte die Architektur eine gestaffelte Offenlegung des Codes ermöglichen, um bestimmten Kunden mehr Informationen zur Verfügung stellen zu können.	H / M
R5 Zwei Kunden wünschen unterschiedliche Werkzeuge im Prototypingsystem: MaSi Lösung	Die Portierung auf ein anderes Modellierungswerkzeug (z. B. DSHplus) sollte von der Architektur unterstützt werden. Dabei sollte die Architektur eine Liste von Anforderungen spezifizieren, die ein zu verwendendes Modellierungswerkzeug zur Verfügung stellen muss.	H / H

Fortsetzung auf der nächsten Seite

Szenario	**Kommentar**	**Bedeutung / Schwierigkeit**
R6 Die von einem Kunden geforderte Einhaltung der Norm IEC 61508 kann innerhalb von drei Monaten erfüllt werden.	Eine Zertifizierung nach IEC 61508 wird für das Prototypen System nicht zu realisieren sein, da eine Änderung der Software jederzeit beliebig vom Kunden vorgenommen werden kann. Aber es besteht Interesse daran, Seriensteuergeräte mit Seriencode später zertifizieren zu lassen.	L / H
R7 (S20) Der Kunde benötigt zur Realisierung der Regelfunktion zeit- und winkelbasierte Ausführungsstränge. Diese kann er durch einheitliche Schnittstellen nutzen.	Grundsätzlich sind für die Steuergerätesoftware mehrere parallel laufende Regler denkbar, die sowohl winkel- als auch zeitbasiert sein können.	M / H
R8 (S21) Bei der Entwicklung soll unter Berücksichtigung von Datenkonsistenz und Zeitanforderungen die Hardware gut ausgenutzt werden	Die Softwarearchitektur soll eine Unabhängigkeit von der verwendeten Hardware ermöglichen. Dies muss jedoch so gestaltet sein, dass jede Hardware mit Ihren Stärken genutzt wird, d. h. für einen Floating Point Prozessor muss Floating Point Code, und für Integer Prozessoren darf nur Integer Code erzeugt werden. Weiterhin sollen alle spezifischen Eigenschaften der Hardware in einer Treiberschicht abstrahiert werden.	H / M

Fortsetzung auf der nächsten Seite

Szenario	Kommentar	Bedeutung / Schwierigkeit
R9 Kunde fordert verschiedene Hardwaremerkmale an. Die VEMAC liefert diese unter Beibehaltung des Gesamtsystems nur durch Tausch der HW und der betroffenen SW Teile.	Dies könnte z. B. durch alternative Library-Bausteine realisiert werden.	H / L
R10 Der Kunde hat die Möglichkeit, ohne manuelle Eingriffe in den erstellten Modellcode einen Seriencode für eine Motorsteuerung abzuleiten.	Dabei sollte die Architektur auch spezifizieren, was einen Seriencode von Entwicklungscode unterscheidet.	M / M
R11 Der Kunde will das Steuergerät in Serie produzieren und muss mit dem automatisch erzeugtem Code die gesetzlichen Vorgaben erfüllen.		L / H
R12 Um von der Modellierungsschicht zum ausführbaren Code für die Hardware zu gelangen, soll eine möglichst weitgehend automatisierte Codegenerierung und Kompilierung realisiert werden.		H / L

Szenarien zur Parametrierung

In diesen Szenarien werden die Anwendungen und Anforderungen zur Parametrierung der Regler dokumentiert.

Szenario	Kommentar	Bedeutung / Schwierigkeit
P1 Kunde möchte das Steuergerät und die Prototypenplattform vor dem Einbau selbst parametrieren bzw. kalibrieren.	Dieses Szenario ist sehr wichtig. Diese Funktionalität soll über eine A2L Schnittstelle integriert werden.	H / H

Fortsetzung auf der nächsten Seite

Szenario	Kommentar	Bedeutung / Schwierigkeit
P2 Ein Kunde übernimmt die entwickelte Steuerung und ändert den Datenstand so, dass ein Schaden entsteht. Die Datenmanipulation kann nachgewiesen werden.	Dieses Szenario greift erst, wenn aus dem Rapid Control Prototyping System ein Seriensteuergerät abgeleitet wird.	L / H
P3 Ein Kunde verwendet ein eigenes Applikationswerkzeug. Der Zugriff über eine eigene Schnittstelle muss möglich sein.	Durch Unterstützung des A2L Protokolls ist eine weitgehende Austauschbarkeit des Applikations- und Kalibriertools gegeben. Der nachträgliche Einbau eines proprietären Kundenprotokolls sollte jedoch möglich sein.	M / M
P4 Der Kunde möchte ein Applikationssystem mit ASAM / MCD Schnittstelle benutzen.	Um eine Einbindung des RCP-Systems in die ASAM / MCD (ASAM AE) Umgebung zu ermöglichen muss in der Modellierungsebene ein A2L Datenexport vorgesehen werden. Auf der Embedded Software Seite muss eine ASAP Schnittstelle vorgesehen werden um Daten während des laufenden Betriebs des Steuergerätes auslesen und ändern zu können. Dieser Datenaustausch muss konsistent erfolgen und möglichst performant sein.	H / H

Szenarien zur Modellierung

Für die Modellierung werden spezielle Anforderungen formuliert. Diese betreffen die Anwendung der Regelung und die Realisierung der Schnittstellenkomponenten.

Szenario	**Kommentar**	**Bedeutung / Schwierigkeit**
M1 Ein neues Projekt ermöglicht die Wiederverwendung bestehender Module, die ohne Seiteneffekte aus einer Bibliothek genommen und verwendet werden können.	Dieses Szenario spiegelt genau die Kernfunktionalität wieder, die die Architektur erreichen soll.	H / M
M2 Die Beschränkungen des Seriensteuergeräts können bei der Entwicklung modelliert und simuliert werden.	Die Simulation sollte die beiden geplanten, einzusetzenden Plattformen beinhalten. In diesem Fall der Integer Prozessor und der Floating Point Prozessor.	M / H
M3 Ein Erdgasmotor soll statt eines Benzinmotors gesteuert werden. Die HW-Schnittstelle kann getauscht werden, ohne Änderungen an der SW-Struktur vorzunehmen.	Das Erhalten der Grundstruktur sollte auch im Rahmen der Matlab Anwendung für den Kunden möglich sein.	H / L
M4 Ein Kunde kann die SW ohne extra HW testen. Das heißt, der Kunde kann ein von ihm erstelltes Reglermodell in der Simulationsumgebung in Co-Simulation mit einem Motormodell testen.	Wichtig ist hierbei, dass das Modell für die Simulation ohne Veränderungen (d. h. auch ohne Austausch der Schnittstellen) für die Codegenerierung benutzt werden kann.	H / M

Fortsetzung auf der nächsten Seite

Szenario	**Kommentar**	**Bedeutung / Schwierigkeit**
M5 Der Kunde soll während des Steuergeräteentwurfs das RCP-System als Datenerfassungsgerät nutzen können.	Das Steuergerät sollte während des Reglerentwurfs als Datenerfassungs- und Ausgabegerät für Matlab dienen können. Dabei werden Sensordaten auf der Hardware erfasst und an Matlab weitergegeben. Diese dienen als Eingangswerte für die Simulation. Die berechneten Ausgangswerte werden ebenfalls an die Aktuatorausgänge des Steuergerätes weitergegeben. (nicht-echtzeitfähige Model-in-the-loop Anwendung)	M / M
M6 Die Möglichkeit der Parametereinstellung kann in Abhängigkeit der Kundenkompetenz gewählt werden	Dies bezieht sich im Wesentlichen auf die Parametrisierbarkeit der Matlab-Library	M / M
M7 Dem Kunden wird die Möglichkeit gegeben, eigene Modelle auf unterschiedlichen Detaillierungsstufen zu erstellen.	Teile der Library muss der Kunde durch eigene Module ersetzen können. Die dazu notwendigen Schnittstellen soll die Architektur spezifizieren.	H / M
Fortsetzung auf der nächsten Seite		

Szenario	Kommentar	Bedeutung / Schwierigkeit
M8 Der Kunde möchte gleichzeitig mit mehreren Entwicklern an den Modellen der RCP Plattform arbeiten.	Die Funktionen des Rapid Control Prototyping Systems können z.B. als Library in der Modellierungsschicht abstrahiert werden. Damit wird den Endkunden die Möglichkeit gegeben eigene Modelle zu erstellen die aus den Basisfunktionen der Library bestehen. Für die Library soll die Architektur ein Konzept anbieten, um die Library in unterschiedliche Detaillierungsstufen auszuführen. Bibliotheksfunktionen unterschiedlicher Detaillierungsstufen sollen untereinander kompatibel sein.	M / H
M9 Die falsche Verwendung der Library durch den Kunden wird bei der Modellierung der Steuerungssoftware aussagekräftig angezeigt.	Dies bezieht sich auf die syntaktisch falsche Benutzung. Semantische Fehler können nicht erkannt werden.	M / M

Entwicklungsprozess

Diese Szenarien spiegeln die Anforderung der VEMAC und Anforderungen an den Entwicklungsprozess mit dem RCP-System wider.

Szenario	Kommentar	Bedeutung / Schwierigkeit
E1 Nach Abwerbung eines maßgeblich beteiligten Entwicklers, kann sich ein neuer Architekt anhand der Dokumentation innerhalb von vier Wochen einarbeiten.	Für dieses Szenario ist es hilfreich, dass derzeit bereits mehere Entwickler an dem Projekt arbeiten und daher bereits erhöhter Druck zur Kommunikation und Dokumentation herrscht.	H /M

Fortsetzung auf der nächsten Seite

Szenario	**Kommentar**	**Bedeutung / Schwierigkeit**
E2 Die Architektur erlaubt bei Bedarf eines größeren Entwicklungsteams die reibungslose Aufteilung auf Entwickler.	Dies ist ein wichtiger Aspekt, der im Moment schon Anwendung findet. Die Entwicklung des Systems wird im Moment von vier Personen übernommen: Ein Entwickler ist zuständig für Hardwaretreiber, ein Entwickler für die "Tool-Chain", zwei weitere Entwickler für die Grundfunktionen.	H / M
E3 Eine verteilte Entwicklung des Frameworks muss unter Berücksichtigung der Architektur ohne Betrachtung des Gesamtsystems möglich sein.		H / L
E4 Die Softwareentwickler sind in der Lage mit geringem Aufwand Treiber, Teile des Frameworks und Teile der Library zu tauschen.		H / M
E5 Die Entwickler sollen nach Abschluss der Entwicklung den erzeugten Quellcode verstehen, ändern, finden, nachvollziehen und erweitern können.	Dies gilt auch bei personellen Änderungen im Entwicklungsteam.	H / M
E6 Neue Versionen der Werkzeuge in der Tool-Chain müssen ohne großen Aufwand eingebunden werden können.		H / M
E7 Veränderungen des Entwicklerteams führen nicht zu einer Verlängerung des Projekts.		H / M

5.2.3. Nicht-funktionale Anforderungen

Die Anforderungserfassung wurde mit einer großen Anzahl von Szenarien dokumentiert. Um eine Analyse der Anforderungen durchzuführen, wurden die Anforderungen geordnet. Die Analyse der nicht-funktionalen Anforderungen erfolgt mit Hilfe eines Qualitätenbaums nach [5, 7].

Dieser Qualitätenbaum ist in Abbildung 5.1 abgebildet. Grundlage des Baums sind nicht-funktionale Qualitäten. Diesen wurden die verschiedenen Szenarien zugeordnet. Die Relevanz der Szenarien läßt sich an der Farbe erkennen.

Der Qualitätenbaum beinhaltet alle Szenarien, die im Rahmen von Befragungen und Workshops erarbeitet wurden. Die Färbung der umgebenden Boxen gibt an, welche Wichtigkeit die einzelnen Szenarien haben. Die Qualitäten wurden in fünf Gruppen unterteilt. Anhand der Nummerierung lässt sich die Zugehörigkeit ableiten. Die Reihenfolge der Szenarien innerhalb des Baums hat keine Bedeutung. Bei Betrachtung des Baums fällt auf, dass die Szenarien, die einer Qualität zugeordnet wurden, eine weitgehend einheitliche Wichtigkeit und Relevanz haben. Daraus folgt, dass die Relevanz der einzelnen Qualität genau bestimmt werden kann.

Aus Platzgründen sind die Szenarien nur durch die entsprechenden Abkürzungen zugeordnet. Diese Abstraktion der Darstellung behinderte allerdings nicht die Bewertung der einzelnen Qualitäten.

Unter der Performanz wird die effektive Ausnutzung der Rechenkapazität verstanden. Diese Performanz wird durch einen möglichen Mehraufwand, welcher durch die Architektur entsteht, bewertet. Das zugehörige Szenario fordert die effektive Nutzung der zur Verfügung stehenden Hardware. Neben der effektiven Ausnutzung der Hardware ist aber auch eine effektive Nutzung der Arbeitszeit eines Ingenieurs im Szenario (R12) beschrieben. Diesem muss eine möglichst einfache Umgebung zur Verfügung gestellt werden.

Die Zuverlässigkeit des RCP-Systems ist kein entscheidendes Kriterium, da es sich um ein Werkzeug in einer prototypischen Entwicklungsphase handelt. Ein wichtiger Punkt ist die Komplexität der zu implementierten Architektur. Dies ist gerade für ein kleines Unternehmen wichtig und drückt sich in der Zuordnung von drei wichtigen Szenarien bei der Time-to-Market aus.

Neben einer schnellen Realisierbarkeit ist auch eine *Modularisierbarkeit* für verschiedene Projekte wichtig. Dies bedeutet, dass ein entsprechender Zuschnitt der Funktionalität möglich sein muss. Diese Eigenschaft hängt mit den Eigenschaften *Wartbarkeit*, *Wiederverwendbarkeit* und *Schutz der Technologie* zusammen, welche alle den Entwicklungsprozess des RCP-Systems betreffen.

Die *Wartbarkeit* des RCP-Systems bezieht sich auf Änderungen an der Werkzeugkette und Hardware. Die Änderungen der Werkzeugkette kann zum einem durch den Austausch des Modellierungswerkzeugs, aber auch durch neue Versionen von verwendeten Werkzeugen kommen. Ein Beispiel ist das Werkzeug MaSi, welches sich

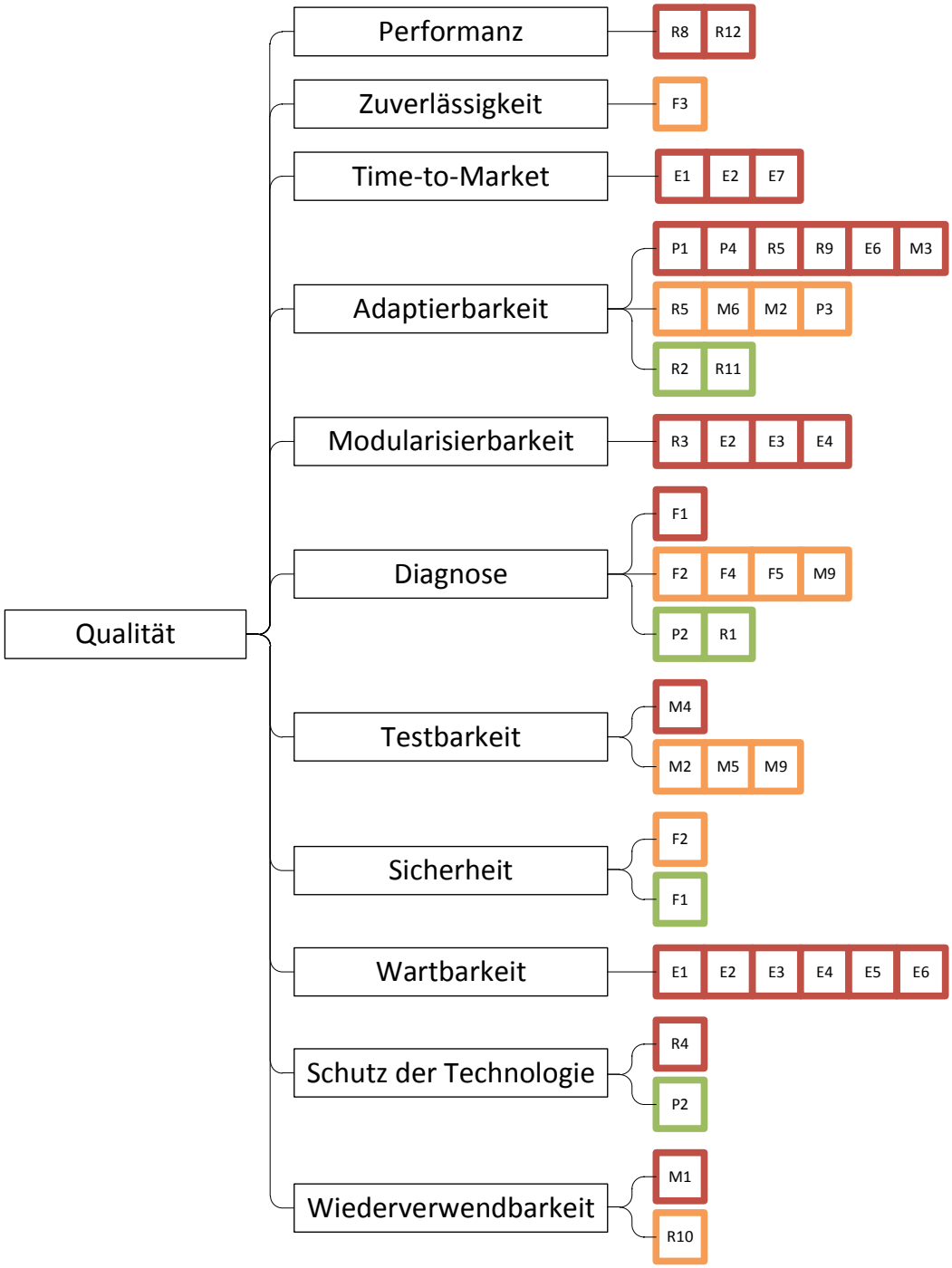

Abbildung 5.1.: Qualitätenbaum mit bewerteter Szenarienzuordnung

bei den Dateiformaten der Versionen stark unterscheidet. Die Modelle im Format neuerer Versionen können meist nicht in alten Versionen verwendet werden.

Die Wiederverwendbarkeit betrifft im Wesentlichen die Schnittstellen zu den Sensoren und Aktuatoren. Durch eine einfache Integration muss ein einfaches Setzen von Aktuatoren und Auslesen von Sensoren auf Modellebene möglich sein. Als Kernkompetenz des Unternehmens wird das Wissen gesehen, wie Sensoren und Aktuatoren elektrisch an die Hardware anzuschließen sind. Dazu gehören auch die notwendige Treibersoftware um die Informationen verfügbar zu machen. Dieses Wissen gilt es zu schützen.

Neben den Anforderungen, die den Entwicklungsprozess betreffen, ist die *Adaptierbarkeit* wichtig für die Architektur. Die Adaptierbarkeit betrifft zum Einen die Modelle auf dem RCP-System, zum Anderen aber auch die Verwendung möglicher Sensoren und Aktuatoren.

5.2.4. Zusammenfassung der Anforderungen

Das wichtigste Argument für das RCP-System wird in der folgenden Aussage zusammengefasst:

> Wir wollen dem Ingenieur ermöglichen, mit dem Rapid-Control-Prototyping System zu spielen, ohne dass er sich um den Anschluss von Sensoren oder Aktuatoren Gedanken machen muss.

Diese Aussage wurde vom Chef der VEMAC über das RCP-System getroffen. Die analysierten Qualitäten spiegeln diese Aussage wider. Die Analyse aufgrund des Qualitätenbaums zeigt, dass die wichtigsten Anforderungen folgende sind:

- Adaptierbarkeit
- Modularisierbarkeit
- Wartbarkeit
- Time-to-Market

Diese sind im Qualtitätenbaum mit vielen Szenarien belegt. Die einzelnen Qualitäten lassen sich bei einer genauen Betrachtung so zusammenfassen, dass die Benutzer in der Lage sein müssen, das RCP-System zu benutzen ohne Details zu den verwendeten Sensoren und Aktuatoren zu kennen.

Die Verwendung verschiedener Sensoren und Aktuatoren wird vor allem durch die Adaptierbarkeit ausgedrückt. Die beschriebenen Szenarien betreffen die technische Anforderung zur Parametrierung des Modells. Zusätzlich wird die Forderung gestellt, dass Sensoren und Aktuatoren einfach ins Modell integriert werden können. Dazu soll dem Ingenieur die Möglichkeit geboten werden, auf das vorhandene Wissen über

den Anschluss von Sensoren und Aktuatoren zuzugreifen. Die Nutzung des Wissens darf aber nicht das Offenlegen der verwendeten Technologie beinhalten.

Das zweite wichtige Ergebnis, dass sich aus der Analyse ergibt ist, die Unterstützung der Entwicklung und des Entwicklungsprozesses von dem RCP-System bei der Herstellerfirma. Ein wichtiger Aspekt ist dabei, die Entwicklung des Systems beim projektspezifischen Entwicklungsprozess zu unterstützen und die Abhängigkeit von einzelnen Entwicklern zu reduzieren.

Eine wichtige Anforderung, die eine hohe Wiederverwendbarkeit von entwickelten Software ermöglicht, ist die Modularisierung. Die projektgetriebene Entwicklung hat zur Folge, dass sich die resultierende Software stark unterscheidet. Eine Modularisierung der entwickelten Software in unabhängige trennbare Module ist die Voraussetzung, um eine Wiederverwendbarkeit in den unterschiedlichen Projekten zu erreichen. Zusätzlich unterstützt die Modularisierung die angestrebte Wartbarkeit. Die entwickelten Softwareteile müssen auch bei der fortlaufenden Weiterentwicklung verwendbar und weiterentwickelbar bleiben.

Die Qualität Time-to-Market spiegelt den innovativen und flexiblen Aspekt kleiner Unternehmen wider. Diese sind sehr flexibel bei der Art der angenommenen Aufträge. Dies bedeutet, dass verschiedenste Aufträge (aus unterschiedlichen Bereichen) angenommen werden, wenn die geforderte Technologie entwickelt werden kann, und wenn entsprechende Kapazitäten vorhanden sind. Die Umsetzung des Auftrags erfolgt meist nicht vordringlich unter dem Aspekt der vorhandenen Architekturansätze, sondern unter der Maßgabe der schnellen Umsetzung.

Insgesamt zeigt sich, dass die Nutzung von vorhandenem Wissen und die Wiederverwendung die zentralen Begriffe sind. In der Literatur werden die Verfahren der Wiederverwendung über verschiedene gleichartige Produkte hinweg mit Produktlinienansätzen abgebildet. Produktlinienansätze gehen von einer gemeinsamen Plattform und definierten Variabilitätspunkten aus. Diese definieren, an welchen Stellen die Variabilität eingebracht werden kann. Kleine Unternehmen haben allerdings nicht die Möglichkeit eine solche gemeinsame Plattform zu definieren, da die Weiterentwicklung sehr stark von den bearbeiteten Aufträgen abhängt. Die Variationspunkte können nicht festgelegt werden.

Das Vorgehen der Produktlinienansätze widerspricht der wichtigen Flexibilität von kleinen Unternehmen. Dort ist es nicht möglich, einen gemeinsamen Kern einer Plattform festzulegen. Der Aufwand und die Komplexität einer flexiblen Lösung sind zu groß. Im folgenden Kapitel wird daher ein Ansatz vorgestellt, welcher die Möglichkeit eines flexiblen Variantenmanagements bietet. Dazu wird auf eine gemeinsame Plattform verzichtet und ein auf Metamodellierung basierendes Architekturkonzept vorgestellt.

5.3. Architektur einer Hardwareabstraktionsschicht für eine modellbasierte Entwicklung

Die Anforderungen aus dem vorherigem Kapitel werden jetzt zu einer Architektur umgesetzt. Das Konzept für die Architektur besteht aus mehreren Teilen und umfasst verschiedene Komponenten und deren Beziehungen, die sich je Einsatzszenario unterscheiden. Die Aspekte des Konzepts werden durch

- eine Beschreibung der Werkzeugkette und deren Interaktion,
- Erläuterungen der Datenmetamodelle und
- Komponentendiagramme in verschiedenen Einsatzszenarien

verdeutlicht. Um das Umfeld des Konzepts einzuordnen, werden erst die Systemgrenzen definiert. Das Architekturkonzept stellt einen Ansatz zur modellbasierten Entwicklung von eingebetteten Systemen vor. Die betrachteten eingebetteten Systeme interagieren über Sensoren oder Aktuatoren mit der Umwelt und haben einen regelnden oder steuernden Einfluss (sie führen Regler aus).

Die Funktionalität wird in der Entwicklungsumgebung MaSi durch Modelle vollständig beschrieben und kann simuliert werden. Bei der Modellierung können domänenspezifische Sprachen und Verfahren eingesetzt werden. Die Modelle werden automatisch in Quelltexte übersetzt.

Das Architekturkonzept betrachtet die Einbindung der Sensoren und Aktuatoren in die Modellierungsebene des Reglers. Zusätzlich wird mit Hilfe von Verwaltungselementen aus der Produktlinienentwicklung die Variabilität der Schnittstellen ausgedrückt. Im Folgenden werden neben der logischen und technischen Systemarchitektur die komplette Werkzeugkette, die Datenmodelle und die Anwendungsszenarien dargestellt.

Die Grundidee der Architektur ist die Einführung einer Hardwareabstraktionsschicht (HAS), die die Sensoren und Aktuatoren auf physikalische Größen zurückführt. Diese HAS soll alle technischen Eigenschaften der Sensoren und Aktuatoren auf der Modellebene kapseln. Dazu wird der Zugriff auf Sensoren und Aktuatoren ausschließlich über physikalische Größen erfolgen.

Durch die HAS und die komplette Kapselung der technischen Eigenschaften ist es möglich, die Sensoren oder Aktuatoren ohne Änderungen am Modell auszutauschen. Das Prinzip soll anhand eines Geschwindigkeitssensors verdeutlicht werden. Auf Modellebene erfolgt die Integration nur die über eine Schnittstelle, die eine Geschwindigkeit zur Verfügung stellt. Die Messung der Geschwindigkeit kann von verschiedenen Sensoren erfolgen, die vollkommen unterschiedliche Messkonzepte beinhalten. Bei Impulssensoren erfolgt z. B. die Messung über die Bildung einer Zeitdifferenz, die zwischen zwei Impulsen verstrichen ist. Der Rückzug auf physikalische

Größen bedingt, dass jeder Schnittstellenkomponente neben dem Hardwaretreiber eine zusätzliche Funktion zur Umrechnung von gemessenen auf physikalische Größen zugeordnet wird.

Um die technische Trennung von Modell und HAS komplett zu vollziehen, wird zusätzlich eine getrennte Konfiguration mit Parametern der Schnittstelle eingeführt. Diese enthält Parameter, die einer abstrakten Modellrepräsentation einen konkreten Sensor zuordnen. Die Parameter enthalten alle Informationen, die zum Betrieb der realen Schnittstellenkomponente notwendig sind. Zusätzlich werden Informationen für die Simulation der Komponente hinterlegt, um den Anforderungen der Regelungstechniker nach einer einfachen Simulierbarkeit des Modells zu genügen.

Die Simulation des Modells ist eine wichtige Anforderung der Regelungstechnik an eine Entwicklungsumgebung. Das Ziel der hier vorgestellten Architektur ist das Ermöglichen einer Simulation ohne aufwändige Änderungen und Verbesserung der Genauigkeit der Simulationsergebnisse. Um dies zu erreichen, wird der HAS situationsabhängiges Verhalten zugeordnet. Dies bedeutet konkret, dass die HAS die Funktionalität sowohl für die *Simulationsphase* als auch für die *Quelltextgenerierungsphase* enthält. Dadurch ist es möglich, eine einheitliche Schnittstelle zu definieren, die auf der Modellebene während der Entwicklung eines Reglers nicht mehr ausgetauscht werden muss.

Während der Simulationsphase stellt die HAS simulierte Ergebnisse der verwendeten Aktuatoren und Sensoren zur Verfügung. Dazu werden wichtige Eigenschaften wie maximaler Mess- oder Stellbereich, Quantisierungsfehler und weitere Eigenschaften in die Simulation des Modells eingebracht. Um die Simulationsergebnisse der HAS im Modell berücksichtigen zu können, müssen die Sensoren und Aktuatoren im geschlossenen Regelkreis betrieben werden können. Dies bedingt, dass sie auf der Modellebene jeweils Ein- und Ausgänge haben. So werden im Modell die Stellwerte an die Aktuatoren gegeben, dort durch die Simulation der HAS angepasst und anschließend weiter an die Strecke des Modells gegeben. Analog werden die Ergebnisse der Strecke durch die Simulation der Sensoren angepasst und weiter an den Regler durchgereicht.

Das zweite Ziel der HAS ist die einfache Bedienung des RCP-Systems. Um dies zu erreichen, sollen jegliche Änderungen am Modell beim Übergang von Simulations- und Quelltextgenierungsphase vermieden werden. Beim Übergang zwischen den beiden Phasen werden überlicherweise zwei Anpassungen vorgenommen:

1. Die Strecke wird aus dem Modell entfernt.

2. Die Sensoren und Aktuatoren werden dem Modell hinzugefügt und mit dem Regler verknüpft.

Diese beiden Arbeitsschritte können nun automatisiert werden. Die Strecke kann mit Hilfe der HAS-Komponenten auf der Modellebene leicht identifiziert werden. Alle

Modellelemente, die zwischen Aktuatoren und Sensoren liegen sind zwingenderweise Streckenelemente und müssen bei der Quelltextgenerierung nicht berücksichtigt werden.

Das Hinzufügen von Sensoren und Aktuatoren ist nun ein Teil der Modellierung. Dies bedeutet zwar in dieser Phase einen erhöhten Aufwand, dieser bringt aber direkt Vorteile, durch die Verbesserung der Simulationsergebnisse.

Die letzte Komponente der Architektur ist eine Verwaltung der Schnittstelle über ein Featuremodell. Die Schnittstellenkonfiguration erfolgt über eine Konfiguration des Featuremodells, welches alle verfügbaren Sensoren enthält und die Konfiguration sowohl auf das Modell als auch auf die Schnittstellenparameter anwendet. Wichtig ist dabei, dass ein Austausch einer HAS-Komponente keine Änderung am Modell nach sich zieht. Es sind aber Änderungen bei der Simulation des geschlossenen Regelkreises zu erwarten, da die neuen Eigenschaften der Komponente berücksichtigt werden.

Das Konzept der Architektur kann durch die folgenden Punkte beschrieben werden:

- Einführung einer Hardwareabstraktionsschicht (HAS) zur vollständigen Trennung der physikalischen und technischen Schnittstelle.
- Verwendung einer Schnittstelle mit physikalischen Größen auf Modellebene
- Einführung einer separaten Konfiguration zur Speicherung von Informationen der Schnittstelle
- Einführung einer Simulation für alle HAS-Komponenten, um die Simulation auf Modellebene zu verbessern und eine Quelltextgenerierung ohne Modelländerung zu ermöglichen.
- Einführung eines Featuremodells, um die Auswahl der Schnittstelle konsistent auf das Implementationsmodell und die Schnittstellenparameter anzupassen.

5.3.1. Logische Systemarchitektur

Die logische Systemarchitektur wird von der Anforderungsanalyse abgeleitet. Das Ziel der logischen Systemarchitektur ist es, in einem Funktionsmodell die Zusammenhänge abstrakt darzustellen [40]. Um eine verständliche logische Systemarchitektur zu erreichen, wird eine Gruppierung und Strukturierung der Funktionalität unabhängig von der realen Implementierung vorgenommen [39].

Die Abbildung 5.2 zeigt die logische Systemarchitektur. Die dargestellten Blöcke repräsentieren Funktionen, die aus der Anforderungsanalyse hergeleitet wurden. Die Übersicht 5.2 zeigt die zwei verschiedenen Szenarien *Simulation* **A** und *Modellausführung* **B**. Beide werden mit der zentralen Konfiguration des *Sensor- und*

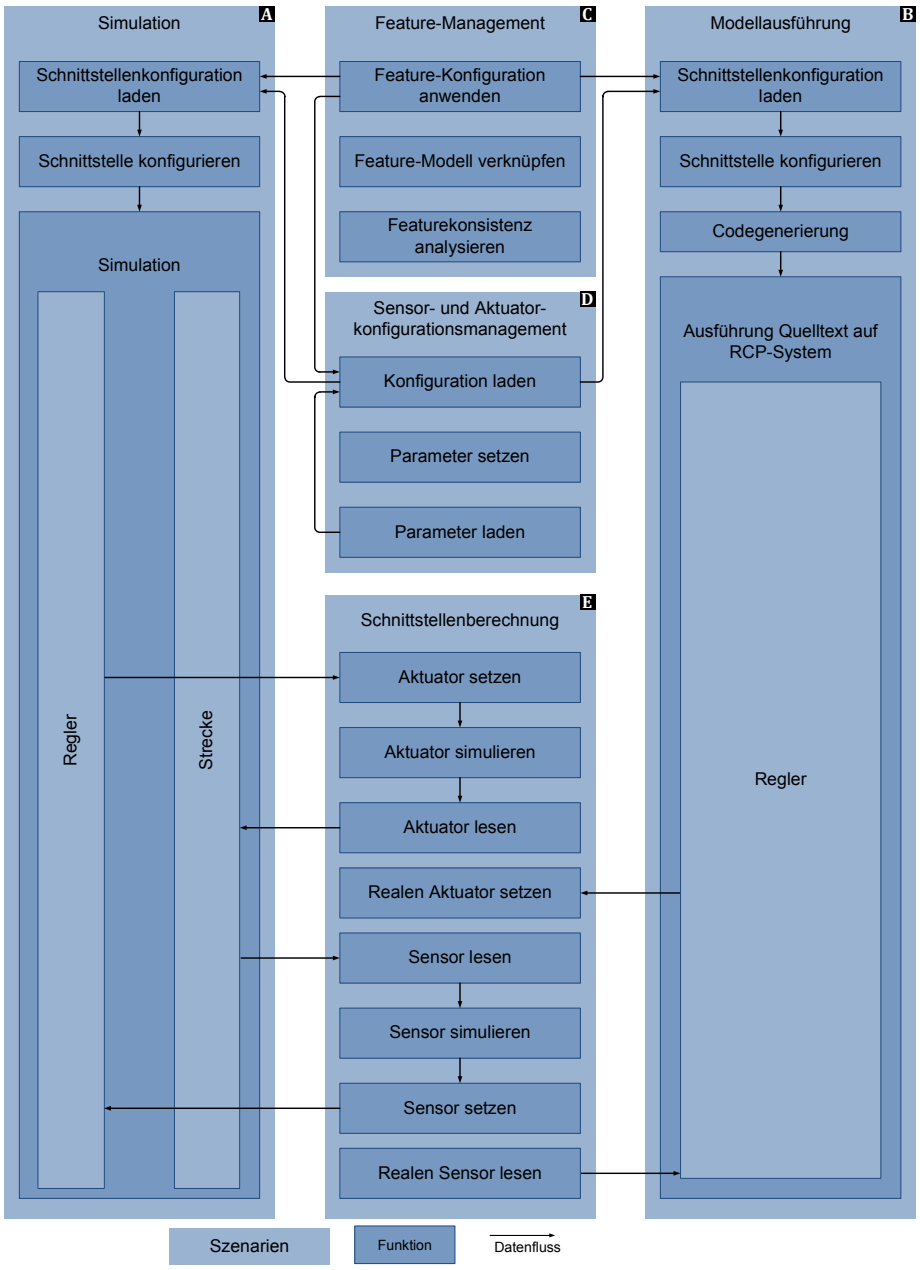

Abbildung 5.2.: Logische Systemarchitektur des Rapid-Control-Prototyping Systems mit Hardwareabstraktionsschicht

Aktuatorkonfigurationsmanagements C gesteuert. Die Konfiguration wird wiederum

vom *Feature-Management* D verwaltet. Die Schnittstellen der HAS sind im Block *Schnittstellenberechnung* B dargestellt.

Im Folgenden wird das Verhalten während der Simulation A und der Modellausführung B beschrieben. Während der Simulation werden zuerst Schnittstellenkonfigurationen der HAS-Komponenten geladen. Die Konfiguration besteht aus der Zuordnung von konkreter HAS-Komponente zu einem Schnittstellenelement der Modellebene und aus den Parametern der HAS-Komponenten.

Die Zuordnung zwischen HAS-Komponente und dem Schnittstellenelement erfolgt durch die Anwendung der Feature-Konfiguration C. Anschließend werden aufgrund der Feature-Konfiguration die Parameter geladen, die das Verhalten der HAS-Komponente während der Simulation beschreiben. Typische Parameter, die das Verhalten beschreiben sind Quantisierungseigenschaften, Begrenzungen und Totzeiten.

Der Ablauf einer Simulation ist in der Abbildung 5.2 B dargestellt. Der Informationsfluss zwischen Aktuator und Sensor unterscheidet sich. Der Aktuator bekommt die vom Regler während der Simulation ermittelten Stellwerte, simuliert diese und gibt die angepassten Werte anschließend an die Strecke. Diese wird dann aufgrund des Stellwerts und weiterer Faktoren weiter simuliert. Der Informationsfluss eines Sensors ist genau entgegengesetzt. Die Reaktion der Strecke gibt der Sensor nach der Simulation der Sensoreigenschaften an den Regler weiter.

Während der Modellausführung (siehe Abbildung 5.2 B) des RCP-Systems wird auf die gleichen Konfigurationskomponenten zurückgegriffen. Dies bedeutet, dass die Konfiguration der Schnittstelle von der Feature-Konfiguration aber auch von den Schnittstellenparametern beeinflusst wird. Es wird exakt die gleiche Feature-Konfiguration verwendet, da die selben HAS-Komponenten verwendet werden (nur ersetzt durch die realen Sensoren und Aktuatoren). Allerdings ändern sich die verwendeten Parameter. Sie enthalten nun Informationen, welche für den realen Betrieb der Schnittstelle benötigt werden. Diese Parameter sind zum Beispiel Hardware-Adressen oder Konfiguration des maximalen Messbereichs.

Die so konfigurierten Schnittstellen werden mittels Quelltextgenerierung zu ausführbarem Quelltext übersetzt und auf dem RCP-System ausgeführt. Der Zugriff auf die Sensoren und Aktuatoren erfolgt dann nicht mehr über die simulierten Schnittstellen, sondern über die Treiber der entsprechenden Sensoren und Aktuatoren.

5.3.2. Technische Systemarchitektur

Die Darstellung der technischen Systemarchitektur beinhaltet die erste konkrete Lösung im Architekturentwurf. Nach [39] besteht die technische Systemarchitektur neben der Zuordnung von realen Komponenten und logischen Funktionen auch aus den Kommunikationsverbindungen und deren Verwendung. Im vorliegenden Fall werden Teile der Kommunikationsstrukturen weiterhin abstrakt dargestellt, da die

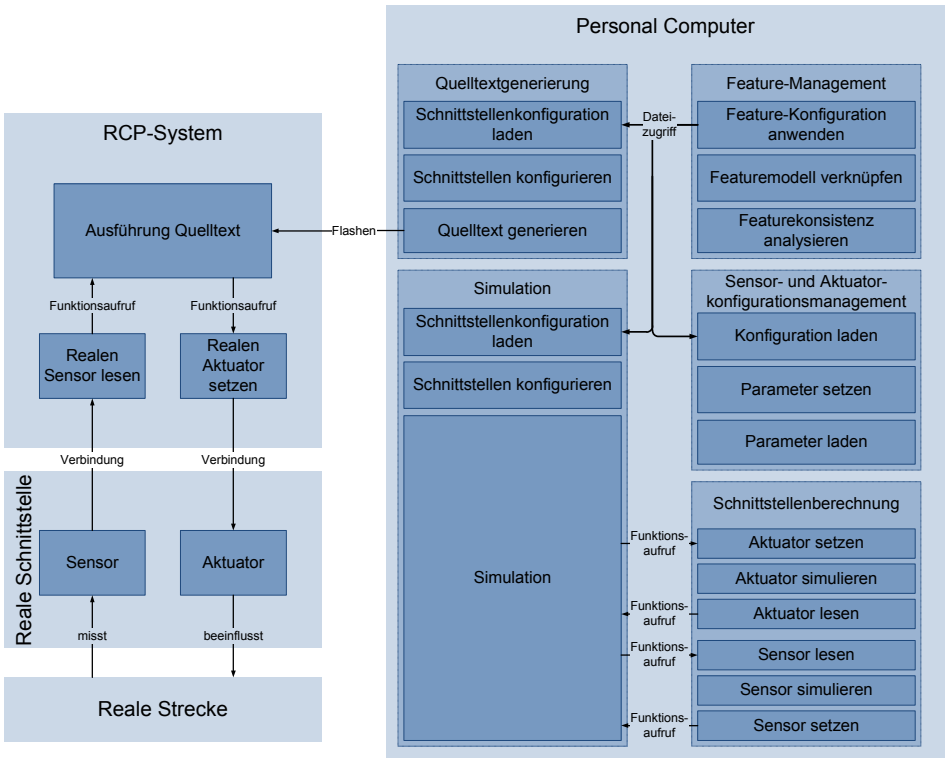

Abbildung 5.3.: Technische Systemarchitektur der Hardwareabstraktionsschicht

konkrete Realisierung erst bei der konkreten Wahl der realen Komponente festgelegt wird.

Die Abbildung 5.3 zeigt einen Überblick über die Verteilung der Funktionalität auf die Hardware und deren Verbindung. Die Rechtecke stellen Funktionen dar. Kommunikationsverbindungen zwischen Funktionen werden durch Pfeile dargestellt. Die verwendete Hardware besteht aus dem

Personal Computer (PC), der zur Entwicklung, Simulation und Quelltextgenerierung eingesetzt wird, dem

RCP-System, das die Ausführung des generierten Quelltextes übernimmt, der

Realen Schnittstelle, welche aus den Sensoren und Aktuatoren besteht und der

Realen Strecke, entspricht dem zu regelndem System.

Auf dem PC werden alle Funktionen ausgeführt, die während der Entwicklung / Modellierung des Reglers benötigt werden. Dazu gehören das Feature-Management,

das Konfigurationsmanagement sowie die Schnittstellenberechnung. Die Funktionalitäten zur Schnittstellenberechnung werden während der Simulationsphase benötigt.

Zusätzlich erfolgt auf dem PC auch die Quelltextgenerierung. Dabei wird aus dem Modell des Reglers ausführbarer Quelltext generiert. Die Konfiguration der Schnittstelle erfolgt über das Feature-Modell und Parameter, die spezifisch für das RCP-System sind.

Auf dem RCP-System wird der Quelltext ausgeführt. Die Kommunikation mit den Schnittstellenkomponenten erfolgt über die integrierten Treiber. Zusätzlich werden die Messwerte für den Regler entsprechend umgerechnet, um die physikalischen Größen zu erhalten. Die elektrische Hardwareschnittstelle des RCP-System ist abstrakt definiert, da diese flexibel gestaltet ist und Änderungen unterliegt.

Die Sensoren und Aktuatoren wiederum messen die benötigten Werte der realen Strecke oder beeinflussen diese über die Aktuatoren. Eine direkte Rückgabe der Messwerte an den PC ist nicht vorgesehen. Eine weitere Schnittstelle des RCP-Systems für die Konfiguration von Reglerparametern ist aus Gründen der Übersichtlichkeit hier nicht dargestellt.

5.3.3. Konzeptueller Aufbau der Werkzeugkette

Die Reglerentwicklung eingebetteter Systeme basiert auf einer Modellierungsumgebung zur Erstellung von Modellen. Diese Modellierungsumgebung ist in der Lage, die Modelle zu simulieren und damit einen Eindruck der Funktionalität zu vermitteln. Eine Transformation des Modells in ausführbaren Quelltext ermöglicht eine schnelle Integration in das eigentliche Zielsystem.

In diesem Ansatz wird die Modellierungsumgebung mit Werkzeugen erweitert, die die Verwaltung der Schnittstellen übernehmen. Einen Überblick über die verschiedenen Werkzeuge ist in der Abbildung 5.4 zu finden. Der Überblick ist verallgemeinert dargestellt und kann durch verschiedene Werkzeuge umgesetzt werden. Die konkrete Umsetzung wird im Abschnitt 5.4 vorgestellt. Aus Abstraktionsgründen ist nur der Datenfluss zwischen Datenmodellen und Werkzeugen dargestellt. Verknüpfungen der Datenmodelle untereinander sind in der Abbildung 5.4 ausgelassen.

Der Ansatz kann in drei Ebenen gegliedert werden. In der obersten Ebene werden grundsätzliche Entscheidungen über die Schnittstellen getroffen. Die Verwaltung dieser Entscheidungen erfolgt über das FM. Die Modelle können mit einem graphischen Werkzeug erstellt und anschließend mit einem Konfigurationswerkzeug konfiguriert werden. Die Feature-Konfiguration ist eine Menge von Features, sodass gilt, dass jedes ausgewählte Feature in dieser Menge enthalten ist. Das FM ist auf das Projekt so zugeschnitten, dass es die verwendbaren Schnittstellen in einer abstrakten Beschreibung enthält. Die konkrete Implementierung der Schnittstellen wird als eine Verfeinerung der abstrakten Schnittstelle dargestellt.

Eine solche Darstellung und Verfeinerung ist in der Abbildung 5.5 am Beispiel

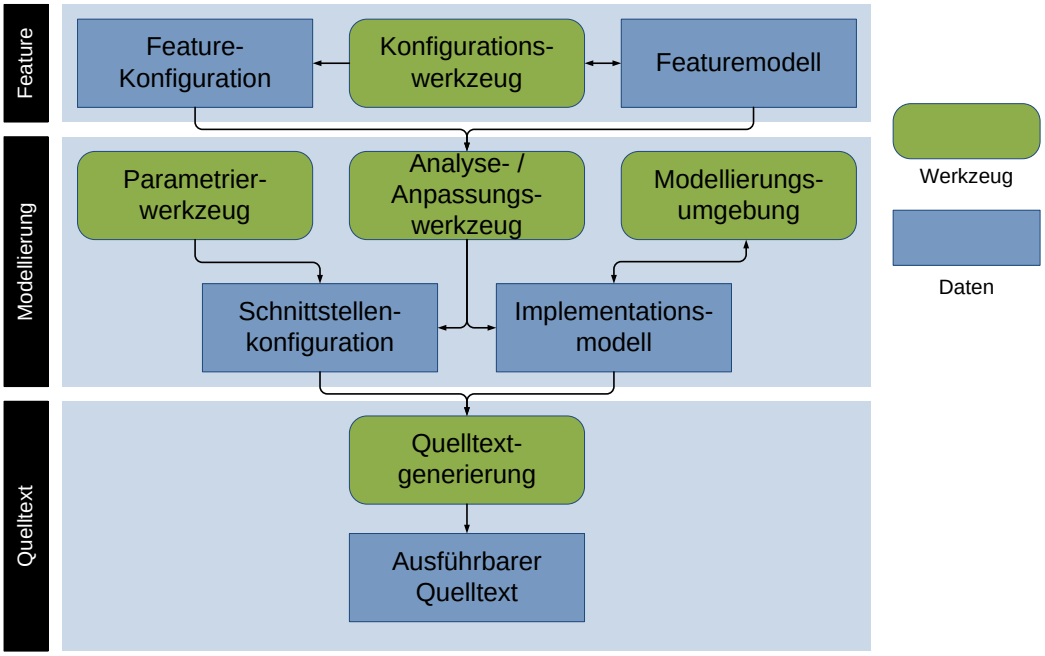

Abbildung 5.4.: Zusammenspiel der Werkzeuge beim Produktlinienkonzept für variable Schnittstellen

eines Geschwindigkeitssensors zu sehen. In diesem Fall können entweder ein *Implus Sensor Vorne* oder eine *Hall Impluse Sensor Hinten* die Geschwindigkeit messen. Da beide über eine *or*-Gruppe ins System integriert sind, können auch beide Sensoren ausgewählt werden, um die Messgenauigkeit oder den Messbereich zu erhöhen.

Das FM und die abgeleitete Konfiguration sind die Eingabe für das *Analyse-* und *Anpassungswerkzeug* der Modellierungsebene. In dieser Ebene wird die Modellierung der Funktionalität und die Parametrierung der Sensoren und Aktuatoren vorgenommen. Durch Modelltransformationen wird in einem ersten Schritt ein inititales Implementationsmodell und eine Schnittstellenkonfiguration abgeleitet. Zusätzliche Ableitungs- und Analyseschritte des Implementationsmodells werden im Kapitel 6 erläutert.

Die Modellierungsumgebung stellt die eigentliche Sprache zur Implementierung der Funktionalität zur Verfügung. In ihr ist es möglich, auf die abstrakten Schnittstellen der Aktuatoren und Sensoren zuzugreifen. Um eine Vermischung der Schnittstellen- und der Modellparameter zu vermeiden sowie die abstrakte Darstellung der Schnittstellen zu erhalten, wird ein weiteres *Parametrierungswerkzeug* der Schnittstellen zur Verfügung gestellt. Dieses Werkzeug ermöglicht die Eingaben und Verwaltung der spezifischen Sensor- und Aktuatorparameter. Diese werden zentral außerhalb

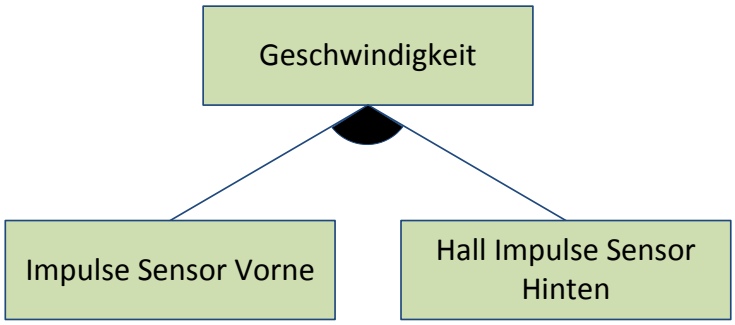

Abbildung 5.5.: Beispiel einer Schnittstellenverfeinerung mit konkreten Sensoren

des Implementationsmodells in der Schnittstellenkonfiguration gespeichert.

Auf der dritten Ebene der Quelltextgenerierung (siehe Abbildung 5.4) werden die Schnittstellenkonfiguration und das Implementationsmodell verwendet, um die entsprechenden Treiber der Schnittstellen zu laden und mit in die Übersetzung des Implementationsmodells zu integrieren. Das Ergebnis ist ausführbarer Quelltext, der direkt auf der Hardware des Zielsystems (RCP-System) ausgeführt werden kann. Der Informationsaustausch zwischen den Datenmodellen wird im folgenden Kapitel erläutert.

5.3.4. Datenmodelle

Innerhalb der Architektur werden mehrere Datenmodelle eingesetzt, um Konfigurationen abzuspeichern und diese weiter während des Entwicklungsprozesses zu nutzen. Die Datenmodelle der Konfigurationen wurden mittels Metamodellierung nach [18] mit dem EMF [14] definiert. Für das Architekturkonzept werden Metamodelle für FM, Simulinkmodelle, Schnittstellenkomponenten und die Verbindung zwischen Modellen benötigt. Die Metamodelle der Schnittenstellenkomponenten und der Verbindungen werden im Folgenden vorgestellt. Das Metamodell für FM wurde von einem vorhandenen Werkzeug übernommen. Das Metamodell der Implementierungsumgebung wird im Kapitel 6 erläutert.

In der Abbildung 5.6 wird das Metamodell einer Schnittstellenkomponente dargestellt. Diese kann entweder durch einen Sensor oder einen Aktuator implementiert werden. Die Typen einer Komponente werden in Enumerationen gespeichert und können durch Erweiterungen im Metamodell angepasst werden. Einer Schnittstellenkomponente ist ein Ausgangs- und ein Eingangswert zugeordnet. Diese entsprechen jeweils einer physikalischen Einheit. Um Erweiterbarkeit zu gewährleisten, existiert zusätzlich die Möglichkeit, beliebige Eigenschaften (*Property*) einer Schnittstelle zuzuordnen.

Die Parameter einer Komponente werden in Simulations- und Hardwarepara-

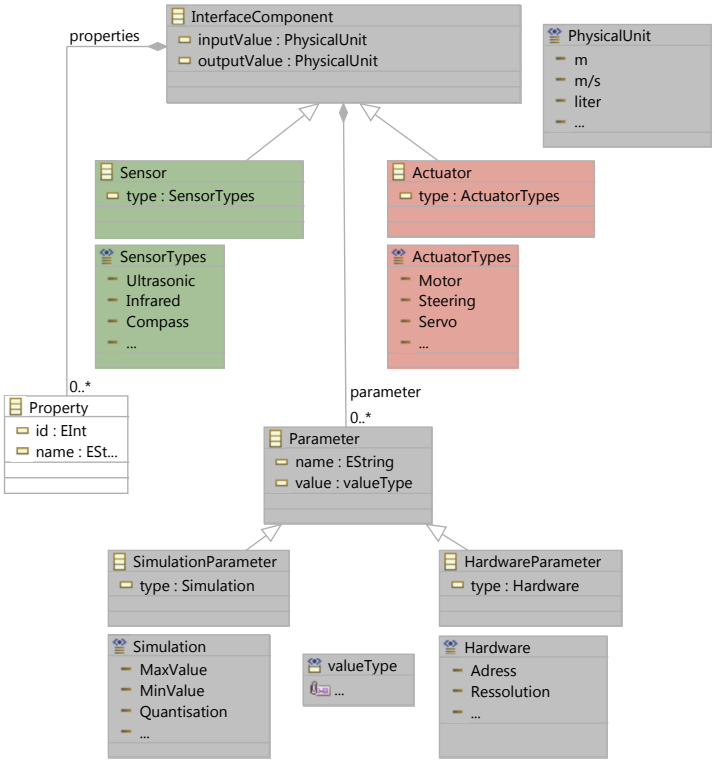

Abbildung 5.6.: Metamodell einer Schnittstellenkomponente

meter aufgeteilt. Diese haben einen Namen und eine Bedeutung, die über eine `type`-Eigenschaft festgelegt wird. Neben dem Namen haben die Parameter noch eine Belegung, die im `value` gespeichert wird. Der Datentyp des Parameters wird über den `valueType` bestimmt. Eine vollständige Konfiguration enthält alle Instanzen von Schnittstellenkomponenten gemäß des vorgestellten Metamodells in der Abbildung 5.6.

Um Verknüpfungen zwischen verschiedenen Modellen zu ermöglichen, existiert ein weiteres Metamodell, welches Referenzen auf die externen Schnittstellenkonfigurationen und Implementationsmodelle enthält. Es ist in der Abbildung 5.7 abgebildet. Das Verknüpfungsdokument (`MappingDocument`) enthält sowohl die Verknüpfung zwischen der Implementation und dem FM (siehe 5.7 `LinkingImplementation`) als auch die Verknüpfungen zwischen dem FM und der Schnittstellenkomponente (siehe 5.7 `LinkingComponent`).

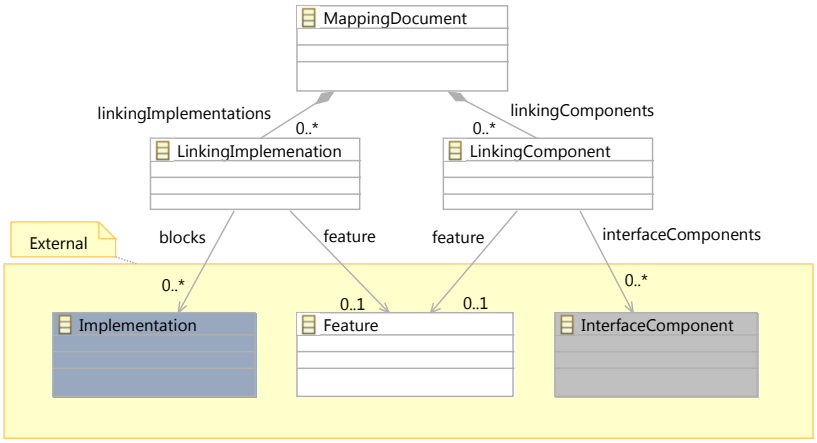

Abbildung 5.7.: Metamodell der Verknüpfungsmodelle

5.3.5. Konzept der Analyse- und Anpassungsszenarien

Das Konzept zur HAS erweitert die modellbasierte Entwicklung um Bausteine der Konfigurations- und Featureverwaltung. Diese müssen in den Entwicklungsprozess integriert werden. In diesem Kapitel werden die verschiedenen Nutzungsszenarien der HAS und die Unterstützung des Entwicklungskonzepts durch Analysen und Anpassungen definiert. Die Szenarien sind Teil eines gesamten Entwicklungsprozesses, welcher von Entwicklern oder Organisationseinheiten definiert wird [27].

Als Voraussetzung wird angenommen, dass eine vollständige Liste von Sensoren und Aktuatoren (gespeichert in einem FM) vorliegt, welche mit dem RCP-System zusammenarbeiten. Die notwendigen Konfigurationen, Treiber und Schnittstellen sind Bestandteil des RCP-Systems. Für die Einführung der Schnittstelle und die Anpassung werden hier die folgenden drei Fälle betrachtet:

1. Verwendung der HAS in einem neuen Projekt ohne vorhandene Grundlagen

2. Erweiterung der HAS durch Integration einer zusätzlichen Schnittstellenkomponente

3. Anpassung der Schnittstelle durch Austausch von HAS-Komponenten

Die folgenden Kapitel erläutern das Vorgehen und das Konzept zur Unterstützung des Entwicklers.

Einrichtung eines neuen modellbasierten Entwicklungsprojekts unter Verwendung der HAS

Im Folgenden wird das Vorgehen bei Einrichtung eines neuen Projektes mit dem RCP-System und der HAS beschrieben. Der Entwickler startet mit der Konfiguration der Schnittstelle. Dazu muss der Ingenieur festlegen, welche physikalischen Werte an der Strecke gemessen werden und wie die Strecke beeinflusst wird. Diese Informationen werden dann im FM für die Konfiguration benötigt. Zusätzlich werden im FM die Komponenten gewählt, welche die Schnittstelle real implementieren. Durch die Mechanismen des FMs spezifiziert der Entwickler, in welcher Beziehung die Komponenten stehen.

Anschließend kann der Entwickler die Generierung der Elemente der Modellierungsebene (siehe Abbildung 5.4) anstoßen. Das Ergebnis dieses Prozesses ist ein initiales Implementationsmodell, in dem die abstrakten Schnittstellenkomponenten enthalten sind. Als zweites Ergebnis wird eine Schnittstellenkonfiguration erzeugt, die Einträge für die realen Sensoren enthält. Diese sind noch nicht definiert und müssen vom Entwickler ausgewählt werden.

Erweiterung einer HAS-Komponente in einem vorhandenen Entwicklungsprojekt

Bei der Erweiterung der Schnittstelle auf Modellebene wird davon ausgegangen, dass eine Anpassung des FMs nicht notwendig ist. Zusätzlich wird ein existierendes Projekt als Voraussetzung angenommen. In diesem Fall ist es notwendig, eine neue Konfiguration des FMs abzuleiten. Die Konfiguration enthält dann die neue Schnittstellenkomponente.

In einem zweiten Schritt müssen das Implementationsmodell und die Schnittstellenkonfiguration angepasst werden. Die Anpassung der Schnittstelle kann wieder durch das Anpassungs- und Analysewerkzeug erfolgen. Die zu aktualisierenden Schnittstellen müssen dafür im FM vom Benutzer gekennzeichnet werden. Das Implementationsmodell und die Schnittstellenkonfiguration werden im Folgenden erweitert.

Anschließend kann der Entwickler die neue Komponente in seine Funktionalität integrieren und simulieren. Dazu muss er zusätzlich noch den Sensor parametrieren. Diese Parameter werden dann sowohl während der Simulation als auch während der Quelltextgenerierung verwendet.

Anpassung der Schnittstelle durch Austausch von HAS-Komponenten

Ein Teil der Entwicklung ist die Wahl der passenden Sensoren und Aktuatoren. Deren Verhalten hat einen großen Einfluss auf die Reglerparameter und muss dem entsprechend während der Simulation berücksichtigt werden. Ein Tausch dieser

Komponenten macht eine Änderung der Konfiguration notwendig. In diesem Fall gibt es zwei Szenarien. Zum einen ist es möglich, die Eigenschaft einer Komponente zu ändern, zum anderen kann die Komponente komplett ausgetauscht werden. Im ersten Fall ist eine Änderung der Schnittstellenparameter notwendig. Diese Änderungen werden automatisch während der Simulation und Quelltextgenerierung berücksichtigt.

Im zweiten Fall wird die Komponente komplett getauscht. Hierbei ist die Voraussetzung, dass sich die gemessene Größe nicht ändert. Ist dies der Fall, erfolgt eine Anpassung der Feature-Konfiguration, und hat zur Folge, dass in der Schnittstellenkonfiguration ein neuer Parametersatz geladen wird. Nach der Parametrierung der neuen Komponente sind keine weiteren Anpassungen notwendig. Insbesondere ist eine Änderung des Implementationsmodells nicht notwendig.

Um eine hohe Benutzerfreundlichkeit zu erreichen, sieht das Konzept vor, dass eine korrekte Konfiguration automatisch auf Implementierungsmodelle und Schnittstellenkonfiguration angewendet wird. Die Anpassungen des Implementierungsmodells erfordern eine Entfernung der Schnittstellenkomponente.

Die Schnittstellenkonfiguration auf Modellebene legt die reale Implementierung fest. Eine Konfiguration bestimmt nun auf dieser Ebene, welche der realen Komponenten (Sensoren oder Aktuatoren) verwendet werden.

5.4. Implementation der RCP-Architektur mit Hardwareabstraktionsschicht

Die Implementation des RCP-Systems mit der HAS besteht aus der Verzahnung verschiedener Werkzeuge und der Entwicklung von Werkzeugen und Datenmodellen. Die Datenmodelle werden im Abschnitt 5.3 vorgestellt. Die Auswahl der Modellierungsumgebung und weiterer Werkzeuge ist zum größten Teil durch die Randbedingungen vorgegeben.

5.4.1. Überblick der Werkzeuge

Ein Überblick der eingesetzten Werkzeuge ist in der Abbildung 5.8 zu sehen. Um das Konzept zu realisieren, werden bestehende Werkzeuge verwendet und mit selbst entwickelten kombiniert.

Die Featuremodellierung und Konfiguration ist mit zwei verschiedenen Werkzeugen realisiert worden, die sich alternativ einsetzen lassen. Das Werkzeug Pure::Variants [38] mit einer Erweiterung um Matlab / Simulinkmodell (SimMo) zu beeinflussen. Als Alternative wurde ein Werkzeug eingesetzt, das vollständig auf dem Eclipse Modeling Framework aufbaut. Dieses Werkzeug besteht aus einem Editor, um Featuremodelle aufzubauen und einem S2T2-Konfigurator [6] für das

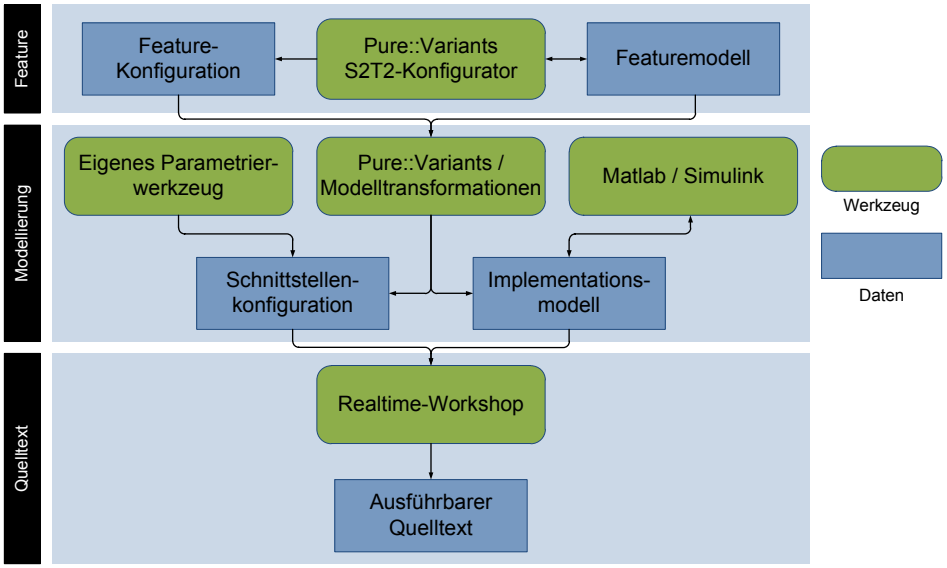

Abbildung 5.8.: Überblick über die verschiedenen Werkzeuge

Ableiten von Featurekonfigurationen dieser Modelle. MaSi wird als Modellierungsumgebung verwendet. Diese Umgebung wird im Bereich der eingebetteten Systeme und Reglerentwicklung als Quasi-Standard angesehen. Die HAS-Komponenten werden als MaSi-Bibliothek zur Verfügung gestellt. Als technische Realisierung nutzen die Schnittstellenkomponenten `S-Functions`. Eine `S-Function` ist ein Konstrukt, das von MaSi zur Verfügung gestellt wird, um spezifische Implementationen zu realisieren und in Modelle zu integrieren. `S-Functions` haben eine graphische Repräsentation und Schnittstellen zur Anbindung von Funktionalität in verschiedenen Repräsentationen (C-Code, Matlab Skript, SimMo und weitere). Zusätzlich können Daten von der Modellebene gelesen und geschrieben werden. Die vom Konzept geforderten Umrechnungen können hiermit während der Simulation vorgenommen werden.

Die Quelltextgenierung aus dem Modell muss passend zur Modellierungsumgebung gewählt werden. In diesem Fall wird der *Realtime-Workshop* verwendet. Dieser ermöglicht die Quelltextgenerierung direkt aus dem Modell für verschiedene Plattformen. Die Anpassung auf das RCP-System ist in diesem Fall nicht notwendig, da alle hardwarerelevanten Informationen entweder durch die HAS von der Modellebene abstrahiert werden oder später erst durch den Kompiler und ein Rahmenwerk in den ausführbaren Quelltext eingebracht werden.

Der Realtime-Workshop ermöglicht auch die direkte Integration der `S-Functions` in den Generierungsprozess. Die `S-Function` bietet dazu neben Schnittstellen während der Simulation auch Schnittstellen für den Quelltextgenerierungsprozess. Das

Verhalten einer `S-Function` kann während des Generierungsprozesses festgelegt und entsprechende Treiber integriert werden.

Um die Schnittstellenkomponenten zu konfigurieren, wird ein Verwaltungswerkzeug zur Verfügung gestellt. Das Werkzeug ermöglicht es die Parameter der Schnittstelle zu konfigurieren. Die Konfiguration wird bei jeder Simulation und bei jedem Quelltextgenerierungsprozess neu ausgelesen und angewendet. Durch das ständige Prüfen der Konfiguration kann sichergestellt werden, dass immer die neueste vorliegende Version wendet wird.

Die Verbindung zwischen der Feature- und Modellierungsebene wird durch das EMF realisiert. Modelltransformationen verwenden die verschiedenen Modelle als Eingabe und passen diese dann entsprechend der ausgewählten Konfiguration an.

5.4.2. Implementierung Hardwareabstraktionsschicht

In diesem Kapitel wird der Aufbau der HAS detailliert erläutert. In der Abbildung 5.9 ist ein Überblick des Aufbaus einer HAS-Komponente gezeigt. Die gesamte HAS setzt sich aus den Implementationen der verfügbaren Schnittstellenkomponenten zusammen.

Die Abbildung 5.9 zeigt, dass auf allen Ebenen eine Schnittstellenkomponente repräsentiert wird. Auf der Feature-Ebene als Feature in einer abstrakten (`Komponente`) und in einer konkreten Form (`Reslisierung 1` und `Realisierung 2`). Die Features sind verknüpft mit der Repräsentation auf Modellebene. Im Modell werden die Komponenten durch einen S-Function-Block realisiert. In der Konfiguration werden die Parameter als XML Eintrag abgelegt.

Die S-Function implementiert den Zugriff während Simulation und Quelltextgenerierung. Während der Simulation wird auf die abstrakte Simulationsberechnung zugegriffen. Diese benutzt die Konfiguration, um die Eigenschaften der konkreten Komponente zu bestimmen.

Auf der Quelltextebene wurde eine hierarchische Schichtenarchitektur implementiert. Die Schichten lassen sich in drei Ebenen einteilen. Die erste Schicht ist dazu notwendig, um die Umrechnung zwischen den Komponenten und der Hardwareabstraktionsschicht zu berechnen. Die zweite Schicht implementiert den Treiber für die Komponente. Diese Schicht kapselt die spezifische Implementierung für die Komponente. In der untersten Schicht werden Hardwaretreiber des Microcontrollers gekapselt. Diese Schicht kapselt die Schnittstelle des Mircocontrollers und ermöglicht so den einheitlichen Zugriff auf Busse, analoge und digitale Ein- und Ausgänge.

In der Abbildung 5.9 ist auf Quelltextebene zusätzlich die Unterscheidung zwischen Sensor und Aktuator erfolgt. Der Informationsfluss unterscheidet sich bei beiden Komponenten. Der Zugriff auf die Hardwareschnittstelle erfolgt über gemeinsame Treiber. Die bidirektionale Kommunikation für Sensoren ist notwendig, da die Abfrage von Informationen oftmals zuerst angestoßen werden muss. Auch für Aktuatoren

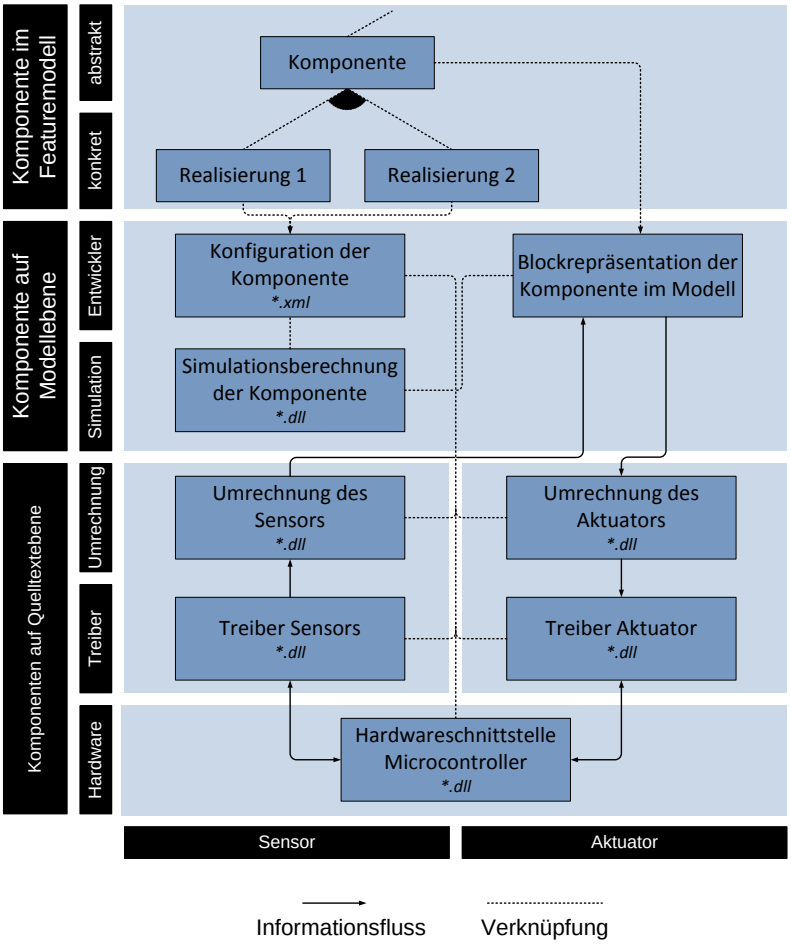

Abbildung 5.9.: Aufbau einer Schnittstellenkomponente der HAS

ist eine bidirektionale Kommunikation notwendig, weil dort beispielsweise über Protokolle eine Ansteuerung erfolgt.

Die Treiber und Simulationen sind in der Sprache C implementiert. Die Komponenten werden anschließend als Bibliothek (im DLL-Format [30]) zur Verfügung gestellt. Dadurch ist es möglich, den Schutz der Komponententechnologie zu gewährleisten.

5.4.3. Werkzeug zur Schnittstellenkonfiguration

Eine zentrale Komponente ist die Konfiguration der HAS-Komponenten auf Modellebene. Die Konfiguration erfolgt über eine graphische Benutzeroberfläche. Das Konfigurationswerkzeug bietet neben der Verwaltung der Parameter für Komponenten

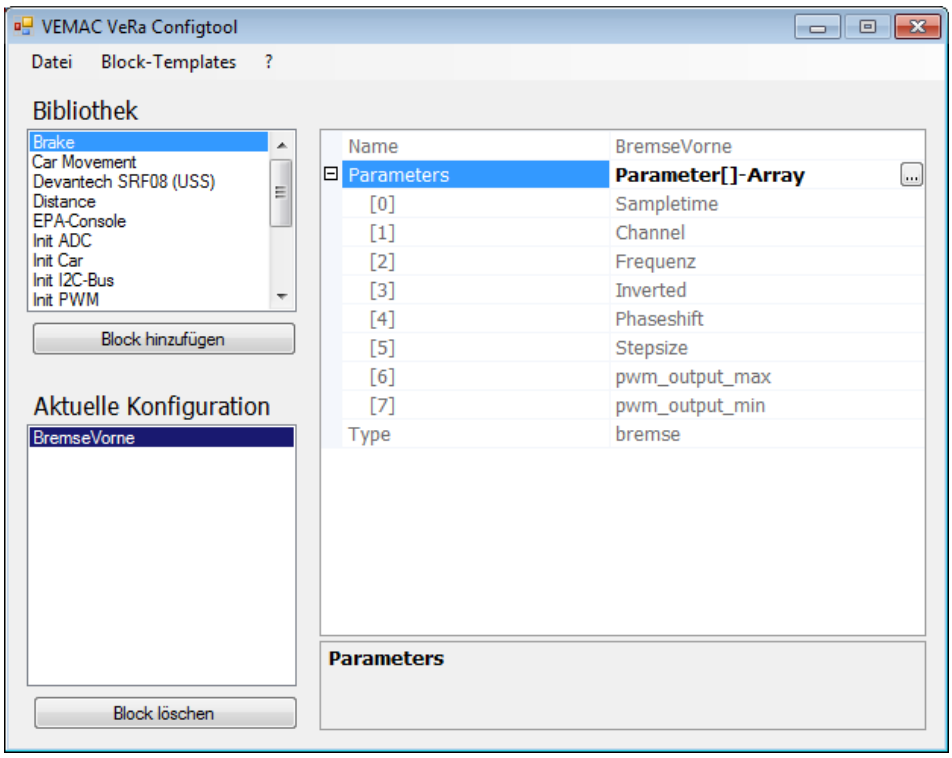

Abbildung 5.10.: Die Hauptansicht des Schnittstellenkonfigurationswerkzeugs

auch die Möglichkeit, die Konfigurationen vor Manipulationen durch Prüfsummen zu sichern. Das Konfigurationswerkzeug ist in `C#` implementiert. Die graphische Oberfläche kann in drei Bereiche eingeteilt werden wie in der Abbildung 5.10 zu sehen:

Bibliothek: Verfügbare Parametersätze für Schnittstellenkomponenten

Aktuelle Konfiguration: Beinhaltet die Schnittstellenkonfiguration

Parameter: Zeigt die Parameter des in „Aktuelle Konfiguration“ markierten Blocks an.

Für die Erstellung einer Konfiguration werden die vorhandenen Parametersätze für die verwendeten HAS-Komponenten gelistet. Die gesetzten Parameter werden mit Prüfsummen versehen, um Änderungen der Parameter ohne Konfigurationswerkzeug nachzuvollziehen. Dies ermöglicht auch ein geschlossenes System zu vertreiben, das eine Änderung der Schnittstellenkonfiguration unterbindet.

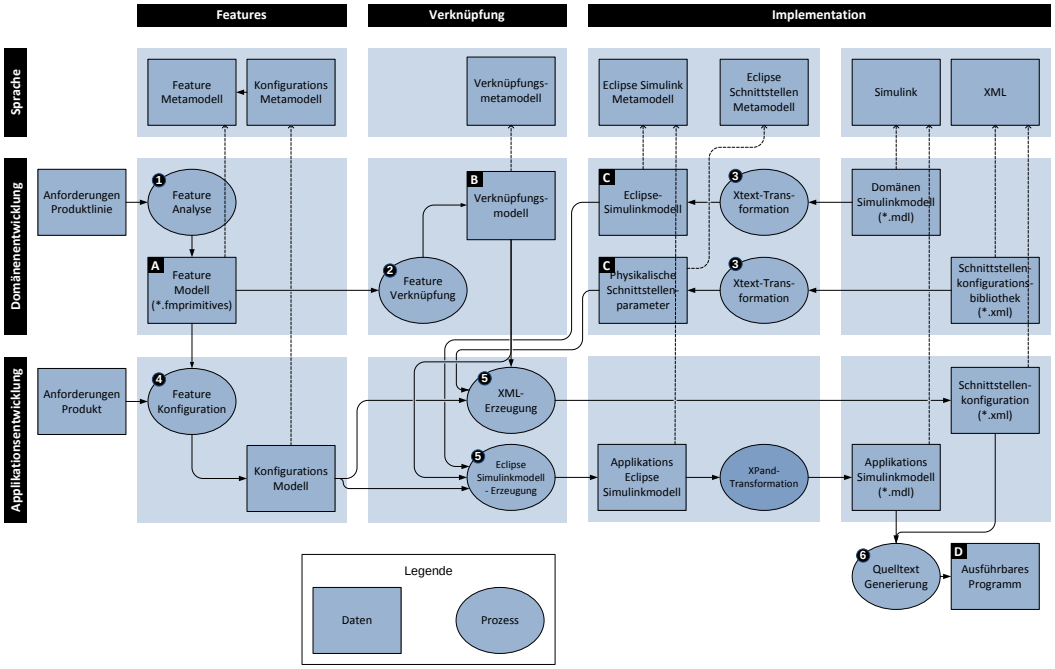

Abbildung 5.11.: Übersicht der Transformationen zur Analyse und Anpassung von Feature-Modell und Modellierungsartefakten

5.4.4. Konfiguration der Hardwareabstraktionsschicht auf Featureebene

Die Konfiguration der HAS auf Featureebene erfolgt durch FMe. Mit der Erstellung des FMs werden vom Entwickler Entscheidungen über die Schnittstelle getroffen. Das FM der Schnittstelle identifiziert die verwendeten Komponenten.

In der Abbildung 5.11 wird der Prozess des Ableitens einer Variante gezeigt. Dies bedeutet, dass die Schnittstelle der HAS entsprechend der Konfiguration angepasst wird. Eine Voraussetzung für eine Ableitung ist, dass alle am Prozess beteiligten Artefakte eine Repräsentation im EMF haben. Im Kapitel 6 werden die dazu notwendigen Import- und Exportfunktionen aller Modelle beschrieben.

Die Übersicht der Abbildung 5.11 ist in vertikale und horizontale Ebenen geteilt. Horizontal werden die Sprache, das Domänenengineering und Applikationsengineering unterschieden. Auf der Sprachebene werden die Sprachdefinitionen der konkreten Instanzen der Domänen- und Applikationsentwicklung gezeigt. Die Einteilung nach Domänen- und Applikationsebene folgt dem Ansatz von Pohl und Böckle [37].

In diesem Fall besteht die Produktlinie aus den HAS-Komponenten. Die zweite Ebene stellt die konkrete Anpassung der Schnittstelle und des SimMo dar. Die Ableitung einer Schnittstellenkonfiguration beginnt mit der Analyse der benötigten

Schnittstellenkomponenten ❶. Das Ergebnis ist ein FM 🅰, das die Schnittstelle mit abstrakten und konkreten HAS-Komponenten enthält. Die Feature-Verknüpfungen ❷ stellt die Verbindung zwischen Feature 🅰 und den Implementationen 🅲 her. Um die Implementationen im EMF repräsentieren zu können, erfolgt ein Import ❸. Die daraus entstehenden Modelle sind das Eclipse-Simulinkmodell 🅲 und die Schnittstellenparameter 🅲.

Für die Ableitung einer Variante erfolgt eine Auswahl von Features, die im Konfigurationsmodell gespeichert werden. Die Transformation ❺ erzeugt die abgeleiteten Varianten. Die Transformation ist in der Modelltransformationssprache Atlas Transformation Language (ATL) [12] implementiert.

Das Ergebnis solcher Transformationen sind Modelle, bei denen Blöcke und Linien existieren, die keinen Beitrag mehr für die Funktionalität liefern. Weitere Transformationen sind notwendig um die Modelle zu bereinigen. Die dazu notwendigen Transformationen sind im Kapitel 6 zu finden. Die Ableitung der Schnittstellenkomponente erfolgt durch eine weitere ATL-Transformation mit gleicher Funktionalität.

Kapitel 6.

Analyse und Unterstützung variantenreicher modellbasierter Entwicklung

Wie im vorherigem Kapitel gezeigt, stützt sich die modellbasierte Entwicklung eingebetteter Systeme auf Methoden und Werkzeuge, die nicht einfach für die variantenreiche Entwicklung geeignet sind. Dies gilt auch für die Entwicklungs- und Modellierungsumgebungen wie zum Beispiel Matlab / Simulink (MaSi). In der Arbeit von Weiland [46] wurden Ansätze entwickelt, um Variabilität in Matlab / Simulinkmodell (SimMo) zu modellieren und mit einem Feature-Modell (FM) zu verknüpfen.

Die Komplexität der variantenreichen Modelle und die Verknüpfung zu anderen Prozessdokumenten haben zur Folge, dass eine Integration von variantenreichen Prozessen ohne Unterstützung für den Entwickler nur mit großem Aufwand möglich ist. Um diese Unterstützung zu entwickeln, wird im folgenden Kapitel eine Anforderungsanalyse auf Modellebene durchgeführt, welche Entwicklungsszenarien zeigt, die durch Werkzeuge, unterstützt werden müssen.

Die Analysen und Werkzeuge die im Folgenden vorgestellt werden, können in die drei Bereiche:

- Analyse von Prozessdokumenten,
- Konsistenz von Prozessdokumenten und
- Unterstützung bei der Ableitung von einem Produkt einer Produktlinie

eingeteilt werden. Die Analysen und Unterstützungskomponenten wurden erweiterbar und anpassbar modellbasiert auf Grundlage des Eclipse Modeling Framework (EMF) entwickelt und betrachten alle am Prozess beteiligten Entitäten (Dokumente, Modelle und Implementationen). Dieses Kapitel beginnt mit einer Anforderungsanalyse. Aus dieser wird ein Rahmenwerk abgeleitet, das die Werkzeuge beinhaltet und die drei oben genannten Bereiche unterstützt. Die Anforderungen, das Konzept und die Umsetzung werden dann anschließend erläutert.

6.1. Anforderungen der variantenreichen modellbasierten Entwicklung

Die Grundlage der im Folgenden vorgestellten Anforderungen ist eine Analyse eines modellbasierten variantenreichen Entwicklungsprozesses aus der Automobilindustrie. In Workshops und Interviews mit erfahrenen Entwicklern wurden die Anforderungen ermittelt.

Der Grund für die Analyse des Prozesses und der Anforderungen ist eine hohe Komplexität bei der Ableitung von Varianten aus den Prozessartefakten. Diese Komplexität betrifft alle am Prozess beteiligten Artefakte und deren Beziehungen. In der Hauptsache sind dies

- Spezifikation / Lastenheft,
- Implementationsmodell und
- Testfälle.

Im bestehenden Ansatz wird die Variabilität hauptsächlich auf der Ebene Lastenhefte dokumentiert. Das Ableiten einer Variante erfolgt dann durch Abzweigen einer Version des Implementationsmodell (IMo) vom Hauptentwicklungszweig. Diese Modellversion wird dann entsprechend den Vorgaben des Lastenhefts angepasst. Zusätzlich müssen noch die Testfälle an die neue Version angepasst werden.

Die Erfahrungen mit diesem Prozess haben gezeigt, dass sich einige Aufgaben im Zusammenhang mit der Ableitung einer Variante nur mit einem großen Aufwand bewerkstelligen lassen. Eine Aufgabe ist dabei die Identifikation der Variabilität. Die Änderungen im Lastenheft werden auf der Ebene der Anforderungen gekennzeichnet. Dies bedeutet, dass die unterschiedlichen Anforderungen für verschiedene Varianten gleichzeitig im Lastenheft vorhanden sind. Um diese umzusetzen, werden alle Anforderungen auf Vollständigkeit geprüft und entsprechend ins Modell eingebracht.

Für die Umsetzung der Anforderungen im Modell müssen die betroffenen Stellen des Modells identifiziert werden. Die Identifikation der Änderungen am IMo ist ein wichtiger Prozessschritt, der zum einen die Konsistenz zwischen Anforderungen und Implementation sicherstellt und zum anderen eine Aufwandsabschätzung ermöglichen soll. Dies ist eine aufwändige Aufgabe, da auch Abhängigkeiten zu anderen Teilen des Modells (und auch deren Anforderungen) betrachtet werden müssen. Zusätzlich muss entschieden werden, wie die Variabilität implementiert wird und welche Auswirkungen bzw. Anpassungen der Testfälle dadurch notwendig werden.

Die genannten Aufgaben und Anpassungen aufgrund einer neuen Variante werden manuell ohne Werkzeugunterstützung durchgeführt. Dies hat einen großen Arbeitsaufwand und eine erhöhte Fehlerwahrscheinlichkeit zur Folge. Um diese Probleme zu lösen, werden im Folgenden die genauen Anforderungen definiert und ein Konzept

für einen Entwicklungsprozess vorgestellt, welcher die variantenreiche Entwicklung unterstützt.

6.1.1. Funktionale Anforderungen

Um die hohe Komplexität des variantenreichen Entwicklungsprozesses zu beherrschen, sind verschiedene Methoden und Analysen notwendig. Um diese zu implementieren soll ein Rahmenwerk geschaffen werden, das alle beteiligten Prozessartefakte berücksichtigt und schnelle prototypische Implementierungen von Methoden und Analysen im Bereich der variantenreichen Entwicklung ermöglicht. Das Ziel der prototypischen Entwicklung ist es, Analysen und Werkzeuge zu evaluieren und eventuell anzupassen. Anschließend sollen die prototypischen Analysen in einem längeren Zeitraum bewertet werden und dann als Vorlage für eine Implementierung dienen.

Das Rahmenwerk muss Methoden zur Ableitung von Varianten aller Prozessartefakte bieten. Bei der Ableitung einer Variante können sowohl positive und als auch negative Variabilität verwendet werden. Eine wichtige Anforderung an den Ableitungsprozess ist, die Konsistenz zwischen den Prozessartefakten sicherzustellen. Das Rahmenwerk sollte zusätzlich die Entwicklung graphisch unterstützen. Die hierfür notwendigen Sichten sollen die Ergebnisse von Analysen, die Strukturen und Zusammenhänge der Prozessdokumente anzeigen.

Neben der Analyse von bestehenden Strukturen gibt es die Anforderung, aus vorhandenen Informationen neue Strukturen und Zusammenhänge abzuleiten. Die Ableitung der Informationen soll gesteuert werden können. Dies bedeutet, dass es für einen Entwickler möglich sein muss, das Ergebnis der Analysen zu steuern und einfach neue Analysen zu entwickeln. Neben den funktionalen Anforderungen spielen nicht-funktionale Anforderungen eine zentrale Rolle, die im folgenden Kapitel aufgeführt werden.

6.1.2. Nicht-funktionale Anforderungen

Die nicht-funktionalen Anforderungen ergeben sich aus der Tatsache, dass die entwickelten Analyse- und Unterstützungsszenarien möglichst schnell bewertbare Ergebnisse liefern sollen. Daraus ergibt sich, dass die benötigte Entwicklungszeit (Time-To-Market) die wichtigste nicht-funktionale Anforderung ist.

Dies bedingt, dass die Umsetzungen leicht zu ändern sein und eine gute Anpassbarkeit haben müssen. Diese Flexibilität gilt auch im Bezug auf neue Informationen, die im Prozess erstellt bzw. verwendet werden und die vom Rahmenwerk berücksichtigt werden müssen. Dazu muss das Rahmenwerk einfach erweiterbar sein und neue Informationen verschiedener Werkzeuge einbinden können.

Neben der schnellen Umsetzung ist die Verwendbarkeit eine wichtige nicht-funktionale Anforderung. Die Entwickler der Produktlinie sollen in der Lage sein, die Analysen und deren Ergebnisse einfach zu erzeugen und zu verwenden. Dazu müssen die Ergebnisse in einer bekannten gut erfassbaren Form präsentiert werden.

Weitere nicht-funktionale Anforderungen müssen von den prototypischen Implementierungen im Allgemeinen beachtet werden. Dazu gehört der Schutz des geistigen Eigentums. Die Entwicklung von Produktlinien findet nicht komplett in einer Firma statt, die einzelnen Varianten werden von verschiedenen Firmen weiterentwickelt. Diese Firmen sollen möglichst keine Einsicht in die Produktlinie erhalten, sondern nur die Informationen der abgeleiteten Varianten bekommen.

6.1.3. Randbedingungen

Die Randbedingungen ergeben sich aus dem zu unterstützenden Entwicklungsprozess und den verwendeten Werkzeugen. Die Artefakte der Entwicklungswerkzeuge müssen im Analyserahmenwerk unterstützt werden. Das bedeutet, dass eine Bearbeitung und Analyse auch der proprietären Modelle möglich sein muss. Die geänderten Modelle sollen in den bestehenden Prozess integriert werden, aus diesem Grund müssen sie auch in den Entwicklungswerkzeugen wieder verwendbar sein.

Neben dem Modellierungswerkzeug MaSi sollen Informationen, die im Anforderungserfassungswerkzeug Doors [21] enthalten sind, analysiert und bearbeitet werden. Die Informationen müssen sowohl strukturell als auch inhaltlich analysiert und angepasst werden können. Auch in diesem Fall müssen Änderungen an den Informationen wieder im ursprünglichen Werkzeug dargestellt werden.

6.1.4. Analyse der Anforderungen

Das Ergebnis der Anforderungserfassung zeigt, dass die nicht-funktionalen Anforderungen auf die schnelle und flexible Entwicklung fokussieren. Das Rahmenwerk muss dazu die vorhandenen Informationen in den Prozessartefakten so aufbereiten, dass eine einfache Verarbeitung möglich ist. Die Informationen müssen verknüpft werden können, um Abhängigkeiten zwischen verschiedenen Dokumenten auszudrücken.

Aufgrund der Verknüpfungen soll es möglich sein, die Prozessartefakte anzupassen und so wiederkehrende Aufgaben zu automatisieren. Aufgrund der prototypischen Implementierung ist eine graphische Aufbereitung der Resultate oder eine Darstellung in der ursprünglichen Werkzeugumgebung wichtig, um Entwicklern die Möglichkeit zu geben, Einfluss auf die Analysen zu nehmen und Verbesserungen einzubringen. Das Rahmenwerk soll in der Lage sein, schnell belastbare und interpretierbare Umsetzungen von Unterstützungsszenarien zu liefern, die dann von einem Domänenexperten bewertet werden können. Die Ergebnisse der Bewertung führen dann zu einer weiteren Anpassung der Implementierung. Das Ergebnis dieses

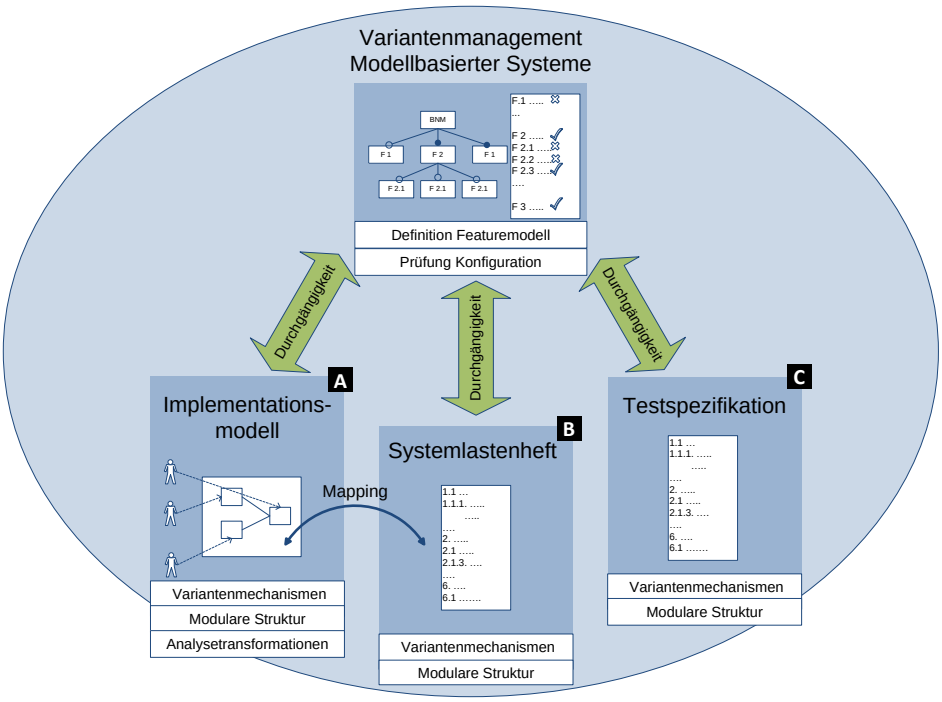

Abbildung 6.1.: Durch Feature gesteuerter modellbasierter Entwicklungsprozess

Prozesses ist eine prototypische Implementierung, die eine verlässliche Formulierung von Anforderungen ermöglicht.

6.2. Konzept des Rahmenwerks

Die Analyse des Prozesses und der Anforderungen ergibt, dass die Verwaltung von Variabilität und Konfiguration in einem separaten Dokument die Komplexität reduziert und für die beteiligten Entwickler besser handhabbar macht. Der neue Prozess basiert auf der Idee, Features als zentrale Instanz der Entwicklung zu nutzen. Die Darstellung der Features erfolgt mit einem Feature-Modell (FM) nach [43]. Eine Übersicht des FMs und der beteiligten Prozessartefakte ist in der Abbildung 6.1 dargestellt.

Das Konzept sieht einen Ansatz vor, der durch eine zentrale Instanz eines FMs gesteuert wird. Der Ansatz folgt dem Konzept von Pohl et al. [37], das einen Domänenansatz vorsieht und die komplette Produktlinie implementiert. Im zweiten Schritt erfolgt die Ableitung einer Applikation durch Anwendung einer Konfiguration.

Wichtiges Merkmal des gezeigten Prozesses ist die Durchgängigkeit, die es ermög-

licht die einzelnen Prozessdokumente (Implementationsmodell A, Systemlastenheft B und Testspezifikation C) immer mit der zentralen Instanz des FMs zu steuern und somit einen konsistenten Zustand für alle Artefakte zu erreichen.

Für das Implementationsmodell (IMo) sieht der Ansatz vor, dass es die gesamte Funktionalität der Produktlinie (150 % Modell) enthält. Um das IMo A im Variantenmanagement für modellbasierte Systeme nutzbar zu machen, werden im Modell Entwurfsmuster eingesetzt, die es ermöglichen, Variantenmechanismen automatisch anzuwenden. Dieses Entwurfsmuster ermöglicht es, bestimmte Teile aus dem Modell zu entfernen und so Produktlinienanpassungen vorzunehmen. Neben der Einführung von Entwurfsmustern muss die Möglichkeit vorgesehen werden, neue Features (Funktionalität) gekapselt einzufügen.

Das Systemlastenheft B enthält die Spezifikation der Applikation. Für die Einführung des Variantenmanagements auf Systemlastenheftebene B, benötigt man neben den Mechanismen zur Darstellung von Variabilität einen Aufbau, der die Ableitung einer Variante ermöglicht. Das gleiche gilt für die Testspezifikation C, die eine Beschreibung der Testfälle enthält. Die Testfallbeschreibung enthält die zu testende Funktionalität mit den entsprechenden Eingaben und den erwarteten Ausgaben. Die Testspezifikation ist die Grundlage zur Implementierung von Testfällen, die in verschiedenen Entwicklungsstadien angewendet werden.

Im Folgenden wird das Konzept der zentralen Variantenverwaltung als Grundlage für ein Rahmenwerk genutzt, das zur Ableitung von Varianten und Analysen von Produktlinien dient.

6.3. Rahmenwerk zur prototypischen Implementierung von Analyse- und Werkzeugunterstützung

In diesem Kapitel wird das Rahmenwerk beschrieben, welches die Anforderungen aus dem vorherigen Kapitel umsetzt. Die Grundlage des Rahmenwerks bildet das Eclipse Modeling Framework (EMF). In der Abbildung 6.2 ist ein Überblick über den Aufbau des Rahmenwerks zu sehen.

Die Abbildung 6.2 zeigt horizontal unterteilt die verwendeten Instanzen und Sprachen, in denen die Artefakte repräsentiert werden. In der vertikalen Ebene wird zwischen den proprietären Werkzeugen und dem Rahmenwerk unterschieden. Die Prozessartefakte der Werkzeugebene sind als Text repräsentiert und werden durch eine Text-Transformation in das Rahmenwerk importiert und als EMF-Modelle gespeichert. Die Text-Transformation basiert auf einem Parser, der mit Hilfe des Werkzeugs Xtext erzeugt wurde. Dazu wurden Regeln aufgestellt, die den Aufbau der Textdokumente beschreiben. Diese Regeln sind die Eingabe, um einen Parser, einen Editor mit Syntaxhighlighting und ein Metamodell des importierten EMF-Modells abzuleiten. Das Xtext-Werkzeug ist Teil des Textual Modeling Framework

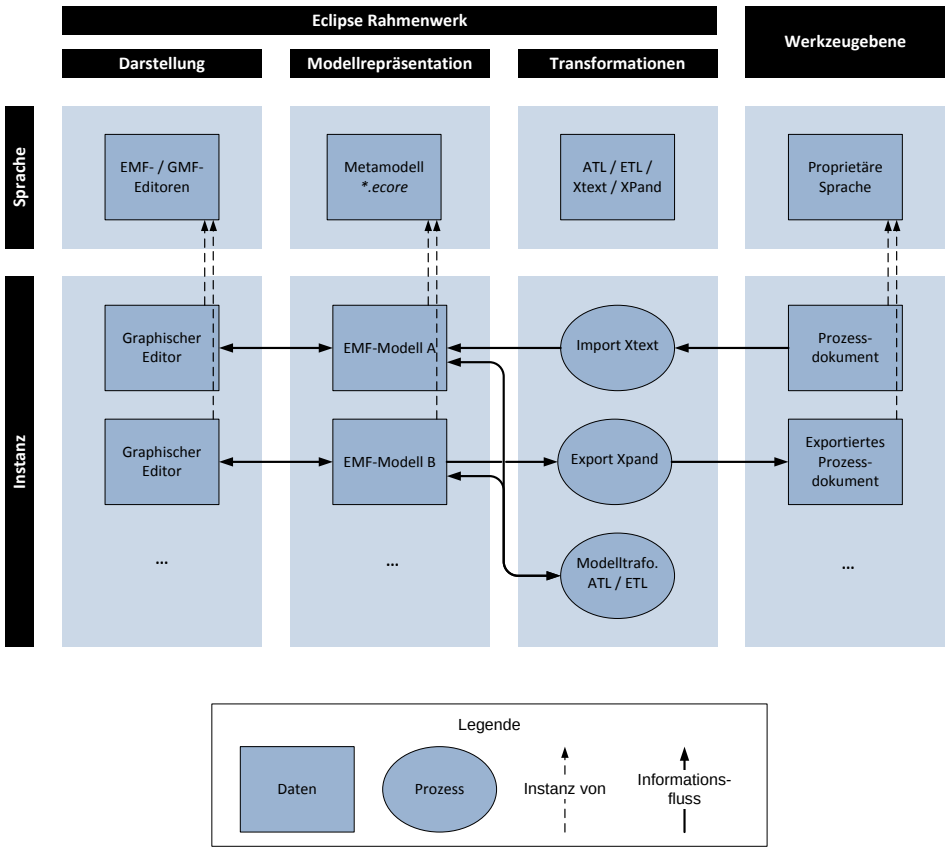

Abbildung 6.2.: Aufbau des Rahmenwerks mit verschiedenen Prozessdokumenten

(TMF) [16].

Für die Rücktransformation aus dem EMF in die Werkzeugebene wird das Werkzeug Xpand benutzt. Xpand hat im Vergleich zum Exportmechanismus von Xtext den Vorteil, dass notwendige Steuerzeichen und Abstände sehr exakt festgelegt werden können. Zusätzlich bietet Xpand die Möglichkeit, zu den vorhandenen Mechanismen benutzerdefinierte Funktionen in die Transformationen zu integrieren. Mit diesem Mechanismus ist es möglich, Textdokumente zu erzeugen, die von den Werkzeugen gelesen werden können.

Die importierten EMF-Modelle enthalten alle Informationen, die in den Prozessartefakten vorhanden sind. Zusätzlich gibt es die Möglichkeit, neue Informationen zu generieren. Diese können in neuen Modellen gespeichert werden oder durch Erweiterung der entsprechenden Metamodelle direkt in die Modelle integriert werden. Dadurch ermöglicht das EMF die geforderte flexible Anpassung und Erweiterung der Modelle, um zusätzlich Informationen zu speichern.

Zusätzlich bietet das Rahmenwerk die Möglichkeit, Modelle miteinander zu verbinden. Der Verbindungsmechanismus wird direkt auf der Modellebene durch weitere Modelle implementiert. Dabei bieten die Metamodelle die Möglichkeit, zu einer Verbindung automatisch die Rückverbingung zu setzen, sodass ein Zugriff von der Quelle zum Ziel und umgekehrt möglich ist.

Die Verbindungen von Modellen können mit Modelltransformationen erzeugt und ausgewertet werden. Die Modelltransformationen haben ein Modell oder mehrere Modelle als Ein- und Ausgabe und können neue Informationen generieren, bewerten und speichern. Es gibt mehrere Transformationssprachen, deren Grundlage das EMF ist. Die mit Hilfe der Sprachen geschriebenen Transformationen können dann EMF-Modelle verarbeiten und erzeugen. In dieser Arbeit werden die Transformationssprachen Atlas Transformation Language (ATL) und Epsilon Transformation Language (ETL) verwendet.

Neben der Flexibilität und Erweiterbarkeit bietet das EMF die Möglichkeit, graphische Editoren modellbasiert zu entwickeln. In dieser Arbeit werden die graphischen Editoren mit dem Graphical Modeling Framework (GMF) [15] implementiert. Das GMF ermöglicht eine modellbasierte Entwicklung der graphischen Editoren. Die Grundlage ist ein EMF-Modell mit zugehörigem Metamodell. Um den graphischen Editor vollständig zu beschreiben, werden zusätzlich Modelle erstellt, die eine graphische Repräsentationen der Modellelemente beinhalten. Anschließend werden über ein weiteres Modell eine Darstellung und eine Instanz des Metamodells verknüpft. Der benötigte Quelltext kann dann generiert werden.

Insgesamt bietet das auf dem EMF basierte Rahmenwerk so die Voraussetzungen, die zur prototypischen Umsetzung von Analysen und Ableitungen von Varianten der modellbasierten Entwicklung notwendig sind. Dies sind eine flexible und erweiterbare Anpassung der Modelle und zusätzlich die Möglichkeit, einen graphischen Editor zu erzeugen. Um einen schnellen Zugriff auf die Informationen zu gewährleisten, müssen in den Modellen Strukturen und Zusammenhänge erstellt werden, die die Analysen unterstützen. Diese Informationen werden zum großen Teil während des Imports der Daten in das Rahmenwerk erstellt. In den folgenden Kapiteln werden die Methoden dargestellt.

6.3.1. Import des Implementationsmodell

Das IMo ist in der Modellierungsumgebung Matlab / Simulink (MaSi) realisiert. MaSi speichert die Modelle als Textdatei ab. Darin sind alle, zur Anpassung und Analyse notwendigen, Informationen vorhanden. Da kein Metamodell für Matlab / Simulinkmodelle (SimMoe) bekannt ist, wird in dieser Arbeit ein eigenes verwendet. Das Metamodell bildet den Aufbau des SimMos nach, welches aus Blöcken besteht, die Informationen austauschen. Der Informationsfluss wird über Linien visualisiert. Die verbreiteten Informationen werden Signale genannt. Blöcke und

Signale können hierarchisch gegliedert werden und ermöglichen so Abstraktionen.

Das resultierende Metamodell für SimMoe ist komplex und umfangreich. Aus diesem Grund wird das Metamodell in einem zweistufigen Prozess erzeugt. Der erste Schritt basiert auf der automatischen Generierung des Metamodells mit Hilfe des Xtext-Werkzeugs. Anschließend erweitert man das Modell manuell, um weitere notwendige Strukturen zu schaffen. Das resultierende EMF-Modell wird als Eclipse-Simulinkmodell (EcSiMo) bezeichnet.

Der eigentliche Import des SimMoe erfolgt mittels des generierten Xtext-Parsers. Die Grundlage für den Xtext-Parser ist eine Grammatik, die in Form von Regeln hinterlegt wird. Die Grammatik hat die Aufgabe, die Strukturen und Blocktypen zu identifizieren, um syntaktische und semantische Aussagen über das Modell treffen zu können. Allerdings ist es nicht möglich, alle Blocktypen zu identifizieren, da es vorkommt, dass benutzerdefinierte Blocktypen erstellt werden. Die Grammatik identifiziert deshalb nur die wichtigen Standard Blocktypen. Die weiteren Blöcke werden vom Typ nicht unterschieden und als *Default-Blöcke* mit beliebigen Eigenschaften behandelt.

Es gibt Informationen, die bei allen Blocktypen vorhanden sind. Jeder Block hat eine Eigenschaft *Name*, welche den Block auf einer Hierarchieebene eindeutig identifiziert. Zusätzlich definiert jeder Block seine Schnittstelle über eine Anzahl von Ein- und Ausgängen, die als Zahlvektor gespeichert wird.

Als separater Typ werden vor allem Blöcke identifiziert, die Einfluss auf die Struktur des Modells haben oder für die Analyse der Modelle und des Informationsflusses von Bedeutung sind. Folgende Blöcke werden von der Grammatik erkannt:

SubSystem: Dieser Blocktyp fasst mehrere Blöcke zusammen und bildet eine Hierarchieebene. Damit hat dieser Blocktyp großen Einfluss auf die Struktur.

BlockInPort / BlockOutPort: Dieser Blocktyp stellt die Verbindung zwischen einem Ein- / Ausgang eines Subsystems und den darin enthaltenen Blöcken her. Diese Blöcke steuern den Informationsfluss innerhalb und außerhalb eines Subsystems.

BlockTriggerPort: Dieser Blocktyp ist Teil eines Subsystems und steuert dessen Aktivierung. Das Aktivieren bzw. Inaktivieren eines Subsystems kann einen Variantenmechanismus darstellen.

BusCreator / BusSelector: Die beiden Blocktypen fassen mehrere Signale in einem Bus zusammen oder selektieren einzelne Signale aus einem Bus. Diese Blocktypen abstrahieren den Informationsfluss.

FromBlock / GotoBlock: Diese Blocktypen senden / empfangen ein Signal. Diese Blöcke müssen analysiert werden, um den Signalfluss zu bestimmen.

ConstantBlock / Groundblock / Terminator: Blöcke, die Signale initialisieren bzw. beenden. Diese Blöcke sind für die Signalanalyse wichtig.

SwitchBlock / MergeBlock: Diese Blocktypen vereinen / wählen aus zwei oder mehreren Signalen eines aus. Mit diesen Blocktypen werden Variantenmechanismen implementiert.

Neben den identifizierten Blöcken sind alle Informationen im importierten Modell enthalten. Diese Informationen sind teilweise nicht direkt verfügbar und müssen aus aus verschiedenen Quellen abgeleitet werden. Ein Beispiel hierfür ist der Signalfluss, der durch Busstrukturen und Subsysteme nur schwierig nachvollziehbar ist. Durch Erweiterungen im Metamodell können solche Informationen gespeichert werden.

In der Abbildung 6.3 ist das Metamodell dargestellt. Die orange gefärbten Rechtecke sind durch Xtext aufgrund der Grammatik erstellt worden. Die grau hinterlegten Rechtecke stellen Erweiterungen dar, die einen vereinfachen Zugriff auf die Struktur ermöglichen. Rechtecke mit grüner Hintergrundfarbe werden zur Anzeige im graphischen Editor benötigt.

Das Metamodell zeigt in orange die eigentliche Struktur des Modells. Ein `Model` besteht aus einen `StateflowModel` und einem `SimulinkModel`. Das `StateflowModel` wird hier nicht weiter betrachtet. Das `SimulinkModel` hat als wichtigsten Bestandteil ein `System`, das sowohl `blocks` (Blöcke) als auch `lines` (Signale) enthält. Ein Block ist eine abstrakte Klasse, die von den verschiedenen Blocktypen als Superklasse verwendet wird. Der Blocktyp `SubSystem` kann wiederum ein System enthalten.

Die Blocktypen `IntegratorBlock` und `Module` sind spezielle Blocktypen, die zur Speicherung von Varianteninformationen benötigt werden. Beide Blöcke sind eigentlich `SubSysteme` mit zusätzlichen Anforderungen an den Aufbau und Funktionen. Die Blocktypen `VariantPoint` und `InterfaceBlock` fassen Blocktypen zusammen.

Die neu eingeführten Blocktypen betreffen die Linien und Signale. Linien sind Verbindungen zwischen zwei oder mehr Blöcken und stellen Informationsfluss dar. Signale hingegen können über mehrere Linien und Blöcke eine Information verteilen. Die abstrakte Entität `Line` kann zur einer `GmfLine`, zu einer `MdlLine` oder `LogicalLine` spezilisiert werden. Die `MdlLine` repräsentiert die ursprüngliche Linie im SimMo, bei der textuell das Ziel und die Quelle hinterlegt sind. Die `LogicalLine` repräsentiert eine direkte Verbindung zwischen mehreren `LogicalPorts`. Die `LogicalPorts` sind zusätzlich neu eingeführt worden, um auf der Blockebene eine Repräsentation zu haben.

Die Signale werden über `LogicalSignal` gespeichert und können in `BusSignal` und `AtomicSignal` unterschieden werden. In einem `LogicalSignal` wird gespeichert, an welchem `SignalTransporter` das Signal anliegt. Ein `SignalTransporter` kann entweder eine `LogicalLine` und ein `LogicalPort` sein.

Um die im Metamodell geschaffenen Strukturen mit den richtigen Informationen zu füllen, werden Modelltransformationen eingesetzt. Insgesamt sind 9 Transforma-

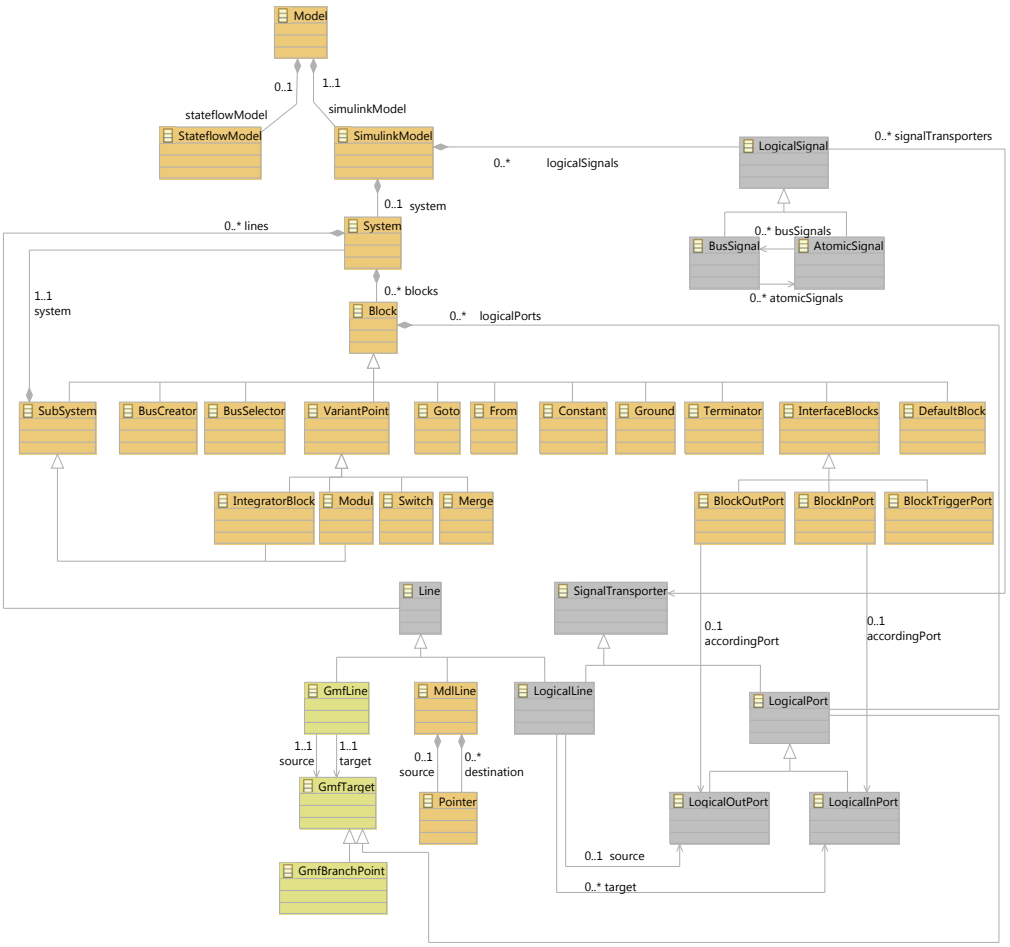

Abbildung 6.3.: Metamodell des Matlab / Simulinkmodells

tionen hintereinander geschaltet. Die 9 Transformationen haben folgende Reihenfolge und Aufgaben:

1. `getPortNames`: Speichert die Namen der Outports des SimMoe direkt in der Implementierung.

2. `createLogicalPorts`: Erzeugt die richtige Anzahl an Ein- und Ausgangsports (`LogicalPort`) für jeden Block.

3. `linkLogicalAndMdlPorts`: Ordnet einem vorhandenen Ausgangsport des SimMos die entsprechenden logischen Ports der vorherigen Transformation zu.

4. `createLogicalLine`: Erzeugt logische Verbindungen aufgrund der im Modell vorhandenen Modelllinien. Das Ziel und die Quelle einer logischen Linie ist immer ein logischer Port.

5. `createLogicalSignals`: Erzeugt alle möglichen logischen Signale. Unterscheidet aber nicht zwischen Bus- und normalen Signalen.

6. `mergeAtomicSignals`: Vereinigt normale Signale und identifiziert sie als atomar (Signale, die nur ein Signal transportieren).

7. `mergeBusSignale`: Vereinigt Bussignale und identifiziert die Signale als Bussignale.

8. `linkBusAndAtomicSignals`: Ordnet den Bussignalen die durch sie transportierten atomaren Signale zu.

9. `removeSignalTransporter`: Diese Transformation entfernt redundante Informationen, sodass Ports und Linien nur noch ein Signal zugeordnet wird. Dieses Signal ist entweder ein Bussignal oder ein atomares Signal. Über die entsprechende Verknüpfung zwischen Bussignalen und atomaren Signalen kann festgestellt werden, welche Signale transportiert werden.

Die Erzeugung der Linien, Ports und Verknüpfungen erfolgt durch Transformationen, die mit der Hilfe von Suchalgorithmen logischen Entitäten den Start und das Ziel von Verknüpfungen bestimmen. Um eine einfache Navigation zu ermöglichen, werden die Eingangs- und Ausgangsports vom Blocktyp `SubSystem` mit den entsprechenden Blöcken verknüpft, die Blöcke innerhalb des `SubSystems` den Signaltransport repräsentieren. Wie in der Abbildung 6.4 zu sehen, ist es damit möglich, direkt von einem Eingangsport zur entsprechenden Verknüpfung innerhalb des Subsystems zu gelangen und wieder zurück.

Für die Berechnung der Signale ist eine Analyse des gesamten Modells notwendig. Als Voraussetzung wird ein EcSiMo erwartet, in dem die logische Struktur des Modells bereits berechnet wurde. Die Signale speichern die logischen Ports und Linien, auf denen sie anliegen, als `SignalTransporter`. Ein Beispiel dieser Signal-Transporter ist in der Abbildung 6.4 zu sehen. Bei der Definition von Signalen kann es zu Widersprüchen im Modell kommen. Ein Widerspruch ergibt sich, wenn Signale doppelt definiert werden. Ein Beispiel für eine doppelte Definition ist in der Abbildung 6.4 zu sehen. Das `Signal B` wird durch den `Default Block` definiert und über die `BlockOutPorts` weitergeben. Daher liegen die Signale am Ausgang des Subsystems an. Erfolgt dort eine weitere Definition eines Signals (`SignalC`) ist der Bezeichner des Signals nicht mehr eindeutig. Beide Bezeichner (`Signal B` und `Signal C`) gelten und werden in der Analyse auch berechnet. Innerhalb der

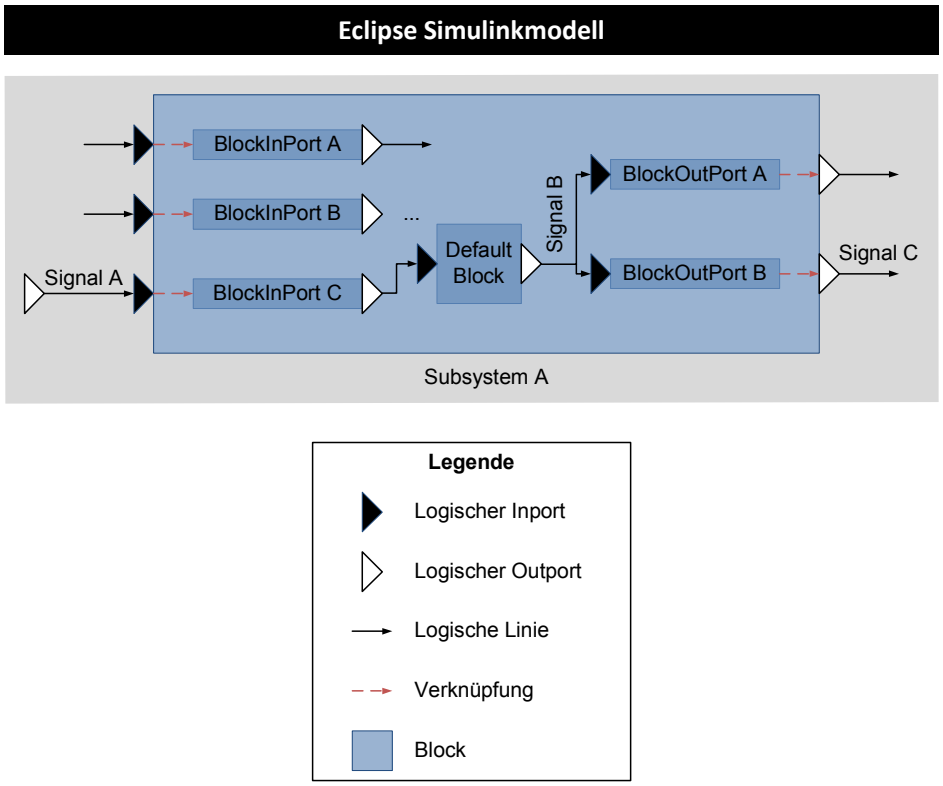

Abbildung 6.4.: Beispiel eines Eclipse-Simulinkmodells mit struktureller Verknüpfung im Subsystem und Signaldefinition

Signalanalyse werden solche doppelten Signale wieder zu einem Signal vereinigt und eine Warnung für den Benutzer ausgegeben.

Insgesamt enthält das EcSiMo nach dem Import und den Transformationen alle strukturellen Informationen, um Analysen der Verbindungen und Signale durchführen zu können.

6.3.2. Import der Anforderungen

Die Anforderungen werden im Werkzeug Doors gespeichert. Die Grundlage von Doors ist eine Datenbank. Der Zugriff auf die Datenbank von außen ist nur mit einem präparierten Klienten möglich. Um den Import sowohl der Anforderungen als auch der Testspezifikation zu ermöglichen, werden die Daten mit Hilfe des Klienten in eine CSV-Datei (Comma-Separated Values [41]) exportiert. Diese kann dann mit dem Xtext-Werkzeug in das EMF importiert werden.

Der Import der CVS-Dateien folgt der Struktur aus der Abbildung 6.2. Ein Parser,

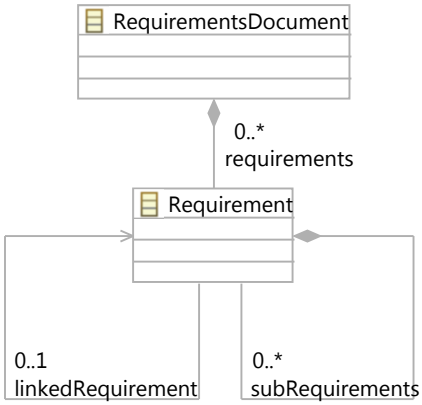

Abbildung 6.5.: Metamodell für hierarchisch strukturierte Anforderungen

der mittels des Werkzeugs Xtext erzeugt wird, übernimmt den Import der Datei in das EMF. Anschließend wird das importierte Modell analysiert und relevante Strukturinformationen hinzugefügt.

Die Strukturinformationen bestehen aus einem hierarchischen Aufbau des gesamten Dokuments. Der hierarchische Aufbau ergibt eine Gliederung in eine Baumstruktur von Kapitelüberschriften mit Unterüberschriften. Die Anforderungen sind jeweils in den Blättern des Baums zu finden. Zusätzlich sind Verknüpfungen innerhalb des Dokuments enthalten. Im zugehörigen Metamodell werden die Überschriften nicht von den Anforderungen unterschieden. In der Abbildung 6.5 ist das Metamodell abgebildet. Die Wurzel der Anforderungen wird in der Entität `RequirementsDocument` gespeichert. Der hierarchische Aufbau wird über die `subRequirements` Beziehung implementiert. Die Verknüpfung kann durch `linkedRequirement` gesetzt werden. Die eigentlichen Informationen sind in Attributen gespeichert, die als Text die Anforderungen beschreiben.

Die Verknüpfungen innerhalb des Dokuments können aufgrund eines eindeutigen Bezeichners gefunden werden. Um die Verknüpfungen zu erzeugen, wird eine weitere Transformation verwendet.

6.3.3. Ableitung einer identischen Transformation

Eine Anforderung an das Rahmenwerk ist es, schnell Analysen und Modellanpassungen implementieren zu können. Der modellbasierte Ansatz mit der Generierung von textuellen und graphischen Editoren ermöglicht die Erfüllung dieser nichtfunktionale Anforderung. Zusätzlich muss auch eine schnelle Implementierung der Modelltransformationen ermöglicht werden.

Während der Entwicklung der Transformationen hat sich gezeigt, dass sich die

Abbildung 6.6.: Metamodell-Transformation in eine identische ATL-Transformation

Analysen und Anpassungen in kleine Funktionalitäten und damit Transformationen einteilen lassen. Dabei werden, wie im Abschnitt 6.3.1 dargestellt, oftmals Modelle um Informationen angereichert oder gefiltert. Diese Transformationen werden für verschiedene Metamodelle immer wieder ausgeführt. Die restlichen Informationen werden einfach übernommen. Aus diesem Grund kann die Transformation grob als eine identische Funktion mit jeweils kleinen Änderungen angesehen werden. Die Größe dieser identischen Transformation hängt von der Größe des Metamodells ab, da das Metamodell die Anzahl der verschiedenen zu kopierenden Entitäten festlegt. Um den Vorgang der Erstellung einer identischen Transformation zu beschleunigen, wird in diesem Rahmenwerk eine Transformation eingesetzt, wie sie in Abbildung 6.6 dargestellt ist. Diese Transformation erzeugt eine identische Transformation zu einem Metamodell.

Die identische Transformation ist in der Lage, für alle vorhandenen Entitäten Regeln zu erzeugen, die eine Kopie anfertigen. Dies gilt auch, wenn abstrakte Entitäten gespeichert werden. Die Zuordnung auf die reale Instanz erfolgt im Fall einer abstrakten Entität über kontextabhängige Hilfsfunktionen, die dann die Implementierung der entsprechenden Regel aufrufen. Ein Beispiel einer Transformation ist in der Abbildung 6.7 zu sehen.

Das zu transformierende Metamodell enthält eine Entität `Abstact`, die zu den Entitäten `Implement1` und `Implement2` konkretisiert wird. Zusätzlich gibt es eine Referenz auf `RefInst` und eine Beziehung der beiden konkreten Implementierungen. Die Metatransformation besteht aus einem `Header`, Kopierregeln (`CopyRules`) und abstrakten Hilfsfunktionen. Zu allen Entitäten werden Kopierregeln erzeugt. Für den Fall, dass es eine abstrahierende Entität gibt, wird zusätzlich noch eine abstrakte Hilfsfunktion erzeugt. In der zweiten Funktion der Metatransformation ist die Generierung einer Hilfsfunktion abgebildet. Die eigentliche Funktionalität muss nur durch die entsprechenden Namen angepasst werden, um die gültige Regel zu erzeugen.

Die resultierende Transformation ist in Auszügen im unteren Teil der Abbildung 6.7 dargestellt. Jede Regel wird immer über eine Hilfsfunktion aufgerufen, um Fehlermeldungen zu verhindern, da eine Regel sonst für nicht vorhandene Objekte ausgeführt wird. Die Transformationsregel wird mit den Schlüsselwörtern `unique` und `lazy` gekennzeichnet. Durch `unique` wird verhindert, dass ein Objekt mehr als einmal transformiert wird. Ein solches Verhalten könnte bei Verknüpfungen

Abbildung 6.7.: Auszüge einer beispielhaften, Metamodell zu identischer Transformation, Umwandlung

innerhalb des Metamodells vorkommen. Die `lazy` Anweisung sorgt dafür, dass die Regeln immer aufgerufen werden müssen, um ausgeführt zu werden. Dieses Verhalten ermöglicht es, Objekte zu filtern und zu entfernen.

Als letzte Funktion in der Abbildung 6.7 ist eine kontextfreie Hilfsfunktion dargestellt, die eine sichere Abfrage ermöglicht. Diese wird innerhalb der Regel im fett markierten Bereich aufgerufen. Die abstrakte Hilfsfunktion ruft dann die kontextsensitive Hilfsfunktion `createAbstractHelperAbstract()` auf. Diese Funktion kann aufgrund des Kontextes entscheiden, welche Regel zu benutzen ist.

Insgesamt bietet diese Transformation die Möglichkeit, zu beliebigen EMF-

Metamodellen eine identische Transformation zu erstellen, um diese dann anzupassen. Durch die Verwendung der `lazy`-Regel ist es möglich, beliebige Veränderungen an dem Modell vorzunehmen (Hinzufügen, Ändern, Löschen von Modellelementen).

6.3.4. Graphischer Editor des Implementationsmodell

Die Anzeige der Analyseergebnisse kann zum einen in den vorhandenen Werkzeugen erfolgen oder in speziellen graphischen Editoren. Diese graphischen Editoren werden mit dem GMF realisiert. Die Prinzipien des Rahmenswerks sind bereits im Grundlagenkapitel beschrieben. Analog zu dem Ansatz der Metatransformation soll ein graphischer Editor als Grundlage zur Verfügung stehen, um durch kleine Veränderungen die Möglichkeit zu haben, schnell die gewünschte Implementierung abzuleiten.

Da die Analysen und Anpassungen meist von einem SimMo ausgehen, ist der Editor in der Lage ein EcSiMo darzustellen. Der Editor hat die gleichen Darstellungsmöglichkeiten wie der MaSi Editor. Dies bedeutet, er kann Blöcke mit entsprechender Schnittstelle und Verbindungen zwischen Blöcken darstellen. Die Positionierung der Blöcke kann über Positionsangaben gesteuert werden. Zusätzlich ist es möglich, die hierarchischen Abstraktionen der Subsysteme und der Busse darzustellen.

Für die Darstellung sind Erweiterungen des EcSiMo-Metamodells notwendig. Die Erweiterungen betreffen zum einen die Linien und zum anderen den Blocktyp `SubSystem`. In der Abbildung 6.3 ist der Linientyp `GmfLine` zu sehen. Dieser Linientyp ist für die Darstellung im graphischen Editor eingeführt worden. Er unterscheidet sich von logischen und MaSi-spezifischen Linien dadurch, dass es immer genau eine Quelle und ein Ziel gibt. Um ein Modell darzustellen, ist es daher notwendig, aus den logischen Linien graphische Linien abzuleiten. Die notwendige Transformation wandelt dazu eine logische Linie mit einer Quelle und `n` Zielen in `n` graphische Linien mit jeweils einer Quelle und einem Ziel um. Die Transformation erzeugt zusätzlich eine Verknüpfung zwischen der logischen und den zugehörigen graphischen Linien. Ziel dieser Verbindungen ist es, bei Änderungen an den graphischen oder logischen Linien wieder einfach einen konsistenten Zustand erzeugen zu können.

Für die Darstellung der Hierarchie auf Blockebene durch `Subsysteme` muss in den automatisch generierten Modellquelltext eingegriffen werden. Der Grund hierfür liegt im Aufbau des SimMos. Der Aufbau ist im Metamodell in der linken oberen Ecke der Abbildung 6.8 zu sehen. Dort ist die Beziehung zwischen `System` und `SubSystem` dargestellt. Ein `System` enthält beliebig viele Blöcke. Ein `SubSystem` enthält hingegen genau ein `System`. Die Modelle zur Erzeugung des graphischen Editors benötigen einen hierarchischen Aufbau, der in der rechten oberen Ecke gezeigt ist. Um beide Anforderungen zu vereinen, ist die Anpassung im automatisch generierten Quelltext des EcSiMoe notwendig. Das neue Metamodell ist in der unteren Fläche der Abbildung 6.8 zu sehen. Dazu wird eine Beziehung für die

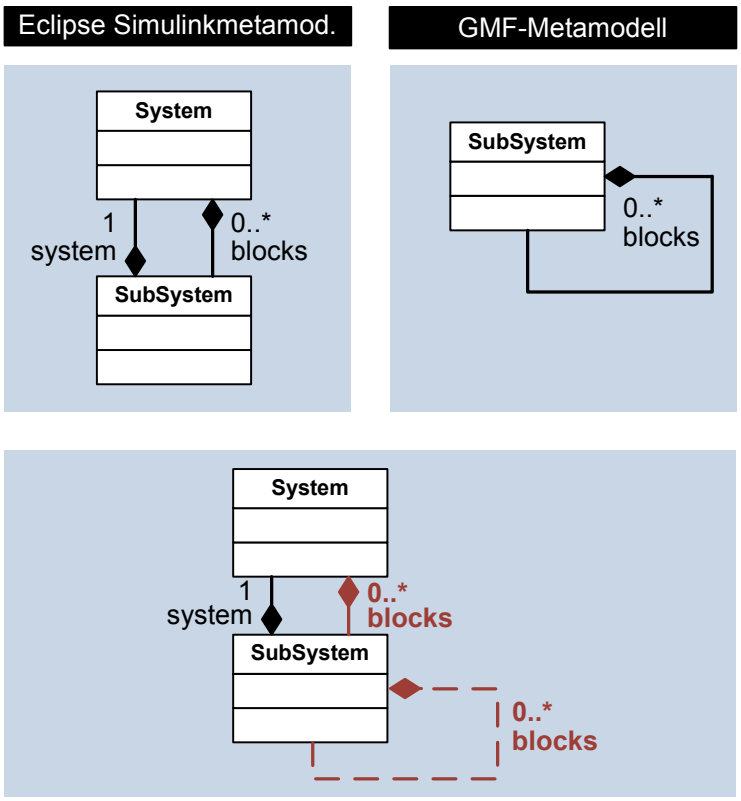

Abbildung 6.8.: Anpassungen im Eclipse-Simulink-Metamodell

Entität `SubSystem` eingeführt. Diese Beziehung wird immer durch die `blocks`-Beziehung von `System` festgelegt. Dies bedeutet, dass die im System enthaltenen Blöcke automatisch dem `SubSystem` zugeordnet werden. Durch diese Konstruktion können die Strukturen beider obigen Metamodelle in einem Metamodell konsistent abgebildet werden.

Weitere konzeptionelle Anpassungen für die Erstellung des graphischen Editors sind nicht notwendig. Die graphische Gestaltung des Editors kann einfach angepasst werden. Die Implementierung erfolgt über die Mechanismen, die schon in den Grundlagenkapiteln erläutert werden. Dazu werden die Blöcke, die darin enthaltenen logischen Ports (Schnittstelle) und die graphischen Verbindungen dargestellt. Zusätzlich gibt es die Möglichkeit, zwischen verschiedenen Hierarchieebenen zu springen, um den Inhalt von Blöcken des Typs `SubSystem` anzuzeigen.

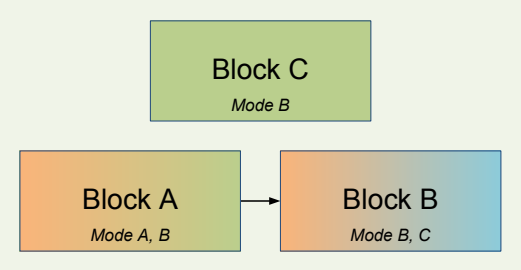

Abbildung 6.9.: Beispiel von einer kritischen Abhängigkeitssituation

6.4. Analysen von modellbasierten Produktlinien

Das vorherige Kapitel hat das Rahmenwerk zur Analyse und Beeinflussung des modellbasierten Entwicklungsprozesses vorgestellt. Aufbauend auf diesem Rahmenwerk werden in den folgenden Kapiteln Anwendungen mit Hilfe des Rahmenwerks umgesetzt. Die Anwendungsfälle sind in Zusammenarbeit mit Entwicklern der Automobilindustrie entstanden. Um die Analysen zu beschreiben, werden die Anforderungen aufgestellt und mittels Szenarios beschrieben. Anschließend werden die für die Umsetzung notwendigen Transformationen und die Verwendung des Rahmenwerks sowie die notwendigen Grundlagen dargestellt. Eine Bewertung des Ergebnisses erfolgt dann im Part III dieser Arbeit (Evaluierung Kapitel 9).

6.4.1. Analyse von Aktivierungsmodi bei Variabilitätspunkten (Aktivierungssicht)

Bei der Entwicklung von automotiver Software werden oftmals domänenspezifische Entwurfsmuster eingesetzt. Eines dieser Entwurfsmuster ist Grundlage der folgenden Analyse. Das Entwurfsmuster steuert die Aktivierung von Teilen des Modells. Die Aktivierung der verschiedenen Modellteile hängt von definierten Betriebsmodi ab. Beispiele für typische Betriebsmodi im Fahrzeug sind *Motor läuft*, *Schlüssel steckt* oder *Zündung eingeschaltet*. Die Betriebsmodi werden für jedes SimMo individuell festgelegt. Der Zustand wird in einem zentralen Teil des SimMoe berechnet und an die abhängigen (zu aktivierenden) Modellteile verteilt.

Ein Beispiel für eine solche Situation der Abhängigkeiten ist in der Abbildung 6.9 zu sehen. Die drei Blöcke A, B, C werden in den drei Modi A, B, C betrieben. Der Block B erwartet Eingaben von Block A. Im Modus C kann es zu einer fehlerhaften Situation kommen, da der Block B Eingaben vom Block A erwartet, dieser aber nicht aktiv ist. Die Situation wird noch komplexer, wenn durch das Variantenmanagement der Block A nicht enthalten ist. Durch diese Situation kommt es zu einem weiteren fehlerhaften Zustand. Ziel der Analyse ist, den Entwickler zu unterstützen, das

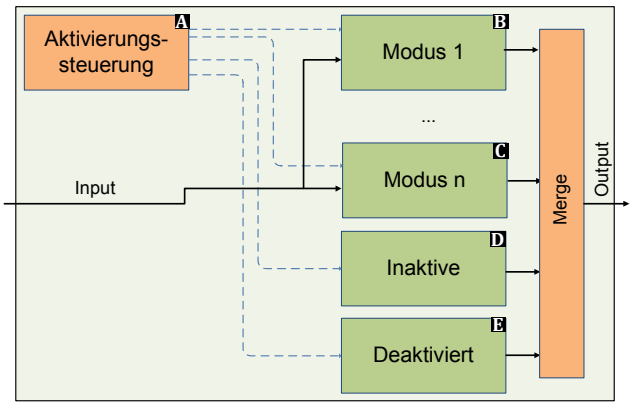

Abbildung 6.10.: Aufbau des Entwurfsmusters Modul

Modell auf diesen Aspekt hin zu untersuchen. Das Ergebnis der Analyse soll ein Modell sein, das nur noch die relevanten Teile des Modells zeigt.

In diesem Fall werden Entwurfsmuster bei der Entwicklung des Modells eingesetzt. Die eingesetzten Entwurfsmuster ermöglichen es, die relevanten Modellteile automatisch zu identifizieren. Das zentrale Entwurfsmuster ist das *Modul*. Das Modul kapselt Funktionalität und wird von Variabilität und den Betriebszuständen (Modi) aktiviert. Als Voraussetzung wird angenommen, dass Funktionalität, die von Betriebszuständen oder vom Variantenmanagement abhängig ist, immer in Modulen gekapselt ist.

Das Entwurfsmuster *Modul* hat einen standardisierten Aufbau, der in der Abbildung 6.10 zu sehen ist. Das Entwurfsmuster besteht aus einen Block vom Typ *SubSystem*. Die innere Struktur des Moduls besteht aus einer `Aktivierungssteuerung` A. Diese aktiviert aufgrund des Betriebszustands (Modus), welche Funktionalität aktiviert wird. Es ist immer nur ein Block gleichzeitig aktiv. Durch den Mergeblock wird das Ergebnis des aktiven Blocks an die Ausgabeschnittstelle weitergeleitet. Die Subsystemblöcke `Modus 1` B bis `Modus n` C beinhalten Funktionalität für den *aktiven* Zustand. Nur diese Blöcke haben auch eine Eingabe von außen. Der `Inaktive` Zustand D wird aktiviert, wenn kein anderer aktiver Block aktiviert wird. Der Block erzeugt Standardwerte. Das Subsystem `Deaktiviert` E wird nicht dynamisch zur Laufzeit verwendet, sondern durch das Variabilitätsmanagement benutzt. Auch im deaktivierten Zustand werden Standardwerte gesetzt. Diese unterscheiden sich teilweise vom inaktiven Zustand.

Weitere Entwurfsmuster legen den Aufbau der Modul-Schnittstellen fest. Die Eingabe und Ausgabe der benötigten und berechneten Signale erfolgt immer über Busse, die mehrere Signale zu einem zusammenfassen. Dadurch wird zum einen die Übersichtlichkeit erhöht, zum anderen geht die Möglichkeit verloren zu sehen,

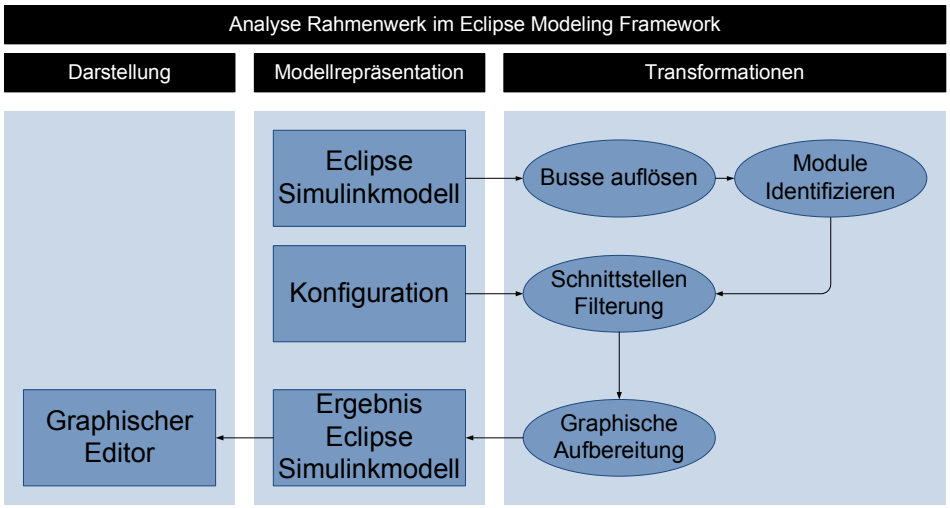

Abbildung 6.11.: Transformationen der Analyse zur Aktivierung von Modulen

welche Signale verwendet bzw. erzeugt werden.

Die Entwurfsmuster sind die Grundlage, um eine Analyse zu implementieren. In der Abbildung 6.11 ist der Ablauf der Transformationen zu sehen. Als Eingabe wird ein EcSiMo und eine Variantenkonfiguration verarbeitet. Das EcSiMo wird zuerst von der Transformation `Busse auflösen` umgewandelt. Dabei entsteht ein Modell, in dem alle Signale einzeln aufgeführt sind. Anschließend werden über die Transformation `Module identifizieren` Blöcke gekennzeichnet, die ein Modul darstellen. Das Modell ohne Busse, die Kennzeichnung der Module und die Konfiguration sind die Eingabe für die eigentliche Analyse (`Schnittstellen Filterung`). Dazu werden nur die relevanten Modellteile ins neue Modell kopiert und eine Analyse der Verbindungsstruktur implementiert. Das Ergebnis wird dann noch graphisch aufbereitet, damit es im Editor entsprechend angezeigt werden kann. Die einzelnen Transformationen werden in den folgenden Kapiteln beschrieben.

Auflösen von Busstrukturen

Das Auflösen der Busstruktur in einem EcSiMo erzeugt ein Modell, das keine Busstrukturen mehr enthält und statt dessen die darin enthaltenen Signale als eine direkte Verbindung nutzt. Das daraus resultierende Modell hat die gleiche Verbindungsstruktur und daher auch die gleiche Semantik. Um die Busstrukturen zu entfernen, werden weitgehende Eingriffe im Modell vorgenommen. Transformationen sind an allen Blocktypen notwendig, die Busstrukturen verarbeiten können. Die hier erstellte Transformation betrachtet aber nicht alle Blocktypen, da einige veraltet sind und nicht mehr verwendet werden sollen. Tauchen dennoch diese Blocktypen auf,

bricht die Transformation ab und gibt eine Warnung aus. Die folgenden Blocktypen werden von der Transformation beachtet:

- `SubSystem` / `BlockInPort` / `BlockOutPort`: Über die Schnittstelle eines Subsystems können Bussignale ausgetauscht werden. Bei Löschung des Busses muss die Schnittstelle und die zugehörigen BlockInPorts und BlockOutPorts angepasst werden.
- `From` / `Goto`: Verbindungen über Goto- und From-Blöcke können Bussignale enthalten. Das Entfernen des Busses bewirkt, dass für jedes Signal eine neue Verbindung über From- und Goto-Blöcke kreiert werden.
- `Terminator`: Ein Terminator Block kann auch ein Bussignal terminieren. Entsprechend müssen nach dem Entfernen des Busses alle einzelnen Signale terminiert werden.
- `Merge` / `Switch`: Ein Merge / Switch Block kann zwischen Bussignalen auswählen. Dieses Verhalten ist auch für die einzelnen Signale notwendig.
- `BusCreator` / `BusSelector`: Diese Blöcke müssen entfernt werden, da sie ohne Busse nicht mehr gebraucht werden.

Die Transformationsregeln der einzelnen Blöcke sind in der Abbildung 6.12 abgebildet. Wenn am Eingang oder Ausgang von Subsystemen Busse benutzt werden, müssen die entsprechenden Schnittstellen angepasst werden. Für die im Bus enthaltenen Signale muss jeweils ein neuer Eingang bzw. Ausgang angelegt werden und zugehörige BlockOutPorts und BlockInPorts eingefügt werden.

Die Blocktypen `Goto` und `From` werden in SimMo genutzt, um Verbindungen herzustellen, ohne sichtbare Linien im Modell zu ziehen. Diese Verbindungen sind über unterschiedliche Hierarchiestufen möglich. Die Verbindung erfolgt über `TAG`s, die angeben, welche Goto- und From-Blöcke verbunden sind. Um die Busstruktur aufzulösen, wird für jedes Signal in einem Bus, der über Goto- und From-Blöcke transportiert wird, ein neues System von Goto- und From-Blöcken angelegt. Dies geschieht auch über die verschiedenen Hierarchiestufen hinweg. Auch für Terminatoren wird wie in der Abbildung 6.12 gezeigt je ein Terminator für ein Signal erzeugt.

Bei Blöcken vom Typ `Merge` wird als Voraussetzung angenommen, dass die Busse gleich sind, d. h. die gleichen Signale transportieren. Die Transformation erstellt für jedes Signal, das in einem Bus enthalten ist, einen neuen Merge-Block und stellt die in der Abbildung 6.12 gezeigten Verbindungen her. Die Anzahl der Eingänge der neuen Merge-Blöcke ist gleich der Anzahl der Eingänge des ursprünglichen Merge-Blocks.

Blöcke vom Typ `Switch` werden ähnlich transformiert wie `Merge`-Blöcke. Der ursprüngliche `Switch`-Block wird durch n neue Switch-Blöcke ersetzt, wobei sich

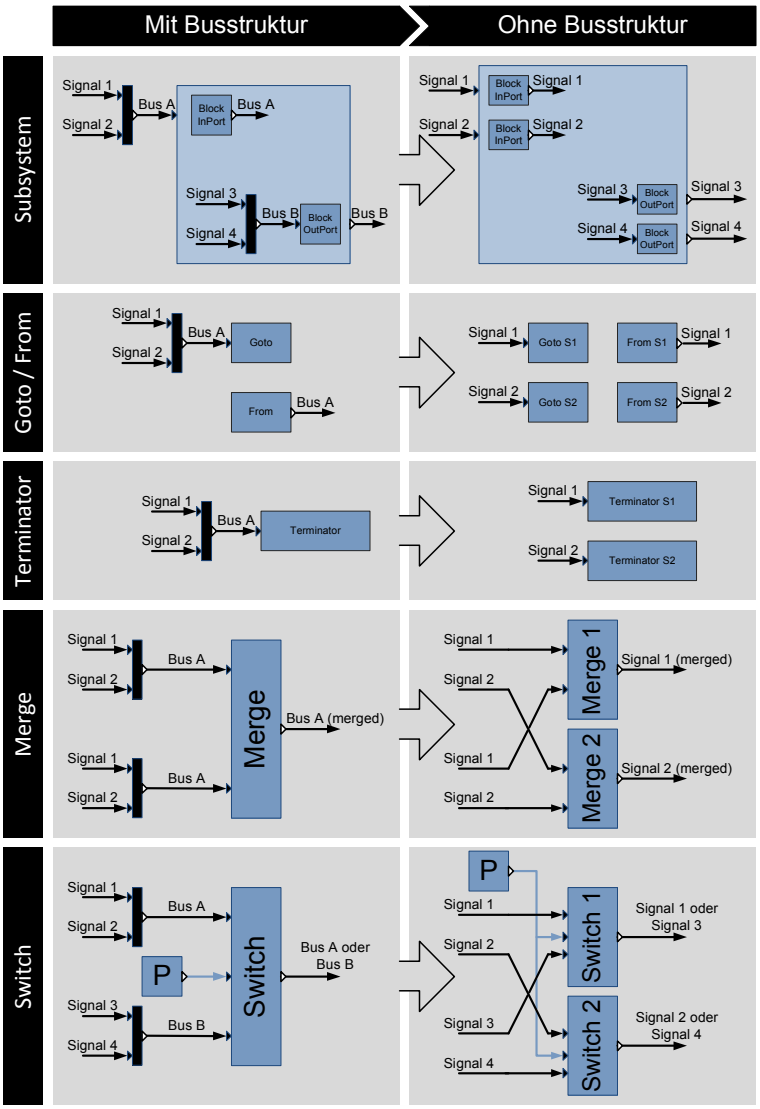

Abbildung 6.12.: Transformationsregeln zur Entfernung von Busstrukturen

deren Anzahl durch die Anzahl der Signale in den Bussen ergibt. Ist die Anzahl an Signalen in den Bussen unterschiedlich, wird eine Warnung ausgegeben und die Transformation abgebrochen. Die Steuerung der Switch-Blöcke wird von einem zentralen Punkt vorgenommen. Dieser wird von dem ursprünglichen `Switch`-Block übernommen.

Die beschriebenen Transformationsregeln werden rekursiv aufgerufen. Für jedes Subsystem werden zu Beginn alle Blöcke integriert, die keine Bussignale transportieren. Der Algorithmus beginnt mit der Bearbeitung der Bussignale. Für das Modell wird angenommen, dass jeder Bus mindestens ein Signal transportiert. Ist dies nicht der Fall, bricht der Algorithmus mit einer Fehlermeldung ab. Der Algorithmus durchläuft auf einer Hierarchieebene alle Verbindungen, die keine Busse sind und erstellt sie neu. Wird ein Signal in einem Bus abstrahiert, werden die neuen Ziele bestimmt. Sind die Ziele Blöcke, die den oben genannten Blocktypen entsprechen, werden erneut die zugehörigen Transformationsregeln angestoßen. Besonders behandelt werden die Busse, die in BlockInPorts in einem Subsystem starten. Hier werden die BlockInPorts entsprechend den Transformationsregeln behandelt und anschließend im obigen Verfahren nach den Zielen für die entsprechenden atomaren Signalen gesucht.

Identifizieren von Modulen

In der Aktivierungssicht wird neben dem Eclipse-Simulinkmodell ohne Busse auch die Identifizierung bestimmter Modellteile benötigt. Im EcSiMo müssen die enthaltenen Module für die folgende Transformation markiert werden.

Um die Module zu erkennen, ist im SimMo eine Kennzeichnung der Modelle erfolgt. Die Kennzeichnung wird über Textblöcke innerhalb des Subsystems vorgenommen. Die Module werden mit Hilfe eines Wizzards erstellt. Dies stellt eine einheitliche Struktur sicher und ermöglicht erst die Identifizierung. Erkannte Module werden im EcSiMo einen neuen Objekttyp zugeordnet. Dieser ist vom Subsystem abgeleitet und kennzeichnet die Module und ihre Eigenschaften. Das Ergebnis der Transformation ist ein Modell, in dem zwei Subsystemtypen unterschieden werden. Das Metamodell für EcSiMo ist um die Entität `Modul` erweitert worden.

Filtern der Schnittstelle

Die Transformation *Schnittstellen Filterung* führt die eigentliche Aufbereitung des EcSiMoe durch, sodass nur die notwendigen Informationen zur Verfügung gestellt werden. Die Transformation basiert auf der Transformation zur Ableitung einer identischen Transformation (Abschnitt 6.3.3).

Die Transformation lässt sich in zwei Schritte unterteilen:

1. Filtern von relevanten Blöcken unter Berücksichtigung von Varianteninformationen

2. Analyse der Verbindungen der relevanten Modellteile

Im ersten Schritt werden alle Module auf die oberste Ebene der Modells platziert. Alle weiteren Blöcke des Modells werden entfernt. Zusätzlich wird der innere

Listing 6.1: ATL-Transformation zur Bestimmung der Abhängigkeiten von Modulen

```
1 helper context MM!Block def: getTargets(modules : Set(MM!Modul))
    : Set(MM!Modul) =
2   self.outPorts -> iterate(op ; result : Set(MM!Modul) = Set{} |
3     result.union( op.linkingLine.target
4         -> iterate(targetPort; linkedSet: Set(MM!Modul)=Set{} |
5           if (thisModule.isIncluded(targetPort)) then
6             linkedSet
7           else
8             if (modules -> includes (targetPort.parent)) then
9               linkedSet.including(targetPort.parent)
10            else
11              if (targetPort.parent.oclIsTypeOf(MM!SubSystem))
                    then
12                linkedSet.union( targetPort.accordingBlock.
                      getTargets(modules))
13              else if (targetPort.parent.oclIsTypeOf(MM!
                    BlockOutPort)) then
14                linkedSet.union( targetPort.parent.accordingPort.
                      parent.getTargets(modules))
15              else
16                linkedSet.union( targetPort.parent.getTargets(
                      modules))
17              endif endif
18            endif
19          endif
20 ))) ;
```

Aufbau der Module angepasst. Die Anpassung unterscheidet, ob ein Modul der `Konfiguration` (siehe Abbildung 6.11) in der Variantenkonfiguration aktiviert ist. Ist das Modul aktiviert, werden die Subsysteme der entsprechenden Modi laut der `Aktivierungssteuerung` (Abbildung 6.10) verwendet. Ist das Modul nicht in der Variantenkonfiguration vorhanden, wird der Inhalt des `Deaktiviert`-Subsystems im Modul integriert. Alle weiteren Informationen werden entfernt.

Im zweiten Schritt werden die Abhängigkeiten zwischen den einzelnen Modulen analysiert. Die Bestimmung der Verbindungen wird über einen rekursiven Algorithmus vorgenommen. Um die Abhängigkeit zwischen Modulen zu bestimmen, wird die folgende Definition von Abhängigkeiten verwendet. Zwei Module sind abhängig, wenn es eine Verbindung zwischen ihnen gibt, die nicht über weitere Module erfolgt. Dies bedeutet, dass Anpassungen oder Abstraktionen keinen Einfluss auf die Abhängigkeit haben.

Der Algorithmus bestimmt die Verbindungen einzeln für jedes Modul. Im Listing 6.1 ist ein Auszug der Transformation gezeigt. Die Hilfsfunktion bestimmt alle Module, die mit einer Quelle verbunden sind. Das Ergebnis wird in `result` gespeichert (Zeile 2). In Zeile 5 wird geprüft, ob die Ergebnisse zur Verbindung schon berechnet wurden. Während der Berechnung werden die schon berechneten Verbindungen gespeichert, um durch die Abfrage eine doppelte Auswertung zu verhindern und einen möglichen exponentiellen Berechnungsaufwand zu begrenzen. Anschließend wird in Zeile 8 geprüft, ob ein Modul erreicht wurde. Das ist die Abbruchbedingung des Algorithmus. In den Zeilen 11-16 erfolgt der rekursive Aufruf der Hilfsfunktion. Bei Aufruf werden drei Fälle unterschieden, zum einen ob in der Subsystemhierarchie abgestiegen (Zeile 11), aufgestiegen (Zeile 13) oder ob es sich um einen anderen Block handelt. Die Hilfsfunktion wird wieder auf den so bestimmten Block angewendet.

Insgesamt liefert die Funktion die Module zurück, die mit dem ursprünglichen Modul verbunden sind. Durch die Überprüfung einer Verbindung und durch die Abbruchbedingung beim Erreichen eines Moduls wird sichergestellt, dass der Algorithmus terminiert und jede Verbindung höchstens einmal ausgewertet wird. Für den Fall, dass die Menge an gefundenen Modulen nicht leer ist, wird eine neue Verbindung im Ergebnismodell erstellt. Das Ergebnis wird in einem EcSiMo gespeichert.

Graphischer Editor

In den vorherigen Kapiteln wird die Erstellung des Ergebnismodells beschrieben. Dieses kann mit Hilfe des Standardeditors angezeigt werden. Diese Anzeige gibt allerdings keinen Überblick über die Beziehung und die Aktivierungszustände des Modells. Dies ist die Hauptanforderungen, die mit der Analyse erfüllt werden soll.

Um eine entsprechende Übersicht zu gewährleisten, müssen die inneren Strukturen der Module auch auf der obersten Ebene einsehbar sein. Dies wird mit einem speziellen graphischen Editor erreicht, der eine neue Sicht auf das Ergebnismodell ermöglicht. In diesem werden auch die inneren Sturkturen der Modelle dargestellt.

Ein Ergebnis der notwendigen Anpassungen der Darstellung ist in der Abbildung 6.13 anhand eines Beispiels zu sehen. Hier sind die inneren Strukturen der einzelnen Module eines SimMo direkt auf der obersten Ebene dargestellt. Hier wird das Modul `SoCProcessing`, in der dargestellten Variante, aktiviert. Die im Modul dargestellte Struktur zeigt, dass dieses Modul in dem Modus `PADJ_SoCEnable` berechnet wird. Zusätzlich ist in der Abbildung 6.13 zu sehen, dass dieses Modul Eingaben für die beiden Module `DynamischesLastManagement` und `Ruhestromschalter` liefert.

Das Modul `DynamischesLastManagement` ist vom Variantenmanagement aktiviert. Es berechnet Ergebnisse in den Betriebsmodi `DrivingMode_b`, `RadioMode_b` `PADJ_DLMEnable`. Die Betriebsmodi unterscheiden sich vom Modul `SoCProcessing`. An dieser Stelle muss der Ingenieur entscheiden, ob dies einen Fehler verursachen

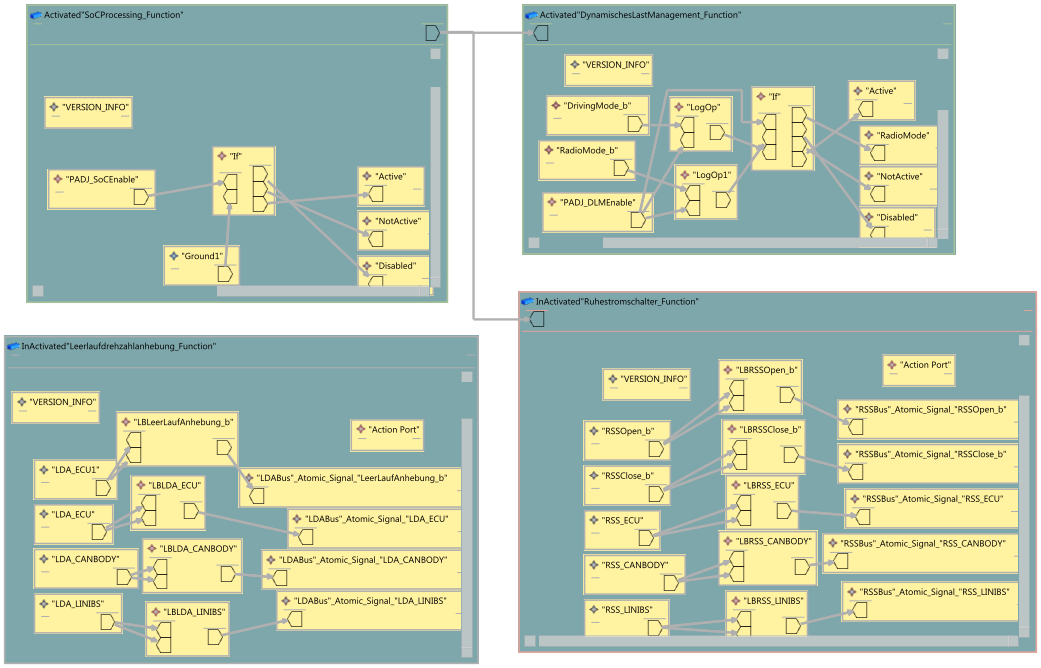

Abbildung 6.13.: Anzeige des graphischen Eclipse-Simulinkeditors

kann. Ein Fehler würde nicht auftreten, wenn der Betriebsmodus `PADJ_SoCEnable` länger aktiv ist als die Betriebszustände des abhängigen Moduls.

Das zweite abhängige Modul `Ruhestromschalter` muss hingegen nicht betrachtet werden, da es im abgeleiteten Produkt enthalten ist. Dies ist zum einen an der Benennung und zum anderen an der dargestellten inneren Struktur zu sehen. Die Struktur zeigt, welche Standardwerte in diesem Fall erzeugt und an das Modell weitergegeben werden.

Der Editor zeigt eine mögliche Analyse für das beschriebene Szenario. Für die Anordnung der inneren Strukturen der Module wird die ursprüngliche Position aus dem Quellmodell übernommen. Der Editor zeigt zusätzlich Abhängigkeiten zwischen den verschiedenen Modulen in einer abstrahierten Form.

6.4.2. Analyse der Schnittstellen

Neben der Analyse, welche Module zu verschiedenen Betriebszuständen laufen, ist es für den Entwickler wichtig, die modellierten Abhängigkeiten zwischen Modulen zu erkennen. Diese Analyse der Schnittstelle muss vor jeder Veränderungen an der Modellstruktur vorgenommen werden. Eine Änderung kann beispielsweise durch eine neue Funktionalität notwendig werden. Auf der Analyse der Schnittstellen, von

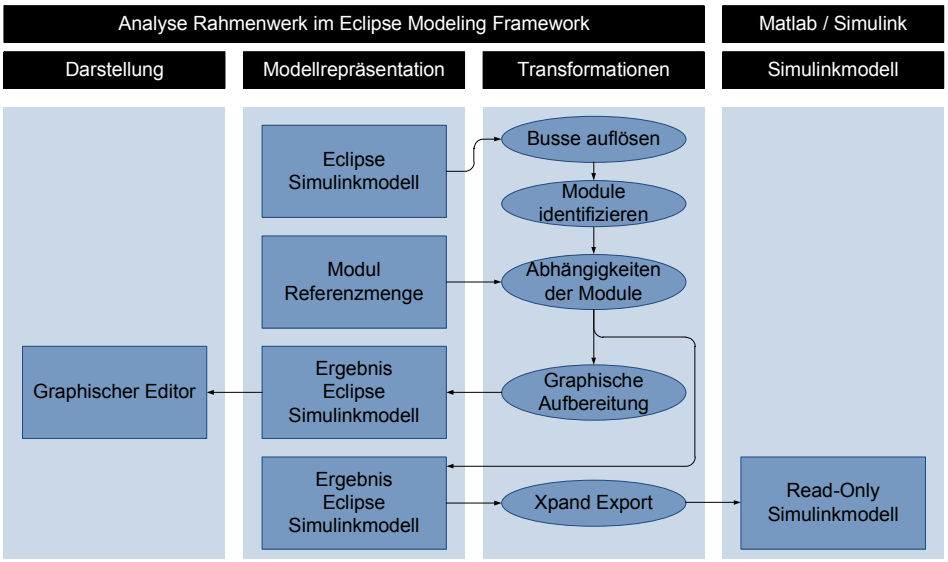

Abbildung 6.14.: Struktur der Transformationen zur Analyse der Modulschnittstellen

bestehenden Module und deren Abhängigkeiten untereinander, basiert beispielsweise die Entscheidung, wie neue Funktionalität integriert wird. Daher ist es eine immer wieder vorkommende Aufgabe, die Schnittstellen und die Abhängigkeiten von einer Menge von Modulen in einem Modell zu bestimmen.

Um die Analyse der Schnittstelle durchzuführen, wird das EcSiMo benötigt. In diesem Modell sind alle notwendigen Informationen enthalten. Die Darstellung des Ergebnisses erfolgt dann entweder im Standard Eclipse-Simulinkeditor (EcSiMo-Editor) oder in MaSi. Der Aufbau der Transformation ist in der Abbildung 6.14 zu sehen. Zusätzlich zum EcSiMo wird als Eingabe noch die Menge der zu analysierenden Module benötigt.

Die Struktur der Transformationen ändert sich wenig gegenüber der Struktur der Aktivierungssicht. Die Transformationen `Busse auflösen` und `Module identifizieren` werden ohne Änderungen übernommen. Die eigentliche Analyse wird in der Transformation `Abhängigkeiten der Module` vorgenommen.

Im Unterschied zur gezeigten Aktivierungssicht im Abschnitt 6.4.1 werden die Abhängigkeiten zwischen den Modulen in dieser Sicht nicht abstrahiert dargestellt. Dies bedeutet, dass jede Verbindung mit Benennung des transportierten Signals dargestellt wird. Zusätzlich werden die Schnittstellen der Module nicht angepasst, d. h. alle Ein- und Ausgänge werden angezeigt.

Die Hauptaufgabe der Transformation ist die Bestimmung der Abhängigkeiten. Der Algorithmus berechnet die bestehenden Verbindungen zwischen den Modulen, indem

für jeden Ausgang die Ziele bestimmt werden und diese Menge dann anschließend mit der Menge der Module geschnitten wird. Zur Bestimmung der Ziele werden die während des Imports berechneten logischen Signale verwendet. Dies ist möglich, da im SimMo alle Verbindungen benannt sein müssen. Für logische Signale gilt zusätzlich, dass diese nur einmal erzeugt und dann an verschiedenen Stellen weitergegeben werden können.

Das so entstandene Modell hat nun alle benötigten Informationen. Die sich anschließenden Transformationen dienen der Darstellung des Ergebnisses. Diese erfolgt sowohl im graphischen EcSiMo-Editor als auch in MaSi. Die beiden notwendigen Transformationen (siehe Abbildung 6.14) sind die Standardtransformationen und erstellen Linien für den GMF-Editor bzw. ein neues SimMo.

6.5. Variabilitätmechanismen in der modellbasierten Entwicklung

Neben der Analyse von variantenreichen Modellen ist die Erstellung und Verwaltung von Produktlinienmechanismen eine immer wiederkehrende Aufgabe. Der Entwickler muss die Prozessartefakte *Implementationsmodell*, *Spezifikation*, *Testfälle* und *Variabilitätsmodell* erstellen und pflegen. Zusätzlich müssen alle Prozessartefakte noch verknüpft werden, um Mechanismen zur Ableitung der Produkte verwenden zu können.

In den folgenden Unterkapiteln werden drei Verfahren vorgestellt zur Unterstützung des Entwicklers bei der Produktlinienentwicklung. Diese sind zum einen ein regelbasiertes Verfahren, um Prozessartefakte zu verknüpfen (Abschnitt 6.5.1). Anschließend werden die Verknüpfungen zwischen den Prozessartefakten genutzt, um mittels einer Transformation die Ableitung von Produkten verschiedener Prozessartefakte zu ermöglichen (Abschnitt 6.5.2). Der letzte Schritt (Abschnitt 6.5.3) bezieht sich auf ein Matlab / Simulinkmodell (SimMo). Nach der Ableitung eines Produkts werden mit Hilfe einer Transformation nicht mehr benötigte Linien und Blöcke entfernt.

6.5.1. Verknüpfung von Prozessartefakten

Die Grundlage der Konzepte in dieser Arbeit zum Variantenmanagement sind Modelle, die miteinander in Beziehung stehen. Diese Verknüpfung zwischen verschiedenen Modellen deutet einen Zusammenhang an, bei dem das zentrale Element ein Feature-Modell (FM) ist. Die Beziehungen über das Feature zeigen, an welcher Stelle Entitäten implementiert, dokumentiert oder getestet werden. Dadurch ist es insgesamt möglich, den Entwickler bei der Arbeit zu unterstützen und verschiedene Analysen und Mechanismen einzusetzen.

Aufgrund der Anzahl möglicher Verknüpfungen ist es sehr aufwändig, die Verknüpfungen manuell zu erzeugen. Eine vollständige automatische Erzeugung der Verknüpfungen ist aber schwierig zu erreichen, da es immer wieder verschiedene Situationen gibt, bei denen spezielle Verknüpfungen benötigt werden.

In dieser Arbeit wird aus diesem Grund ein Ansatz gewählt, der den Entwickler durch eine Kombination von automatischen und manuellen Mechanismen unterstützt, die Verknüpfung von verschiedenen Modellen vorzunehmen. Die automatischen Mechanismen basieren auf konfigurierbaren Transformationen, die eine Beeinflussung der Verknüpfungen ermöglichen. Die manuelle Verknüpfung wird durch einen graphischen Editor unterstützt, der bestehende Verknüpfungen übersichtlich darstellt und es ermöglicht, diese zu verändern oder zu ergänzen (Der graphische Editor wird im Abschnitt 6.5.1 vorgestellt). Mit diesem Konzept ist es möglich, automatisch erzeugte Verknüpfungen zu korrigieren und zu ergänzen und insgesamt beide Techniken zusammen zu benutzen.

Im Folgenden wird der regelbasierte Ansatz zur automatischen Ableitung von Verknüpfungen vorgestellt. Das zentrale Element stellt in diesem Produktlinienansatz das Variabilitätsmodell dar. Die Verknüpfungen werden aus diesem Grund immer von einem Feature zu einem der anderen Prozessdokumente hergestellt. Die Abbildung 6.15 zeigt ein Metamodell mit den Verknüfungsstrukturen. Der linke Teil der Abbildung 6.15 zeigt die zentrale Position des FMs, mit dem alle weiteren Prozessartefakte verknüpft werden. Auf der rechten Seite ist das Metamodell der Verknüpfungsmodelle. Die Entitäten mit grauem Hintergrund stammen aus den entsprechenden verknüpften Metamodellen der Prozessartefakte. Die weißen Entitäten sind Teil des eigentlichen Metamodells. Wichtig ist die abstrakte Entität `LinkedItem`. Diese ermöglicht im Folgenden eine verallgemeinerte Entwicklung des Verknüpfungsregelkonzepts.

Die automatische Verknüpfung der Prozessartefakte ist mittels einer Transformation realisiert. Um diese Transformation steuern zu können, sind verschiedene Typen von Regeln entwickelt worden, die der Entwickler nutzen kann. Der Ablauf der Transformation ist in der Abbildung 6.16 zu sehen. Die Verknüpfungsregeln werden in einer textbasierten Form als Eingabe der Transformation verwendet. Eine Übersetzung in ein *Modell der Verknüpfungsregeln* erfolgt mittels einer Xtext-Transformation. Die Regeln und die beiden zu verknüpfenden Modelle bilden dann die Eingabe für die Transformation zur *Erstellung der Verknüpfungen*. Diese erzeugt ein Verknüpfungsmodell, das dem Metamodell der Transformation entspricht.

Die verschiedenen Arten der Verknüpfungsregeln wurden mit Entwicklern der Industrie definiert. Für die Regeln sind die Annahmen getroffen worden, dass jede zu verknüpfende Entität einen Namen und eine eindeutige Kennung besitzt. Diese Annahme gilt sowohl für die Features als auch für die Entitäten der Prozessartefakte. Diese Annahmen entsprechen den oftmals in Unternehmen geltenden Namenskonventionen und auch der Möglichkeit des Eclipse Modeling Frameworks

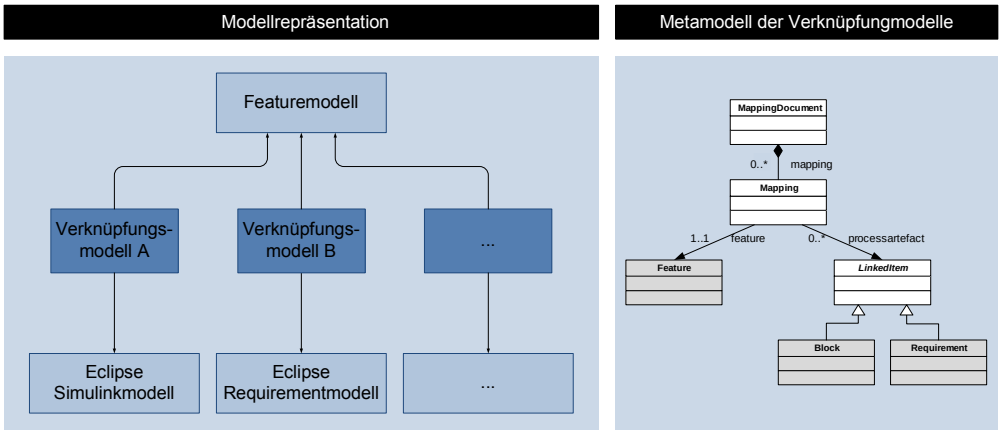

Abbildung 6.15.: Verknüpfung der Prozessartefakte untereinander

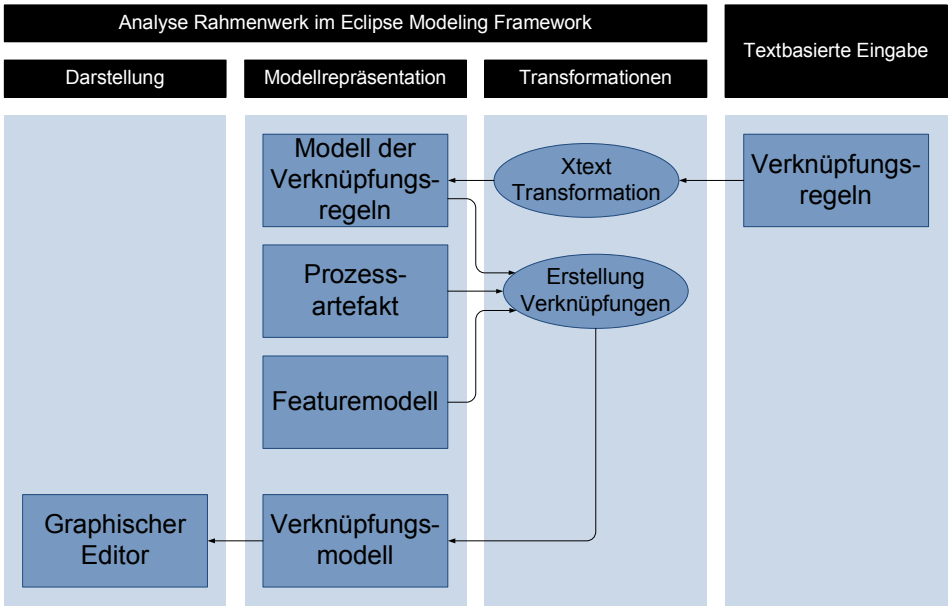

Abbildung 6.16.: Regelbasierte Erstellung von Verknüpfungen

(EMFs), eindeutige Kennungen zu generieren. Aus den Namenskonventionen und den eindeutigen Kennungen sind die folgenden Regelarten abgeleitet worden:

1. Feature und Entität des Prozessartefakts sind bekannt und werden eindeutig durch den Namen oder Kennung als zu verknüpfend angegeben.

2. Die Beziehung der Feature und Entität der Prozessartefakte sind für ein

spezielles Feature bekannt. Das Feature wird über einen Namen oder Kennung identifiziert. Die abhängigen Komponenten des Prozessartefakts werden durch den Namenspräfix des Features festgelegt.

3. Eine weitere Regel erweitert die vorherige Regel, indem das Namenspräfix durch eine gleiche Zeichenkette einer bestimmten Länge festgelegt wird.

4. Feature und Entitäten eines Prozessartefakts werden verknüpft, bis die Namensgleichheit zu einem festgelegten Trennzeichen vorliegt.

5. Zusätzlich gibt es eine Verallgemeinerung der vorherigen Regel, bei der ein globales Trennzeichen angegeben wird, das alle Feature und Entitäten mit gleichem Namen bis zum Trennzeichen verknüpft.

6. Als letzte Regel gibt es die Möglichkeit, alle Feature und Entitäten des Prozessartefakts miteinander zu Verknüpfen, die genau die gleichen Namen haben.

Die Eingabe der Regeln erfolgt mittels einer domänenspezifischen Sprachen. Die Implementierung der Sprache erfolgt mittels Xtext. Der Entwickler erhält damit einen Editor, der ihn bei der Erstellung der Regeln mit Syntaxhighlighting, Autocompletion, Warnungen und Fehlermeldungen unterstützt. Zusätzlich können die Eingabe in Form von EMF-Modellen abgespeichert werden und somit direkt für weitere Transformationen genutzt werden. Dieses Modell dient als Eingabe, um die Verknüpfungen zwischen den Modellen zu generieren. Die dazu notwendige Transformation wird im folgenden Kapitel beschrieben.

Erstellung von Verknüpfungen

Die Umsetzung der Verknüpfungsregeln in Verknüpfungen zwischen den einzelnen Artefakten wird von einer Transformation durchgeführt. Diese erzeugt aus jeder Regel eine oder mehrere Verknüpfungen. Jede Regelart wird separat betrachtet und umgesetzt. Daher werden zum Teil redundante Verknüpfungen erzeugt. Die Umsetzung der verschiedenen Regeln wird im Einzelnen betrachtet.

Bei der Transformation werden die Regeln nacheinander abgearbeitet. Es möglich ist, dass sich Regeln widersprechen. Es hat sich aber gezeigt, dass eine lineare Bearbeitung der eingegebenen Regeln gut nachzuvollziehen ist. Das Ergebnis der Regeln hängt dann neben der Interpretation der Funktionalität auch von der Implementierung (z. B. Reihenfolge der Regeln) ab. Um die Ergebnisse bewerten zu können, wird in den Verknüpfungen gespeichert, durch welche Regel sie erzeugt worden ist.

Um die Ergebnisse der Regeln zusätzlich besser interpretieren zu können, werden die Regeln auch dann angewendet, wenn die Voraussetzung der Eindeutigkeit von Featurenamen nicht gegeben ist. Diese Regeln werden dann als nicht eindeutig

gekennzeichnet. Zusätzlich werden Warnungen durch die Transformation ausgegeben. Durch dieses Vorgehen wird es dem Entwickler ermöglicht zu sehen, welche der getroffenen Annahmen verletzt wurde.

Im Listing 6.2 ist auszugsweise die Implementierung der Hardlink-Regeln gezeigt. Eine Voraussetzung der Regel ist ein Feature, dass durch einen Namen oder eine Kennung eindeutig identifiziert wird. Zusätzlich muss gelten, dass die Entitäten des zu verknüpfenden Prozessdokuments eindeutig sind. Sind die Voraussetzungen erfüllt, hat das zur Folge, dass eine Hardlink-Regel genau zu einer Verknüpfung führt. Diese Bedingung wird in den Zeilen 4 und 5 geprüft.

Verknüpfungseditor

Wie in den vorherigen Kapiteln beschrieben, ist neben der regelbasierten auch eine manuelle Möglichkeit vorgesehen, Verknüpfungen zu erstellen. Die manuelle Eingabe der Verknüpfungen erfolgt mit einem graphischen Editor. Dieser hat die Aufgabe, alle möglichen Verknüpfungen zwischen den Features und den weiteren Prozessartefakten darzustellen.

Der Verknüpfungseditor wird eingesetzt, um die Ergebnisse der automatischen Transformationen zu überprüfen und gegebenenfalls zu korrigieren. Zentrale Elemente sind Features, von denen alle Verknüpfungen ausgehen. Die Verknüpfungen können mit verschiedenen Prozessartefakten erfolgen, deren Darstellung immer gleich ist.

Die Anzahl der darzustellenden Verbindungen ist bei realen Beispielen meist hoch. Um den Entwickler hier zu unterstützen, sind verschiedene Darstellungen und Filter vorgesehen. Dabei ist es möglich, die hierarchischen Eigenschaften der Prozessartefakte verschiedener Hierarchieebenen darzustellen. Als Voraussetzung für diese Darstellung wird davon ausgegangen, dass die Verbindungen eindeutig sind und immer nur auf der niedrigsten Hierachieebenen erfolgen. Eine abstrakte Darstellung kann dann durch Modelltransformationen abgeleitet werden. Zusätzlich werden zwei Verknüpfungstypen (manuell und automatisch) unterschieden. Bei den automatisch erzeugten Verknüpfungen wird angegeben, welche Regel der Verknüpfung zu Grunde liegt und von welchem Typ die Regel ist.

Mit Hilfe dieses Verknüpfungseditors ist es zum einen möglich, manuell neue Verknüpfungen zu erzeugen, zum anderen ist eine Überprüfung und eventuelle Korrektur der regelbasierten Verknüpfung möglich. Der Editor hat eine einheitliche Darstellung für alle Prozessartefakte. Dadurch wird eine Unabhängigkeit und leichte Erweiterbarkeit erreicht.

Listing 6.2: ATL-Transformation zur Erzeugung von Verknüpfungen

```
1  rule HardLink2Link {
2    from hl : MMFsmaprules!HardLink(
3      ( not (MMFmprimitives!Feature.allInstances()->select(f|f.name
           = hl.feature.name))->first().oclIsUndefined()
4        and MMFmprimitives!Feature.allInstances()->select(f|f.name
             = hl.feature.name).size()<2 )
5      or not (MMFmprimitives!Feature.allInstances()->select(f|f.id
           = hl.feature.id))->first().oclIsUndefined() )
6    to l : MMFsmap!Link (
7      [...]
8      linkedFeature<-(if (hl.feature.name.oclIsUndefined())
9        then
10         (MMFmprimitives!Feature.allInstances()->select(f|f.id =
               hl.feature.id))->first()
11       else
12         (MMFmprimitives!Feature.allInstances()->select(f|f.name =
               hl.feature.name))->first()
13       endif
14   )   )
15   do {
16     for (mItem in hl.item) {
17       for (fItem in mItem.getItem()) {
18         l.linkedItem<-(fItem);
19     }   }
20     l.intoMap();
21 }   }
22
23 rule HardLink2ManyLinks {
24   from hl : MMFsmaprules!HardLink(
25       not (MMFmprimitives!Feature.allInstances()->select(f|f.name
           = hl.feature.name))->first().oclIsUndefined()
26       and MMFmprimitives!Feature.allInstances()->select(f|f.name =
             hl.feature.name).size()>1
27       and (MMFmprimitives!Feature.allInstances()->select(f|f.id =
             hl.feature.id))->first().oclIsUndefined()
28   )
29   using {
30     n : Integer = 0;
31   }
32   to l : MMFsmap!Link
33   do {
34     for (f in MMFmprimitives!Feature.allInstances()->select(f|f.
           name = hl.feature.name)) {
35       thisModule.HardLink2Links(f, hl, n);
36       n <- n + 1;
37 }   }   }
```

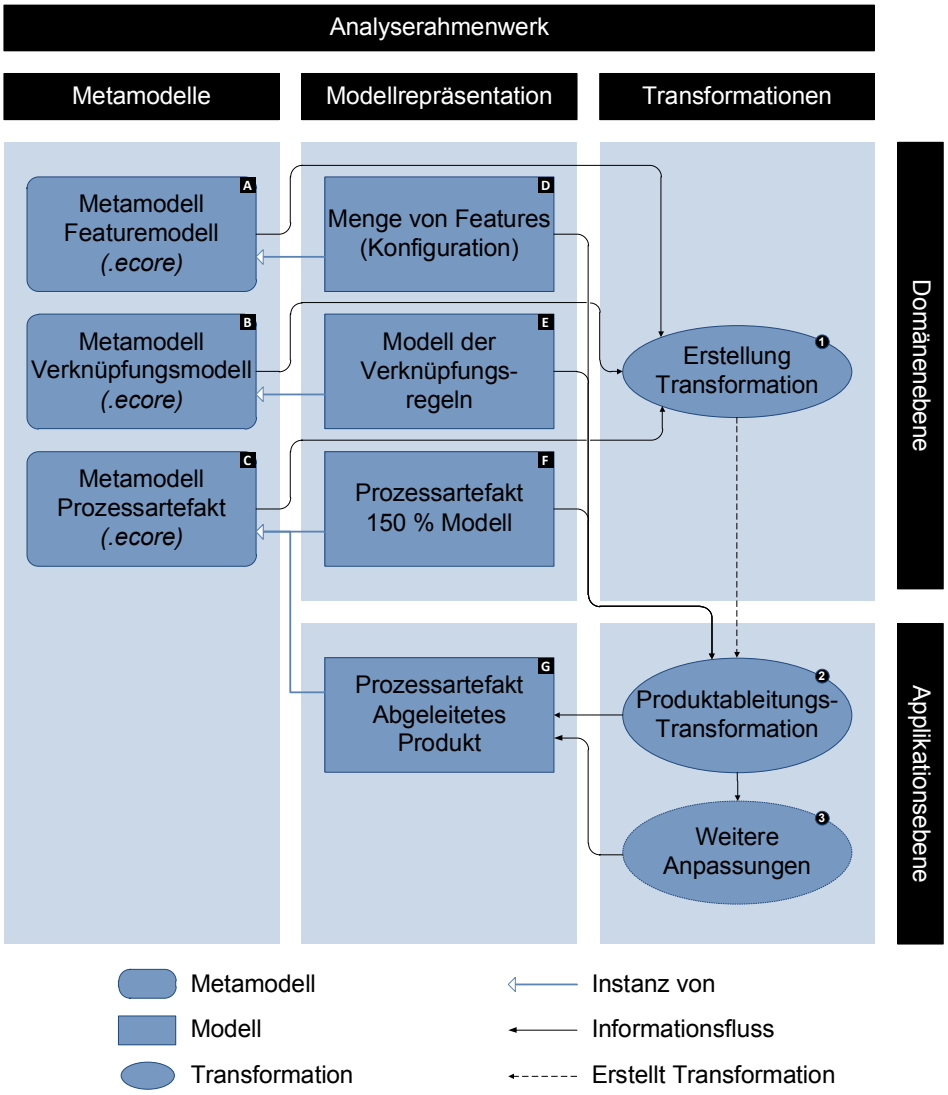

Abbildung 6.17.: Regelbasierte Erstellung von Verknüpfungen

6.5.2. Ableiten von Produkten

Die Ableitung von Produkten erfolgt mit dem Mechanismus der negativen Variabilität. Die Dokumente der Domänenebene sind dann 150 % Modelle und enthalten alle möglichen Entitäten. Ein abgeleitetes Produkt der Produktlinie wird dann durch Löschen von einzelnen Teilen erzeugt. Im Rahmenwerk ist die Funktionalität unabhängig von den abzuleitenden Dokumenten zur Verfügung gestellt.

Um die Funktionalität der Ableitung unabhängig von den abzuleitenden Produkten

zu machen, wird die Funktionalität allgemein für Metamodelle definiert und nicht direkt für die Produkte. Dadurch ist es möglich, die Ableitungsfunktionalität wieder zu verwenden.

Einen Überblick über die Eingaben ist in der Abbildung 6.17 zu sehen. Die Ableitung erfolgt in einem zweistufigen Prozess. In der ersten Phase wird aus dem dem Metamodell der Features **A**, dem Metamodell der Verknüpfungen **B** und dem Metamodell der Prozessartefakte **C** (Domänenmodell) eine *Produktableitungs-Transformation* ❷ erstellt. Diese Transformation ❶ ist in der Lage, eine identische Kopie des Dokuments zu erstellen. Dazu werden einzelne Kopierregeln erstellt, die alle Entitäten des Metamodells erfassen.

Zusätzlich wird die Menge der Entitäten des abzuleitenden Modells bestimmt, welche von der Ableitungsfunktionalität betroffen ist. Diese Menge wird bestimmt durch das Metamodell der Verknüpfungen. In der Menge sind genau die Entitäten enthalten, die von der abstrakten Klasse `LinkedItem` abgeleitet werden (siehe Abbildung 6.15). Die Elemente dieser Menge erhalten in der Produktableitungs-Transformation spezielle Kopierregeln. Diese Kopierregeln beachten die Produktkonfigurationen und transformieren nur die Entitäten, die in der Konfiguration enthalten sind.

In der zweiten Phase erfolgt die Anwendung der Produktableitungs-Transformation ❷. Die Transformation erstellt aus einem Prozessartefakt (Domänenmodell) **F**, einer Produktkonfiguration **D** und einem Verknüpfungsmodell **E** das abgeleitete Modell der Applikationsebene **G**. Die Anwendung der angepassten Kopierregeln stellt sicher, dass nur die Entitäten im Modell vorhanden sind, die in der Konfiguration enthalten sind. Um den allgemeinen Charakter einer Transformation zu erhalten, werden notwendige Anpassungen des SimMos in einer weiteren Transformation ❸ ausgelagert. Für die Ableitung der anderen Prozessartefakte ist keine Anpassung notwendig.

6.5.3. Anpassung des Implementationsmodells

Auf der Applikationsebene sind für das Implementationsmodell besondere Schritte notwendig, um ein abgeleitetes Produkt und eine korrekte Struktur des SimMos zu erhalten. Der eigentliche Ableitungsvorgang findet in zwei Schritten statt. Der erste Schritt ist die Anwendung der Konfiguration, wie sie im vorherigen Abschnitt (siehe Abschnitt 6.5.2) beschrieben wird.

Die Ableitung des Modells wird auf Blöcke des Modells angewendet. Dies bedeutet, dass Blöcke, die nicht benötigte Funktionalität beinhalten, aus dem Modell entfernt werden. Eine zusätzliche Anpassung findet nicht statt. Durch das Löschen der Blöcke entstehen im Modell Situationen, deren Struktur nicht immer den üblichen Modellierungsrichtlinien entsprechen. Dies betrifft vornehmlich die Verbindungsstrukturen im Modell wie beispielsweise Verbindungslinien und Schnittstellen verschiedener Blocktypen.

Um diese Strukturen wieder den Modellierungsrichtlinien entsprechend anzupassen, wird eine Verbindungsanalyse des Modells durchgeführt. Dazu wird getestet, ob alle Verbindungen jeweils einen Start- und Zielpunkt haben. Ist dies nicht der Fall, werden die entsprechenden Verbindungen entfernt. Dazu werden auch die Schnittstellen der Blocktypen `Merge`, `Subsystem`, Busstrukturen und `Goto`- bzw. `From`-Blöcke angepasst und rekursiv weiterverfolgt. In einer zweiten Phase werden ausgehend von allen Schnittstellen die oben genannten Blocktypen geprüft. Wird eine Schnittstelle gefunden, bei der kein Signal vorhanden ist, werden wieder rekursiv alle Strukturen (Verbindungen und Schnittstellen) entfernt, bis wieder ein richtlinienkonformes Modell entsteht.

Ein konformes Modell wird nicht immer erreicht. Die Transformation stoppt, sobald andere als die beschriebenen Blocktypen erreicht werden mit einer entsprechenden Warnung. In diesem Fall wird angenommen, dass dort weitere Funktionalität implementiert ist, die von der Analyse sonst eventuell entfernt wird. Dies widerspricht der Annahme, dass die Transformation nur strukturelle Anpassungen vornimmt und keine Funktionalität verändert.

Kapitel 7.

Verwandte Arbeiten

In diesem Kapitel werden verwandte Forschungsarbeiten vorgestellt, die sich wie diese Arbeit mit der modellbasierten Produktlinienentwicklung für eingebettete Systeme beschäftigen. Eine allgemeine Plattform, deren Quelltexte frei verfügbar sind, ist *OpenEmbeDD* [22]. Diese Plattform konzentriert sich auf eingebettete Systeme mit Echtzeitanforderungen. Sie stellt eine Menge von Werkzeugen bereit, die auf dem EMF basieren. Diese Werkzeuge unterstützen Entwickler bei der Modellierung, Simulation und Validierung von eingebetteten Systemen. Die Plattform bezieht sich nicht auf die Produktlinienentwicklung.

Im Folgenden werden Ansätze zu verschiedenen Mechanismen aus dem Bereich der Produktlinienentwicklung betrachtet:

Agrawal et al. [2] präsentiert in diesem Zusammenhang einen Ansatz, der sich auf die Semantik und Verifikation konzentriert. Sie analysieren in ihrer Arbeit Matlab / Simulinkmodelle und Stateflow-Modelle. Diese Modelle werden mit Hilfe der Graph Rewriting und Transformations Sprache (GReAT) in das Hybride Systeme Interchange Format (HSIF) übersetzt. Im Gegensatz zum Ansatz der vorliegenden Arbeit liegt der Fokus von Agrawl et al. aber nicht in der Ableitung von Produkten, sondern auf der semantischen Analyse. Zu diesem Zweck führen sie eine Interpretation der Modelle durch.

In den wissenschaftlichen Arbeiten von Biehl et al. [4] werden Modelltransformationen eingesetzt, um sicherheitskritische eingebettete Systeme zu analysieren. Dabei werden Beschreibungen in der Beschreibungssprache EAST-ADL3 als Eingaben für das Werkzeug HiP-HOPS genutzt. Ähnlich zu unserem Ansatz werden kontextabhängige Sichten genutzt, um die sicherheitskritischen Systeme zu analysieren. Der Fokus von Biehl et al. liegt nicht auf der Produktlinienentwicklung.

Im Bereich der Produktableitung gibt es einige verwandte Froschungen. Czarnecki und Antkiewicz [10] zeigen in ihrem template-basierten Ansatz eine Möglichkeit, um Verknüpfungen zwischen Feature-Modell und dem Implementationsmodell herzustellen. Hierfür nutzen sie die Object Constraint Language (OCL) [34]. In ihrem Konzept nutzen sie eine Idee vergleichbar zu einem 150% Modell und leiten Produkte durch negative Variabilität ab. In der OCL hinterlegen sie Abhängigkeiten der UML Fragmenten.

In den Arbeiten von Heidenreich et al. [19] wird das Werkzeug FeatureMapper präsentiert. Dieses Werkzeug ermöglicht die Verknüpfung von Features und Implementationsmodell-Elementen. Dieser Forschungsansatz ähnelt dem in dieser Arbeit gezeigten Verfahren zur Verknüpfung. Allerdings nutzt FeatureMapper allgemeine EMF-Modelle im Gegensatz zu dem in dieser Arbeit genutzten Verknüpfungen, die direkt auf die konkreten Modelleelemente referenziert.

Voelter und Groher [44] beschreiben aspekt- und modellbasierte Lösungen, um Produktlinienentwicklung zu betreiben. Ihre Forschung basiert auf Variabilitätsmechanismen in openArchtitektureWare [35]. Dort wurden Werkzeuge wie Xvar und XWeave entwickelt, um Produktableitungen von EMF-Modellen zu ermöglichen. Als beispielhafte Implementierung nutzten sie ein Heimautomatisierungssystem. Dabei konzentierten sie sich auf die aspektorientierte Ableitung von Produkten. Ihre Ansätze der aspektorientierten Produktlinienentwicklung könnten auch in diesem Bereich der eingebetteten Systeme eingesetzt werden.

Teil III.

Evaluierung

Kapitel 8.

Anwendung und Evaluierung der Rapid-Control-Prototyping Architektur

Die vorangegangenen Kapitel beschreiben Konzepte und Umsetzungen eines Produktlinienansatzes für die modellbasierte Entwicklung von eingebetteten Systemen. Der Ansatz kombiniert die Entwicklungsumgebung Matlab / Simulink (MaSi) mit Konzepten zur Integration von Sensoren und Aktuatoren sowie Analysen und Anpassungen um Variabilität auf verschiedenen Ebenen zu verwalten. Die Anwendung der Konzepte erfolgt im ersten Teil am Beispiel eines Parkassistenten, Im zweiten Teil erfolgt eine Evaluierung durch eine Befragung von Entwicklern und einer Analyse deren Bewertungen.

8.1. Parkassistent

Die Implementierung eines Parkassistenten dient als Beispiel zur Entwicklung von Regelungssoftware. Dieser Assistent basiert auf einem experimentellen Fahrzeug dessen technische Architektur in Abbildung 8.1 dargestellt ist.

Die gesamte Applikation wird gesteuert über einen Mikrocontroller, der über eine Vielzahl von Sensoren die Umgebung erfassen kann. Der Mikrocontroller hat Zugriff auf die Geschwindigkeitsinformation des Fahrzeugs, Distanzen in verschiedenen Richtungen und die Ausrichtung des Fahrzeugs. Zusätzlich werden noch Eingaben, die mittels einer Fernbedienung vom Benutzer erfolgen, eingelesen. Diese Informationen werden über im Modellbau übliche Komponenten übertragen.

Neben den Sensoren hat der Mikrocontroller auch die Möglichkeit, das Fahrzeug zu beeinflussen. Dazu können drei Servomotoren angesteuert werden, die zum einen die Lenkung und zum anderen zwei Bremskreisläufe (hintere und vordere Achse) beeinflussen. Das Fahrzeug kann über einen Elektromotor beschleunigt werden.

Die Sensoren haben verschiedene Charakteristiken, die Einfluss auf das Verhalten des Reglers haben. Die Distanzsenoren liegen in den zwei Ausführungen (als Ultraschall- und Infrarotsenor) vor. Diese beiden Sensortypen unterscheiden sich in der Zeitdauer einer Messung, der Genauigkeit, der Zuverlässigkeit und dem Messbereich.

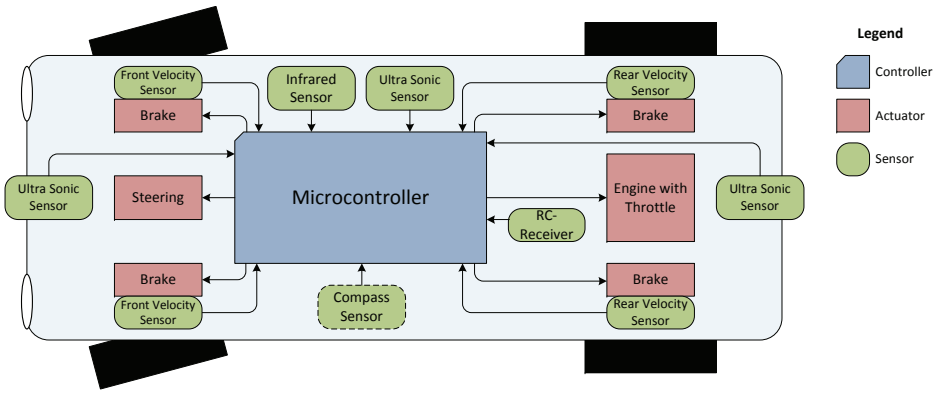

Abbildung 8.1.: Technische Architektur mit Komponenten und Verbindungen des Versuchsfahrzeugs

Der Einfluss der Sensoren auf einen Regler kann im Folgenden am Beispiel eines Ultraschallsensors verdeutlicht werden. Dieser benötigt für eine Entfernungsmessung eine Zeitspanne von 60 Millisekunden (abhängig von der eingestellten maximalen Messweite). Jede Berechnung des Regler benötigt einen neuen Messwert. Daraus folgt, dass der Regler eine minimale Schleifenzeit von mehr als 60 Millisekunden hat. Diese Abhängigkeit zwischen Sensoren bzw. Aktuatoren und dem Regler zeigen, dass Veränderungen an den unteren Schichten große Änderungen der funktionalen Eigenschaften zur Folge haben. Für den Fall des Parkassistenten kann beispielsweise eine Umstellung von Infrarot- zu Ultraschallsensoren bedeuten, dass die Einparkgeschwindigkeit deutlich herabgesetzt werden muss.

Unabhängig von der Sensor- / Aktuatorkonfiguration wird der Einparkvorgang in drei Phasen ausgeführt. In der ersten Phase wird die Parklücke vermessen. Die Vorbeifahrt an der Lücke wird durch den Benutzer manuell ausgeführt. Bei entsprechender Aktivierung messen die Geschwindigkeits- und Entfernungssensoren passende Parklücken auf der rechten Seite aus. Während jedes Durchlaufs der Applikation auf dem Mikrocontroller wird die zurückgelegte Distanz berechnet und die Tiefe einer möglichen Parklücke vermessen. Die Parklücken müssen eine Mindesttiefe und -länge haben, um von dem System als gültig akzeptiert zu werden. In der zweiten Phase hat der Bediener des Systems zehn Sekunden Zeit, um den Einparkassistenten zu aktivieren. In der dritten Phase erfolgt dann der Einparkvorgang. Dieser beginnt mit einer Initialisierungsphase, in der das Fahrzeug durch den Regler in eine definierte Position zur Parklücke gebracht wird. Anschließend erfolgt das Einparken. Dabei wird durch jeden Sensor fortwährend geprüft, ob Hindernisse im Weg sind. Wird ein Hindernis erkannt, bricht der Vorgang ab. Zusätzlich hat der Benutzer die Möglichkeit, ständig einzugreifen und damit den Vorgang manuell abzubrechen.

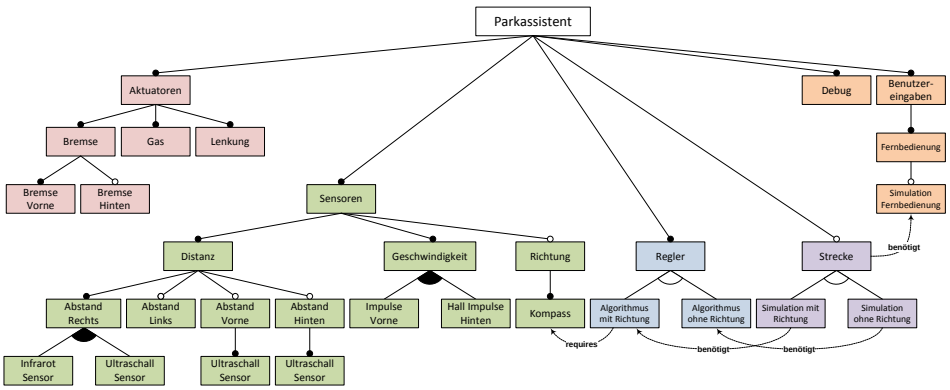

Abbildung 8.2.: Feature-Modell des Einparkassistenten

Der Regler fährt eine vorher fest vorgegebene Trajektorie ab, die das Fahrzeug in die Parklücke lenkt. Ist eine Hardwareabstraktionsschicht (HAS)-Komponente als Kompass vorhanden und in der entsprechenden Anwendung implementiert wird das Fahrzeug in der Parklücke zusätzlich ausgerichtet.

8.2. Feature-Modell des Parkassistenten

Die Variabilität des Parkassistenten wird durch ein Feature-Modell (FM) ausgedrückt. Das FM ist in der Abbildung 8.2 zu sehen. Die Produktlinie des Parkassistenten besteht aus *Aktuatoren*, *Sensoren*, *Regler*, *Strecke* und dem Feature *Benutzereingaben*.

Die Gruppe der Aktuatoren besteht hauptsächlich aus verpflichteten Features (*Gas*, *Lenkung* und *Bremse vorne*). Die Gruppe der Bremse lässt sich durch die *Bremsen vorne* und den optionalen *Bremsen hinten* implementieren. Dazu wird angenommen, dass diese im Fall des Modellautos nicht immer notwendig sind.

Die Featuregruppe der Sensoren drückt eine große Vielfalt an Variabilität aus. Das spiegelt die Tatsache wider, dass die Auswahl von passenden Sensoren Teil des Entwicklungsprozesses ist. Für einen Großteil der Abstandssensoren gilt, dass diese nicht unbedingt notwendig sind und deshalb als *optionale* Features im Modell abgebildet werden. Einzig ein *Abstandssensor Rechts* ist notwendig, um den Parkasssistenten zu implementieren. Zur Realisierung der Abstandsmessung stehen verschiedene Technologien zur Verfügung, die gewählt werden können. Die Geschwindigkeit kann über Implussensoren an den vorderen Rädern oder Hall Sensoren an den hinteren Rädern bestimmt werden. Die beiden Geschwindigkeitssensoren unterscheiden sich in der Auflösung des zurückgelegten Wegs und in der maximal messbaren Geschwindigkeit.

Der Regler stellt zwei Grundimplementierungen zur Verfügung. Diese unterscheiden sich in der Implementierung der Richtungsauswertung. Im ersten Fall werden

die Richtungsinformationen ausgewertet und nach dem Parkvorgang eine Korrektur vorgenommen. Im zweiten Fall erfolgt diese Korrektur nicht. Die Richtungsinformationen werden von der Simulationsumgebung (Strecke) entsprechend generiert oder nicht.

Ein rein nicht-funktionales Feature ist das *Debug*-Feature. Hiermit werden verschiedene Ausgaben und Überprüfungsmechanismen gesteuert. Die sind auf Modellebene durch *Scope*-Blöcke eingebracht. Diese Blöcke können die Daten und Zustände des Modells während der Simulation visualisieren. Während des Testens auf dem Rapid-Control-Prototyping System (RCP-System) erfolgt eine Ausgabe der Daten über eine serielle Schnittstelle des Microcontrollers.

Das Feature *Benutzerschnittstelle* stellt die Funktionalität der Fernbedienung bereit. Dies bedeutet, dass die Signale einer Fernbedienung gelesen, interpretiert und dann durch den Mircocontroller in analoge und binäre Fahrbefehle umgesetzt werden. Die Fahrbefehle während der Simulation werden mittels eines Gamepads am PC erzeugt und während der Ausführung des Modells direkt in die Simulation mit eingebracht.

Kapitel 9.

Evaluation der modellbasierten Transformationen

Die in den vorherigen Kapiteln beschriebenen Implementierungen sollen zeigen, welches Potential der Einsatz des Eclipse Modeling Frameworks zur Unterstützung des modellbasierten Entwicklungsprozesses hat. Dazu wurden wiederkehrende Arbeiten und der Aufwand des Entwicklungsprozesses untersucht und prototypisch Lösungen erarbeitet.

Zur Bewertung des Potentials werden verschiedene Kriterien herangezogen. Das erste Kriterium ist die Umsetzbarkeit der geforderten Analysen. Mittels der Transformationen muss es möglich sein, die geforderten Analysen, Verknüpfungen und Darstellungen zu implementieren.

Die Effizienz des Systems ist ein weiteres Kriterium zur Bewertung des Potentials. Die Effizienz wird auf der einen Seite in Bezug auf den Speicherverbrauch, Rechenzeit und Anforderungen an das System (Betriebssystem und Systemkomponenten) evaluiert. Auf der anderen Seite wird die Effizienz im Bezug auf den Umsetzungsaufwand der Transformationen bewertet. Der Aufwand der Umsetzung wird durch die benötigte Zeit bewertet.

Ein weiteres Kriterium zielt auf die Anwendbarkeit durch den Entwickler des Matlab / Simulinkmodells (SimMo) ab. Die Anwendung der Transformationen soll sich durch möglichst einfache Eingabe auszeichnen und wenig von der bekannten Entwicklungsumgebung unterscheiden. Die Eingaben sollen möglichst direkt aus dem bekannten Modellierungswerkzeug erfolgen. Das Potential der Transformationen wird zusätzlich im Bezug auf den Nutzen der Transformationen bewertet. Der Nutzen einer Transformation misst sich durch die Reduzierung des Aufwands für die Analyse. Zusätzlich muss betrachtet werden, ob sich die Genauigkeit der Analyse verbessert. Diese misst sich darin, alle Aspekte der Prozessartefakte betrachtet zu haben. Ein weiterer Aspekt des Potentials der modellbasierten Analyse ist der Aufwand einer möglichen alternativen Implementierung mit anderen Werkzeugen. Die Matlab / Simulink (MaSi)-Umgebung bietet die Möglichkeit, mit einer eigenen Programmiersprache Modellveränderungen vorzunehmen. Zu diesem Zweck wird eine spezielle Schnittstelle zur Verfügung gestellt, die als Alternative in Betracht

gezogen wird.

Die Implementation der Transformationen erfolgt mit Hilfe der Modelltransformationssprachen ETL und ATL. Die Prozessartefakte werden durch Import aus Textdateien (Simulinkdateien direkt als MDL-Datei; Doors Anforderungen werden durch Export als CSV-Datei) bewerkstelligt. Die Eingaben des Benutzers erfolgen direkt im programmierten Framework in der Eclipse-Umgebung (z. B. Bestimmung des entsprechenden Ports). Die Anzeige der Analyseergebnisse erfolgt direkt in der Eclipse-Umgebung (mittels eines graphischen Editors) oder durch Rücktransformation in ein Matlab / Simulinkmodell (SimMo).

9.1. Potential von graphisch-unterstützen Modellanalysen

Die Implementierung von graphisch-unterstützten Modellanalysen (Sichten) ist anhand von verschiedenen Anwendungsfällen untersucht worden. Die Aktivierungssicht beinhaltet die Filterung von Modulen und eine spezielle Darstellung.

Umsetzung: Die geforderte Darstellung wurde mehrmals angepasst und konnte gut realisiert werden. Um die Umsetzung zu ermöglichen, musste ein Mechanismus zur Identifikation von Modulen eingeführt werden.

Effizienz / Aufwand: Die Darstellung konnte in einer ersten Version innerhalb von 8 Personenstunden erstellt werden. Die Ausführung der Transformationen innerhalb des importierten Eclipse-Simulink-Modells erfolgt in unter 10 Sekunden (Eingabe: Beispielmodell 1,5 Mb).

Anwendbarkeit: Für diese Transformationen werden keine Eingaben benötigt. Ausgenommen sind die notwendigen Anpassungen des Modells zur Identifikation der Module. Die Darstellung erfolgt im Eclipse Editor.

Nutzen: Die Abhängigkeiten nachzuvollziehen, ist für den Modellierer schwierig und fehleranfällig. Aus diesem Grund ist der Nutzen dieser Analyse groß.

Alternativen: Die Signalverfolgung mit m-Dateien in der Simulinkumgebung ist schwierig, da langwierige Implementierungen vorangehen müssen. Eine Implementierung der Analyse in MaSi ist daher aufwändig.

Eine zweite Analyse wurde zur Signalverfolgung implementiert. Dazu wurden sowohl eine Vorwärts- als auch eine Rückwärtsanalyse in verschiedenen Abstraktionsniveaus implementiert:

Umsetzung: Die Implementierung dieser Transformation konnte auf allen Abstraktionsniveaus umgesetzt werden.

Effizienz / Aufwand: Die Analyse benötigt zur Durchführung auf dem Referenzmodell unter 5 Sekunden. Der Aufwand der ersten prototypischen Implementierung ist mit 5 Personenstunden für die Vorwärts- und Rückwärtsanalyse anzugeben.

Anwendbarkeit: Zur Anwendung der Signalverfolgung muss das Modell importiert werden und als Eingabe ein Port im importierten Modell ausgewählt werden.

Nutzen: Die Signalverfolgung ist ein wichtiges Hilfsmittel der Modellanalyse. Es erleichtert das Auffinden von Abhängigkeiten und ist daher eine wichtige Analyse.

Alternativen: Eine alternative Implementierung in MaSi ist einfach möglich, ist aber nur mäßig erweiterbar. Die Anbindung externer Konfigurationen an eine mögliche Darstellung ist schwierig.

Bei dieser Analyse wird eine Konfiguration im Modell visualisiert. Dazu werden Module die in einer Konfiguration enthalten sind, grün dargestellt und Module, die durch Konfiguration entfernt werden, rot.

Umsetzung: Zur Konfigurationsbestimmung wurden Parameter einer m-Datei eingelesen und mit den entsprechenden Parametern im SimMo verknüpft. Die notwendigen Werkzeuge zur Analyse der Parameter und Verknüpfung mit dem Feature-Modell (FM) sind vorhanden.

Effizienz / Aufwand: Zur Darstellung werden zwei Eingabemodelle verarbeitet. Insgesamt benötigt die Transformation mit Schreibvorgängen ca. 20 s auf dem Referenzmodell. Der Implementierungsaufwand beträgt 20 Personenstunden.

Anwendbarkeit: Die Darstellung erfolgt im Simulinkmodell oder in der Eclipse-Umgebung. Weitere Eingaben sind nicht erforderlich.

Nutzen: Der Nutzen dieser Analyse ergibt sich aus der besseren Übersicht für den Entwickler. Ein direkter Nutzen lässt sich daher ohne Entwicklerbefragung schwer ermitteln.

Alternativen: Eine alternative Implementierung in MaSi ist einfach möglich, ist aber nur mäßig erweiterbar. Die Anbindung externer Konfigurationen an eine mögliche Darstellung ist schwierig.

Die einfache Darstellung der Schnittstellen einer Menge von Modulen ist Inhalt der Analyse. Die Analyse stellt alle Verbindungen zwischen einer vorgegebenen Menge von Modulen dar und entfernt die restlichen Verbindungen. Die Bewertung dieser Analyse:

Umsetzung: Die Analyse nutzt die Ergebnisse der Transformation Auflösen von Bussen und bestimmt durch einfaches Filtern die benötigten Modulstrukturen und Eingangs- und Ausgangssignale.

Effizienz / Aufwand: Die Transformation besteht aus zwei Transformationen, welche auf dem Referenzmodell eine Ausführungszeit von ca. 10 s haben. Der Aufwand für die Implementierung beträgt ca. 10 Personenstunden.

Anwendbarkeit: Als Eingabe für diese Analyse müssen die referenzierten Blöcke eingegeben werden. Dies kann zur Zeit erst nach dem zweiten Transformationsschritt geschehen, so dass zwei Durchläufe der Transformation notwendig sind. Dies erschwert die Anwendbarkeit.

Nutzen: Der Nutzen dieser Analyse liegt in der fehlerfreien, vollständigeren und weniger aufwändigen Dokumentation. Aus diesem Grund wird der Nutzen dieser Transformation ebenfalls als hoch eingestuft.

Alternativen: Eine Analyse der Schnittstellen und der zugehörigen Signale ist unter Matlab aus den schon weiter oben genannten Gründen schwierig. Da dies der Hauptteil der Analyse ist, würde eine Implementierung in MaSi-Umgebung einen großen Aufwand bedeuten.

Die Auflösung der Busstruktur ist die letzte Transformation, welche eine neue Sicht auf das Modell erzeugt. Dazu werden alle Bussignale aus dem Modell entfernt und durch atomare Signale ersetzt.

Umsetzung: Für die Umsetzung dieser Analyse müssen alle strukturellen Blöcke gekennzeichnet und in der Transformation behandelt werden. Die entsprechende Erweiterung des Ursprungsmodells um neue Blöcke ist möglich.

Effizienz / Aufwand: Die Transformation benötigt auf dem Referenzmodell eine Rechenzeit von ca. 10 s. Der Aufwand für die Analyse ist mit ca. 30 Personenstunden anzugeben. Die Umsetzung der Blöcke und die rekursiv implementierte Verfolgung der Bussignale machen eine komplizierte Transformation notwendig, die nicht in kleine Einheiten zerlegt werden kann.

Anwendbarkeit: Die Transformation benötigt keine zusätzliche Eingaben, so dass nach dem Import eine einfache Anwendbarkeit gegeben ist.

Nutzen: Das Ergebnis dieser Transformation ist als Eingabe für weitere Transformationen zu sehen. Ein direkter Nutzen für den Modellierer kann in der Analyse der verwendeten Signale liegen, da nur benötigte Signale weitergegeben werden.

Alternativen: Eine alternative Implementierung ist wiederum aufgrund der aufwändigen Signalverfolgung schwierig. Deshalb gibt es in diesem Fall keine sinnvolle Alternative mit werkzeugeigenen Verfahren.

Zusammenfassend sieht man bei allen Transformationen / Analysen, dass die Anwendungsfälle umsetzbar sind und die entsprechenden Sichten erzeugt werden können. Die Analysen können mit einem geringen Aufwand durchgeführt und implementiert werden. Daraus folgt ein großes Potential zur Erstellung weiterer Sichten.

9.2. Verknüpfung von Prozessartefakten

Die Technologie zur Verknüpfung von Prozessartefakten ist im Abschnitt 6.5.1 beschrieben. Die Prozessartefakte Requirements, Feature und Simulinkmodell werden in Beziehung gesetzt. Die Beziehungen werden kontextabhängig über die zentrale Instanz des Features verknüpft.

Die Implementierung hat das Potential, verschiedene Analysen zu implementieren und damit verschiedene Anfragen des Modellierers zu beantworten. Diese können in Richtung von Verknüpfungen aber auch in Richtung von komplexeren Analysen gehen. Die Bewertung des Potentials erfolgt durch folgende Kriterien:

Umsetzung: Für die Implementierung der Prozessbeziehungen werden vorhandene Verknüpfungsmechanismen des Eclipse Modeling Framework (EMF) verwendet. Diese ermöglichen einen transparenten Zugriff auf referenzierte Artefakte.

Effizienz / Aufwand: Der Aufbau eines kompletten Verknüpfungsmodells braucht keine nennenswerte Rechenzeit. Der Aufwand der Entwicklung des Konzepts und der Umsetzung kann mit 25 Personenstunden abgeschätzt werden.

Anwendbarkeit: Die Ergebnisse werden durch einem graphischen Editor angezeigt. Die zur Erstellung notwendigen Verknüpfungen müssen innerhalb der Eclipse-Umgebung erfolgen. Die Unterstützung ist für den Modellierer in einem Werkzeug vorhanden. Die Ergebnisse werden nicht innerhalb der bisher verwendeten Werkzeuge dargestellt. Der neue Ansatz hat eine einheitliche, einfache Visualisierung.

Nutzen: Der Nutzen des Frameworks für den Entwickler ist hoch. Bei der Entwicklung bzw. Weiterentwicklung von Funktionalität ist die Konsistenz der einzelnen Prozessartefakte sehr wichtig. Der Nachweis, dass diese Konsistenz gegeben ist, ist auf manuellen Weg nur sehr schwierig zu erreichen. Aus diesem Grund sind automatische Analysen in Verbindung mit Verknüpfungen wichtig, um den Zustand der Konsistenz nachzuweisen. Neben der Verknüpfung ist auch die zeitliche Entwicklung der Artefakte wichtig. Diese ist noch nicht im

Fokus dieser Implementierung gewesen. Die Ansätze zeigen jedoch, dass die Anwendungsfälle mit diesem Konzept unterstützt werden.

Alternativen: Eine alternative Implementierung ist vorhanden, wird aber aus verschiedenen Gründen als nicht zielführend angesehen und daher nicht verwendet.

Das Konzept zur Verknüpfung von Prozessartefakten hat gezeigt, dass mit EMF-Mechanismen verschiedene Anwendungsfälle umgesetzt werden können.

9.3. Generelles Anaylserahmenwerk

Das Potential zur Entwicklung einer spezialisierten Analyse-Sprache ist durch Implementierung einer domänenspezifischen Sprache für die Verknüpfungskonfiguration untersucht worden. Die entwickelte Sprache ist in der Lage, Transformationen zur Artefaktverknüpfung zu konfigurieren und zu steuern. Ziel der generellen Sprache ist die Erstellung von Transformationen.

Der Aufbau aller Analysen wurde so gestaltet, dass einzelne wiederkehrende Aufgaben getrennt implementiert wurden. Dadurch ist eine Kombination verschiedener Transformationen möglich. Um die endgültigen Analyseergebnisse mittels eines generellen Rahmenwerks zu erreichen, müssen die Transformationen konfigurierbar gestaltet werden.

Ein analoges Vorgehen existiert bereits im Bereich der relationalen Datenbank. Die Daten werden analysiert und durch eine benutzerspezifische Sprache (Structured Query Language - SQL) abgerufen und analysiert. Teile dieser Analysen sind vordefinierte Standardlösungen. Es können aber zusätzlich erweiterte Analysen selbständig implementiert werden. Grundlage der Analyse sind die Eclipse-Artefaktmodelle. Diese müssen die Informationen für die Analysen bereitstellen. Diese Informationen sind zum Teil in den Modellen enthalten. Durch die Anreicherung der Metamodelle sind sie klar und explizit zugreifbar. Um das Potential der generischen Sprache zu bestimmen, müssen auch die Standardtransformationen vorhanden sein. Diese ermöglichen dann durch Kombination eine Analyse. Bei der Implementierung konnten folgende Standardaufgaben identifiziert werden:

- Filtern von Modellelementen
- Verfolgung von Modellstrukturen
- Verflachung von Hierarchien
- Extrahieren, Bestimmen und Verändern von Modellelementen

Mit Hilfe der beschriebenen Transformationen lassen sich die Analysen umsetzen. Eine genaue Bestimmung des Abstraktionsniveaus ist notwendig, um die möglichen Eingaben zur Konfiguration und die Ausgaben der Transformationen zu betrachten.

Abschließend zeigen die Analysen, dass das EMF und die entstandenen Transformationen ein großes Potential bieten, um eine generische Sprache zu entwickeln. Die Konfigurierbarkeit von Transformationen konnte durch gesteuerte Artefaktverknüpfungen nachgewiesen werden. Das Verfahren der Kombination von Transformationen mit begrenzter Funktionalität wurde ebenfalls erfolgreich eingesetzt. Zu untersuchen bleibt, welche Randbedingungen im Bezug auf die Analyse und der Datenübergabe zwischen den einzelnen Transformationen beachtet werden müssen. Die Entwicklung einer Programmier- oder Modelliersprache, welche eine entsprechende Analyse für Modellierer durch Kombination von Transformationen umsetzbar macht, ist durch Werkzeuge des EMF (Textual Modeling Framework (TMF) und Graphical Modeling Framework (GMF)) sichergestellt.

Kapitel 10.

Abschließende Bewertung

Bei der vorherigen Evaluierung wird der im Kapitel 5 vorgestellte Ansatz umgesetzt. Die Umsetzung enthält eine komplette Implementierung, ausgehend von Hardwaretreibern für Busse, bis his zur Anbindung von Sensoren auf Modellebene. Die Informationen auf Modellebene werden durch einen Einparkassistenten genutzt. Anschließend werden die Unterstützungsmethoden auf Modellebene evaluiert. Dabei liegt der Fokus auf schneller und einfacher Umsetzbarkeit, die mit Hilfe eines Rahmenwerks erreicht werden soll.

Das wichtigste Ziel der Arbeit ist, die Wiederverwendbarkeit einzelner Komponenten auf verschiedenen Ebenen zu erreichen. Auf der tiefsten (hardwarenahen) Ebene sind die Treiber hierarchisch aufgebaut. Diese abstrahieren die einzelnen Hardwarefunktionen und fassen diese dann weiter zusammen. Eine Wiederverwendung der Hardwaretreiber ist daher gut möglich. Sie werden auch genutzt, um komplexere Funktionalität (z. B. Busstandards wie I2C) zu implementieren.

Auf der nächst höheren Abstraktionsebene sind die Implementierungen der Schnittstellenkomponenten (Sensoren und Aktuatoren) zu finden. In dieser Studie sind die Implementierungen der Komponenten unabhängig vom eigentlichen Modell erfolgt. Sie enthalten die Funktionalitäten für Umrechnungen und Simulationen. Für die Anbindung an die Simulations- und Modellierungsumgebung Matlab / Simulink sind Anpassungen notwendig, die die Funktionsaufrufe der Simulationsumgebung auf die entsprechenden Funktionen der Komponenten abbilden. Diese Anpassung ist für alle Simulationsumgebungen möglich, die eine zeitliche Simulation aufgrund von Eingangs- und Ausgangssignalen durchführen.

Die Anbindung der Schnittstellenkomponenten im Modell ist nicht wiederverwendbar für verschiedene Entwicklungsumgebungen. Die Funktionalität auf dieser Ebene beschränkt sich auf die Weiterleitung von Informationen. Allerdings ist die Wiederverwendbarkeit der Schnittstellenkomponenten aus Sicht des Modellierers hoch. So ist es möglich, die Schnittstelle zu verwenden, ohne dass eine Festlegung auf die konkrete Schnittstellenrealisierung notwendig ist. Dies erhöht die Wiederverwendbarkeit innerhalb des Modells.

Diese innerhalb des Modells zu steigern, ist auch das Ziel des zweiten vorgestellten Ansatzes. Hier wurden die Analysen und Transformationen so gestaltet, dass die

Ergebnisse einfach zu verwenden sind. Um die Wiederverwendbarkeit der Transformationen zu steigern, wurden zwei Verfahren gewählt. Das erste Verfahren nutzt die vorhandenen Metamodelle, um die Mechanismen auf dieser Ebene abstrakt zu definieren. Ein Beispiel dazu ist die Möglichkeit, den Variantenmechanismus unabhängig vom verwendeten Domänenmodell zu definieren. Das zweite Verfahren nutzt die Möglichkeit, durch konfigurierbare und kombinierbare Transformationen kleinteilige Aufgaben zu implementieren, die wiederverwendet werden können. Die verschiedenen Analysen stützen sich zum größten Teil auf diese Transformationen. Ein Beispiel einer solch zentralen Funktionalität, die oft wiederverwendet wurde, ist das Auflösen der Busstrukturen im Matlab / Simulinkmodell.

Eingeschränkt wird der Ansatz dadurch, dass so genannte „Quick Hacks“ mit dem gezeigten Ansatz nicht möglich sind. Diese widerspricht dem Drang des Ingenieurs, schnell Verfahren und Methoden auszuprobieren. Bei dem hier vorgestellten Ansatz zur Realisierung ist ein initialer Aufwand zu treiben, um Zugriff auf die Sensoren und Aktuatoren zu gewähren. Solange sich die Entwicklung aber auf bekannte Sensoren und Aktuatoren beschränkt, ist die Einbindung ins Modell einfach zu bewerkstelligen.

Zusätzlich profitiert der Ingenieur von der nahtlosen Umsetzung des Modells von der Simulationsumgebung zur Implementierung auf dem Rapid-Control-Prototyping System. Während der Entwicklung sind meist mehrere Schleifen notwendig, um das eingebettete System zu entwickeln. Durch die nahtlose Integration der Schnittstellen müssen keine Änderungen am Modell mehr vorgenommen werden.

Diese Integration der Schnittstellen hat zur Folge, dass der Ressourcenverbrauch der Anwendung höher ist, da zusätzliche Umrechnungen benötigt werden, um auf die eigentlichen Informationen zuzugreifen. Diese kann beispielsweise durch veränderte Totzeiten Einfluss auf die Reglergebnisse haben. Es ist aber möglich, den Einfluss der Umrechnung schon während der Simulation im Modell aufzuzeigen. Zusätzlich muss bei der Betrachtung des größeren Berechnungsaufwands auch berücksichtigt werden, dass ähnliche Umrechnungen sonst meist in der eigentlichen Reglerfunktionalität vorgenommen werden, sodass sich dann der Aufwand nur vom Regler zur Schnittstelle verschoben hat.

Die Schnittstellenarchitektur ermöglicht neben der nahtlosen Integration auch die Verwaltung der Schnittstelle mittels Feature-Modellen. Die gezeigte Anbindung der Schnittstelle und des Modells ermöglichen es, konsistente Modelle abzuleiten. Diese wird in dieser Arbeit am Beispiel des Features Richtungssenor gezeigt. Ist dieser im Versuchsfahrzeug verfügbar, wird durch das Featuremodell ein Algorithmus gewählt, welcher auf die Informationen des Sensors zurückgreift, um den Parkvorgang zu verbessern. Ist der Sensor nicht verfügbar, muss auf einen einfacheren Algorithmus zurückgegriffen werden. Durch die gemeinsame Anbindung der Schnittstelle und des Modells an das Feature-Modell ist es möglich, solche Abhängigkeiten auszudrücken und durch automatische Anpassungen zu berücksichtigen.

Die Möglichkeit der automatischen Anpassung der am Entwicklungsprozess beteiligten Prozessartefakte hat zur Folge, dass 150% Modelle implementiert werden müssen. Der Aufwand diese zu realisieren ist größer als ein einzelnes Modell zu implementieren. Zusätzlich ist zu erwarten, dass ein Modell, das speziell für ein Produkt implementiert wird, effektiver ist als ein Modell, das durch Ableitung die geforderte Funktionalität erhält. Neben der Effektivität ist auch eine höhere Komplexität für die Realisierung zu erwarten. Dem gegenüber steht, dass beim Ableiten mehrerer Produkte der Aufwand für jedes einzelne Produkt mit dem Produktlinienansatz sinkt. Zusätzlich sind die gezeigten Analysen und Sichten in der Lage, einen Teil der Komplexität des Produktlinienansatzes zu reduzieren. Es gilt aber auch, dass die Analysen auf Modelle ohne Produktlinienansatz angewendet werden können, um die Komplexität besser beherrschbar zu machen.

Der in dieser Arbeit verwendete Produktlinienansatz mit einem zentralen Feature-Modell bietet zusätzlich die Möglichkeit, verschiedenste Prozessartefakte konsistent abzuleiten. Zusätzlich ermöglicht die Verknüpfung der Prozessartefakte eine gewisse Zuordnung von Teilen der einzelnen Modelle zueinander. Abschließend kann gesagt werden, dass die Integration von Produktlinienansätzen in die modellbasierte Entwicklung von eingebetteten Systemen erfolgreich ist. Diese Aussage ist in dem Sinne zu verstehen, dass die Modellierung und das Testen auf einem Rapid-Control-Prototyping System eine nahtlose und konfigurierbare Verknüpfung erhalten haben, die kürzere Entwicklungszeiten erwarten lässt. Zusätzlich ist die Modellierung einer Produktlinie in einem 150% Modell für die einzelnen Prozessartefakte eine Möglichkeit, einen konsistenten Entwicklungsstand für abzuleitende Produkte zu erhalten.

Kapitel 11.

Zusammenfassung

In der vorliegenden Arbeit werden zwei Ansätze für eine modellbasierte Produktlinienentwicklung von eingebetteten Systemen vorgestellt. Im Fokus der beiden Ansätze liegen die Anforderungen, die aus den verschiedenen Domänen der Regelungsentwicklung und der Softwareentwicklung für eingebettete Systeme stammen. Aus diesen Anforderungen werden Ansätze für eine variantenreiche, modellbasierte Entwicklung abgeleitet. Diese konzentrieren sich im ersten Ansatz auf die Schnittstellen der Modelle zur realen Welt. Der zweite Ansatz konzentriert sich auf die Verwaltung von Variabilität der Modellebene. In beiden Ansätzen wird die Entwicklungsumgebung Matlab / Simulink verwendet, um Funktionalität zu implementieren.

Durch eine ausführliche Analyse der Anforderungen aus Software- und Regelungstechnik wird in dieser Arbeit eine neue Architektur entwickelt, welche eine Entwicklungsumgebung mit einem Rapid-Control-Prototyping System für die modellbasierte Entwicklung beschreibt.

Der zentrale Teil des neuen Systems ist eine Hardwareabstraktionsschicht (HAS), die alle Sensoren und Aktuatoren als Schnittstellenkomponenten auffasst und in einer abstrahierten Form in die Modellierungsumgebung einbettet. Die Abstraktion der Schnittstellenkomponenten wird durch Ausgabe und Eingabe von physikalischen Größen realisiert. Der Entwickler bekommt durch die Abstraktion nur die zur Berechnung notwendigen Informationen zur Verfügung gestellt. Die Ansteuerung der Aktuatoren erfolgt ebenfalls über physikalische Größen. Durch diese Abstraktion ist es möglich, die Wiederverwendung von Treibern der Schnittstellenkomponenten von der Modellierungsebene zu trennen. Eine so neu eingeführte Konfiguration ermöglicht die Speicherung notwendiger Informationen und die Wiederverwendung von Treibern und notwendiger Software.

Neben der Wiederverwendung berücksichtigt die HAS den Aspekt der Hardwareressourcen während der Entwicklung. Die Implementation der Schnittstellenkomponenten wurde zu diesem Zweck um Simulationskomponenten erweitert. So ist es möglich, die Beeinflussung der Schnittstellenkomponente schon während der Simulation auf Modellebene in den Entwicklungsprozess aufzunehmen. Zusätzlich wird dem Ingenieur während der Entwicklung die Möglichkeit gegeben, verschiedene Schnittstellenkomponenten zu testen und den Einfluss auf die Anwendung abzu-

schätzen. Ohne Änderungen am Implementationsmodell (IMo) ist es möglich, die konkreten Realisierungen der Schnittstelle zu tauschen.

Die Architektur des Rapid-Control-Prototyping Systems (RCP-Systems) ermöglicht die Umsetzung weiterer nicht-funktionaler Anforderungen. Eine wichtige nicht-funktionale Anforderung ist der Schutz des geistigen Eigentums. Die Komponenten können ohne Preisgabe der eigentlichen Implementierung in Form von Binärdateien an Kunden gegeben werden. Es ist zusätzlich möglich, fest vorgegebene Schnittstellenkonfigurationen zur Verfügung zu stellen.

Zur Verwaltung der Hardwareabstraktionsschicht (HAS) wird ein Feature-Modell (FM) verwendet, das die Möglichkeit bietet, am Prozess beteiligte Dokumente und notwendige Verknüpfungen anzulegen. Features im FM repräsentieren hierarchisch sowohl die abstrakten als auch die konkreten Komponenten. Diese Informationen können genutzt werden, um die Komponenten initial aufzubauen und so den Entwickler zu unterstützen.

Das FM bildet die Grundlage des Analyse- und Unterstützungsrahmenwerks. Dieses hat das Ziel, die variantenreiche modellbasierte Entwicklung mit verschiedenen Werkzeugen zu unterstützen. Die Unterstützung findet in drei Bereichen statt. Die Analyse von variantenreichen Prozessdokumenten ist der erste Bereich. Im zweiten Bereich werden Anpassungen von Prozessdokumenten vorgenommen. Der dritte vorgestellte Bereich betrifft die Evolution von Produktlinien.

Die Verwaltung der Schnittstelle über ein FM bietet die Möglichkeit, Abhängigkeiten zwischen Komponenten und weiteren Features auszudrücken und durch Konfigurationswerkzeuge überprüfen zu lassen. Durch dieses Vorgehen kann sichergestellt werden, dass die abgeleiteten Konfigurationen richtig sind und alle Abhängigkeiten beachtet werden.

Das Rahmenwerk ist mit dem Eclipse Modeling Framework (EMF) realisiert und beinhaltet Repräsentationen von allen Prozessdokumenten als Modelle des Frameworks. Die importierten Modelle werden zusätzlich für eine einfache Bearbeitung angepasst. Das EMF stellt Mechanismen zur Verfügung zur schnellen Entwicklung von graphischen Editoren mit Hilfe des Graphical Modeling Frameworks (GMFs), zum Einsatz von Modelltransformationssprachen (Atlas Transformation Language (ATL) [12] und Epsilon Transformation Language (ETL) [25]) und zur konsitenten Modellverwaltung durch Verknüpfungsmechanismen.

Um die Entwicklung von variantenreichen Modellen zu erleichtern, werden verschiedene Analysemethoden vorgestellt. Deren Ergebnisse unterstützen den Modellierer in bestimmten Entwicklungsszenarien. Die Analysen basieren auf Modelltransformationen, welche Entwurfsmuster in den Prozessdokumenten ausnutzen, um spezielle Sichten zu extrahieren. Die vorgestellte Analyse der Aktivierungsmodi filtert zum Einen alle unwichtigen Informationen aus dem Modell und bestimmt zum Anderen für den Modellierer interessante Verknüpfungen und beachtet Varianteninformationen.

Die Anpassung der Prozessdokumente hat zum Ziel, die Konsistenz zwischen den Enitäten der Entwicklung sicherzustellen. Es werden Konfigurationsinformationen verwendet, um die Prozessdokumente anzupassen. Durch die gleichzeitige Anwendung der Konfiguration auf alle Elemente ist sichergestellt, dass alle Dokumente in der gleichen Variante bearbeitet werden. Die dafür notwendige Verknüpfung der Entitäten erfolgt regelbasiert mit dem zentralen Element des FMs. Die Ergebnisse der automatischen Transformationen können im Rahmenwerk graphisch aufbereitet und manuell angepasst werden.

Die Evaluation der vorgestellten Architektur und des Rahmenwerks erfolgt anhand zweier Fallstudien. Innerhalb der ersten Fallstudie wird ein autonomer Einparkassistent entwickelt, dessen Schnittstelle auf der vorgestellten Architektur basiert. In der Studie kann durch den Einsatz der Hardwareabstraktionsschicht die Unabhängigkeit von der konkreten Realisierung der Schnittstelle gezeigt werden. Z. B. können ohne Änderungen am Modell verschiedene Entfernungssensoren mit unterschiedlichen Hardwareschnittstellen und Verhalten im Modell verwendet werden. Die Charakteristik der Schnittstelle wird bei dem Ansatz sowohl in der Simulation als auch während der Quelltextgenerierung berücksichtigt. Die Anpassungen können an einer zentralen Konfiguration vorgenommen werden. In der Fallstudie konnten alle benötigten Sensoren, Aktuatoren und Benutzerschnittstellen konsistent eingebunden werden. So ist es möglich, die Benutzerschnittstelle während der Simulation über ein Gamepad und während des Tests auf dem Rapid-Control-Prototyping System über einer Fernbedienung aus dem Modellbau zu steuern.

In einer zweiten Fallstudie werden Analyse- und Unterstützungswerkzeuge evaluiert. Grundlage der Evaluierung ist ein Implementationsmodell, das alle Eigenschaften eines realen Implementationsmodells aufweist und verschiedene Variantenmechansimen beinhaltet. Das Matlab / Simulinkmodell enthält ca. 1200 Blöcke, implementiert sechs Features und stammt, wie die weiteren Prozessdokumente, aus einem realen Entwicklungsprozess. Die prototypisch implementierten Transformationen benötigten weniger als 30 Sekunden Rechenzeit und konnten mit geringen Entwicklungszeiten vor Ort mit verantwortlichen Modellierern realisiert werden. Schon die prototypischen Umsetzungen zeigen in den drei Bereichen Analyse, Anpassung und Ableitung von Produkten, dass die Ergebnisse den Entwickler substanziell unterstützen und die Komplexität beherrschbar machen.

Insgesamt zeigt die Arbeit verschiedene Ansätze zur Unterstützung der variantenreichen Entwicklung durch einen modellbasierten Ansatz. Diese Unterstützung erstreckt sich auf den hardwarenahen Bereich der Schnittstellenkomponenten und deren Integration unter Berücksichtigung von regelungs- und softwaretechnischen Anforderungen. In einem zweiten Teil werden Ansätze der variantenreichen modellbasierten Entwicklung auf Modellebene durch Analysen und Anpassungen von Prozessdokumenten durch ein Rahmenwerk und darin realisierten Transformationen untersucht.

Kapitel 12.

Ausblick

Die in dieser Arbeit vorgestellten Ansätze zur Unterstützung einer variantenreichen, modellbasierten Entwicklung können in verschiedenen Bereichen ausgebaut und fortgeführt werden. Die weitere Verbesserung der Benutzerfreundlichkeit und Performanz kann durch eine weitgehende Integration in ein Werkzeug erfolgen. So kann eine große Automatisierung erreicht werden. Ziel wäre eine Steigerung der Effizienz. Hierfür ist der Einsatz von Datenbanken denkbar. Im Eclipse Modeling Framework (EMF) existiert das Connected Data Object (CDO) [13] Rahmenwerk, welches eine Verwaltung von Modellen in Datenbanken ermöglicht. Mit diesem Ansatz kann neben einer zentralen Verwaltung auch der Mehrbenutzerbetrieb gesichert werden. Zusätzlich ist eine verbesserte Performanz bei großen Modellen zu erwarten. Erste Tests haben aber gezeigt, dass zur Zeit eine Anbindung an die Transformationssprachen noch nicht nahtlos möglich ist.

In dieser Arbeit wurden Ansätze zum Anreichern von Modellen um zusätzliche Metainformationen für Analysen und Unterstützung des Entwicklers vorgestellt. Neben diesen Informationen werden in den realen Prozessen zahlreiche Mechansimen verwendet, um weitere Metainformationen in Prozessdokumenten zu speichern. Mit Hilfe der Datenbanken ist ein neues Konzept vorstellbar, das einen einheitlichen Zugriff auf alle Arten von Annotationen ermöglicht und von proprietären Werkzeugen unabhängig ist.

Eine zentrale Datenbank kann zusätzlich die Entwicklung und Verwaltung der Hardwareabstraktionsschicht (HAS) vereinfachen. Über Datenbankanfragen, die über das Internet mit den Datenbanken verbunden werden, könnte es möglich sein, die neusten Versionen der benötigten Treiber oder neue Schnittstellenkomponenten auf Modellebene zur Verfügung zu stellen. Neue Komponenten könnten auch nur zur Simulation zur Verfügung gestellt werden, um eine Bewertung neuer Komponenten vornehmen zu können. Dadurch ist es möglich, dass kleine Unternehmen ihr Wissen zur Verfügung stellen, ohne die Kontrolle über den Einsatz ihres Wissens zu verlieren.

Auf Ebene des Produktlinienansatzes ist es notwendig, die Integration mit dem AUTOSAR-Ansatz zu untersuchen [24]. Die AUTOSAR-konformen Modelle ermöglichen eine automatische Ableitung der Integrationsumgebungen. Die benötigten Informationen werden in Datenbanken vorgehalten. Integrationen der beiden Ansätze

ermöglichen es, alle Informationen zentral zu verwalten und durch Konfigurationen der Produktlinien anzupassen.

Zusätzlich sollte der Modellierer neben der semantischen auch auf der syntaktischen Ebene mit Analysen unterstützt werden. Diese syntaktischen Analysen der Modelle erfordern ein Verständnis der Funktionalität und können im Zuge von regelmäßigen Prüfungen auf mögliche Fehler hinweisen. Denkbar ist es dabei, Prüfungen wie in modernen Entwicklungsumgebungen hierarchisch zu gliedern. In kurzen Abständen könnten dann einfache Hinweise und Warnungen im Modell angezeigt werden. Aufwändigere Analysen könnten dann in größeren zeitlichen Abständen durchgeführt und deren Ergebnisse dem Modellierer in Form einer Sicht auf das Modell präsentiert werden.

Literaturverzeichnis

[1] Xml metadata interchange specification.

[2] Aditya Agrawal, Gyula Simon, and Gabor Karsai. Semantic translation of simulink/stateflow models to hybrid automata using graph transformations. *Electron. Notes Theor. Comput. Sci.*, 109:43–56, 2004. ISSN 1571-0661.

[3] J. Banks. *Handbook of simulation: principles, methodology, advances, applications, and practice*. Wiley-Interscience, 1998. ISBN 0471134031.

[4] Matthias Biehl, Chen DeJiu, and Martin Törngren. Integrating safety analysis into the model-based development toolchain of automotive embedded systems. In *LCTES '10*, pages 125–132, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-953-4.

[5] B.W. Boehm, J.R. Brown, H. Kaspar, M. Lipow, G.J. MacLeod, and M.J. Merrit. Characteristics of software quality. 1978.

[6] Goetz Botterweck, Mikolas Janota, and Denny Schneeweiss. A design of a configurable feature model configurator. In *Proceedings of the 3rd International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS 09)*, pages 165–168, January 2009. URL `http://www.vamos-workshop.net/proceedings/VaMoS_2009_Proceedings.pdf`.

[7] L. Chung and J. do Prado Leite. On non-functional requirements in software engineering. *Conceptual Modeling: Foundations and Applications*, pages 363–379, 2009.

[8] P. Clements, R. Kazman, and M. Klein. *Evaluating software architectures: methods and case studies*. Addison-Wesley Reading, MA, 2002. ISBN 020170482X.

[9] Paul Clements and Linda M. Northrop. *Software Product Lines: Practices and Patterns*. The SEI series in software engineering. Addison-Wesley, Boston, MA, USA, 2002. ISBN 978-0201703320.

[10] Krzysztof Czarnecki and Michal Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *GPCE'05*, 2005.

[11] Krzysztof Czarnecki and Chang Hwan Peter Kim. Cardinality-based feature modeling and constraints: A progress report. http://scholar.google.com/url?sa=U&q=http://softwarefactories.com/workshops/OOPSLA-2005/Papers/Czarnecki.pdf, 2005.

[12] Eclipse-Foundation. Atl (ATLAS Transformation Language), . URL `http://www.eclipse.org/m2m/atl/`. `http://www.eclipse.org/m2m/atl/`.

[13] Eclipse-Foundation. Cdo (Connected Data Objects), . `http://www.eclipse.org/cdo/`.

[14] Eclipse-Foundation. EMF - Eclipse Modelling Framework, . URL `http://www.eclipse.org/modeling/emf/`. `http://www.eclipse.org/modeling/emf/`.

[15] Eclipse-Foundation. GMF - Graphical Modelling Framework, . `http://www.eclipse.org/modeling/gmf/`.

[16] Eclipse-Foundation. TMF - Textual Modeling Framework, . `http://www.eclipse.org/modeling/tmf/`.

[17] M. Griss, J. Favaro, and M. d'Alessandro. Integrating feature modeling with the RSEB. In *Proceedings of theFifthInternational Conference on Software Reuse*, pages 76–85, Vancouver, BC, Canada, 1998.

[18] Object Management Group. Omg unifired modeling language specification. online, May 2010. Version 2.3.

[19] Florian Heidenreich, Jan Kopcsek, and Christian Wende. Featuremapper: mapping features to models. In *ICSE Companion '08*, pages 943–944, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1.

[20] Veit Hoffmann and Horst Lichter. A model based narrative use case simulation environment. In J. Cordeiro, M. Virvou, and B. Shishkov, editors, *Conference on Software and Data Technologies (ICSOFT 2010)*, volume 2, pages 63–72, Athens, Greece, July 2010.

[21] IBM-Corporation. IBM Rational DOORS. `http://www-01.ibm.com/software/awdtools/doors/`.

[22] IRISA-Campus de Beaulieu. OpenEmbeDD. `http://openembedd.org/`.

[23] R. Kazman, G. Abowd, L. Bass, and P. Clements. Scenario-based analysis of software architecture. *Software, IEEE*, 13(6):47–55, 2002. ISSN 0740-7459.

[24] Olaf Kindel and Mario Friedrich. *Softwareentwicklung mit AUTOSAR. Grundlagen, Engineering, Management für die Praxis.* dpunkt-Verlag, 2009.

[25] Dimitrios Kolovos, Richard Paige, and Fiona Polack. The epsilon transformation language (etl). In A. Vallecillo, J. Gray, and A. Pierantonio, editors, *ICMT 2008*. Springer-Verlag Berlin Heidelberg, 2008.

[26] J. Kramer. Is abstraction the key to computing? *Communications of the ACM*, 50(4):36–42, 2007. ISSN 0001-0782.

[27] J. Ludewig and H. Lichter. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. dpunkt-Verl., 2007. ISBN 3898642682.

[28] J. Lunze. *Automatisierungstechnik*. Oldenbourg Wissenschaftsverlag, 2003. ISBN 3486274309.

[29] P. Marwedel. *Embedded system design*. Kluwer Academic Pub, 2003.

[30] Microsoft. Executable-file header format. online, August 2004. Revision: 3.1.

[31] M. Nagl. *Softwaretechnik: Methodisches Programmieren im Großen*. Springer, 1990. ISBN 0387527052.

[32] Object Management Group. MOF Version 2.0: MOF Core Specification. online, Jannuar 2006. URL `http://www.omg.org/spec/MOF/2.0/PDF/`.

[33] Object Management Group. UML Version 2.3: Infrastructure Specification. online, November 2010. URL `http://www.omg.org/spec/UML/2.3/Infrastructure/PDF/`.

[34] *Object Constraint Language, Version 2.2*. Object Management Group (OMG), 2010. `http://www.omg.org/spec/OCL/2.2`.

[35] openarchitectureware.org. openarchitectureware.org - official open architecture ware homepage. `http://www.openarchitectureware.org/`.

[36] David Lorge Parnas. On the design and development of program families. *IEEE Trans. Software Eng.*, 2(1):1–9, 1976.

[37] Klaus Pohl, Guenter Boeckle, and Frank van der Linden. *Software Product Line Engineering : Foundations, Principles, and Techniques*. Springer, New York, NY, 2005. ISBN 978-3540243724.

[38] Pure::systems. pure::variants Connector for Simulink. `http://www.mathworks.com/products/connections/product_main.html?prod_id=732`.

[39] K. Reif. *Automobilelektronik: Eine Einführung für Ingenieure*. Vieweg+ Teubner Verlag, 2008. ISBN 3834804460.

[40] J. Schäuffele and T. Zurawka. *Automotive Software Engineering*. Vieweg, 2003. ISBN 3528010401.

[41] Y. Shafranovich. Common Format and MIME Type for Comma-Separated Values (CSV) File. Technical report, RFC 4180, October 2005.

[42] H. Stachowiak. *Allgemeine Modelltheorie*. Springer-Verlag, 1973. ISBN 3211811060.

[43] Jacques Thomas, Christian Dziobek, and Bernd Hedenetz. Variability management in the autosar-based development of applications for in vehicle sytems. In *Procccdings of Fifth International Workshop on Variability Modelling of Software-intensive Systems (Vamos)*, page to appear, 2011.

[44] Markus Voelter and Iris Groher. Product line implementation using aspect-oriented and model-driven software development. In *SPLC 2007*, Kyoto, Japan, September 2007.

[45] Thomas von der Maßen, Klaus Müller, John MacGregor, Eva Geisberger, Jörg Dörr, Frank Houdek, Harbhajan Singh, Holger Wußmann, Hans-Veit Bacher, and Barbara Paech. Einsatz von Features im Software-Entwicklungsprozess. Aachener Informatik Berichte AIB-2005-18, 2005.

[46] Jens Weiland. *Variantenkonfiguration eingebetteter automotiver Software mit Simulink*. PhD thesis, Universität Leipzig, 2008.

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

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
2013-14	Jörg Brauer: Automatic Abstraction for Bit-Vectors using Decision Procedures
2013-16	Carsten Otto: Java Program Analysis by Symbolic Execution
2013-19	Florian Schmidt, David Orlea, and Klaus Wehrle: Support for error tolerance in the Real-Time Transport Protocol
2013-20	Jacob Palczynski: Time-Continuous Behaviour Comparison Based on Abstract Models
2014-01 *	Fachgruppe Informatik: Annual Report 2014
2014-02	Daniel Merschen: Integration und Analyse von Artefakten in der modellbasierten Entwicklung eingebetteter Software

2014-03 Uwe Naumann, Klaus Leppkes, and Johannes Lotz: dco/c++ User Guide

2014-04 Namit Chaturvedi: Languages of Infinite Traces and Deterministic Asynchronous Automata

2014-05 Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, and Peter Schneider-Kamp: Automated Termination Analysis for Programs with Pointer Arithmetic

2014-06 Esther Horbert, Germán Martín García, Simone Frintrop, and Bastian Leibe: Sequence Level Salient Object Proposals for Generic Object Detection in Video

2014-07 Niloofar Safiran, Johannes Lotz, and Uwe Naumann: Algorithmic Differentiation of Numerical Methods: Second-Order Tangent and Adjoint Solvers for Systems of Parametrized Nonlinear Equations

2014-08 Christina Jansen, Florian Göbe, and Thomas Noll: Generating Inductive Predicates for Symbolic Execution of Pointer-Manipulating Programs

2014-09 Thomas Ströder and Terrance Swift (Editors): Proceedings of the International Joint Workshop on Implementation of Constraint and Logic Programming Systems and Logic-based Methods in Programming Environments 2014

2014-14 Florian Schmidt, Matteo Ceriotti, Niklas Hauser, and Klaus Wehrle: HotBox: Testing Temperature Effects in Sensor Networks

2014-15 Dominique Gückel: Synthesis of State Space Generators for Model Checking Microcontroller Code

2014-16 Hongfei Fu: Verifying Probabilistic Systems: New Algorithms and Complexity Results

2015-01 * Fachgruppe Informatik: Annual Report 2015

2015-02 Dominik Franke: Testing Life Cycle-related Properties of Mobile Applications

2015-05 Florian Frohn, Jürgen Giesl, Jera Hensel, Cornelius Aschermann, and Thomas Ströder: Inferring Lower Bounds for Runtime Complexity

2015-06 Thomas Ströder and Wolfgang Thomas (Editors): Proceedings of the Young Researchers' Conference "Frontiers of Formal Methods"

2015-07 Hilal Diab: Experimental Validation and Mathematical Analysis of Cooperative Vehicles in a Platoon

2015-08 Mathias Pelka, Jó Agila Bitsch, Horst Hellbrück, and Klaus Wehrle (Editors): Proceedings of the 1st KuVS Expert Talk on Localization

2015-09 Xin Chen: Reachability Analysis of Non-Linear Hybrid Systems Using Taylor Models

2015-11 Stefan Wüller, Marián Kühnel, and Ulrike Meyer: Information Hiding in the Public RSA Modulus

2015-12 Christoph Matheja, Christina Jansen, and Thomas Noll: Tree-like Grammars and Separation Logic

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