Institut für Architektur von Anwendungssystemen
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3130

Variabilitätsmanagement in der Fahrzeugsimulation

Michael Müller

Studiengang:	Informatik
Prüfer:	Jun.-Prof. Dr.-Ing. Dimka Karastoyanova
Betreuer:	Dipl.-Inf. Katharina Görlach Dr. Uwe Scholz
begonnen am:	19. November 2010
beendet am:	26. Mai 2011
CR-Klassifikation:	D.2.11, D.2.13, D.2.2, I.6.7

Zusammenfassung

Variabilität in Software ermöglicht die Anpassung ausgewählter Bestandteile einer Software an die jeweiligen Bedürfnisse eines Nutzers. Es stützt sich auf eine gemeinsame, identische Architektur und zusätzlich je nach Ausprägung auf spezifische Elemente. Durch die Nutzung von gemeinsamen Einheiten in entwickelten und geplanten Produkten mit gemeinsamer, variabler Architektur und dessen gezielte Wiederverwendung, kann die Entwicklungszeit beschleunigt und die Kosten gesenkt werden.

In vorliegender Arbeit wird ein Konzept zur Entwicklung eines Variabilitätsmanagements vorgestellt, das den Nutzer bei der Erstellung einer Fahrzeugsimulation unterstützen soll. Die Auflösung der Variabilitäten erfolgt dabei durch eine automatische Entscheidungsfindung anhand aussagenlogischer Regeln.
Darüber hinaus wurde ein prototypisches Variabiltitätsmanagement für die Firma Bosch GmbH entwickelt und in dieser Arbeit beschrieben.

Inhaltsverzeichnis

1 Einführung **5**
- 1.1 Motivation . . . 6
- 1.2 Einordnung . . . 8
- 1.3 Ziele der Arbeit . . . 10

2 Grundlagen **13**
- 2.1 Software Produktlinien . . . 14
  - 2.1.1 Domänenanalyse . . . 17
  - 2.1.2 Feature Oriented Domain Analysis . . . 19
- 2.2 Variabilität in Software . . . 24
  - 2.2.1 Variabilitätspunkte und Varianten . . . 25
  - 2.2.2 Variabilitätsmodellierung . . . 27
  - 2.2.3 Typen von Variabilität . . . 31
  - 2.2.4 Variabilität in verschiedenen Architekturebenen . . . 33
  - 2.2.5 Bindungszeitpunkte . . . 36

3 Konzeption eines Variabilitätsmanagements in der Fahrzeugsimulation **41**
- 3.1 Kontextanalyse und Domänenmodellierung nach FODA . . . 42
- 3.2 Abstraktionsschicht durch Features und Variabilität . . . 45
- 3.3 Herausforderungen . . . 48

4 Prototypische Implementierung eines regelbasierten Variabilitätsmanagements **51**
- 4.1 Aktivität der EVA Fahrzeugsimulation ohne Variabilitätsmanagement . . . 51
- 4.2 Ablauf und Integration des Variabilitätsmanagements in EVA . . . 54
- 4.3 Architektur und Realisierung des Variabilitätsmanagements . . . 60
- 4.4 Bedingungsregeln . . . 63
  - 4.4.1 Syntax und Semantik . . . 63
  - 4.4.2 Realisierung der Auswertung . . . 71

5 Zusammenfassung und Ausblick **75**

Literaturverzeichnis **79**

Abbildungsverzeichnis

1.1 Standard- und Individualsoftware vs. Variabilitätsmanagement 7
1.2 Produktlinienausschnitt des Polo Trendline von Volkswagen 8
1.3 Wiederverwendungspotential von Software-Einheiten 10

2.1 Referenzprozess für Software-Produktlinienentwicklung 15
2.2 Domänenanalyse im Kontextdiagramm 18
2.3 Hauptaktivitäten in FODA 20
2.4 Strukturdiagramm eines Bordcomputers 21
2.5 Kontextdiagramm eines Bordcomputers 22
2.6 Beziehungen eines Feature Modells 23
2.7 Feature-Modell des Bordcomputers 24
2.8 Metamodell für Variabilitäten als ER-Modell 26
2.9 Modellierung von Variabilitäten in UML Use-Case Diagrammen 29
2.10 Modellierung von Variabilitäten in UML-Aktivitätsdiagrammen 30
2.11 Modellierung von Variabilitäten in UML Diagrammen 30
2.12 Schichtendiagramm - Fachliche und technische Variabilität 31
2.13 Drei-Schichten-Architekturmuster 34
2.14 Variabilität in der Drei-Schichten-Architektur 35
2.15 Das Wasserfallmodell 37

3.1 Strukturdiagramm von EVA 43
3.2 Kontextdiagramm von EVA 44
3.3 Feature-Modell - Motorkonfiguration 45
3.4 Abstraktion durch Features und Variabilität - Einbindung der Regeln 46
3.5 Steigende Komplexität bei hoher Anzahl variabler Elemente 50

4.1 UML2 Aktivitätsdiagramm der EVA Simulation 52
4.2 Parameterabfrage in EVA 54
4.3 UML2 Aktivitätsdiagramm der EVA Simulation 56
4.4 Parameterabfrage in EVA mit Variabilitätsmanagement 58
4.5 Architektur des Variabilitätsmanagement 61
4.6 Klassen- und Methodenaufruf der Regeldatei von EVA 67
4.7 Bedingung einer Entscheidungsregel, dargestellt als XML Baum 69
4.8 Bedingung II, dargestellt als XML Baum 72
4.9 Beispielhafte Auswertung einer Bedingung 73

Tabellenverzeichnis

2.1 Variabilitätsabhängigkeiten 27

4.1 Verfügbare XML Elemente 70
4.2 Implementierte erweiterbare Klassen 71
4.3 Verfügbare erweiterbare Operatoren 71

Verzeichnis der Listings

4.1 if-then-else Verzeweigung 59
4.2 Mögliches „if-then-else“ Schema für Elektro- Hybrid oder Benzinfahrzeuge . . 59
4.3 Regeldatei - Beispiel 1 64
4.4 XML Schema für Regeldatei - Beispiel 1 64
4.5 Regeldatei - Beispiel 2 68

1 Einführung

Wiederverwendung ist ein fundamentales Konzept der Informatik im Umgang mit Software. Bereits durch die Auslagerung des prozeduralen Programmcodes in Funktionen oder in eine objektorientierte Datenstruktur entstanden wiederverwendbare Programmfragmente. Durch das Konzept der Wiederverwendung können identische Funktionalitäten ohne Codewiederholung abgebildet werden. Dabei ist die Wiederverwendung die Grundlage um Variabilität und Modularisierung von Software zu erreichen. Diverse Ansätze von aktuellen Entwicklungen wie das Cloud Computing, Software as a Service etc. arbeiten bereits mit Variabilität und Anpassbarkeit für den Nutzer. Variabilität in Software bedeutet, dass ausgewählte Module, Klassen, Funktionen oder andere Elemente einer Software durch Selektion an die Bedürfnisse des Nutzers angepasst werden können.

Nicht nur in der Informatik ist das Konzept der Wiederverwendung die Basis für Variabilität. Auch die Automobilindustrie erkannte, dass sich der Markt besser nutzen lässt, wenn es möglich ist, einen Fahrzeugtyp für jedermann anzubieten. Für den Kunden ergibt sich damit die Möglichkeit, ein Auto nach seinen Wünschen und entsprechend seinem Budget auszuwählen.

Mit dem T-Model verwirklichte Henry Ford seinen Traum, ein zuverlässiges und effizientes Automobil zu einem vernünftigen Preis zu bauen. Durch den Bau und Verkauf von mehr als 15 Millionen Exemplaren wurde eine neue Ära im Personenverkehr eingeläutet. Bereits das T-Model gab es mit ein paar Wahlmöglichkeiten - als zweisitziges Cabriolet, als Coupé, Limousine oder als Kleinlastwagen [1].

In dieser Arbeit wird die Thematik der Variabilität in Software untersucht mit dem Ziel, diese auf Automobile - speziell Fahrzeugsimulationssysteme - und deren wiederverwendbaren Technologien zu übertragen. Dabei werden die Ansätze der Software Produktlinien, das Domain- und Application Engineering sowie diverse Konzepte der Variabilisierung von Software in dieser Arbeit berücksichtigt.

[1] Spiegel 2008, J. Pander, Und Lizzy knattert immer noch

1.1 Motivation

Variabilität in Software ermöglicht die Anpassung ausgewählter Bestandteile einer Software an die jeweiligen Bedürfnisse eines Nutzers. Dies impliziert eine leichtere und flexiblere Individualisierung, womit die gesamten Entwicklungskosten der Software gegenüber einer angepassten Standardlösung verringert werden sollen.

Variabilität in Software stützt sich auf eine gemeinsame, identische Architektur und zusätzlich je nach Ausprägung auf spezifische Elemente. Durch die Nutzung von gemeinsamen Einheiten in entwickelten und geplanten Produkten mit gemeinsamer, variabler Architektur und dessen gezielte Wiederverwendung kann die Entwicklungszeit beschleunigt und die Kosten gesenkt werden. Abbildung 1.1 zeigt die Einordnung eines Variabilitätsmanagements in Bezug zu Standard- und Individualsoftware. Die Initialkosten eines Variabilitätsmanagements sind gegenüber einer Standardlösung aufgrund der zu berücksichtigenden Variabilität in der Architektur höher. Jede abgeleitete `Variante` eines Variabilitätsmanagements erhöht die Gesamtkosten jedoch nur in geringem Maße. Im Gegensatz dazu sind `Modifikationen` an einer Standardsoftware deutlich aufwändiger, da diese in der Architektur nicht vorgesehen sind. Die Steigung der Geraden, welche das Variabilitätsmanagement repräsentiert, ist daher größer als die Steigung der Standardsoftware. Bei niedriger Anzahl an Modifikationen sind die gesamten Entwicklungskosten der individualisierten Standardlösung geringer. Bei wachsender Anzahl der Varianten sowie einem höherem `Grad an Individualisierung` lassen sich die `gesamten Entwicklungskosten` trotz größerem initialen Aufwand durch ein Variabilitätsmanagement senken. Oberhalb des `Break-Even-Points` (Schnittpunkt der Geraden) verringern sich die durchschnittlichen Entwicklungskosten des Variabilitätsmanagements mit jeder weiteren Variante.

Ein möglicher Anwendungsfall für Variabilität in Software ist die Fahrzeugsimulation. Die Fahrzeugsimulation versucht Erkenntnisse über das reelle Fahrzeug zu gewinnen. Sie macht sich zur Aufgabe bestimmte Funktionen oder Verhaltensweisen des zu simulierenden Systems abzubilden [Har05]. Dabei werden in der Fahrzeugsimulation der Gesamtenergieverbrauch, Ströme, Beschleunigungen, auftretende Kräfte sowie andere physikalische Größen betrachtet. Oftmals ist auch der Vergleich zwischen unterschiedlichen Konfigurationen oder die Auswirkungen von Neuentwicklungen auf das Gesamtmodell von Interesse.
Um Vergleichsmöglichkeiten zwischen unterschiedlichen Automobilkonfigurationen zu erhalten, ist die Wiederverwendbarkeit sowie die flexible Zusammenstellung und Parameterisierung der Simulationsmodule notwendig. Dabei entsprechen die Simulationsmodule den wiederverwendbaren, zu simulierenden Teilmodellen des Gesamtmodells. Eine denkbare Simulation wäre beispielsweise das Ermitteln der Höchstgeschwindigkeit eines Automobils mit einem Verbrennungsmotor oder einem Elektromotor im Vergleich.

Produktlinien der Automobilindustrie verdeutlichen wie hoch die Variabilität und der Wiederverwendungsgrad innerhalb einer Produktlinie des Automobils und damit auch die eines Fahrzeugsimulationssystems sein muss. Abbildung 1.2 zeigt einen Ausschnitt einer Produktlinie des Polo Trendline von Volkswagen. Der Ausschnitt der möglichen

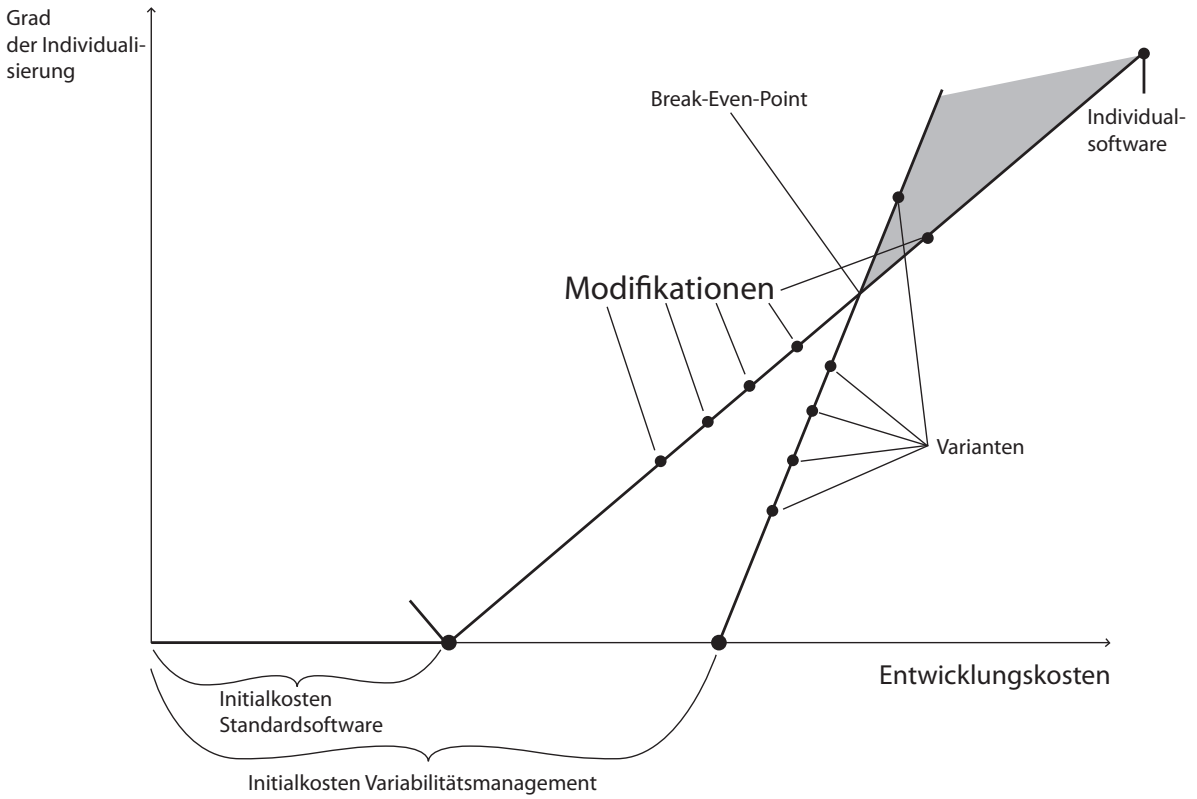

Abbildung 1.1: Die Abbildung zeigt die Einordnung der Entwicklungskosten eines Variabilitätsmanagements gegenüber einer Standardsoftware, welche durch einzelne Modifikationen individualisiert wird. Die Initialkosten eines Variabilitätsmanagements sind gegenüber den Initialkosten einer Standardlösung höher. Beim wachsendem Grad an Individualisierung sind die gesamten Entwicklungskosten eines Variabilitätsmanagements gegenüber einer Standardlösung günstiger. Die Rentabilität eines Variabilitätsmanagements tritt oberhalb des Schnittpunkts der beiden Geraden, auch als Break-Even-Point bezeichnet, ein.

Konfigurationen wurde aus der Internetseite von Volkswagen[2] entnommen. Dabei lässt sich leicht erkennen, dass sich aus dieser Produktlinie viele kombinatorische Möglichkeiten für Produkte ergeben. Welche Elemente optional, alternativ oder verpflichtend sind, wurden der Einfachheit halber an dieser Stelle außen vor gelassen.

Produktlinien von Automobilen sind die Vorreiter bezüglich den Ansprüchen an Softwaresysteme hinsichtlich großer Flexibilität zu vergleichsweise geringen Kosten [BKPS04]. Auf Basis einer gemeinsamen Architektur enthält eine Produktlinie mehrere Ausprägungen

[2] www.volkswagen.de

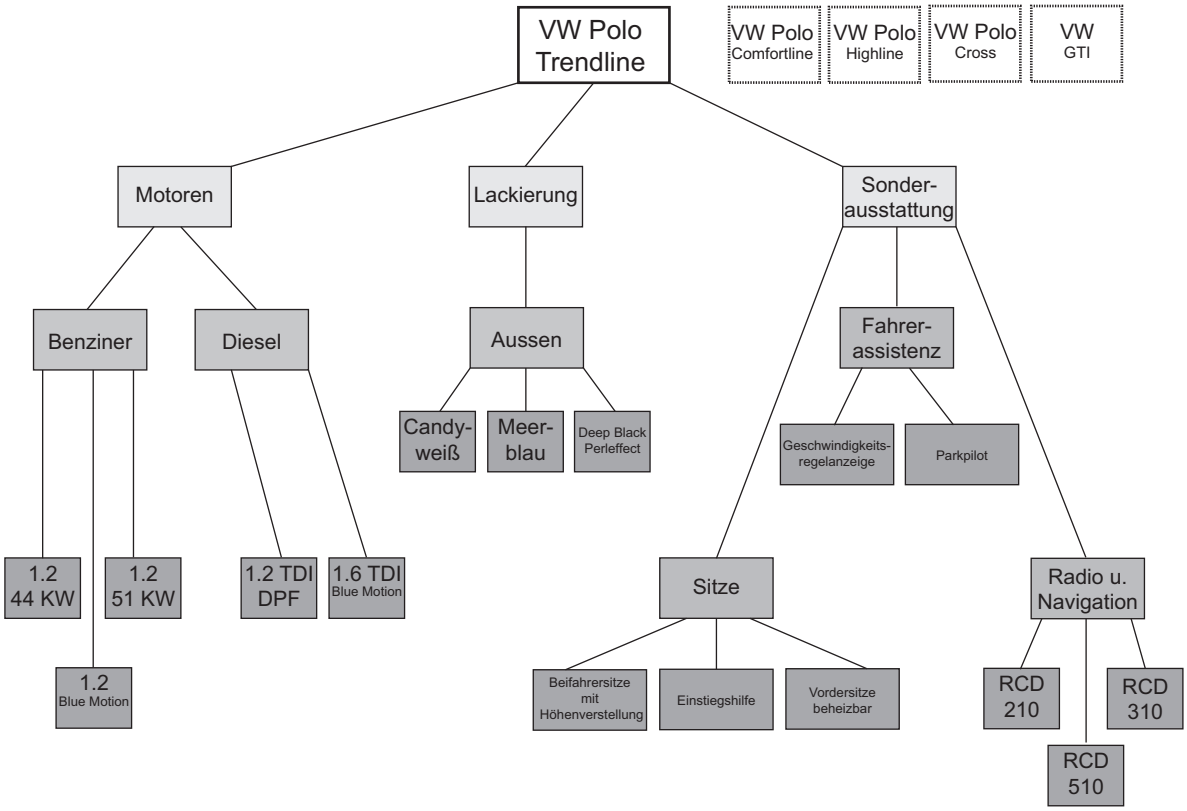

Abbildung 1.2: Der Produktlinienausschnitt zeigt mögliche Konfigurationen eines Polo Trendline. Die Produktlinie wurde aus der Internetseite von Volkswagen entnommen.

eines Produktes. Das Produkt selbst ist eine individuelle Konfiguration im Rahmen der Produktlinie. Die Produktlinie wird daher durch ihre Variabilität und deren wiederverwendbaren Einheiten bestimmt. An einem bestimmten Punkt, dem sogenannten Variabilitätspunkt innerhalb der Produktlinie, wird erstmal die Entscheidung offen gelassen, welches Teilprodukt gebunden werden soll. Um ein konkretes Produkt aus der Produktlinie abzuleiten, werden alle Variabilitätspunkte anhand der jeweiligen Entscheidungsregeln aufgelöst und dann gebunden.

1.2 Einordnung

Diese Arbeit ist in das Umfeld der Wiederverwendung von Softwaremodulen und Variabilitätsmanagementsystemen einzuordnen. Die Wiederverwendung ist die Grundlage um Variabilität in Software zu erzeugen.

Die IEEE-Norm 610.12 (1990) definiert Wiederwendbarkeit wie folgt:

> *„reusability - The degree to which a software module or other work product can be used in more than one computer program or software system."*

Die Wiederverwendung eines Softwaremoduls zeichnet sich dadurch aus, dass es mehrfach und in unterschiedlichem Kontext einsetzbar ist. Zusätzlich sollte es möglichst wenig Schnittstellen nach außen besitzen, um die Integration zu erleichtern und einen spezifischen, allgemeinen Aufgabenbereich abdecken. Die Module sollten also möglichst anwendungsunabhängig und technikneutral sein.

Abbildung 1.3 zeigt das Potential zur Wiederverwendung von Software-Einheiten. Das Wiederverwendungspotential ist abhängig davon, in welchem Umfang die Software-Einheiten technik- und anwendungsspezifisch entwickelt wurden. Rechts oben sind die technik- und anwendungsneutralen Software-Einheiten einzuordnen, sie sind `vielseitig wiederverwendbar`. Wenn eine Software-Einheit technikneutral und anwendungsspezifisch ist, kann sie innerhalb der Anwendung ggf. mehrfach eingesetzt werden, ist jedoch `an die Anwendung selbst gebunden`. Dies ist in der Abbildung links oben dargestellt. Rechts unten werden die technikspezifischen und anwendungsneutralen Software-Einheiten dargestellt, diese sind `an die Technologie gebunden`. Eine `anwendungs- und technikspezifische` Software-Einheit ist `kaum wiederverwendbar` und wird links unten eingeordnet.

Oftmals ist die Anwendungsunabhängigkeit der Software-Einheiten nicht ohne weiteres machbar. Bereits beim Zerlegen des Gesamtsystems in einzelne Elemente sind Abhängigkeiten zu berücksichtigen. Diese Abhängigkeiten zwischen den Elementen müssen im Kontext aufgelöst werden. Dabei können einfache oder auch mehrfach transitive Abhängigkeiten zwischen den Elementen vorkommen. Ein Variabilitätsmanagement hat die Aufgabe, die Auflösung dieser Abhängigkeiten zu unterstützen. Dabei greift es auf Entscheidungsregeln zurück, welche die Abhängigkeiten der wiederverwendbaren (variablen) Elemente darstellen. Es lassen sich komplexe und mehrfache Abhängigkeiten mit Hilfe aussagenlogischer Formeln definieren.

Wiederverwendung von bisherigen Entwicklungen ist nicht nur in der Informatik ein relevantes Thema um Entwicklungskosten zu senken. Sondern auch in Produktlinien von Automobilen existieren diese Variabilitäten und komplexen Abhängigkeiten zwischen den variablen Elementen. Der Nutzen einer Fahrzeugsimulation ist die Simulation auf Basis unterschiedlicher Konfigurationen von Automobilen durchzuführen. Damit ist die Repräsentation einer Teilmenge der Produktlinien und damit auch dessen Variabilität in der Simulation notwendig.

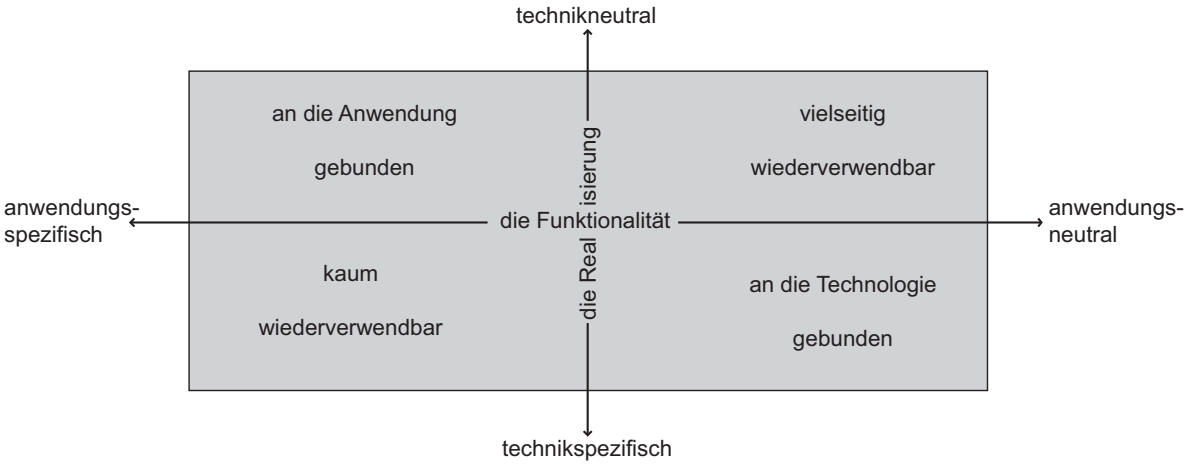

Abbildung 1.3: Die Abbildung beschreibt das Wiederverwendungspotential von entwickelten Software-Einheiten. Um eine vielseitige Wiederverwendbarkeit zu erreichen, muss eine Software-Einheit weitgehend technik- und anwendungsneutral sein. In der Abbildung ist dies rechts oben dargestellt. Hingegen ist eine Einheit, welche anwendungs- und technikspezifisch ist, kaum wiederverwendbar, dargestellt links unten. Links oben in der Abbildung werden Software-Einheiten eingeordnet, welche anwendungsspezifisch und technikneutral sind. Diese sind an die Anwendung gebunden, jedoch innerhalb der Anwendung mehrfach einsetzbar. Software-Einheiten welche anwendungsneutral, jedoch technikspezifisch sind, sind an die Technologie gebunden und werden rechts unten in der Grafik eingeordnet. [LL07]

1.3 Ziele der Arbeit

Das Ziel dieser Arbeit ist die Konzeption und Entwicklung eines Variabilitätsmanagements, das den Nutzer bei der Erstellung einer Fahrzeugsimulation unterstützen soll. Die Auflösung der Variabilitäten erfolgt durch eine automatisierte Entscheidungsfindung anhand aussagenlogischer Regeln. Bei der Konzeption und Umsetzung werden die grundlegenden Ansätze der Software Produktlinien, FODA (Feature-Orientated Domain Analysis) als auch diverse Konzepte von Variabilitätsmodellen in diese Arbeit einbezogen. Anschließend wird die Entwicklung des Variabilitätsmanagements in ein bestehendes Simulationsframework der Firma Bosch GmbH beispielhaft integriert.

Durch die Integration eines Konzepts zum Variabilitätsmanagement wird dem Nutzer ermöglicht, vordefinierte Aspekte der Applikation an die eigenen Anforderungen anzupassen. Dies wird erreicht, indem an bewusst unspezifizierten Stellen sogenannte Variabilitätspunkte gesetzt werden. An Stelle der Variabilitätspunkte können alternative, optionale oder verpflichtende Simulationsmodule gebunden oder parameterisiert werden. Die Bindung bezeichnet dabei die Zuweisung eines konkreten Simulationsmoduls, um

die Variabilität an der jeweiligen Stelle aufzulösen. Die Entscheidungsfindung, welche Simulationsmodule gebunden oder parameterisiert werden sollen, erfolgt durch die automatische Auflösung der Regeln. Die Realisierung der Entscheidungsfindung sowie die Definition eines Formats für die Entscheidungsregeln sind ebenso Bestandteil dieser Arbeit.

Die Regeln geben aussagenlogische Informationen wieder und spezifizieren, ob und wie die Simulationsmodule zusammenhängen. Dadurch lassen sich komplexe Abhängigkeiten zwischen unterschiedlichen Komponenten des Simulationsmodells darstellen. Das Variabilitätsmanagement wertet diese Regeln unter Beachtung der Simulationsmodule aus. Dadurch werden die Simulationsmodule konfiguriert sowie parameterisiert.

Durch den Einsatz der Variabilitätspunkte mit zugehörigen Entscheidungsregeln wird eine theoretische Abstraktionsschicht zwischen der direkten Implementierung und der Parameterisierung erzeugt. Eine direkte Konfiguration, wie sie bisher im betrieblichen Ablauf gehandhabt wurde, wird damit unnötig. Die Entscheidungsregeln beziehen sich dabei auf die vom Systemnutzer ausgewählten Features. Features bezeichnet man als wesentliche, meist für den Benutzer sichtbare funktionale oder nichtfunktionale Leistungsmerkmale eines Produkts [BKPS04]. Ein mögliches Feature in einem Automobil wäre beispielsweise ein "Winterpaket", welches eine Sitz- und Seitenspiegelheizung beinhaltet. Dabei muss mit Auswahl des Features automatisch der richtige Sitz, Seitenspiegel und die Bedienungselemente (z.B. zur Regulierung der Temperatur der Sitzheizung) integriert werden, ohne dass der Nutzer die Details kennen muss. Dem Nutzer soll durch die veränderte Konfiguration das nötige Wissen über das Systemmodell vereinfacht werden und der Umgang mit dessen direkter Parameterkonfiguration entfallen. Statt der direkten Konfiguration ist nur noch die Auswahl von „Features“ und deren Konfiguration erforderlich. Durch diesen variablen Ansatz erhalten wir eine spätmöglichste und flexible Bindung der Simulationsmodule.

Im zweiten Teil dieser Arbeit wird eine prototypische Implementierung eines Ansatzes zum Variabilitätsmanagement in einem Fahrzeugsimulationssystem der Firma Bosch GmbH vorgestellt. Unter zu Hilfenahme des vorgegeben Simulationsframework wird das entwickelte Variabilitätsmanagement integriert. Im konkreten Anwendungsfall werden die Systemmodule anhand vorgegebener Regeln, welche den Modulen selbst zugeordnet sind, parameterisiert.

2 Grundlagen

Für das Verständnis der Variabilität in Software insbesondere in der Fahrzeugsimulation bedarf es einiger einführender Informationen zu den verwendeten Technologien und Vorgehensweisen. In diesem Kapitel gehen wir nicht speziell auf die Variabilität in der Fahrzeugsimulation, sondern auf allgemeine Konzepte und Vorgehensweisen für Analyse von Variabilität in Software ein.

Variabilität ist die Vorraussetzung für die gezielte Konstruktion und Wiederverwendung von Produktartefakten [BKPS04]. Mit Hilfe des Domain Engineering werden gemeinsame und variable Artefakte definiert und entwickelt. Durch gültige Kombination der verschiedenen variablen Artefakte kann ein Produkt erzeugt werden. Produkt bedeutet in diesem Kontext eine abgeleitete Variante, also die komplette Bindung aller Variabilitäten mit dem Ergebnis einer Software. Durch Variabiltiät ist es möglich, auf die unterschiedlichen Kundenbedürfnisse im Rahmen der zur Verfügung stehenden Alternativen einzugehen. Dies bedeutet, dass die Softwarearchitektur in allen Alternativen identisch ist und sich nur die Funktionalitäten je nach Anforderung unterscheiden.

Kapitel 2.1 betrachtet Software Produktlinien. Diese haben die Wiederverwendung, Kosteneinsparungen sowie qualitätsgesicherte Software zum Ziel. Das wird durch die zwei zentralen Entwicklungsprozesse Domain- und Application Engineering erreicht. Die Domänenanalyse untersucht einen Bereich auf Zusammenhänge und Abläufe, mit dem Ziel, wiederverwendbaren Artefakte zu gewinnen. Dieses wird in Kapitel 2.1.1 vorgestellt. Feature Oriented Domain Analysis ist ein gängiges Verfahren für die Domainanalyse und wird in Kapitel 2.1.2 behandelt.

Kapitel 2.2 diskutiert die fundamentalen Konzepte um Variabilität in Software zu erreichen. Dabei wird in Kapitel 2.2.1 auf die Funktionsweise von Variabilitätspunkten und Varianten eingegangen. Die grundlegenden Modellierungmöglichkeiten von Variabilität werden in Kapitel 2.2.1 betrachtet. Zusätzlich wird in Kapitel 2.2.3 eine Klassifizierung der Variabilitätstypen vorgenommen sowie anhand eines Drei-Schichten-Architekturmusters die Variabilität in den jeweiligen Ebenen beschrieben (2.2.4). Kapitel 2.2.5 beschreibt zu welchem Zeitpunkt von der Entwicklung einer Software bis zum Betrieb eine Variabilität aufgelöst werden kann.

2.1 Software Produktlinien

Ähnliche Produkte erzeugen automatisch immer wieder identische oder ähnliche Abläufe. Der Vorreiter einer organisierter Wiederverwendung ist die Software-Produktlinie [BK04]. Sie unterstützt die Entwicklung von Produkten, die einen gemeinsamen *Kern* haben und zusätzlich über *spezifische Funktionalitäten* verfügen. Die spezifischen Funktionalitäten können vom Kunden gewählt oder abgewählt werden, um so ein für ihn angepasstes Produkt zusammenzustellen. Durch die gezielte Wiederverwendung verspricht man sich eine bessere Qualität, erprobte Systemkomponenten, kürzere Entwicklungszeiten und letztendlich geringere Kosten.
[BKPS04]

Dabei ist nicht allein die Identifikation der Komponenten, die für alle Produkte wiederverwendet werden können, von Interesse, sondern vielmehr sollte der Kern in möglichst vielen Produkten wiederverwendet werden. Die Entwicklung des Kerns als größter gemeinsamer Nenner aller Produkte führt dazu, dass der produktspezifische Teil wächst und keine Komponenten mehrfach entwickelt werden müssen.

Es existieren unterschiedliche Definitionen von Software-Produktlinien. Eine in diversen Literaturen häufig zitierte Definition stammt vom Software Engineering Institute der Carnegie Mellon Universität. Diese definieren eine Software-Produktlinie wie folgt:

> *„Software Productline - A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.“ [BCC+10]*

Der Referenzprozess für das Domain- und Applicationengineering wurde von F. van der Linden entworfen und ist in Abbildung 2.1 dargestellt.
Die Entwicklung einer Produktlinie erfolgt im wesentlichen in den zwei Hauptaktivitäten, nämlich Domain- und Applikation Engineering. Diese sind durch ein Rechteck mit abgerundeten Ecken (in der Abbildung oben und unten) dargestellt.

Das *Domain Engineering* identifiziert und entwickelt die gemeinsamen und variablen Artefakte, die Bestandteile der Produktlinie werden. Es kann als grundlegendes Werkzeug der Software-Produktlinien in diesem Bereich angesehen werden [Puk06]. Dies bildet die Grundlage für eine gemeinsame Plattform und variable Artefakte. Die Variabilität wird dadurch erreicht, dass eine vordefinierte Menge Software-Einheiten durch Selektion und Konfiguration an die unterschiedlichen Kundenbedürfnisse angepasst werden können. Die gemeinsamen Elemente aller Produkte bilden schließlich die Plattform . Das Domain Engineering repräsentiert also einen zusammenhängenden Funktionsbereich der Problemseite, also die komplette Produktlinie mit allen gemeinsamen und variablen Elementen.
[BKPS04]

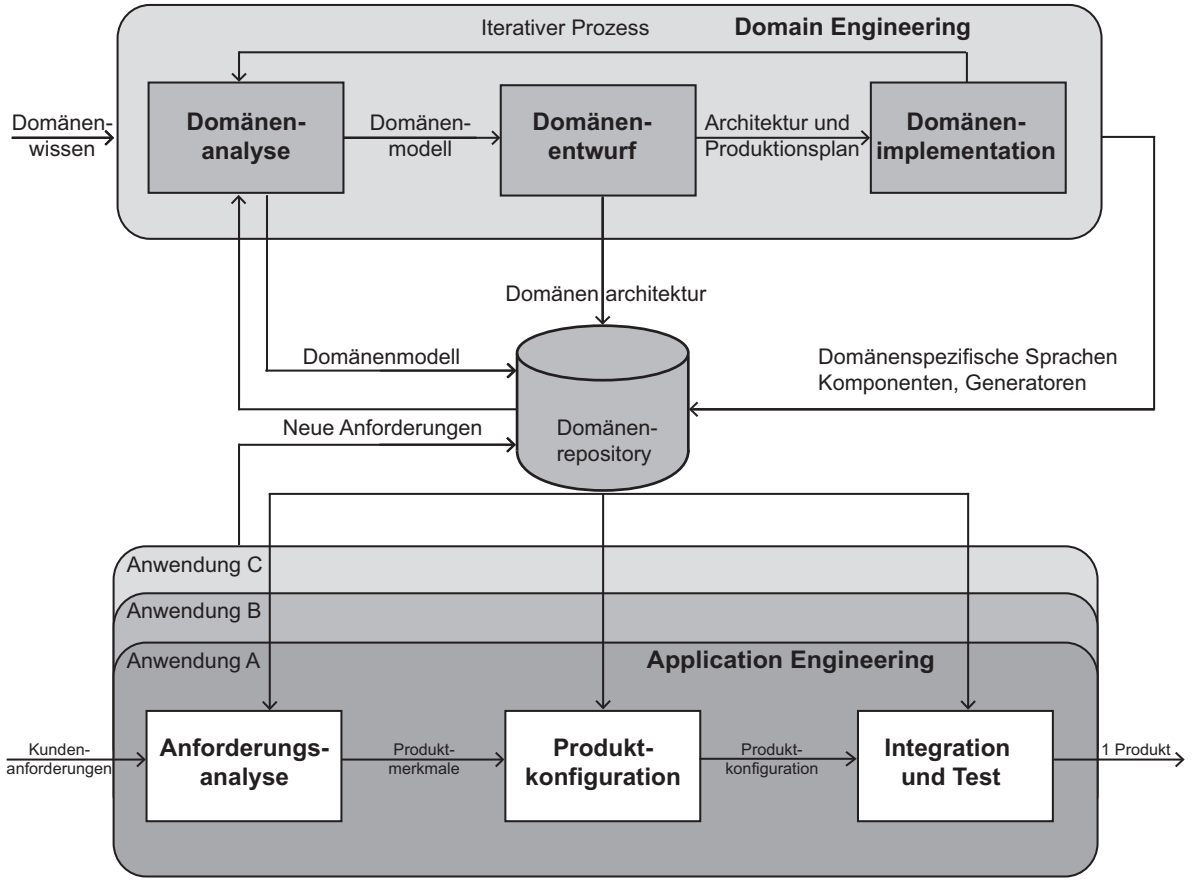

Abbildung 2.1: Die Abbildung stellt den Referenzprozess für Software-Produktlinienentwicklung dar. Die Domänenentwicklung erfolgt durch einen iterativen Prozess in den drei Phasen: Domänenanalyse, Domänenentwurf und Domämenimplementation. Die Datenflüsse zwischen den Phasen sind durch Kanten dargestellt. Das Domänenrepository stellt einen Datenspeicher zum Austausch der Daten zwischen dem Domain- und Application Engineering dar. Die Ableitung einer konkreten Anwendung wird durch das Application Engineering vorgenommen. In den Phasen der Anforderungsanalyse, Produktkonfiguration sowie Integration und Test werden die Kundenanforderungen angenommen und daraus ein fertiges Produkt entwickelt.

Das *Application Engineering* hingegen wird meist durch eine technische Domäne der Lösungsseite repräsentiert, also ein konkreter Teilbereich der Produktlinie [BKPS04]. Es werden die Produkte aus der Produktlinie abgeleitet. Diese werden dabei aus den gemeinsamen und teilweise variablen Software-Einheiten, welche im Domain Engineering entwickelt wurden, zusammengefügt.

Ein Funktionsbereich der Problemseite ist beispielsweise die Produktlinie eines Automobils, da sie alle gemeinsamen und variablen Elemente enthält. Hingegen ist ein fertig erstelltes Automobil eine konkrete Ausprägung der Produktlinie und repräsentiert einen konkreten Teilbereich davon.

Das in der Abbildung dargestellte `Domänenrepository` stellt die „Ergebnissammlung“ dar, auf welchen die jeweiligen Aktivitäten des Domain- und Applicationengineering lesend und schreibend zugreifen können. Der Schreibezugriff wird durch eingehende Kanten, der Lesezugriff durch ausgehende Kanten dargestellt.
Das Domain Engineering wird als iterativer Prozess gesehen, in dem drei Aktivitäten vorkommen. Die `Domänenanalyse`, der `Domänenentwurf` sowie die Domänenimplementierung. In der Abbildung 2.1 sind diese durch Rechtecke dargestellt und grau hinterlegt. Der Datenfluss zwischen den Aktivitäten wird jeweils mit Kanten dargestellt.

Ausgehend vom `Domänenwissen`, dargestellt durch eine Kante in der Abbildung und den neuen Anforderungen aus bisherigen `Iterationen`, beginnt der Entwicklungsprozess mit der `Domänenanalyse`.

In der *Domänenanalyse* werden relevante Informationen zur Domäne gesammelt und eine Abgrenzung zu angrenzenden Domänen und Systemen geschaffen. Ebenso werden die gemeinsamen und unterschiedlichen Merkmale der Gesamtdomäne analysiert und abgegrenzt. Die Ergebnisse werden in Domänenmodellen festgehalten. Diese sind Grundlage für den Domänenentwurf und das Application Engineering. In Abbildung 2.1 wird dies durch die Kanten `Domänenmodell` zum `Domänenrepository` und `Domänenentwurf` dargestellt.
Im *Domänenentwurf* wird der Entwurf einer Domänenarchitektur für eine Familie von Systemen erstellt. Ebenso wird ein Dokument erzeugt, in dem der Prozess zum Bau konkreter Systeme dargestellt ist. Die `Domänenarchitektur` wird im `Domänenrepository` hinterlegt. Dies ist in der Abbildung 2.1 durch eine Kante dargestellt. Die Architektur und der Produktionsplan werden zudem der `Domänenimplementation` zur Verfügung gestellt.
In der *Domänenimplementierung* werden die Architektur und die variablen Komponenten umgesetzt. Die Komponenten bilden die Grundlage für weitere Iterationen zur Optimierung der Anwendung.

Die im Domain Engineering entwickelten Komponenten, Generatoren und domänenspezifischen Sprachen werden in der `Domänenrepository` hinterlegt.

Mit der Anforderung eines Kunden beginnt eine Entwicklung im Application Engineering. Diese läuft ebenfalls in drei Phasen, nämlich der `Anforderungsanalyse`,

`Produktkonfiguration` sowie `Integration und Test`. Diese sind in der Abbildung durch weiß hinterlegte Rechtecke dargestellt.

In der *Anforderungsanalyse* werden die Anforderungen des Kunden aufgenommen und anhand der zur Verfügung stehenden Informationen in der `Domänenrepository` überprüft, ob die Kundenanforderungen erfüllt werden können. Ebenso werden anhand der Kundenanforderungen die Produktmerkmale definiert. Damit wird die `Produktkonfiguration` durchgeführt.
Die *Produktkonfiguration* stellt mittels der zur Verfügung stehenden gemeinsamen und variablen Komponenten aus dem Domänenentwurf die Produktkonfiguration her. Die `Produktkonfiguration` wird dann zur Integration in der letzten Phase `Integration und Test` benötigt. In der Phase *Integration und Test* wird mit den ausgewählten Komponenten der Produktkonfiguration die Integration des Produktes vorgenommen. Dazu werden alle Komponenten gebunden und eine fertiges Produkt abgeleitet.

Die Erkenntnisse und Anforderungen, die sich während der Produktlinienentwicklungen ergeben, werden in der Domänenrepository abgelegt. Diese werden bei weiteren Iterationen der Domänenentwicklung einbezogen.

Im folgenden betrachten wir die Domänenanalyse als Teil des Domain Engineering detaillierter, welche für diese Arbeit hauptsächlich relevant ist.

2.1.1 Domänenanalyse

Unter Domänenanalyse (Domain Analysis) versteht man eine systematische Untersuchung eines Bereichs auf Zusammenhänge und Abläufe mit dem Ziel, wiederverwendbare Artefakte zu gewinnen. Um ein Domänenmodell zu erstellen, unterscheidet man in der Vorgehensweise folgende drei aufeinanderfolgende Phasen: Die Kontextanalyse, die Domänenmodellierung und die Architekturmodellierung. In der Kontextanalyse wird der Umfang und die Grenzen der Domäne analysiert. Ebenso werden die Beziehungen der Domäne zu den benachbarten Domänen beschrieben. In der zweiten Phase, der Domänenmodellierung, wird die innere Struktur einer Domäne untersucht und dokumentiert. Hieraus entsteht dann das eigentliche Modell. In der Architekturmodellierungsphase werden letztendlich die Softwarelösungen aufgrund der Ergebnisse der Domänenmodellierung entworfen. Die Phasen werden in der Regel mehrfach durchlaufen, um Fragen und Schwierigkeiten, die in vorherigen Iterationen auftraten, zu berücksichtigen [Sch03].

Die möglichen Informationsquellen, um das nötige Domänenwissen zu erlangen, sind vielseitig. Üblicherweise werden bereits existierende Systeme herangezogen. Deren Quellcode, Benutzerhandbücher und Anforderungsspezifikationen können als Grundlage verwendet werden. Experten, welche Erfahrung mit bereits existierenden Anwendungen in dieser Domäne haben, sollten zusätzlich hinzugezogen werden, ebenso technische Literatur,

Kundenbefragungen und Marktanalysen. Das Untersuchen bereits existierender Systeme kann Schwachpunkte und Stärken vorheriger Entwicklungen aufdecken. Abbildung 2.2 stellt die Domänenanalyse in einem anschaulichen Diagramm dar.

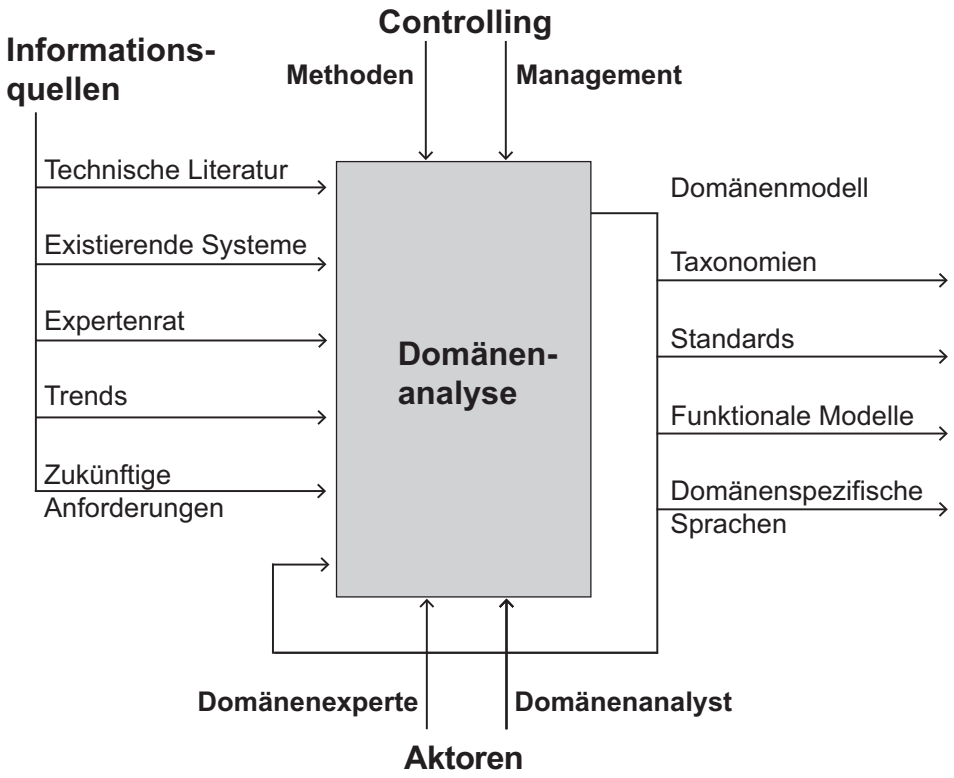

Abbildung 2.2: Diese Abbildung stellt die Domänenanalyse in einem Kontextdiagramm dar. Die Durchführung einer Kontextanalyse erfordert diverse Informationsquellen, z.B. Technische Literatur, Existierende Systeme, Expertenrat, Trends oder künftige Anforderungen. In der Abbildung sind diese mit eingehenden Kanten dargestellt. Zusätzlich gibt ein Controlling die Methoden zur Analyse vor. Gängige Verfahren sind „PuLSE CDA“ und Feature Oriented Domain Analysis. Die Aktoren, häufig Domänenexperten und Domänenanalysisten führen diese Verfahren aus. Das Ergebnis ist ein Domänenmodell das aus Taxonomien, funktionale Modelle, Domänenspezifische Sprachen oder Festlegung von Standards besteht. Die Ergebnisse werden dann für weitere Iterationen oder für die Weiterverwendung im Domänenentwurf verwendet. [APD91]

Das graue Rechteck repräsentiert die `Domänenanalyse`. Diese arbeitet mit den Informationen aus den diversen `Informationsquellen`, welche in der Abbildung links dargestellt sind. Das `Controlling` gibt die `Methoden` für die Domänenanalyse vor, mit welchen die `Aktoren` (`Domänenexperte` und `Domänenanalyst`) arbeiten. Gängige Verfahren zur Domänenanalyse

sind beispielsweise „PuLSE CDA“ und "Feature Oriented Domain Analysis", auf welches wir in Kapitel 2.1.2 näher eingehen.
Schließlich entsteht ein sogenanntes `Domänenmodell`. In der Praxis ist das Domänenmodell eine Menge von Dokumenten mit extrahiertem Wissen aus unterschiedlichen Sichten [Sch03]. Dieses liefert Informationen über Objekte, Funktionen, Daten und Beziehungen in diversen Modellen, welche in der Gesamtheit die jeweilige Problemdomäne spezifizieren. Genaue Bestandteile dieses Modells sind nicht vorgeschrieben und werden in diversen Literaturen unterschiedlich behandelt. Oftmals wird der Einsatz von `Taxonomien`, `Dokumentationen`, `Schnittstellenbeschreibung` und `funktionalen Modelle` wie Datenflussdiagramme, UML- oder Aktivitätsdiagrammen, genauso wie ein Begriffslexikon vorgeschlagen [Beh02].

Einen allgemeinen Standard für das Domänenmodell, gibt es bisher nicht. Dennoch wird der „Feature Oriented Domain Analysis“-Ansatz, den wir in Kapitel 2.1.2 betrachten, als elementarer Ansatz eingestuft. Er dient dazu, relevante Funktionsbereiche für die Wiederverwendung zu identifizieren und abzugrenzen. Außerdem stellt „Feature Oriented Domain Analysis“ eine Möglichkeit für Modellierung in Variabilität vor.

2.1.2 Feature Oriented Domain Analysis

Zu den wichtigsten Methoden der Domänenanalyse zählt die vom Software Engineering Institute der Carnegie Mellon Universität in Pissburg entwickelte „Feature Oriented Domain Analysis“ (FODA). Das in diesem Kapitel vorgestellte Verfahren basiert auf deren technischem Report [KCH+90] und einer Zusammenfassung [Sch03].

FODA stützt sich wie diverse andere Ansätze darauf, die gemeinsamen und unterschiedlichen Elemente zu identifizieren. Das Ziel der Feature Oriented Domain Analysis ist, die generischen Domänenmodule herauszuarbeiten, welche stark wiederverwendbar sind. FODA wird in drei Hauptaktivitäten unterteilt: Kontextanalyse, Domänenmodellierung und Architekturmodellierung. Bei diesen Aktivitäten werden diverse Diagramme und Modelle erzeugt.

Abbildung 2.3 stellt die FODA Hauptaktivitäten sowie deren zugehörigen Modelle und Diagramme dar.
Die FODA Hauptaktivitäten (`Kontextanalyse`, `Domänen- und Architekturmodellierung`) sind horizontal in Rechtecken abgebildet. Die Kanten zwischen den Rechtecken geben die Reihenfolge an, in welcher die Phasen abgearbeitet werden. Jedem der Phasen sind Modelle und Diagramme zugeordnet, welche üblicherweise in jeder dieser Phasen erstellt werden. Die `Kontextanalyse` erzeugt ein Struktur- und Kontextmodell. Das Informationsmodell, Feature-Modell, diverse funktionale Modelle sowie ein Begriffslexikon sind das Ergebnis der `Domänenmodellierung`. In der Phase der `Architekturmodellierung` entstehen ein Prozess-Interaktionsmodell sowie ein Modul-Strukturdiagramm. Alle Modelle und Diagramme bilden das sogenannte `Domänenmodell`, das dann im Domänenentwurf weiterverwendet wird.

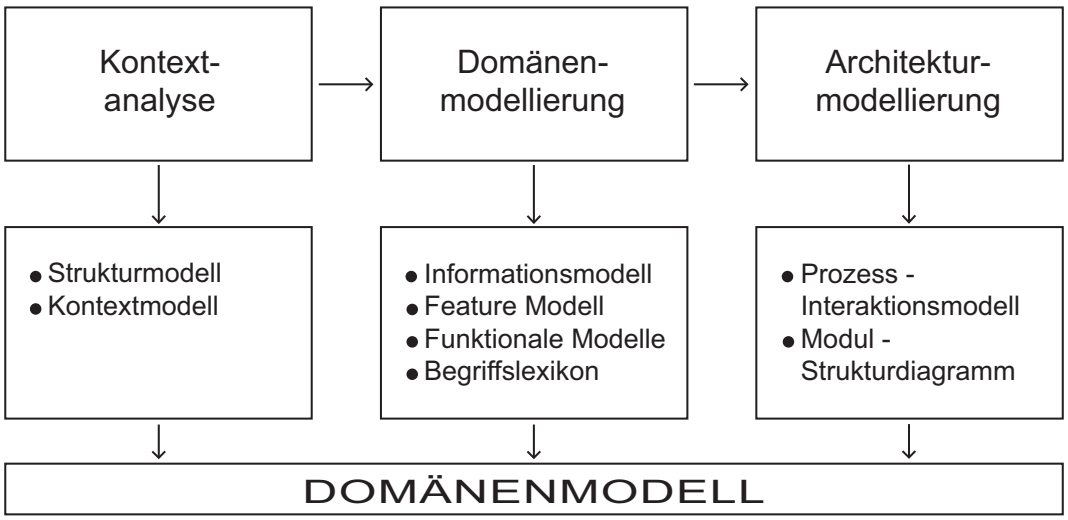

Abbildung 2.3: Die Abbildung zeigt die Hauptaktivitäten der FODA Methode und die Modelle, welche in den jeweiligen Aktivitäten erstellt werden. In der Kontextanalyse wird ein Strukturmodell und ein Kontextmodell erzeugt. In der Domänenmodellierung geht es darum, ein Informationsmodell, ein Feature Modell, funktionale Modelle und ein Begriffslexikon zu erzeugen. Während der Architekturmodellierung wird ein Prozess-Interaktionsmodell sowie ein Modul-Strukturdiagramm erstellt. Alle Modelle zusammen bilden das Domänenmodell.

Im Folgenden wird ein Bordcomputer für eine beispielhafte Analyse verwendet. Es gilt zu beachten, dass die Ausführung anhand des Beispiels unvollständig sind, sondern nur zur Verbesserung des Verständnisses beitragen sollen. Ein Bordcomputer ist ein Anzeigegerät, an dem diverse Informationen eingestellt und abgefragt werden können. Typische Informationen bei einem Bordcomputer für Automobile sind Restkraftstoffmenge, momentane Geschwindigkeit, Reifendruck und der Durchschnittsverbrauch. Häufig werden auch Funktionen angeboten, um z.B. das Radio, die Klimaanlage, die Sitzheizung oder andere Geräte zu steuern. In gehobenen Ausstattungslinien kann der Bordcomputer auch Navigationsinformationen bereitstellen.

1. Kontextanalyse:

In der Kontextanalyse soll die Problematik der kompletten Domäne erfasst werden. Dazu wird deren Umfang definiert und zu benachbarten Domänen abgegrenzt. Es soll ein generelles Verständnis der Domäne sowie deren Eingliederung im Kontext erfolgen. Die Ergebnisse hierzu werden in einem Modell fixiert, welches sich aus einem Struktur- und Kontextmodell zusammensetzt.
Ein Strukturdiagramm wird meist durch eine Schichtendarstellung realisiert. Die Kommunikation in Schichtendiagrammen verläuft dabei immer nur zwischen übereinander

liegenden Schichten. Direkte Kommunikation zu anderen Schichten sollte vermieden werden. Abbildung 2.4 zeigt ein mögliches Strukturdiagramm des Bordcomputers. Die Software des Bordcomputers kommuniziert mit entfernten Treibern, dargestellt durch eine `Kommunikationsschicht` und einer Treiberschicht. Die beispielhaften Treiber zeigen den direkten Zugriff auf Hardwarekomponenten. In der Abbildung bietet `Treiber 1` Zugriff auf den Ölstandsensor, während `Treiber n` eine Schnittstelle zum Reifendruckmesser bildet. Der `Ölstandsensor` gibt Auskunft darüber, ob genügend Motoröl vorhanden ist. Der `Reifendruckmesser` zeigt den aktuellen Reifendruck für jedes Rad an. Da weitere Hardwarekomponenten, die über den Bordcomputer angesprochen werden können, hinzukommen können, sind die Treiber in der Abbildung mit 1 bis n bezeichnet. In einem

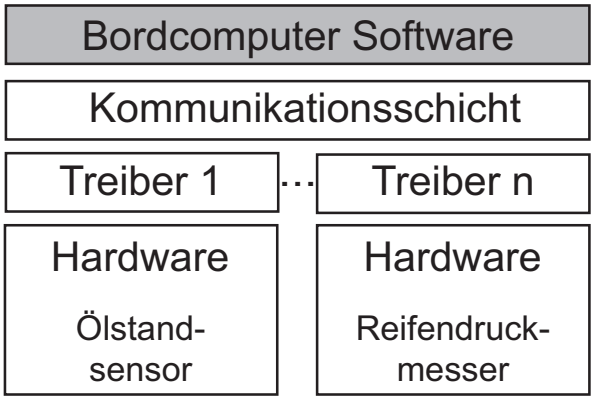

Abbildung 2.4: Der Bordcomputer greift über eine Kommunikationsschicht auf entfernte Treiber zu, welche Hardware ansteuern und auslesen. Die beispielhaft dargestellten Treiber sind für den Ölstdandsensor und Reifendruckmesser.

Kontextdiagramm siehe Abbildung 2.5, wird dann der Datenfluss zu den umliegenden Domänen dargestellt. Es soll dazu dienen, die benachbarten Domänen abzugrenzen sowie den Datenfluss zu beschreiben. Abbildung 2.5 zeigt die Einordnung der Bordcomputersoftware in Bezug zu der `Hardware` und dem `Fahrerdisplay`. Der Fahrer will beispielsweise über die Bordcomputeranzeige die Innentemperatur seines Fahrzeugs verändern. Der Befehl wird an die Bordcomputersoftware übertragen. Diese kommunziert mit dem `Treiber` der jeweiligen Hardware und gibt dann eine Erfolgsmeldung zurück, welche in der Bordcomputeranzeige dargestellt wird.

2. Domänenmodellierung:

Der Haupteil der FODA-Analyse ist die Domänenmodellierung. In dieser wird die Problemdomäne analysiert sowie die Gemeinsamkeiten und Unterschiede herausgearbeitet und in Komponenten aufgeteilt. Es entstehen ein Informationsmodell, ein Feature-Modell und ein operationales Modell. Ebenso wird ein Begriffslexikon eingeführt.

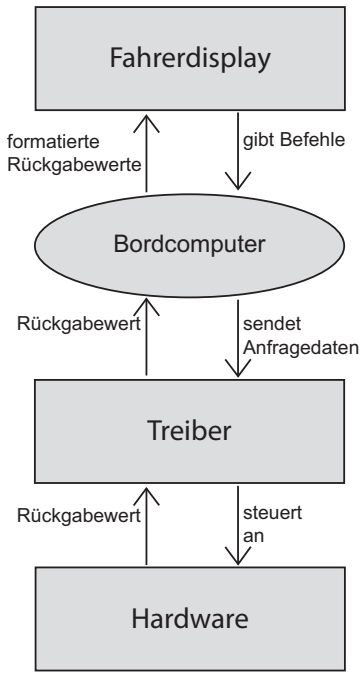

Abbildung 2.5: Kontextdiagramm eines Bordcomputers

Das Feature-Modell entsteht durch eine sogenannte Feature Analyse. Die Feature Analyse ist systematischer Ansatz zum Erfassen von wesentlichen Merkmalen, sogenannten Features [Maro6]. Durch eine hierarchische Anordnung erhält man eine grafische Darstellung der Zusammenhänge einer Domäne. Aufgabe dieses Diagramms ist es, die Struktur der Domäne mit Hilfe der Features zu spezifizieren. Um die Gemeinsamkeiten und die Variabilität in einem Feature-Diagramm darzustellen, gibt es alternative, optionale, obligatorische und auch „Oder“ Beziehungen. In Abbildung 2.6 werden diese dargestellt. Sie stellen Auszüge aus dem Feature-Modell eines Bordcomputers dar.
`Obligatorische Features` sind immer dann Teil der Konfiguration, wenn das Elternelement Bestandteil der Konfiguration ist. Die Darstellung erfolgt durch einen ausgefüllten Kreis am Ende der Kante. In Abbildung 2.6 ist das Anzeigeelement das Elternteil, die `Uhranzeige` sowie `Geschwindigkeitsanzeige` stellen die obligatorischen Features dar. In mancher Literatur wird das obligatorische Feature ohne ausgefüllten Kreis und nur als Kante dargestellt.
Bei der Erstellung der Konfiguration kann die Auswahl eines `optionalen Features` erfolgen. Das `optionale Feature` wird mit einem nicht ausgefüllten Kreis am Ende der Kante dargestellt. Steuerung ist in der Abbildung 2.6 das Elternelement. die Steuerung Radio sowie Steuerung Klimaanlage sind die optionalen Features.
Bei `Alternativen Features` muss bei Auswahl des Elternteils genau ein Kindelement ausgewählt werden. Eine Menge von `alternativen Features` können durch einen Kreisbogen an den jeweiligen Kanten der Features dargestellt werden. In Abbildung 2.6 sind die alternativen Features die Kindelemente digitale Anzeige und analoge Anzeige.

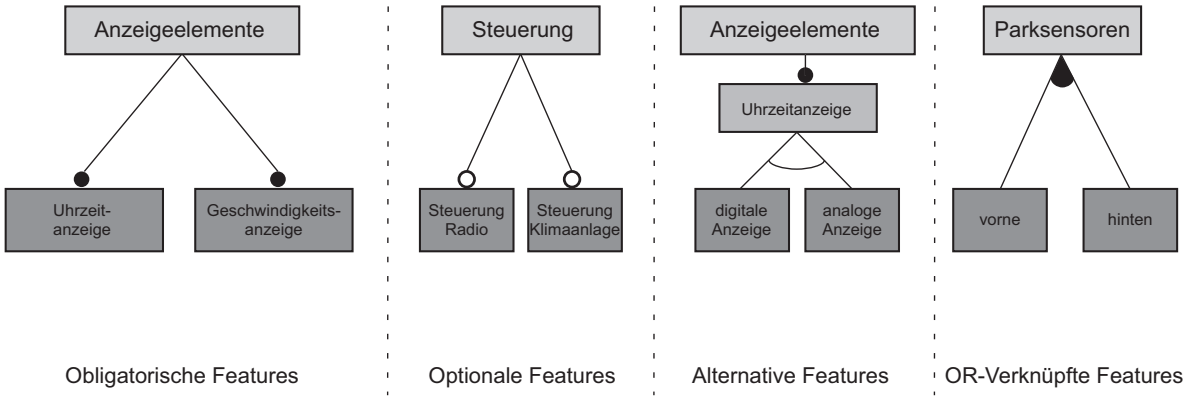

Abbildung 2.6: Die Abbildung stellt die Beziehungen des Feature Modells dar. Obligatorische Features werden mit ausgefüllten Kreisen am Ende der Kante dargestellt, optionale Features durch nicht ausgefüllte Kreise. Alternative Features können durch einen Kreisbogen an den Kanten der Features dargestellt werden, OR-verknüpfte Features durch einen ausgefüllten Kreisbogen.

Bei einer `Oder-Beziehung` muss mindestens ein Kindelement ausgewählt werden. Die `Oder-Beziehung` wird durch einen ausgefüllten Kreisbogen dargestellt. In Abbildung 2.6 ist dies durch die Parksensoren dargestellt. Es besteht die Möglichkeit, die Parksensoren vorne oder hinten oder vorne und hinten auszuwählen.

Abbildung 2.7 zeigt beispielhaft das Feature-Modell des Bordcomputers. Die Anzeigeelemente `Uhr` und `Geschwindkeitsanzeige` sind obligatorische Features. Dabei ist die Anzeige der Uhr entweder analog oder digital möglich und stellt ein alternatives Feature dar. Die Parksensoren sind optional und können entweder sowohl vorne als auch hinten am Fahrzeug angebracht sein. Die Steuerelemente für das Radio, die Klimaanlage und die Sitzheizung sind ebenfalls optional.

Das Informationsmodell als Teilmodell der Domänenmodellierung wird über eine Informationsanalyse erstellt. Es dient dazu, das extrahierte domänenspezifische Wissen in Form von Objekten und Beziehungen untereinander darzustellen. Ebenso werden Attribute dieser Objekte notiert. Die Darstellung erfolgt häufig als Entity-Relationship-Diagramm. Anhand dieser Modellierung können später im Rahmen des Domain Design die konkreten Datenstrukturen erstellt werden.

Im Rahmen der funktionalen Analyse werden Gemeinsamkeiten und Unterschiede hinsichtlich des Kontroll- und Datenflusses der domänenspezifischen Operationen identifiziert und im sog. funktionalen Modell modelliert. Das funktionale Modell dient zum Verständnis, wie die Anwendung in der Problemdomäne interagiert. Die als gemeinsam erkannten Features des Feature-Modells und die Objekte des Informationsmodells dienen als Basis, um dieses abstrakte funktionale Modell zu beschreiben.
[Beh02]

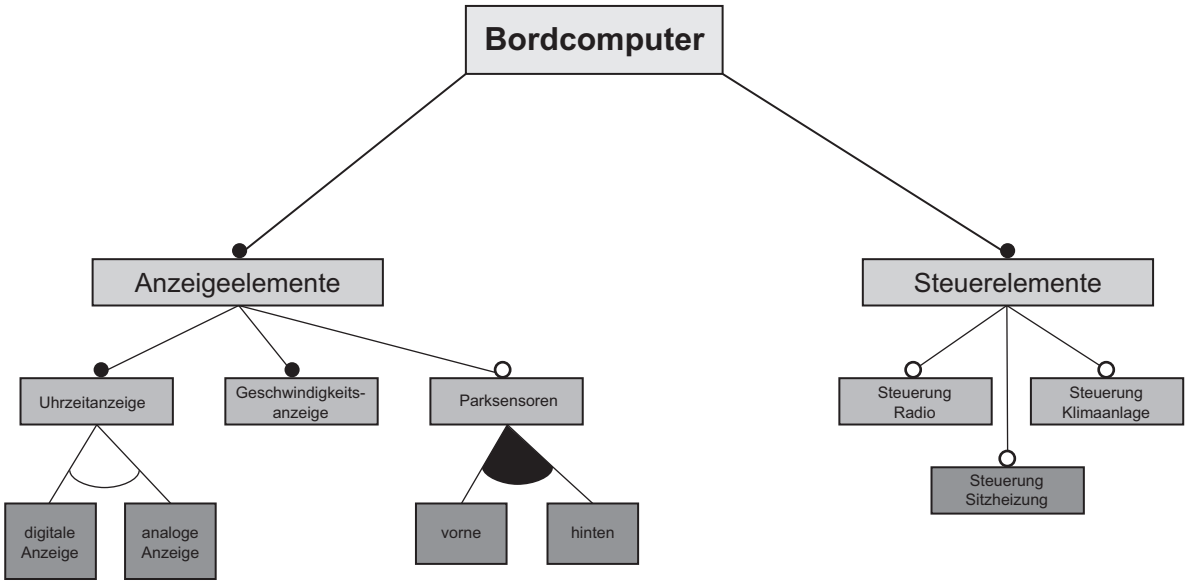

Abbildung 2.7

3. Architekturmodellierung:

Die Architekturmodellierung dient dazu, eine konkrete Lösung für die domänenspezifischen Probleme zu skizzieren. Das Architekturmodell soll dabei nur einen Grobentwurf der Produktlinienarchitektur darstellen. Die Produktlinienarchitektur beinhaltet die gemeinsamen und variablen Elemente der Produktlinien. Das Hauptaugenmerk dieser Modellierung liegt darauf, parallelablaufende Prozesse sowie gemeinsame Module bzw. Komponenten der Applikationen zu identifizieren und schließlich die erfassten Features, Funktionen und Datenobjekte den zugehörigen Prozessen und Modulen zuzuteilen.
[Beho2]

2.2 Variabilität in Software

Als Variabilität in Software werden die Teile einer Software bezeichnet, die durch Selektion an die unterschiedlichen Kundenbedürfnisse angepasst werden können. Variabilität ist die Voraussetzung für gezielte Wiederwendung.
[LLo7]

Zunächst (Kapitel 2.2.1) betrachten wir, was Variabilitätspunkte sind und wie sich mit deren Hilfe unspezifizierte Stellen in einer Software erstellen lassen. In Kapitel 2.2.2 wird dann auf die verschiedenen Modellierungsmöglichkeiten von Variabilität eingegangen. Einen Überblick über die existierenden Typen von Variabilität erhält man in Kapitel 2.2.3.

Beispielhaft betrachten wir in Kapitel 2.2.4 das Drei-Schichten-Architekturmuster, welches das mögliche Vorkommen der Typen von Variabilität in den jeweiligen Architekturebenen beschreibt. Letztendlich wird in Kapitel 2.2.5 auf die verschiedenen Bindungszeiten eingegangen, die angeben wann eine Variabilität aufgelöst wird.

2.2.1 Variabilitätspunkte und Varianten

Um Variabilität in Software zu erzielen, erstellt man noch während der Entwicklung sogenannte Variabilitätspunkte. *Variabilitätspunkte* sind bewusst unspezifizierte Stellen in der Implementierung. Ein Variabilitätspunkt gibt die Stelle an, an welcher eine konkrete Bindung eines sogenannten Artefakts zu einem späteren Zeitpunkt erfolgen wird. Ein *Artefakt* sind alle in der Software enthaltenen wiederverwendbaren, variablen Komponenten. Diese wurden in der Domänenentwicklung bereits definiert und entwickelt. Ihre Wiederverwendbarkeit in verschiedenen Kontexten soll durch die explizite Modellierung von Variabilität erreicht werden [BKPS04].

Abbildung 2.8 zeigt in Anlehnung an [ML08], [ML09] und [BHP03] ein Metamodell für Variabilitäten in Software.

`Variabilitätspunkt` und `Artefakte` stellen in der Abbildung Entitäten dar. Ein Artefakt ist eine Software-Einheit, welche an der Stelle des Variablitätspunktes gebunden werden kann. Jeder Variabilitätspunkt kann mit n Artefakte eine Bindung eingehen. Diese ist in der Abbildung durch die Relation `bindet` dargestellt. Jedes Artefakt kann ebenso n Variabilitätspunkten zugeordnet werden.
Die Entität `Variabilitätsabhängigkeit` gibt dabei an, ob und ggf. wie ein Artefakt einem Variabilitätspunkt zugeordnet (gebunden) werden kann. Daher besitzt die Entität `Variabilitätsabhängigkeit` ebenso eine Kante zur Relation `bindet`. Die verfügbaren Abhängigkeiten können aus logischer oder technischer Sicht definiert werden. Die technische Sicht gibt dabei an, ob das jeweilige Artefakt an diesem Variabilitätspunkt funktionieren kann. Die logische Sicht hingegen gibt an, ob die Bindung des Artefakts an dieser Stelle für den Nutzer Sinn ergibt. Die Beziehungen, die zwischen dem Variabilitätspunkt und dem Artefakten existieren, können optional, alternativ, sich gegenseitig ausschließend oder verpflichtend sein (vgl. auch [BHP03], [BKPS04]). Dies ist in der Abbildung durch eine Spezialisierung des Entitätstyps `Variabilitätsabhängigkeit` und den jeweiligen Entitäten `Optional`, `Alternativ`, `Pflicht` und `Ausschließend` dargestellt.
Bei *optionalen Beziehungen* zwischen dem Variabilitätspunkt und den Artefakten lässt sich das Artefakt aus- oder abwählen. Bei der *alternativen Beziehung* muss genau ein Artefakt aus der Menge der variablen Elemente gewählt werden. Eine *Pflichtbeziehung* zwischen den Elementen gibt an, dass mindestens ein Artefakt gebunden werden muss. Hingegen darf bei einer *Ausschlussbeziehung* das gegebene Artefakt auf keinem Fall gebunden werden.

Tabelle 2.1 stellt die Variabilitätsabhängigkeiten nochmals gegenüber.

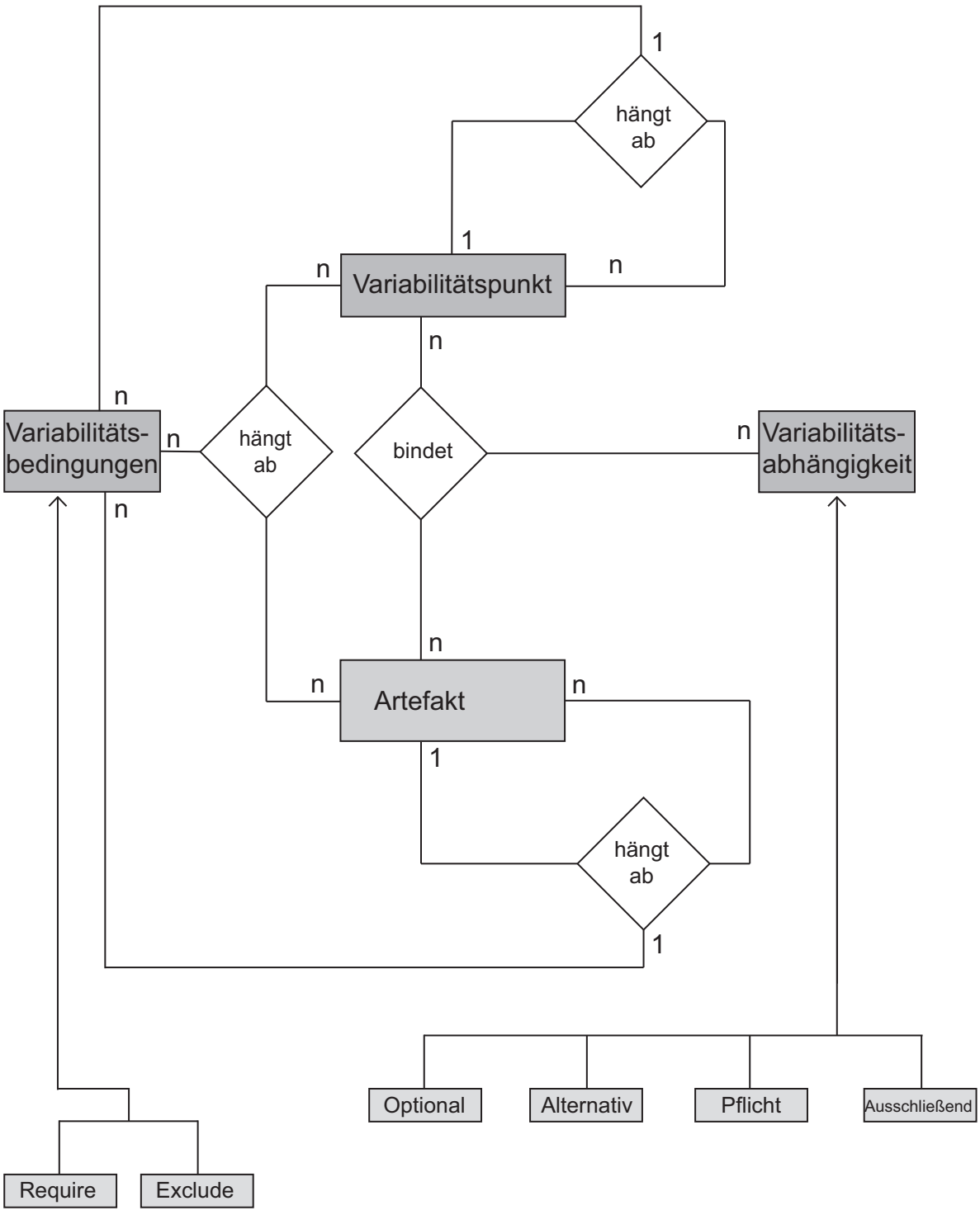

Abbildung 2.8: Metamodell für Variabilitäten als ER-Modell

Variabilitätsabhängigkeit	
Optionale Beziehungen	n<=1, mit |A|=1
Alternative Beziehungen	n=1, mit |A|>0
Pflichtbeziehungen	n>=1, mit |A|>0
Ausschlussbeziehungen	n=0, mit |A|>=0

Tabelle 2.1: Variabilitätsabhängigkeiten, A spezifiziert die Menge der Artefakte und n gibt die Anzahl der auszuwählenden Elemente an.

Abhängig von den spezifizierten Artefakten und dessen Abhängigkeiten sind nur bestimmte Werte oder Bindungen für einen Variabilitätspunkt zulässig. Es ist möglich, dass ein Variabilitätspunkt von n Variabilitätspunkten abhängen. Ebenso kann ein Artefakt von n Artefakten abhängen. Dies ist durch Relationen `hängt ab` und der Kardinalität 1 zu n bei den jeweiligen Entitäten `Variabilitätspunkte` und `Artefakt` dargestellt. Außerdem können Variabilitätspunkte von Artefakten abhängen. Die Abhängigkeit ist ebenso durch die Relation `hängt ab` dargestellt und verbindet die Entitäten `Variabilitätspunkte` und `Artefakt` mit der Kardinalität n zu n. Die Art der Zusammenhänge werden mit `Variabilitätsbedingungen` bezeichnet. In der Abbildung ist dieser als Entitätstyp dargestellt. Die Spezialisierung von `Variabilitätsbedingungen` ist eine `Requires` und `Exclude` Beziehung. Requires-Beziehungen definieren dabei, dass ein Variabilitätspunkt oder Artefakt einen anderen Variabilitätspunkt bzw. ein anderes Artefakt benötigt. Exclude-Beziehungen hingegen schließen zwei Artefakte oder zwei Variabilitätspunkte gegenseitig aus.

Soll ein neues Produkt erstellt werden, müssen die Variabilitätspunkte aufgelöst werden. Die Software, die dann durch die Bindung der Variabilitätspunkte entsteht, wird als *Variante* bezeichnet. Formal gesehen kann man einen Variabilitätspunkt also eine hinausgezögerte Designentscheidung realisieren [BKPS04].

2.2.2 Variabilitätsmodellierung

Für die Modellierung von Variabilitäten gibt es diverse Ansätze. Hauptsächlich sind dies Änderungen und Erweiterungen von bereits bestehenden allgemeinen Modellierungsansätzen. Die Notation von Feature-Modellen wurde bereits in Kapitel 2.1.2 ausführlich vorgestellt. Es gibt jedoch noch weitere Modellierungsansätze, die alle unterschiedliche Vorteile bringen. Aufgrund der Komplexität jedes einzelnen Ansatzes werden diese hier nur in groben Zügen vorgestellt. Die Ansätze um Variabilität zu modellieren sind sehr ähnlich. In jedem wird eine Stelle definiert, an welcher die Variabilität auftritt und welche Auswahlmöglichkeiten zur Verfügung stehen.

Eine mögliche Modellierung von Variabilität lässt sich beispielsweise mit Hilfe von

UML-Use-Case-Diagrammen [1] durchführen. Die Modellierung von Variabilität in Use-Case-Diagrammen eignet sich besonders, wenn versucht wird, die Variabilität aus Sicht des Nutzers darzustellen. In Anlehnung an [ML03] und [JRH+04] zeigt Abbildung 2.9 eine mögliche Modellierung der Anzeigeelemente „Uhr“ und „Parksensor“ eines Bordcomputers mit UML Use-Case. Die mögliche Variante sowie der Variabilitätspunkt ist in der Abbildung mit einem schwarzen Dreieck gekennzeichnet. Zusätzlich besteht die Möglichkeit, eine Einschränkung mit Hilfe einer Kardinalitätsbeschreibung zu definieren. Eine Option wird mit 0..1 definiert, eine Alternative mit 1...1 bzw 1...x. Die Anzeige des Parksensors hängt in diesem Fallbeispiel von der vorhandenen Hardware im Automobil ab. Das Anzeigenelement bindet den Variabiltätspunkt “Parksensor“ ein. Zur Anzeige der Distanzinformationen des Parksensors müssen an mindestens einer Stelle, im Front- oder Heckbereich, Parksensoren angebracht sein. Abhängig davon an welcher Position (vorne oder hinten oder in Kombination) die Parksensoren angebracht sind, wird die entsprechende Distanzinformation im Bordcomputer angezeigt. Die Kardinalität 0...2 in der Abbildung zeigt, dass es sich bei den Parksensoren um optionale Elemente handelt. Im Gegensatz dazu ist die Anzeige der Uhrzeit nicht an externe Hardware gekoppelt. Die Uhrzeit wird auf jeden Fall angezeigt, dies wird durch die Kardinalität 1...1 dargestellt. Welche der beiden Möglichkeiten gebunden wird, ist abhängig von der Wahl des Nutzers oder der Softwareversion.

Eine weitere Modellierungsmöglichkeit ergibt sich nach [PA08] durch UML-Aktivitätsdiagramme [2] , wie in Abbildung 2.10 dargestellt. UML-Aktivitätsdiagramme werden bei der Modellierung eingesetzt, um die dynamischen Aspekte von Software darzustellen. In Abbildung 2.10 wird der Variabilitätspunkt des Aktivitätsdiagramms mit Hilfe einer schwarzen Raute sowie dem Hinweis „«VP»“ dargestellt. An dieser Stelle soll eine entsprechende Variante gebunden werden. Diese sind in Kreisen abgebildet und durch gestrichelte Kanten mit dem Variabilitätspunkt verbunden. Die ausgehende Kanten der Varianten führen zu einem Verbindungsknoten (in der Abbildung die Raute). Zusätzlich besteht die Möglichkeit neben dem Variabilitätspunkt eine Kardinalitätsbeschreibung, der Form x...y, zu definieren.
In Abbildung 2.10 wird nach Start des Aktivitätsdiagramms die Aktion `Uhr anzeigen` aufgerufen. Der darauf folgende Variabilitätspunkt gibt die Möglichkeit die Variante `Digitale Anzeige` oder die Variante `Analoge Anzeige` zu binden. Abhängig der gewählten Variante wird nach Ablauf des Aktivitätsdiagrammes eine analoge oder digitale Uhr angezeigt.

Die letzte hier aufgezeigte Modellierungsmöglichkeit ist die Variantenmodellierung in UML. Stereotypen sind ein Mechanismus zur Erweiterung von UML auf der UML-Metamodellebene[3]. Das entsprechende Modellierungselement wird direkt durch die definierte Semantik beeinflußt. Auch Attribute, Operationen und Assoziationen können mit Stereotypen klassifiziert werden [Dum03]. Für die Definition neuer Stereotypen sind

[1] http://www.uml.org/
[2] http://www.uml.org/
[3] http://www.omg.org/spec/UML/2.1.1/

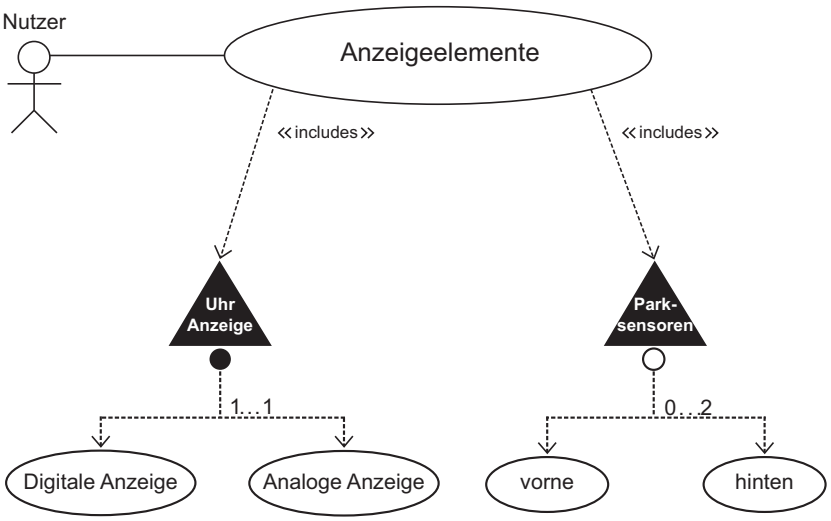

Abbildung 2.9: Das Diagramm stellt die Modellierung von Variablitätspunkten mit UML Use-Case Diagrammen dar. Zur Darstellung der Variabilität wird das Diagramm um schwarze Dreiecke und deren abgehende Kanten erweitert. Die schwarzen Dreiecke symbolisieren die Variabilitätspunkte. Die obligatorischen werden durch gefüllte und die optionalen durch leere Kreise unterhalb des schwarzen Dreiecks dargestellt. Die Kanten welche vom Variabilitätspunkt zu den Varianten führen, können zusätzlich eine Kardinalitiätsbeschreibungen erhalten.

daher Kenntnisse über das Metamodell erforderlich. Die Darstellung erfolgt oberhalb des Klassennamens mit einem Schlüsselwort zwischen spitzen Klammern in der Form «StereoType». Zur Modellierung von Geräten kann beispielsweise der Stereotyp «Device» eingeführt werden, welcher die Metaklasse „Class“ erweitert. Eine mögliche Klasse in der Modellebene ist dann „Computer“ oder „Festplatte“.

Für die Modellierung für Variabilität können ebenso Stereotypen genutzt werden, wie auch in Abbildung 2.11 dargestellt. Der Variabilitätspunkt wird durch den Stereotyp «VariabilityPoint», Varianten durch «Variant» identifiziert. Zusätzlich kann bei dem «VariabilityPoint» Objekt durch die Attribute `minSelection` und `maxSelection` eine Kardinalität angegeben werden. In der Abbildung wird der Variabilitätspunkt der `Uhr Anzeige` dargestellt, was durch den Stereotyp «VariabilityPoint> definiert ist. Die Varianten, dargestellt durch «Variant», sind die `Digitale Anzeige` und `Analoge Anzeige`. Die Kardinialität des Variabilitätspunkts definiert durch die Auswahl von `minSelection=1` und `maxSelection=1`, dass es ein alternatives Feature ist. Der Variabilitätspunkt `Parksensoren` mit den Varianten `vorne` und `hinten` wird mit der Kardinalität `minSelection=0` und `maxSelection=2` dargestellt. Daher sind die Varianten optional.

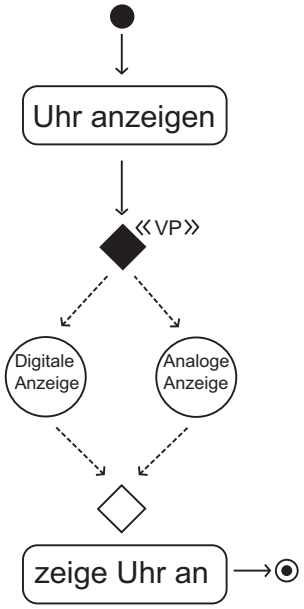

Abbildung 2.10: In dem abgebildeten UML-Aktivitätsdiagramm wird die Modellierung von Variabilitätspunkten dargestellt. Dem Diagramm wurden zur Darstellung der Variabiltätspunkte schwarze Rauten hinzugefügt. Bei den Rauten kann wahlweise eine Kardinalitätsbeschreibung angegeben werden. Die ausgehende Kanten der Rauten zeigen auf die Varianten.

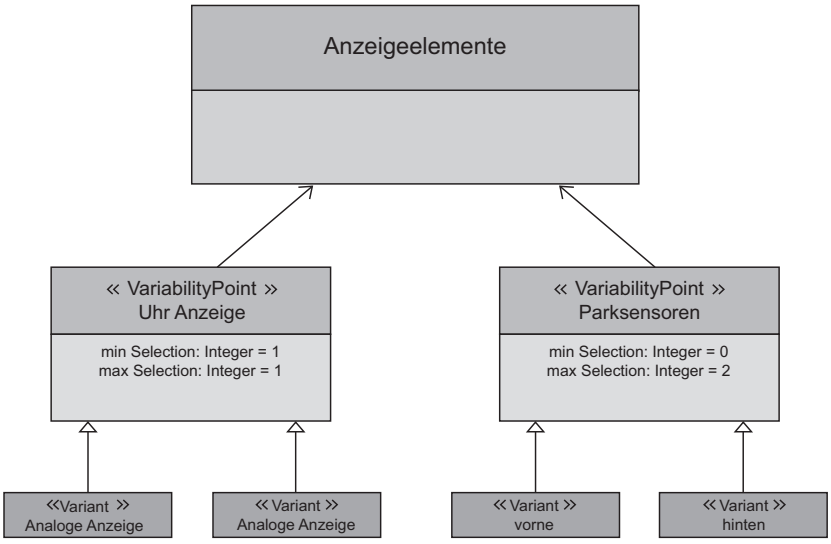

Abbildung 2.11: Modellierung von Variabilitäten in UML Diagrammen

2.2.3 Typen von Variabilität

Dieser Abschnitt gibt in Anlehnung an Arbeiten von [Wüb10], [BKPS04], [BB01], [PBL05] und [PA08] einen Einblick in Typen und Klassifizierungen von Variabilitäten.

Variablitäten werden häufig in sogenannte technische und fachliche Variabilitäten eingeteilt. Die *technische* Variabilität umfasst alle Variabilitäten, die sich mit dem Umfeld der jeweiligen Software beschäftigen, beispielsweise mit der zugrundeliegenden Hardware oder dem Betriebssystem. Die *fachliche* Variabilität umfasst alle Aspekte der Variabilität, die sich auf den Einsatz des Produktes beim Nutzer beziehen, wie beispielsweise Funktionalität, Systemumgebung oder betriebliche Einbettung [PH10]. Die fachliche Variabilität definiert daher die Arten der Variabilität, die für den Kunden relevant und sichtbar sind und somit die Variabilität bezüglich des Produkteinsatzes [PH10].

Abbildung 2.12 stellt die Klassifizierungen in einem Schichtenmodell dar. Es lässt sich erkennen, dass die technische Variabilität dem System nahe liegt. Die *technische Variabilität* wird von den Eigenschaften des Systems beeinflust. Hingegen wird die fachliche Variabilität von der zugrundeliegenden technischen Variabilität und den Ansprüchen des Nutzers umgesetzt.

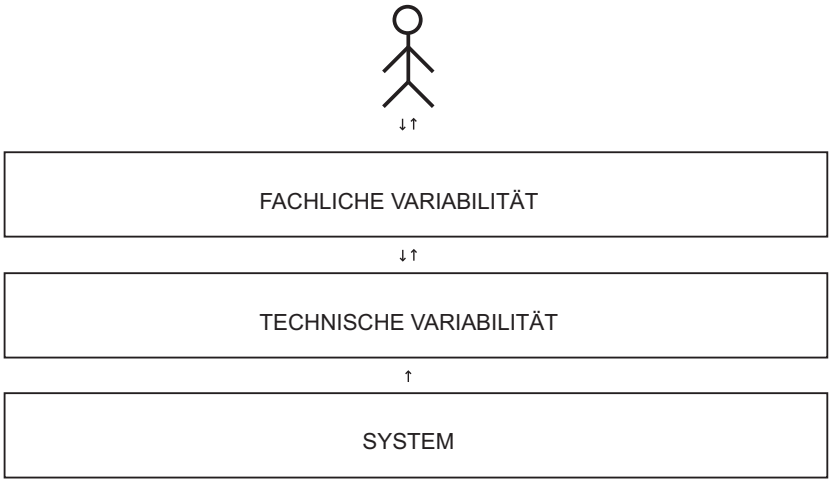

Abbildung 2.12: Die Abbildung zeigt eine Einteilung der fachlichen und technischen Variabilität in Bezug zu Nutzer und dem System. Die technische Variabilität basiert auf der zugrundeliegenden Hardware oder dem Betriebssystem. Bei Auflösung einer technischen Variabilität werden die Informationen des Systems einbezogen. Die fachliche Variabilität basiert auf Aspekte der Variabilität, die sich auf den Einsatzes des Produktes beim Nutzer beziehen.

Im Rahmen der technischen und fachlichen Variabilität kann man weiter untergliedern.

Einer der möglichen Typen der technischen Variabilität ist:

- Variabilität in der Infrastruktur: Variabilitätspunkte dieses Typs können auf die unterschiedlichen Infrastrukturen eingehen. Um die Lauffähigkeit einer Sofware zu gewährleisten, muss die gegebene Infrastruktur kompatibel zum Produkt sein. Dies betrifft die Interoperabilität mit anderen Software-und Hardwaresystemen, die Netzwerkumgebungen oder auch örtliche Gegebenheiten. Ebenso denkbar wären unterschiedliche Chipsätze, den zugrundeliegenden Befehlssatz des Prozessors oder die technische Internetverbindung (WLAN, Modem, LAN).

Mögliche Typen der fachlichen Variabilität sind:

- **Variabilität in Features:** Features sind Charakteristika eines Produktes. Die selbe Software enthält nach Bindung der Variabilitätspunkte einen unterschiedlichen Funktionsumfang. Zum Beispiel könnte ein Bordcomputer zusätzlich ein integriertes Navigationssystem enthalten.
- **Variabilität in Abläufen:** Das Auftreten von Variabilität in Abläufen bedeutet, dass auf Basis einer modularen Software die Prozessschritte variieren, aber das gleiche Ergebnis liefern. Beispielsweise könnte bei manchen Bordcomputern bei Start des Automobils ein PIN abgefragt werden, während dies in anderen Bordcomputern wegfällt.
- **Variabilität im Datenformat:** Variabilität im Datenformat kann eingesetzt werden, um identische Daten mit unterschiedlichem Format abzuleiten, z.B. könnte eine Speicherung des Ölstands in Litern oder in Prozent für weitere Berechnungen genutzt werden. Ebenso ist eine unterschiedliche Präsentation der Daten möglich.
- **Variabilität im Datenumfang:** Das Auftreten von Variabilität im Datenformat bedeutet, dass Produkte mit unterschiedlichen Attributen abgeleitet werden können. Beispielsweise bedingt eine zusätzliche Funktion wie „Ölstand anzeigen“, dass im Bordcomputer mehr Daten gespeichert werden müssen.
- **Variabilität im Systemzugang:** Die Variabilität im Systemzugang lässt die Möglichkeiten offen, die Authentifizierung je nach Bedarf auszutauschen und anzupassen. Beispielsweise könnte der Zugriff auf den Bordcomputer direkt im Fahrzeug und/oder alternativ über das Internet erfolgen. Hieraus ergibt sich eine mögliche Anwendung für der Diebstahlschutz.
- **Variabilität in Benutzerschnittstellen:** Mit der Variabilität einer Benutzerschnittstelle lässt sich die Präsentation gegenüber dem Nutzers anpassen. Hiermit ist es dann möglich, für die selben Daten und Funktionen diverse Ansichten zu liefern. Beispielsweise könnte zusätzlich zur Textausgabe auch eine sprachgesteuerte Ausgabe erfolgen.
- **Variabilität in Systemschnittstellen:** Variabilität in Systemschnittstellen bezeichnet, dass Produkte mit unterschiedlichen Schnittstellen zu externen Systemen, ausgeliefert werden können. Durch Variabiltität in der Systemschnittstelle könnten beispielsweise

eine Anbindung an die Verkehrsüberwachung, wahlweise über das Internet (GPRS[4]) oder Radiofunk (TMC [5]), erstellt werden.

- **Variabilität in der Qualität:** Die Variabilität in der Qualität bedeutet, dass identische Software mit unterschiedlichen Qualitätsaspekten- und Ausprägungen ausgeliefert wird - oftmals in Verbindung mit einem „Quality-of-Service"-Vertrag. Im Rahmen dessen werden z.B. Aspekte wie Verfügbarkeit, Zuverlässigkeit und Leistung festgelegt.

Es ist nur in seltenen Fällen möglich, die Typen der Variabilität isoliert zu betrachten. Eine Auswahl eines Features erfordert beispielsweise eine Veränderung der Benutzerschnittstelle und des Datenumfangs. Ebenso kann eine Auswahl eines Features die Deaktivierung eines anderen Features verursachen.

2.2.4 Variabilität in verschiedenen Architekturebenen

Die Strukturierung einer Anwendung wird häufig durch ein Schichtenmodell realisiert. Um die Variabilität in verschiedenen Architekturebenen zu erläutern, verwenden wir beispielhaft das Drei-Schichten-Architekturmuster, wie es häufig in interaktiven Software-Systemen zum Einsatz kommt. Das Drei-Schichten-Architekturmuster besteht aus einer Präsentationsschicht, einer Anwendungsschicht und einer Datenhaltungsschicht. Dies wird in Abbildung 2.13 dargestellt. Die Präsentationsschicht realisiert die Darstellung der Daten für den Nutzer. Ebenso übernimmt sie die Interaktion mit diesem und übergibt die Daten an die Anwendungsschicht. Die Anwendungsschicht realisiert die fachlichen Funktionen. Um Daten darzustellen oder zu manipulieren, greifen die Elemente der Anwendungsschicht auf die Datenhaltungsschicht zu und leiten diese an die Präsentationsschicht weiter. Die Datenhaltungsschicht sorgt ausschließlich dafür, dass die Daten persistent gehalten sind, beispielsweise auf dem lokalen Dateisystem oder in einer Datenbank. Ebenso sind auch Schnittstellen zu externen Dienstleistern möglich, die einen Service zur Datenspeicherung anbieten.

Variabilität kann in einer Architekturebene, in mehreren oder auch in allen Ebenen Anwendung finden. Zusätzlich können Variabilitätspunkte in mehreren Ebenen voneinander abhängen oder sich gegenseitig beeinflussen.
Variabilität in der *Präsentationsschicht* ist in Software relativ häufig und findet sich beispielsweise im Web oder auch in gängigen Softwarelösungen wie beispielsweise Textverarbeitungsprogrammen wieder. Diese Programme lassen die Ein- und Ausblendung und die Sortierung von Funktionen zu. SaaS[6] Lösungen beispielsweise Online-Shops beginnen bei der Inbetriebnahme häufig mit einen „Setup-Prozess", bei dem die Anwendung personalisiert werden kann. Die Anpassungen betrifft meist Farben oder die Einbindung eines

[4]General Packet Radio Service
[5]Traffic Message Channel
[6]Software as a Service

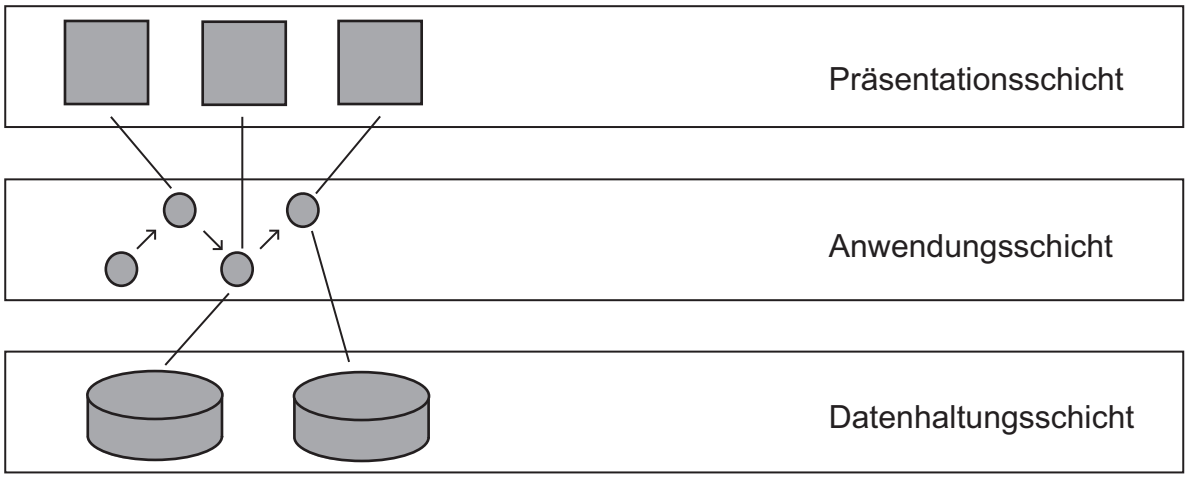

Abbildung 2.13: Drei-Schichten-Architekturmuster

eigenen Firmenlogos. An der Stelle der hinterlegten Farbwerte oder des Firmenlogos stehen die Variabilitätspunkte, an welche beim Setup-Prozess konkrete Werte gebunden werden.

Durch Variabilität in der *Anwendungsschicht* lässt sich die Funktionalität einer Anwendung anpassen. Die Datenstruktur ist dabei irrelevant. Vorteilhaft ist jedoch eine objektorientierte oder modulartige Struktur mit loser Kopplung. Die lose Kopplung bezeichnet eine geringe Anzahl von Abhängigkeiten zwischen Komponenten. Mit dieser Eigenschaft wird das flexible Austauschen bzw. Binden von Modulen unterstützt. Ein möglicher Anwendungsfall findet sich ebenfalls in Online-Shop Lösungen wieder. Während des „Setup-Prozess“ werden z.B. vom Betreiber die Zahlungsmöglichkeiten festgelegt. Jede Zahlungsart kann einen Variabilitätspunkt darstellen, welcher eine optionale Bindung zu einer Zahlungsart ermöglicht. Bei Bindung der ausgewählten Zahlungsmöglichkeiten wird die Software dahingehend modifiziert, dass dem Kunden bei Bestellung nur die ausgewählten Zahlungsarten zur Verfügung stehen.

Mit der Variabilität in der *Datenschicht* lassen sich beispielsweise die Speichermedien oder Datenquellen dynamisch binden. Dabei ist die Datenhaltung in Datenbanken, lokalen oder externen Dateisystemen oder andere Speichermedien sowie bei externen Dienstleistern durchaus möglich. Ein konkreter Anwendungsfall ist beispielsweise das Cloud-Computing. Die Unabhängigkeit der Datenhaltung wird immer relevanter. Beispielsweise handelt es sich bei Amazon Simple Storage Service[7] (S3) um einen Webservice, welcher darauf ausgelegt ist, eine beliebige Datenmenge zu jeder Zeit und an jedem Ort zu speichern oder abzurufen. Dabei wird der genaue physikalische Ort der Daten verborgen. Software-Angebote von diversen Softwareunternehmen arbeiten bereits mit dieser Technologie und lagern Daten in der „Cloud“. Eine Software könnte beispielsweise

[7]http://aws.amazon.com/de/s3

die Möglichkeit bieten, dynamisch Speichermedien oder Datenquellen zu binden. Jeder Variabilitätspunkt kann dabei mit einer oder mehreren Datenquellen assoziert werden. Bei Start der Software werden anhand diverser Kriterien die Varianten gebunden.

In häufigen Fällen tritt Variabilität in mehr als nur einer Schicht auf. Beispielsweise bedingt eine Änderung der Ablauflogik oftmals eine Änderung der Präsentation. Das heißt, dass es durchaus Anwendungsfälle geben kann, in denen die Variabilität über mehrere oder alle Schichten verteilt ist und sich gegenseitig beeinflusst. Um diese Thematik zu verdeutlichen, wird im Folgenden erneut eine Softwarelösung für den Bordcomputer vorgestellt, welche ebenfalls auf einem Drei-Schichten-Architekturmuster aufgebaut ist. Die Anwendung ist in Abbildung 2.14 dargestellt.

Bordcomputer geben dem Nutzer diverse Informationen über das Auto und ermöglichen ihm Einstellungen, zum Beispiel Temperaturregelung der Klimaanlage, Auswahl des Radiosenders, usw. Die Steuerung einer Klimaanlage über den Bordcomputer des Automobils ist ein möglicher Fall für Variabilität in allen drei Ebenen. Abhängig davon, ob eine Klimaanlage vorhanden ist und welches Modell eingesetzt wird (Datenschicht), müssen die Prozeduren zur Steuerung der Klimaanlage integriert werden (Anwendungsschicht). Die Einstellungsmöglichkeiten für den Nutzer sind davon abhängig ob und welcher Typ von Klimaanalage vorhanden ist (Präsentationsschicht).

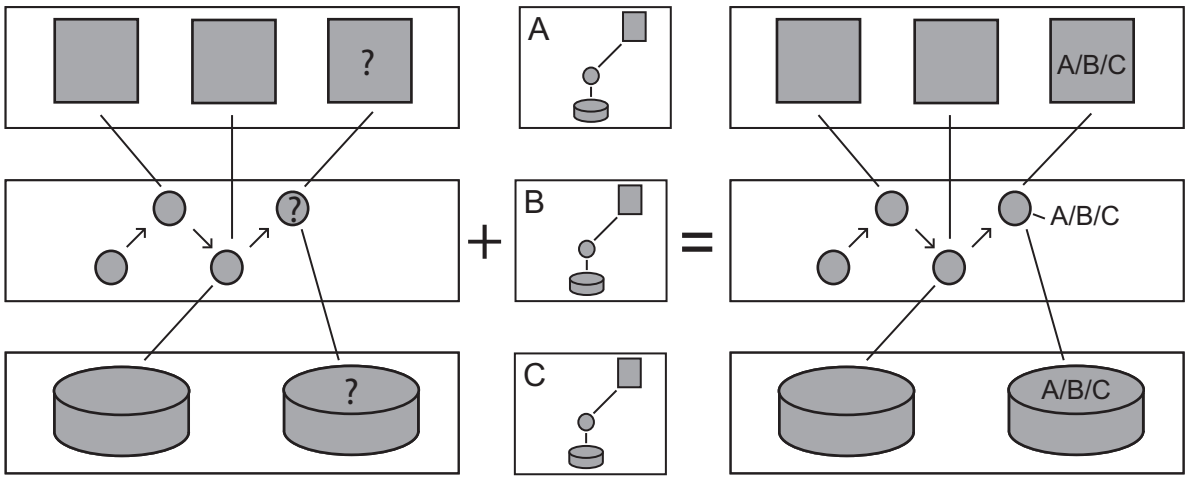

Abbildung 2.14: Die Abbildung zeigt Variabilität in der Drei-Schichten-Architektur. Der linke Teil der Abbildung zeigt die drei Schichten nämlich die Präsentationsschicht, die Anwendungsschicht und Datenschicht. Die „?“ definieren die Variabilitätspunkte der Anwendung, an welche noch keine konkreten Elemente gebunden sind. Die Rechtecke A, B und C definieren Auflösungsmöglichkeiten, um eine konkrete Variante zu erzeugen. Sofern eine der Varianten A, B oder C gebunden wird, werden die Variabilitäten in allen Ebenen aufgelöst. (dargestellt rechts von „=“)

Der Hersteller des Bordcomputers hat nun die Möglichkeit, für jedes einzelne Fahrzeug eine individuelle Software zu implementieren, welche die Unterschiedlichkeit der Fahrzeuge berücksichtigt. Alternativ könnten die Unterschiede (Variabilitäten) mit Hilfe von Variabilitätspunkten in den verschiedenen Ebenen integriert werden. Die Variabilitätspunkte in Abbildung 2.14 werden mit „?“ dargestellt. Die Rechtecke A, B und C stehen für Software der unterschiedlichen Klimaanlagentypen, welche deren jeweilige Funktionalität repräsentiert. Durch Bindung der konkreten Software (A, B, C) wird die Variabilität aufgelöst und

Die funktionale Logik ist in der Abbildung 2.14 durch Kreise mit Kanten dargestellt. Die Kanten stellen dabei jeweils ein Element (Funktion oder Klasse) dar. Das letzte Element enthält einen Variabilitätspunkt, der die Bindung der funktionalen Logik einer Klimaanlage ermöglicht. Wenn in dem Fahrzeug eine Klimaanlage eingebaut ist, müssen Prozeduren integriert werden, um eine Interaktion zwischen der Hardware und dem Nutzer zu ermöglichen.

Außerdem muss dem Nutzer eine Ansicht zur Einstellung von Parametern, beispielsweise Temperaturwerte, geboten werden. Dies erfordert eine Darstellung der Klimaanlage in der Präsentationsebene. In der Abbildung wird die Variabiltät in der Präsentationsebene durch das Rechteck mit dem „?“ dargestellt. Besitzt das Fahrzeug keine Klimaanlage, ist eine Anzeige in der Präsentationsschicht hinfällig.

Ebenso ist in unserem Beispiel eine Variabilität in der Datenschicht vorhanden. Nach der Wahl der Temperatur im Bordcomputer muss dieser Wert an die Klimaanlage übermittelt und dort gespeichert werden. Der Bordcomputer könnte diverse Typen von Klimaanlagen steuern. Eine Bindung in der Datenschicht kann also erst zu dem Zeitpunkt erfolgen, in dem klar ist, welcher Typ von Klimaanlage eingebaut wird. Zum Bindungszeitpunkt wird entsprechend des Klimaanlagentyps A, B oder C die entsprechende Variabilität aufgelöst.

2.2.5 Bindungszeitpunkte

Um eine konkrete Ausprägung eines variablen Produktes zu erhalten, müssen die Variabilitätspunkte gebunden werden. Als Bindung bezeichnet man das Hinzufügen bzw. Entfernen von Funktionalitäten sowie das Aus- bzw. Abwählen von Werten. Der Bindungszeitpunkt kann in verschiedenen Phasen während oder nach der Entwicklung also bei Inbetriebnahme oder während der Laufzeit des Programms liegen. Auch in unserem Fallbeispiel, der Software für Bordcomputer, könnte dies während der Entwicklung, bei der Installation oder während der Laufzeit geschehen. Je nach Anwendung und ggf. auch nach Variabilitätspunkt muss dies fallweise entschieden werden.

Mit Zuhilfenahme des Wasserfallmodells in Abbildung 2.15 betrachten wir die Entwicklung einer Software von der Systemanalyse bis zur Inbetriebnahme und analysieren dabei die möglichen Bindungszeitpunkte der Variabilitäten. Ohne genauer auf die

Bedeutung des Wasserfallmodells einzugehen kann man es als ein Modell zur Darstellung der Tätigkeiten bei einer Softwareentwicklung betrachten [LL07]. Man beginnt links oben und arbeitet sich nach rechts unten vor. Das Wasserfallmodell lässt sich unterteilen

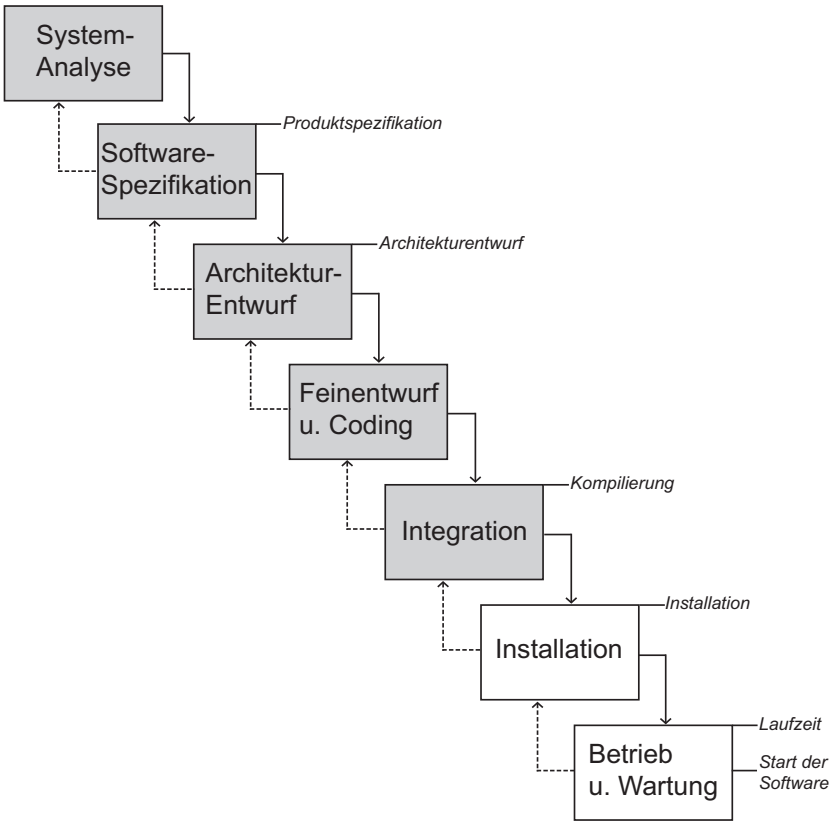

Abbildung 2.15: Das Wasserfallmodell [LL07]: Der Entwicklungsprozess ist grau hinterlegt. Die Inbetriebnahme und Betrieb sind weiß dargestellt. Zusätzlich sind den Phasen des Wasserfallmodells die möglichen Bindungszeitpunkte zugeordnet. Diese sind in kursiver Schrift dargestellt.

in den Entwicklungsprozess, welcher in der Abbildung grau hinterlegt ist, sowie in Inbetriebnahme und Betrieb. Der *Entwicklungsprozess* wird in die Phasen System-Analyse, Software-Spezifikation, Architekturentwurf, Feinentwurf u. Coding sowie Integration eingeteilt. Die *Inbetriebnahme und Betrieb* in die Phasen Installation und Betrieb u. Wartung. Zusätzlich sind den Phasen die möglichen Bindungszeitpunkte zugeordnet. Diese sind in der Abbildung kursiv dargestellt.

Mögliche Bindungszeitpunkte im Entwicklungsprozess sind: (vgl. [Wüb10])

- Produktspezifikation: Der frühstmögliche Zeitpunkt zu dem eine Bindung der Variabilitätspunkte vorgenommen werden kann, liegt in der Phase `Softwarespezifikation`, also direkt nach der Systemanalyse und noch deutlich vor dem Beginn der Codierung.

- Architekturentwurf: Die Bindung der Variabilität wird in der Phase `Architekturentwurf` vorgenommen. Die Bindung erfolgt somit noch während der Planung der Struktur und deutlich vor dem eigentlichen Feinentwurf der Architektur.
- Kompilierung: Während der Übersetzung des Quellcodes in Maschinencode werden die Variabilitätspunkte gebunden. Dies ist der spätest mögliche Zeitpunkt während des Entwicklungsprozesses.

Die Bindung der Variabilität in einer der Entwicklungsphasen bedeutet, dass der Abnehmer eine Software ohne Variabiltät erhält. Er bekommt eine Software, in welcher sämtliche Variabilitätspunkte bereits gelöst sind. Betrachten wir erneut die Software für Bordcomputer aus Kapitel 2.1.2. Die Entwickler setzen Variabilitätspunkte an Stellen, an denen die spezifische Elemente für die Fahrzeuge gebunden werden sollen, z.B. zur Steuerung eines Autoradios. Wenn die Auflösung der Variabilität im Entwicklungsprozess erfolgt, dann sind vor der Installation bereits alle Variabilitäten gebunden. Dies bedeutet, dass im Fall des Autoradios bereits ein konkretes Modell ausgewählt und gebunden worden ist. Dies hat den Vorteil, dass der Bordcomputer nur mit der nötigen Software (ohne variable Anteile) ausgestattet ist. Bei der Ausrüstung mit anderen Fahrzeugextras, zum Beispiel einem anderem Autoradio, muss der Bordcomputer mit einer anderen Softwareversion versehen werden.

Die Bindung der Variabilität bei der Inbetriebnahme oder während des Betriebs bedingt, dass die Bordcomputersoftware ohne vorherige Auflösung der Variabilitätspunkte installiert wird. Die Auflösung erfolgt dann automatisch gemäß der Systemumgebung oder durch manuelle Bedienung des Nutzers. Für die Bindung während der Inbetriebnahme oder im Betrieb gibt es ebenfalls unterschiedliche Zeitpunkte, an denen die Auflösung stattfinden kann: (vgl. [Wüb10])

- Installation: Die Bindung der Variabilitätspunkte erfolgt während der Phase der `Installation` auf dem jeweiligen Zielsystem. Der Nutzer bekommt die Software mit allen Varianten. Erst während der Installation werden die Variablitätspunkte durch einen „Setup-Prozess“ oder automatisch aufgrund der Hardware gebunden.
- Start der Software: Die Auflösung der Variabilität erfolgt beim Start der Software. Damit kann auf eine veränderte Systemumgebung, auf angepasste Konfigurationsdateien oder auf andere Abhängigkeiten dynamisch reagiert werden.
- Laufzeit: Die Variabilität wird während des Betriebs der Software gebunden. Die Anwendung kann damit dynamisch auf sich ändernde Bedingungen, wie beispielsweise Peripherieeingaben, Systemzeiten, Auslastung o.ä. reagieren.

Beim Bordcomputer wäre eine Bindung während der Installation oder beim Start der Software denkbar. In diesem Fall würde die Variabilität während der Installation aufgelöst und es werden nur die nötigen Komponenten installiert. Damit ensteht jedoch dasselbe Problem wie bei der Bindung während der Entwicklungsphasen. Bei Veränderung der Ausstattung des Fahrzeugs muss die Installation des Software neu ausgeführt werden.
Bei der Auflösung der Variabilität während des Starts muss die Bindung beim Anlassen des Fahrzeugs erfolgen. Dies hat den Nachteil, dass der Startvorgang des Bordcomputers unter

Umständen verlängert wird. Dafür würden aber neue Komponenten im Automobil direkt erkannt und übernommen.
Die Variabilität während der Laufzeit aufzulösen ist nachteilig. Bei jedem Menüaufruf muss geprüft werden, ob ein mögliches Artefakt gebunden werden kann/muss. Somit entsteht bei jedem Aufruf eine Verzögerung, obwohl zu erwarten ist, dass diverse Komponenten eines Automobils während des Betrieb des Bordcomputers nicht getauscht werden.

Wie man sehen konnte, ist beim Bordcomputer eine Bindung während der Entwicklungszeit, während der Installation oder auch während des Starts der Software denkbar. Im Weiteren werden wir feststellen, dass der Bindungszeitpunkt für die Fahrzeugsimulation innerhalb der Laufzeit liegen muss.

3 Konzeption eines Variabilitätsmanagements in der Fahrzeugsimulation

Die Konzeption des regelbasierten Variabilitätsmanagements in der Fahrzeugsimulation wird im Folgenden beispielhaft an Enhanced Vehicle Analysis (EVA) vorgenommen. EVA ist eine Entwicklung der Firma Bosch GmbH. Es ist ein Simulationsframework um Fahrzeugsimulationen durchzuführen. Typische Fragen, welche die Simulationen beantworten sollen, betreffen häufig den Energieverbrauch oder die Geschwindigkeit des Fahrzeugs. Beispielsweise wird simuliert, welche der Komponenten den höchsten Verbrauch haben oder die meiste Energie benötigen. Ebenso könnte simuliert werden, ob die Kühlung ausreichend ist, um eine Beschleunigung von null auf 100 km/h in 7 Sekunden zu erreichen. Das primäre Ziel der Entwicklung der Fahrzeugsimulation ist es, die Entwicklungskosten und -risiken zu senken.

In der Entwicklung von EVA wurden einzelne Bestandteile des Automobils, wie beispielsweise Motor, Kupplung etc. komponentenbasiert erstellt. Dadurch entstand ein flexibler und modularer Aufbau, mit dem es möglich ist, vereinzelte Elemente wieder zu verwenden oder auszutauschen. Durch die gegebene Flexibilität können mit Hilfe unterschiedlicher Zusammenstellungen von Komponenten die Auswirkungen in auf das Fahrzeug betrachtet werden. Häufig bestehen zwischen den einzelnen Elementen in der Fahrzeugsimulation auch Simulationsmodule genannt Abhängigkeiten. Eine Abhängigkeit stellt beispielsweise der Benzinmotor zum Tank dar. Eine Benzinmotor benötigt zwangsläufig einen Tank.

Beim Austausch eines Simulationsmoduls müssen Abhängigkeiten zu anderen Modulen beachtet werden. Damit eine korrekte Simulation durchgeführt werden kann, muss die Konsistenz des Gesamtmodells gewahrt werden, indem die Abhängigkeiten der Bestandteile untereinander erkannt und aufgelöst werden. Eine Simulation mit fehlerhaften Abhängigkeiten führt zum Absturz oder liefert falsche Simulationsergebnisse. Die manuelle Auflösung der Abhängigkeiten bei wachsenden Anzahl von Bestandteilen ist daher nur begrenzt möglich.

Durch die regelbasierte Entscheidungsfindung kann ein Variabiliabilitätsmanagement selbstständig Abhängigkeiten erkennen und diese auflösen. Besonders bei mehrfachen Transitivitäten ist es schwierig, dies durch manuelle Konfiguration in Dateien vorzunehmen. Durch eine Abstraktionsschicht zwischen der direkten Implementierung und der Nutzerkonfiguration, die durch das Variabilitätsmanagement eingeführt wird, benötigt der Nutzer nur ein reduziertes Verständnis über das Gesamtsystem und das notwendig Wissen über

Abhängigkeiten zwischen den Simulationsmodulen verringert sich.

Kapitel 3.1 beschreibt die Kontextanalyse und Domänenentwicklung nach FODA. Die Kontextanalyse dient dazu, einen Überblick über die Anforderungen einer Fahrzeugsimulation zu erlangen. In der Domänenenanalyse wird dann eine Trennung zwischen den gemeinsamen und variablen Simulationsmodulen vorgenommen. In Kapitel 3.2 wird erläutert, wie mit Hilfe von sogenannten Features eine Abstraktionsschicht zwischen der Nutzerkonfiguration und der systemnahen Simulationsmodulkonfiguration erreicht wird. Ebenso wird erläutert an welcher Stelle die Variabilitätspunkte integriert werden und welche Typen von Variabilität dadurch möglich sind. In Kapitel 3.3 werden letztendlich die Schwierigkeiten und Herausforderungen durch Variabilität in der Fahrzeugsimulation erläutert.

3.1 Kontextanalyse und Domänenmodellierung nach FODA

Die Kontextanalyse und Domänenmodellierung nach FODA sind Teilgebiete der Domänenanalyse. Die Domänenanalyse beginnt damit, die Variabilitäten eines Systems zu erkennen und diese gegenüber den gemeinsamen Elementen abzugrenzen. Es hat die systematische Untersuchung eines Bereichs auf Zusammenhänge und Abläufe zur Gewinnung der wiederverwendbaren Elemente zum Ziel.

1. Kontextanalyse:

Die *Kontextanalyse* soll die Problematik und Anforderungen der kompletten Domäne erfassen. Um ein generelles Verständnis der Domäne zu erlangen wird deren Umfang definiert sowie eine Eingliederung in deren Kontext durchgeführt.

Zusammenfassend besteht das Ziel von EVA darin, vergleichende Fahrzeugsimulationen durch den gezielten Austausch von Simulationsmodulen durchzuführen. Die bisherige Konfiguration, die sogenannte „direkte Konfiguration“, bildet eine direkte Abbildung zwischen den Simulationsmodulen und einer in XML Syntax erstellten Konfigurationsdatei. Um die Konfiguration zu erleichtern, soll die Konfigurationsdatei nicht mehr auf Basis der direkten Konfiguration, sondern abstrahiert in Form von Features dargestellt werden. Als Feature bezeichnet man wesentliche, meist für den Benutzer sichtbare funktionale oder nicht funktionale Leistungsmerkmale eines Produkts [BKPS04]. Dabei sollen Abhängigkeiten zwischen Simulationsmodulen automatisiert erkannt werden, um die Fehleranfälligkeit der Simulationsergebnisse zu verringern. EVA ist durch die komponentenbasierte Entwicklung bereits sehr flexibel, eine automatische Auflösung der Abhängigkeiten existiert bisher jedoch nicht. Die Konfiguration der Simulationsmodule wird durch eine zentrale Konfigurationsdatei vorgenommen, welche nach Einlesen nicht auf korrekte Abhängigkeiten validiert wird.

Das Simulationsframework EVA basiert auf Matlab für Windows. Matlab wird von

der Firma "The MathWorks"[1] entwickelt und dient primär zur Lösung von mathematischen Problemen. Visualisierung und Analyse von Daten sowie numerische Berechnungen lassen sich mit Matlab schnell und effizient lösen. Zusätzlich wird „Simulink“, eine Erweiterung für Matlab zur komfortableren Modellierung von physikalischen Systemen, verwendet.

Abbildung 3.1 zeigt die Einordnung von EVA in einem Strukturdiagramm. Das Strukturdiagramm ist Teil der Kontextanalyse von FODA, Kapitel 2.1.2. Es soll eine Einordnung und Abgrenzung zur benachbarten Domäne darstellen. Die unterste Ebene stellt das Betriebssystem `Microsoft Windows` dar. Auf diesem basiert `Matlab` und `Simulink`. Matlab und Simulink gibt es auch für andere Betriebssysteme wie beispielsweise MacOS. Diese sind jedoch im Kontext mit EVA nicht funktionsfähig. Die `Fahrzeugsimulation EVA` basiert auf `Matlab` und `Simulink` welche vor Inbetriebnahme von EVA installiert sein müssen.

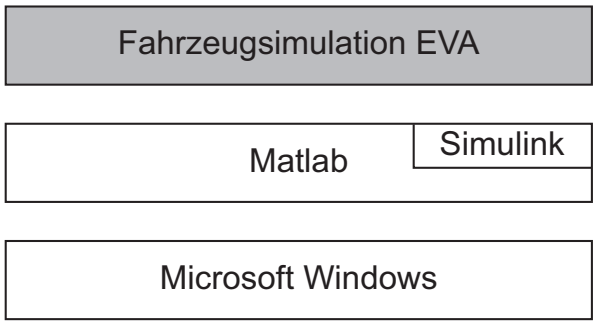

Abbildung 3.1: Das Strukturdiagramm zeigt die Einordnung der Fahrzeugsimulationssoftware EVA. EVA basiert auf Matlab und Simulink für Windows. Die unterste Ebene bildet das Betriebssystem Microsoft Windows.

Zusätzlich wird in Abbildung 3.2 der Datenfluss von EVA dargestellt. Dies wird Hilfe eines Kontextdiagramms erreicht. Das Kontextdiagramm stellt die Datenflüsse zwischen benachbarten Domänen z.B. EVA und Matlab dar.

Der Nutzer hat die Möglichkeit, die Konfiguration des Gesamtmoduls also unter anderem die Auswahl der Simulationsmodule in EVA vorzunehmen. Nach der Konfiguration startet der Nutzer die Simulation. EVA erstellt dann anhand der Konfiguration die zu berechnenden Funktionen und übergibt diese an Matlab. Matlab führt die Berechnung, eine numerische Integration, durch und gibt die Ergebnisse zurück an EVA. Das EVA Framework strukturiert die Ergebnisse und gibt sie formatiert an den Nutzer zurück.

2. Domänenmodellierung:

Die Domänenmodellierung ist der Hauptbestandteil der FODA-Analyse. In dieser

[1] http://www.mathworks.de

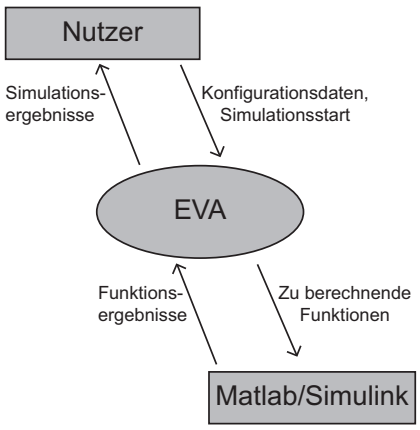

Abbildung 3.2: Das Kontextdiagramm stellt den Datenfluss von EVA dar. Der Nutzer konfiguriert EVA und startet die Simulation. EVA übergibt die zu berechnenden Funktionen an Matlab und Simulink. Diese berechnen die Funktionen und geben die Ergebnisse zurück an EVA. EVA strukturiert diese und gibt sie formatiert zurück an den Nutzer.

wird das Gesamtmodell in logische Komponenten eingeteilt um diese dann in „gemeinsame“ und „variable“ Elemente einzuordnen. Eine Trennung von Elementen muss so erfolgen, dass jedes Element eine eigene Zuständigkeit erhält und somit einen eigenen Aufgabenbereich abdeckt. Dies ist von zentraler Bedeutung, damit die einzelnen Elemente anwendungsneutral und damit in der Anwendung wiederwendbar sind.

Die Trennung des Gesamtmodells einer Fahrzeugsimulation nutzt das reale Automobil zur Definition der Elementgrenzen, da die Zuständigkeiten in materiellen Gegenständen bereits definiert sind. Ein Motor, eine Kupplung oder ein Rad stellen in der Realität jeweils eine eigene Komponente dar. Somit können diese auch in der Fahrzeugsimulation jeweils ein eigenes Simulationsmodul bilden.

Weiterhin muss entschieden werden, wie hoch die Granularität bei der Trennung der Zuständigkeiten sein muss. Ein Motor könnte beispielsweise in weitere Bestandteile getrennt werden. Die Granularität muss anhand der Aufgaben der jeweiligen Software, in diesem Fall der Fahrzeugsimulationssoftware entschieden werden. Sofern die Simulationssoftware beispielsweise in unterschiedlichen Simulationsdurchläufen denselben Motor mit unterschiedlichen Zylindern verwendet, ist eine weitere Auftrennung des Motors in mehrere Simulationsmodule nötig. Wird die Simulation immer mit denselben Zylindern ausgeführt, ist die Auftrennung unnötig.

Sobald die Auftrennung erfolgt ist, wird eine Einteilung in variable und gemeinsame Elemente vorgenommen. Ein Element wird als „gemeinsames Element“ bezeichnet, wenn es nicht optional ist und keine Alternative dafür existiert. Ein Element wird als variabel bezeichnet, wenn es optional ist oder mindestens eine Alternative dafür existiert. Variable

Elemente werden in der Fahrzeugsimulation und im Folgenden auch als *Simulationsmodul* bezeichnet.

Abbildung 3.3 zeigt beispielhaft ein Feature-Modell, welches einen möglichen Ausschnitt aus dem Gesamtmodell einer Fahrzeugsimulation darstellt. Der `Motor` stellt ein variables Element also ein Simulationsmodul dar. Es gibt in der Darstellung zwei Arten von Motoren: Einen `Benzinmotor` und einen `Elektromotor`. Diese können entweder `vorne` oder `hinten` am Fahrzeug angebracht sein. Die Motoren sind „Oder-Verknüpft", was bedeutet, dass entweder nur der Benzinmotor, nur der Elektromotor oder beide hinten am Fahrzeug angebracht sind. Ein Fahrzeug mit Elektro- und Benzinmotor wird als Hybridfahrzeug bezeichnet.

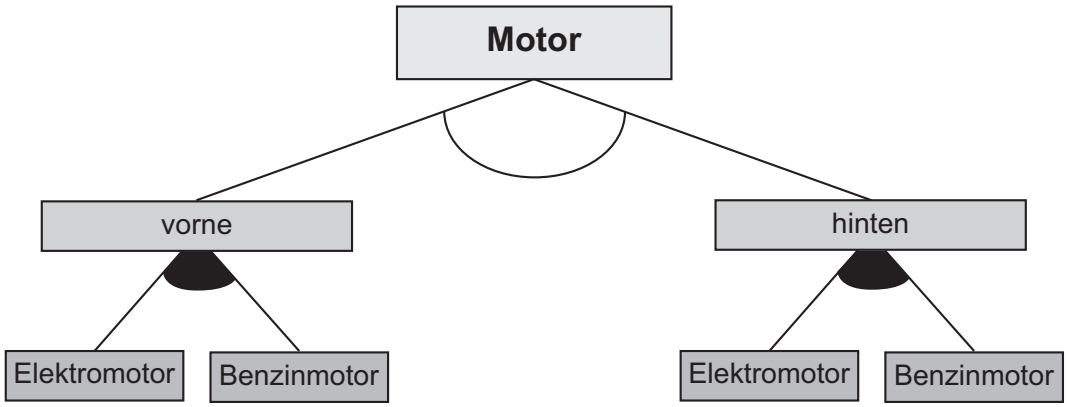

Abbildung 3.3: Die Abbildung zeigt beispielhaft ein Feature-Modell einer Motorkonfiguration. Die Motoren können vorne oder hinten am Fahrzeug angebracht sein. Es kann sowohl ein Elektromotor als auch ein Benzinmotor oder beides integriert werden.

Durch die Kontextanalyse und Domänenmodellierung sowie der Erstellung eines Feature Baums sind die Simulationsmodule sowie deren Alternativen bekannt und dargestellt. Im EVA Simulationsframework wird der Baum, welcher alle Simulationsmodule enthält, auch als „Family Baum“ bezeichnet.

3.2 Abstraktionsschicht durch Features und Variabilität

Durch die Integration von Features wird die direkte Konfiguration der Simulationsmodule unnötig. Features sind Abbildungen bestimmter Leistungsmerkmale, beispielsweise „Vorderradantrieb“. Ein Feature impliziert damit automatisch eine Menge von Simulationsmodulen. Im Fall des Features „Vorderradantrieb“ mindestens einen Motor, eine Achse und Räder. Durch eine Konfiguration mit Hilfe von Features entsteht eine theoretische Abstraktionsschicht zwischen dem Nutzer und der direkten Konfiguration. Die Zuordnung

der Simulationsmodule zu den Features, genauso wie die Abhängigkeiten der Simulationsmodule untereinander wird über Entscheidungsregeln abgebildet. Die Regeln werden durch die Variabilitätspunkte direkt mit dem Simulationsmodul verknüpft.

Abbildung 3.4 zeigt die Zuordnung zwischen der vom Nutzer definierten Feature Konfigurationsdatei und der Auswahl der Simulationsmodule.

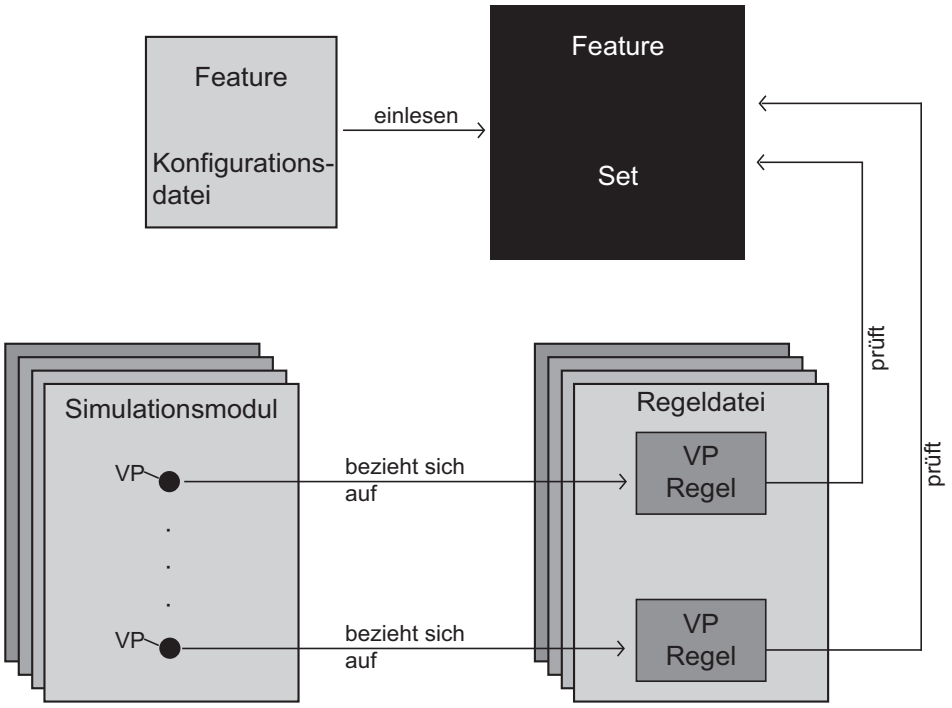

Abbildung 3.4: Variabilitätspunkte, in der Abbildung mit VP dargestellt, sind Simulationsmodulen zugeordnet. Einem Simulationsmodul können ein oder mehrere Variabilitätspunkte zugeordnet werden. Zu jedem Simulationsmodul gehört eine Regeldatei, in welcher die Regeln zu jeweiligem Variabilitätspunkt definiert sind. Dies ist mit der Kante „bezieht sich auf" dargestellt. Die Bedingungen der Regeldateien bestehen aus Bedingungen und Anweisungen. Die Bedingungen beziehen sich auf die vom Nutzer ausgewählten Features im Feature Set. Das Feature Set ist ein Datenspeicher, der bei Start der Simulation durch die Feature Konfigurationsdatei beschrieben wird.

Beim Start der Simulation werden die vom Nutzer gewählten Features aus einer `Feature Konfigurationsdatei` gelesen und in das `Feature Set`, welches auch in Abbildung 3.4 zu sehen ist, geschrieben. Das `Feature Set` stellt einen Datenspeicher dar, welcher die vom Nutzer ausgewählten Features repräsentiert. Jedem Simulationsmodul können ein oder mehrere Variabilitätspunkte zugeordnet werden. Diese sind in der Abbildung mit `VP` gekennzeichnet. Zu jedem Simulationsmodul existiert eine `Regeldatei`, welche die

Entscheidungsregeln zu dem jeweiligem Variabilitätspunkt enthält. Die Beziehung ist in der Abbildung durch die Kante `bezieht sich auf` dargestellt. Eine Regel besteht aus einer Bedingung und einer Anweisung, welche bei Erfüllung der Bedingung ausgeführt wird. Die Bedingungen beziehen sich auf die vom Nutzer ausgewählten Features im Feature Set. Möglicherweise könnte eine Regel, welche zu einem Simulationsmoduls eines Elektromotor gehört, prüfen, ob das Feature „Elektroautomobil“ ausgewählt ist. In diesem Fall würde sich das Simulationsmodul selbständig im Gesamtmodell aktivieren.

Im Gegensatz zu diversen anderen Arbeiten, wie [ML08] und [MALP09], erfolgt die Modellierung der Variabilität implizit. Eine implizierte Variabilität bedeutet, dass durch die Zuordnung von Simulationsmodulen zu den Regeldateien zwar eine Variabilität geschaffen wird, diese jedoch nicht durch fest definierte Stellen im Quelltext integriert ist. Beispielsweise wird keine Klassenmethode „set_variability()“ im Quelltext eingebunden, welche explizit an dieser Stelle einen variablen Inhalt bindet. Stattdessen entsteht die Variabilität dadurch, dass die Regeln, welche dem Simulationsmodul zugeordnet sind, über Parameter des Simulationsmodul entscheiden. Beispielsweise besitzt das Simulationsmodul den Parameter „Anzahl der Räder„. Eine Regel kann also anhand des Fahrzeugtyps Motorrad oder Automobil entscheiden, ob die Anzahl der Räder auf 2 bzw. 4 gesetzt wird.

Zusammenfassend erhält man durch die Einführung von Features und das automatisierte Auflösen von Variabilitäten eine Abstraktion der Konfiguration. Das benötigte Wissen um eine Konfiguration vornehmen zu können, wird für einen Nutzer dadurch verringert. Zusätzlich werden die Abhängigkeiten der Simulationsmodule mit Hilfe der Entscheidungsregeln automatisch gelöst, was Fehlkonfigurationen vorbeugt. Das Detailwissen wird also vom Nutzer weg in die Entscheidungsregeln verlagert.

Die durch Entscheidungsregeln eingeführte Variabilität betrifft vor allem die „Variabilität in Features“, um den Funktionsumfang dynamisch an die Anforderung des Nutzers anzupassen. Dadurch sind vergleichende Simulationen inklusive automatischer Auflösung der Abhängigkeiten möglich. Die EVA Fahrzeugsimulation wird bisher auf Kommandozeilenbasis bedient. Bei Integration einer grafischen Oberfläche, was vorgesehen ist, muss darauf geachtet werden, dass die Variabilität auch die Präsentationsebene betrifft und nicht isoliert betrachtet werden kann.

Des Weiteren kann mit Hilfe der Entscheidungsregeln die Variabilität im Datenformat oder -umfang verwendet werden, um Datenmanipulationen vorzunehmen. Die Variabilität im Datenformat ist in der Lage beispielsweise Umrechnungen von Zahlenwerten für die Weiterverarbeitung vornehmen. Ein mögliches Feature „mm_to_m“ könnte veranlassen, dass sämtliche Werte mit der Einheit Millimeter in Meter umgerechnet werden. Variabiltät in Systemschnittstellen kann zusätzlich interessant sein, um in zukünftigen Entwicklungen externe Anwendungen anzubinden. Durch Auswahl des Features „Externe Anwendung X“ könnte ein Simulationsmodul integriert werden, welches eine Anbindung zur „Anwendung X“ nutzt.

Durch die Variabilität in Software entsteht Flexibilität, welche gezielt für Anpassun-

gen und Erweiterungen genutzt werden kann, ohne dabei die Gesamtarchitektur ändern zu müssen. Variabiltiät und Flexibilität bedeuten aber auch eine Herausforderung.

3.3 Herausforderungen

Wie in Abbildung 1.1 bereits dargestellt, sinken mit zunehmenden Grad der Individualisierung die Entwicklungskosten des Variabilitätsmanagement im Verhältnis zu Standard bzw. Individualsoftware. Dabei ist zu beachten, dass ein hoher Grad an Individualisierung gleichermaßen Herausforderungen und Schwierigkeiten aufwirft.

Bereits das *Zerlegen des Gesamtsystems* in variable und gemeinsame Elemente ist entscheidend. Eine hohe und vor allem unnötige Variabilität erschwert das Verständnis und führt zu ineffizienter Nutzung des Systems. Elemente, welche eigentlich nicht variabel sind, jedoch als variabel angelegt werden, müssen künstlich bei der Bindung mit integriert werden und verursachen unnötigen Mehraufwand. Im Gegensatz dazu führt eine zu geringe Variabilität zur Ablehnung des Systems vom Nutzer, da es seinen Zweck nicht erfüllt.

Ebenso muss die *Grenze und Granularität eines Elements* korrekt festgelegt werden. Beispielsweise könnte der Benzinmotor eines Automobils als ein Simulationsmodul angesehen werden. Denkbar wäre jedoch auch, dass jeder einzelne Zylinder des Motors ein Simulationsmodul darstellt. Die Variabiltät bis ins letzte Detail ist nur dann sinnvoll, wenn diese auch genutzt wird. Sofern der Motor immer mit denselben Zylindern simuliert wird, wäre eine Auftrennung des Motors in weitere Bestandteile unnötig. Generell ist zu empfehlen, dass die Granularität zu Beginn eher zu grob als zu fein sein sollte. Eine weitere Auftrennung im Verlauf ist leichter möglich, als das Zusammenfügen mehrerer variabler Bestandteile.

Weitere Herausforderungen entstehen vor allem in *Testphasen* oder beim *Debugging*. Die Durchführung von Einzeltests sogenannten „Unit Tests“, bei denen kleinere Programmeinheiten getestet werden, sind für die Entwicklung nicht ausreichend. Durch die Variabilität und das automatisierte Einbinden von variablen Elementen gibt es eine hohe Anzahl an Kombinationen für Gesamtsimulationen. Bereits bei 3 optionalen Features werden 2^3=8 Kombinationen erzielt. In der Realität sind bei einer Fahrzeugsimulation mehrere tausend Kombinationen von Simulationsmodulen möglich. Ein „Unit Test“ kann für zwei ausgewählte Simulationsmodule erfolgreich verlaufen. In Kombination zeigen diese jedoch ein unerwartetes Verhalten, wie beispielsweise „Race Conditions“. Eine „Race Condition“ kann als eine Art Wettlauf zwischen Komponenten betrachtet werden. Greifen beispielsweise zwei Komponenten auf die gleichen Ressourcen lesend und schreibend im Wechsel zu, ist das Ergebnis, abhängig davon welche Komponente zu erst zugreift, nicht eindeutig [Tano3]. Dies führt dann zu unerwartetem Verhalten der Software. Die Auswirkungen aller Variabiltitäten auf das Gesamtsystem sind im Kontext schwierig zu prüfen. Dazu müssten alle Kombinationen des Systems durch „White- und Black-Box-Tests“ geprüft werden. Bei einem “Black-Box-Test“ wird im Gegensatz zu einem “White-Box-Test“ die

innere Struktur des Systems außer acht gelassen. Das Hauptaugenmerk richtet sich auf einen Funktionstest des Gesamtsystems bei einer Reihe von Konfigurationen. Ziel des "White-Box-Tests" ist es, die innere Funktionsweise, also den Quellcode zu testen. Dazu muss anhand des Quellcodes die Kombinationen der Konfiguration so gewählt werden, dass ein möglichst hoher Prozentsatz an Codeüberdeckung erreicht wird [LL07]. Dies ist bei den bereits erwähnten vielen Kombinationen des Systems eine schwierige und zeitaufwändige Aufgabe.

Doch nicht nur die Prüfung aller Kombinationen ist eine Herausforderung. Bereits das *Finden aller möglichen Kombinationen der Module* durch Regeln stellt weitere Schwierigkeiten dar. Durch das Einbinden eines einzelnen Simulationsmoduls werden Ketten von Abhängigkeiten automatisch aufgelöst. Sich einen Überblick zu verschaffen, welche Kombinationen durch die Regeln theoretisch möglich sind, wird bei wachsender Zahl von Simulationsmodulen immer komplexer.

Mit jedem variablen Element oder Simulationsmodul wächst gleichzeitig die Anzahl der Regeln, welche zur Entscheidungsfindung genutzt werden, um mögliche Abhängigkeiten aufzulösen. Auch die Entscheidungsregeln bilden großes Fehlerpotential, basieren auf menschlichem Wissen und können nicht vollständig automatisiert überprüft und auf Korrektheit geprüft werden. Eine Überprüfung kann nur stichprobenartig erfolgen. Beispielsweise könnte geprüft werden, ob ein Automobil mehr als einen Benzinmotor besitzt. Eine vollständige Prüfung würde bedeuten, dass der Sinn aller möglichen Kombinationen von Simulationsmodule geprüft werden müsste. Dies stellt jedoch implizites Wissen dar und kann nicht automatisiert entschieden werden.

Zusammenfassend kann man erkennen, dass bei mit wachsender Anzahl variabler Elemente die Anzahl der Modulkombinationen steigt. Bei einer geringeren Anzahl an variablen Elementen und Abhängigkeiten hingegen ist ein Variabilitätsmanagement unnötig. Abbildung 3.5 stellt dies grafisch dar.

Bei optionalen Features steigen die *Kombinationsmöglichkeiten* exponentiell an und bilden damit die Obergrenze. Jedes variable Element kann aktiviert oder deaktiviert werden. Die Grenze ist damit 2^n. Bei obligatorischen Features hingegen wächst die Anzahl der Möglichkeiten linear mit der Anzahl der variablen Elementen. Dies bildet die Untergrenze, da das Wachstum von n kleiner ist als das Wachstum von 2^n. Bei der Entwicklung sollte also darauf geachtet werden, die Variabilität minimal zu gestalten, um die Anzahl der Kombinationsmöglichkeiten zu senken. Dennoch muss ein Mindestmaß an Variabilität vorhanden sein, damit die Anwendung seinen Zweck erfüllt und beim Nutzer akzeptiert wird. Bei sehr geringer Variabilität ist das Variabilitätsmanagementsystem unnötig, da es die Inititalkosten der Architektur nicht rechtfertigt. Hier wäre auf eine manuelle Konfiguration zurückzugreifen. Dies ist in der Abbildung 3.5 mit einer Schranke dargestellt.

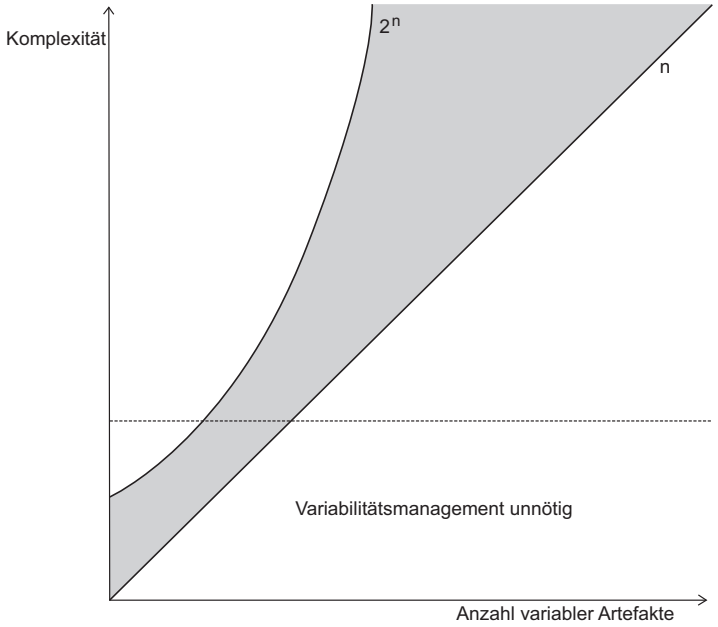

Abbildung 3.5: Die Abbildung zeigt das Wachstum optionalen Features (2^n) gegenüber obligatorischen Features (n). Obligatorische Features bilden die Untergrenze, optionale die Obergrenze da das Wachstum von n kleiner ist als das Wachstum von 2^n. Bei sehr geringer Variabilität ist das Variabilitätsmangement unnötig. In der Abbildung ist dies mit der Schranke dargestellt.
Dies ist in der Grafik mit der Schranke dargestellt.

4 Prototypische Implementierung eines regelbasierten Variabilitätsmanagements

Im Rahmen dieser Arbeit wurde in Kooperation mit der Firma Bosch GmbH ein prototypisches Variabilitätsmanagement implementiert, welches in dem bestehenden Simulationsframework EVA eingesetzt werden soll. Kapitel 4.1 beschreibt die Aktivität des Simulationsframeworks EVA ohne Variabilitätsmanagement.

In Kapitel 4.2 wird schließlich die veränderte Aktivität nach der Integration des Variabilitätsmanagements vorgestellt. Eine genaue Beschreibung der Architektur erfolgt in 4.4.2. Die Syntax und Semantik sowie die Vorgehensweise bei der Auswertung der Regellogik ist in Kapitel 4.4 beschrieben.

4.1 Aktivität der EVA Fahrzeugsimulation ohne Variabilitätsmanagement

Um die Funktionsweise des entwickelten Variabilitätsmanagements besser erläutern zu können, wird die bisherige Aktivität des Simulationsframeworks beschrieben. Abbildung 4.1 zeigt ein UML2-Aktivitätsdiagramm[1] mit dem wesentlichem Ablauf der Simulation.

Die `Parameter Konfigurationsdatei` ist in Extensible Markup Language[2] (XML) definiert und dient zur Parameterisierung der Simulationsmodule. Sie enthält Daten, die darüber entscheiden, welche Simulationsmodule in das Gesamtmodell integriert werden sollen und Parameter mit denen Einstellungen an den Simulationsmodulen vorgenommen werden können, zum Beispiel Gewicht des Fahrzeuges, Radgröße, etc. Die XML Elemente werden bijektiv auf die einzelnen Simulationsmodule abgebildet, weshalb dies als „direkte Konfiguration“ bezeichnet wird.

Nach Ausführung der Simulationssoftware, wird die Konfigurationsdatei eingelesen. In der Abbildung 4.1 wird dies durch die Aktion `Konfigurationsdatei einlesen` dargestellt. Die Konfigurationsdatei wird durch den Objektknoten `Parameter Konfigurationsdatei` beschrieben. Die Konfigurationsdatei wird manuell gepflegt und muss für jede Simulation neu erstellt bzw. angepasst werden. Die Komplexität und Schwierigkeit dieser direkten

[1] http://www.omg.org/spec/UML/
[2] http://www.w3.org/XML/

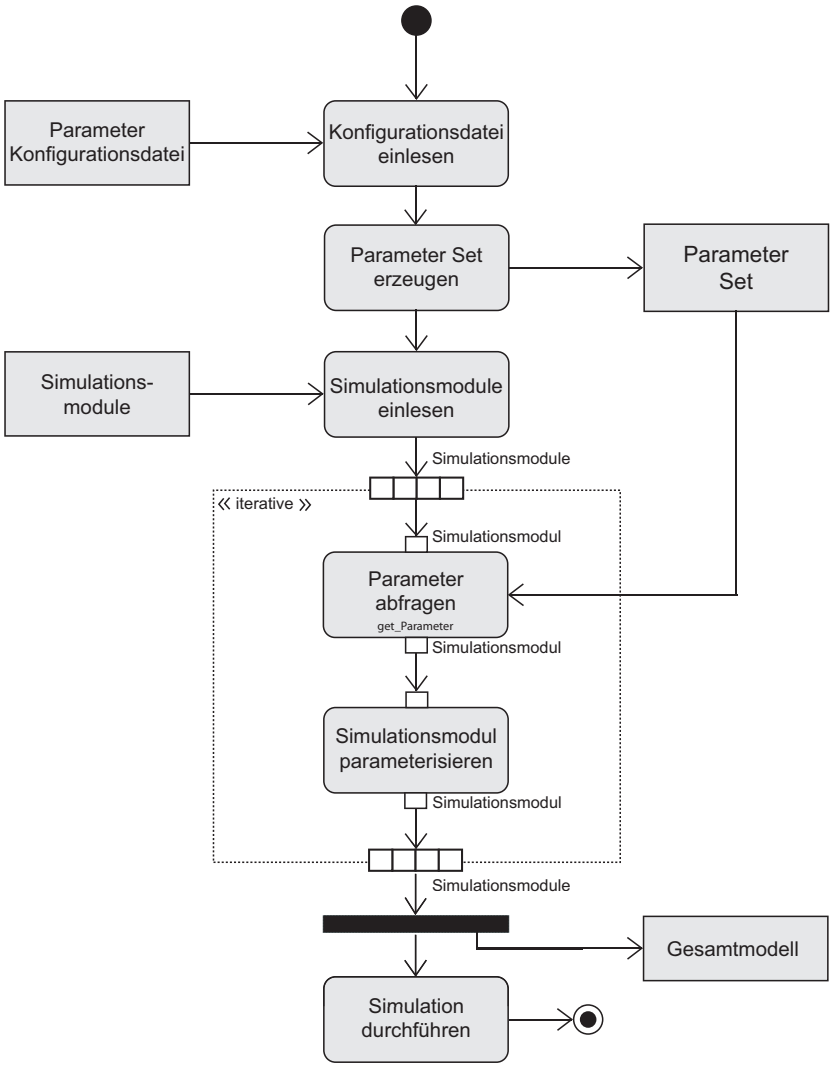

Abbildung 4.1: Die Abbildung zeigt ein UML2-Aktivitätsdiagramm des EVA Simulationsframeworks ohne Variabilitätsmanagement

Konfiguration besteht darin, dass bei Änderung von Teilen eines Automobils, mehrere Parameter geändert werden müssen. Dies betrifft beispielsweise der Austausch eines Benzinmotors gegen einen Elektromotor. Das erfordert Detailwissen über die vorhandenen Simulationsmodule und deren Abhängigkeiten um die Konfiguration gezielt anpassen zu können. Gerade der Austausch von Bestandteilen am Automobil ist von besonderem Interesse um Vergleichssimulationen durchführen zu können.

Nach dem Einlesen der Konfigurationsdatei wird die in der Abbildung 4.1 dargestellte Aktion `Parameter Set erzeugen` ausgeführt. Das Parameter Set kapselt die Daten und bietet Methoden zur Speicherung und Abfrage. Der Lesezugriff auf die Parameter

erfolgt über die Methode „get_Parameter(ParameterName)“. Im Folgenden wird der Übergabeparameter „ParameterName“ der Übersichtlichkeit wegen nicht mehr aufgeführt.

Nach dem Erzeugen des `Parameter Set` werden die Simulationsmodule eingelesen. In der Abbildung 4.1 ist diese Aktion mit `Simulationsmodule einlesen` und die Simulationsmodule durch das Objekt `Simulationsmodule` bezeichnet. Die Simulationsmodule sind Klassen, die in eigenen Datei im Dateisystem abgelegt sind. Einlesen bedeutet, dass die Dateien geöffnet und die Klassen der Simulationsmodule instanziiert werden. Die Instanzen und die Namen der Simulationsmodule sind in einer Liste gespeichert. Die Namen der Simulationsmodule entsprechen der Ordnerstruktur, z.B. wird Simulationsmodul des Elektromotors im Pfad „antrieb/motor/elektromotor.mat“ als „antrieb_motor_elektromotor"benannt.

Durch die Liste der Instanzen und Namen der Simulationsmodule wird iteriert, was in der Abbildung 4.1 mit dem iterativen Ausdehnungsbereich dargestellt ist. In diesem wird die Parameterisierung aller Simulationsmodule vorgenommen. Grundsätzlich sind alle Simulationsmodule Bestandteil des Gesamtmodells und werden während der Iteration durch die entsprechenden Parameter konfiguriert oder deaktiviert. Zusätzlich benötigen die meisten Simulationsmodule weitere Parameter wie z.B. den Raddurchmesser, Reibungskoeffizienten, usw.. Die Parameter wurden bereits im zweiten Schritt der Aktivität in das `Parameter Set` geschrieben. Aus diesem werden nun alle Parameter, die für das Simulationsmodul notwendig sind, abgerufen. In der Abbildung 4.1 wird diese Aktion mit `Parameter abfragen (get_Parameter())` bezeichnet. Sofern ein angefragter Parameter im Parameter Set nicht vorhanden ist (also in der Konfigurationsdatei nicht konfiguriert wurde), wird der Standardwert, der im Modul hinterlegt ist, übernommen. Sofern auch kein Standardwert hinterlegt ist, wird eine Fehlermeldung ausgegeben und terminiert. Nach Abfrage der Parameter werden diese dem Simulationsmodul zugewiesen. Dieser Vorgang nennt sich `Simulationsmodul parameterisieren` und ist in der Abbildung 4.1 mit der gleichnamigen Aktion bezeichnet. Der Zusammenhang zwischen den Simulationsmodulen und dem Parameter Set ist zusätzlich in Abbildung 4.2 dargestellt.

Das Schreiben der Daten der `Parameter Konfigurationsdatei` in das `Parameter Set` ist in der Abbildung 4.2 durch die beiden Rechtecke `Parameter Konfigurationsdatei` und `Parameter Set` sowie einer verbindenden Kante symbolisiert. Abhängig von der Anzahl der Parameter pro Simulationsmodul kann „get_Parameter()“ von jedem Simulationsmodul mehrfach, also entsprechender Anzahl der Parameter, aufgerufen werden. Die Abfrage der Parameter eines Simulationsmodules an das Parameter Set wird durch die Kante `get_Parameter()` beschreiben. Die Simulationsmodule 1 bis n zählen zu dem Gesamtmodell.

Nach Beendigung der Iteration durch die Liste der Instanzen und Namen der Simulationsmodule, wurde jedes Simulationsmodul parameterisiert, so dass dann ein Gesamtmodell entstanden ist. Dies ist in Abbildung 4.1 durch das Objekt `Gesamtmodell` dargestellt. Nach der Iteration durch die Liste der Simulationsmodule ist, bedingt durch die Parameterisierung der Simulationsmodule, ein Gesamtmodell erstellt. Anhand dessen kann

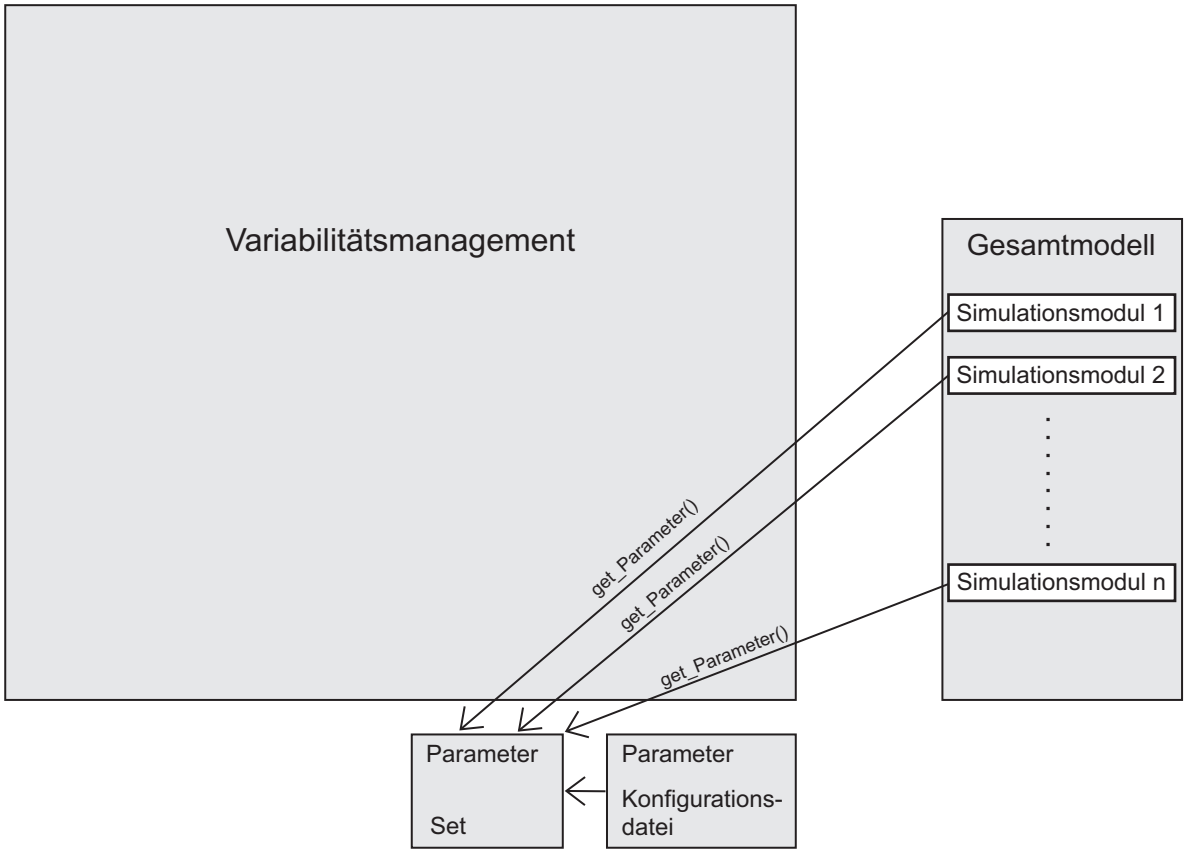

Abbildung 4.2: Die Abbildung stellt den Aufruf der Simulationsmodule am Parameter Set dar. Der Aufruf erfolgt über den Aufruf „get_Parameter()“ und ist durch eine Kante dargestellt. Das Parameter Set wird durch die Konfigurationsdatei erzeugt.

nun die Simulation des Gesamtmodells durchgeführt werden. In der Abbildung 4.1 ist dies durch die Aktion `Simulation durchführen` dargestellt.

4.2 Ablauf und Integration des Variabilitätsmanagements in EVA

Ziel dieser Arbeit ist es, ein Variabilitätsmanagementsystem zu entwickeln in dies in das EVA Frameworks zu integrieren. Bei der Planung und Realisierung des prototypischen Variabilitätsmanagements wurden diverse Aspekte berücksichtigt.
Einer dieser Aspekte ist eine *einfache Integrationsmöglichkeit*. Damit soll erreicht werden, ohne detailliertes Wissen über genaue Funktionalitäten einzelner Objekte und Methoden des vorhandenen Systems das Variabilitätsmanagement zu integrieren. Durch die Kapselung wird die Hauptaktivität des Variabilitätsmanagement isoliert. Zusätzlich wurde darauf

geachtet, dass nur eine geringe Anzahl von Schnittstellen zu externen Objekten des Frameworks notwendig sind. Die bereits existierenden Technologien wurden weitgehend übernommen, um so wenige wie möglich der bisherigen Objekte und Schnittstellen anpassen zu müssen. Damit fungiert der Kern des Variabilitätsmanagement als eine Art „Black Box“ System. Eine Black Box kann man als System betrachten von dem man seine Aufgabe kennt, dessen Funktionsweise aber nicht genau bekannt ist.

Ein weiterer Aspekt der Entwicklung des Variabilitätsmanagements ist es, eine einfache, verständliche und leistungsvollständige *Syntax und Semantik der Regeln* zu definieren. Mit der *Einfachheit* und *Verständlichkeit* der Regel steigt die Akzeptanz beim Nutzer und der Umgang mit der Software wird erleichtert. Die *Leistungsvollständigkeit* der Syntax und Semantik sorgt dafür, dass alle Leistungen die von einer Software gefordert tatsächlich auch erbracht werden. Dies impliziert, dass die Syntax so definiert ist, um alle möglichen Abhängigkeiten zwischen den Simulationsmodulen beschreiben zu können.
Die Leistungsvollständigkeit der Regeln erzeugt automatisch eine weitere Notwendigkeit, nämlich die *Erweiterbarkeit*. Die Regeln bestehen aus einer Bedingung und einer Anweisung. Die Anweisung wird ausgeführt, wenn die Bedingung zutrifft. Optional gibt es eine weitere Anweisung, welche ausgeführt wird, wenn die Bedingung nicht zutrifft. Weitere Informationen zur Syntax und Semantik der Regeln findet sich in Kapitel 4.4.1. Die Bedingung einer Entscheidungsregel besteht generell aus Operatoren und Operanden. Operatoren sind die Verknüpfungen einer Bedingung, beispielsweise AND, OR oder NOT. Mit diesen lassen sich die Operanden, in diesem Fall Klassenmethoden, zu logischen Formeln kombinieren. Die Klassenmethoden (Operanden) prüfen beispielsweise, ob ein Elektro- oder ein Benzinmotor existiert. Durch Weiterentwicklung des Simulationsframeworks kann es nötig werden die Bedingungsregeln zu erweitern. Daher muss ein Mechanismus geschaffen werden, mit dem es möglich ist, die Operanden zu erweitern. Ebenso sollten die Operatoren erweiterbar sein. Generell ist zwar mit den Operatoren AND, OR und NOT jeglicher weiterer Operator, z.B. XOR, modellierbar, jedoch nur mit erhöhtem Schreibaufwand. Dies widerspricht dem Ziel der einfachen und verständlichen Syntax der Regeln.

Ein weiteres Hauptaugenmerk der Entwicklung des Variabilitätsmanagements ist ausgerichtet auf die *Korrektheit* und *Zuverlässigkeit* des Systems. Mit der Einführung werden alle Abhängigkeiten zwischen den Simulationsmodulen regelbasiert aufgelöst, um den Nutzer bei der Konfiguration zu unterstützen. Dies erfordert eine korrekte und zuverlässige Auflösung der Abhängigkeiten und Interpretation der Regeln.

Abbildung 4.3 zeigt ein UML2-Aktivitätsdiagramm mit einem durch das Variabilitätsmanagement geänderten Ablauf der Simulation. Die Veränderungen, die sich zum Ablauf der direkten Konfiguration ergeben, werden durch einen dunklen Grauton hervorgehoben.

Die Konfiguration des veränderten Simulationsframework erfolgt featurebasiert. Wie bereits in Kapitel 3.2 erläutert, sind Features Leistungsmerkmale des Gesamtmodells. Ein Feature impliziert immer eine Menge von Simulationsmodulen. Die Zuordnung der Simulationsmodule zu den Features wird durch Entscheidungsregeln implementiert. Die

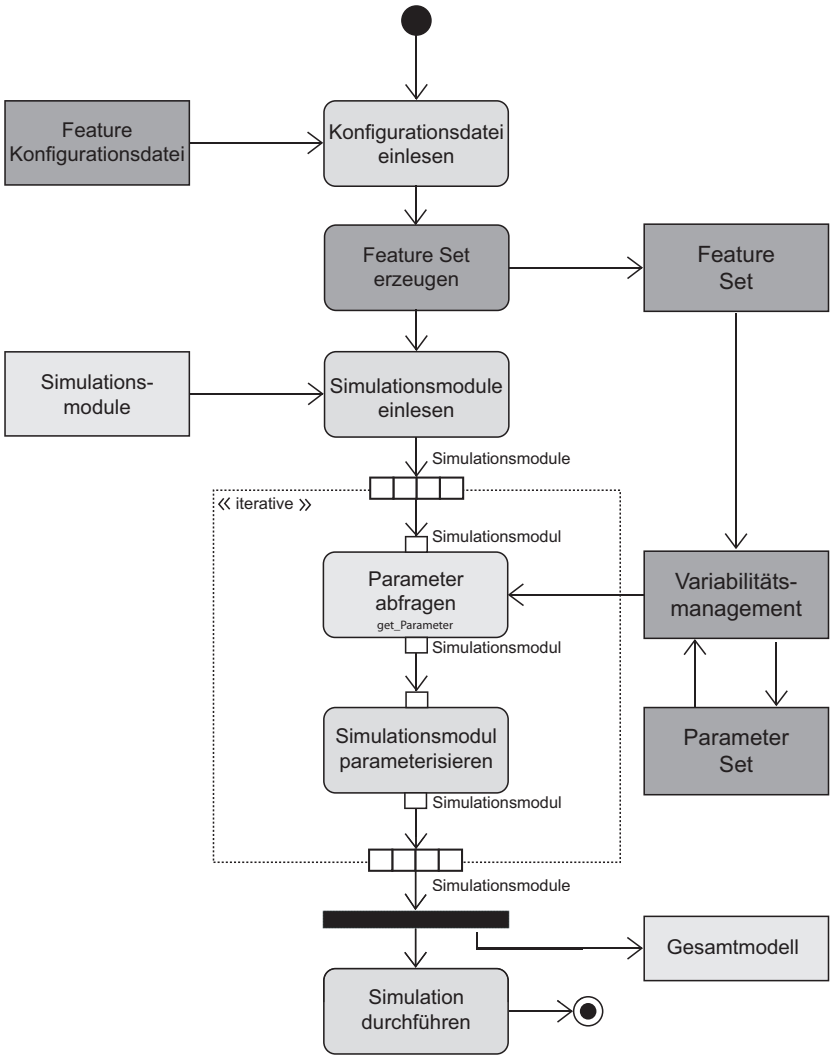

Abbildung 4.3: Die Abbildung zeigt ein UML2-Aktivitätsdiagramm des EVA Simulationsframeworks ohne Variabilitätsmanagement

Feature Datei dient zur Konfiguration der Features und damit zur indirekten Konfiguration den Simulationsmodule. Die `Feature Konfigurationsdatei` ist genauso wie die Parameter Konfigurationsdatei in XML definiert und als Objektknoten (`Feature Konfigurationsdatei`) links oben in der Abbildung dargestellt.

Nach Ausführung der Simulationssoftware, wird die Feature Konfigurationsdatei eingelesen. In der Abbildung 4.3 ist dies mit der Aktion `Konfigurationsdatei einlesen` bezeichnet. Diese Feature Konfigurationsdatei wird, genauso wie bei der direkten Konfiguration, manuell gepflegt und muss für jede Simulation neu erstellt bzw. angepasst werden. In der Abbildung 4.3 ist dieses Objekt mit `Feature Konfigurationsdatei` bezeichnet. Der

grundlegende Unterschied zwischen den beiden Konfigurationsarten ist, dass beim Austausch einzelner Features, wie zum Beispiel „Vorderradantrieb“ gegen „Hinterradantrieb“, keine zusätzlichen Parameter geändert werden müssen. Die Zusammenhänge zwischen den Simulationsmodule werden anhand der Features automatisch erkannt. Dies reduziert das benötigte Wissen über die vorhandenen Simulationsmodule und deren Abhängigkeiten. Es beugt Fehlkonfigurationen vor und erzeugt Akzeptanz beim Nutzer.

Nach dem Einlesen der Feature Konfigurationsdatei wird ein sogenanntes `Feature Set` erzeugt. In der Abbildung 4.3 ist dies mit einem Objektknoten `Feature Set` dargestellt. Das Feature Set ist eine Datenstruktur in der die Daten aus der ausgelesenen Feature Konfigurationsdatei gespeichert werden und kann als eine Art Liste vom Nutzer ausgewählten Features interpretiert werden. Das Feature Set stellt zwei Interaktionsmöglichkeiten zur Verfügung: Speicherung und Anfrage, ob ein Element existiert. Die Anfrage über die Existenz eines Features, erfolgt über die Methode „isset_Feature(FeatureName)“. Im Folgenden wird der Übergabeparameter „FeatureName“ der Übersichtlichkeit wegen nicht mehr erwähnt.

Nach Erzeugung des `Feature Set` werden die Simulationsmodule eingelesen. Dieser Vorgang hat sich verglichen mit der direkten Parameterisierung nicht geändert. Alle Klassen der jeweiligen Simulationsmodule werden eingelesen und die Instanzen und Namen werden in einer Liste gespeichert. Die Namen der Simulationsmodule entsprechen der Ordnerstruktur, z.B. wird das Simulationsmodul des Elektromotors im Pfad „simulationsmodule/antrieb/motor/elektromotor.mat“ den Namen „antrieb_motor_elektromotor“ bezeichnet. In der Abbildung 4.3 sind die Simulationsmodule mit dem Objektknoten `Simulationsmodule` und die Aktion zum Einlesen als `Simulationsmodule einlesen` dargestellt.

Ebenso wird, wie auch in der direkten Konfiguration, die Liste der Simulationsmodule iteriert, was in der Abbildung 4.1 mittels iterativen Ausdehnungsbereich dargestellt ist. Für jedes Simulationsmodul werden die zugehörigen Parameter abgefragt. Der grundlegende Unterschied in der Simulation mit dem Variabilitätsmanagement im Gegensatz zur Simulation ohne Variabilitätsmanagement liegt darin, dass bei der Iteration durch die Liste der Simulationsmodule die „get_Parameter()“-Anfrage nicht mehr an das Parameter Set, sondern an das Variabilitätsmanagement gestellt wird. In der direkten Konfiguration hat bislang „get_Parameter()“ die Daten direkt aus dem Parameter Set abgerufen. Das Parameter Set wurde aus der „direkten Konfigurationsdatei“ des Nutzers erstellt.

Das Variabilitätsmanagement entscheidet anhand von Regeln und Features, welche Werte aufgrund des „get_Parameter()“ Aufruf zurückgegeben werden. Die Abhängigkeiten zwischen den Regeln, Features und den Simulationsmodulen sind in Abbildung 4.4 beschrieben. Die Elemente, die im Vergleich der direkten Konfiguration dazugekommen sind, sind dunkelgrau hinterlegt. Die Parameterkonfigurationsdatei, die durch den Einsatz des Variabilitätsmanagement nicht mehr gebraucht wird, ist hellgrau ausgeführt.

In jedem Iterationsschritt durch die Liste wird für jedes Simulationsmodul die Para-

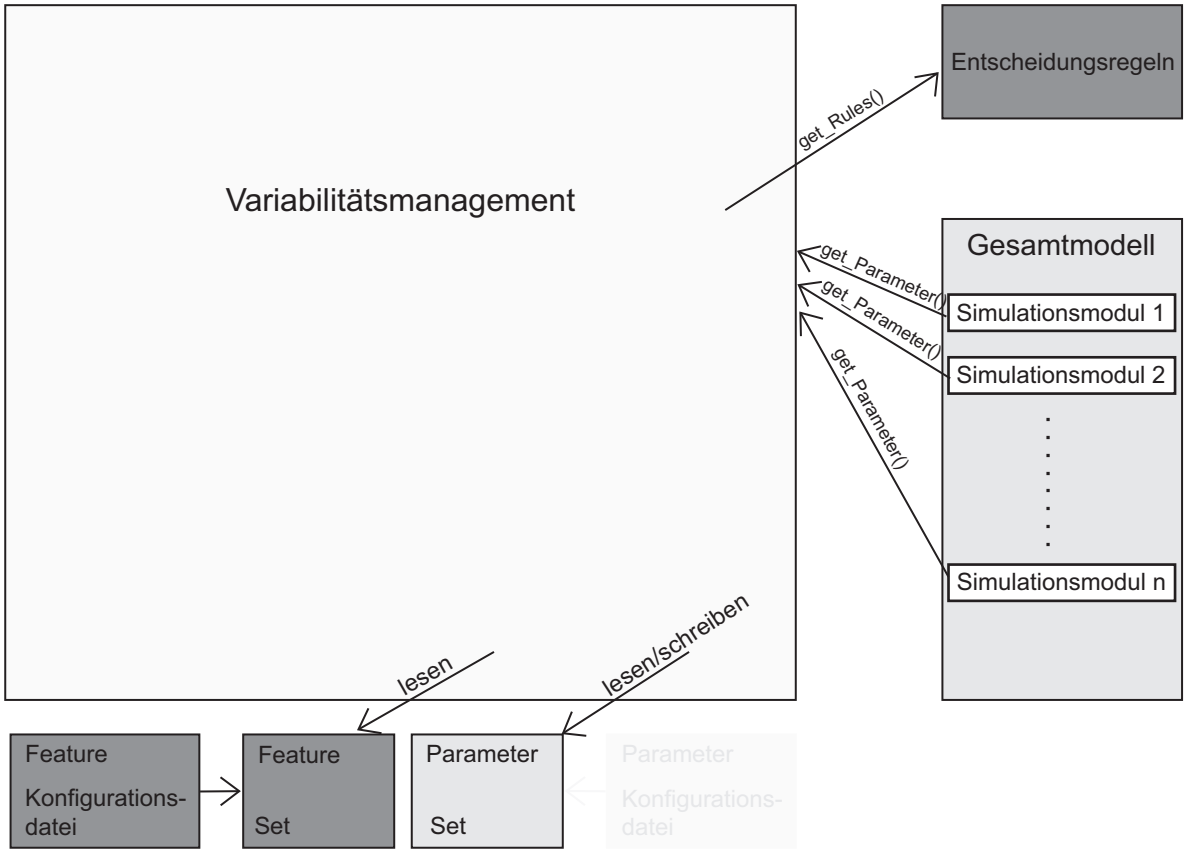

Abbildung 4.4: Die Abbildung stellt die Parameterabfrage der Simulationsmodule an das Variabilitätsmanagement dar. Der Aufruf ist durch die Kante „get_Parameter()“ dargestellt. Das Variabilitätsmanagement analysiert anhand der Regeln, welche Parameter gesetzt werden. Die Regeln, welche durch „get_Rules()“ abgefragt werden, beziehen sich zur Entscheidungsfindung auf Features, welche vom Feature Set abgerufen werden können. Dies ist durch die Kante „lesen“ dargestellt. Die Parameter werden im Parameter Set zwischengespeichert.

meter Anfrage „get_Parameter()“ gestartet. Der Aufruf erfolgt, wie auch in Abbildung 4.4 durch die Kante `get_Parameter()` dargestellt, an das Variabilitätsmanagement. Dieses ermittelt die zu jedem Simulationsmodul zugehörigen Regeln. Die Regeln sind in XML Syntax definiert und für jedes Simulationsmodul in einer eigenen Regeldatei gespeichert, die sich anhand des Namens ermitteln lässt. Eine direkte Integration der Regeln in den Simulationsmodulen wurde vermieden, damit keine Änderungen an vorhandenen Simulationsmodulen vorgenommen werden müssen. Der Name der Regeldatei mit zusätzlichem Präfix ist identisch mit dem Pfad zur jeweiligen Klassendatei des Simulationsmoduls. Beispielsweise trägt das Simulationsmodul des Elektromotors den Namen „antrieb_motor_elektromotor“ und der Pfad lautet „simulationsmodule/antrieb/motor/-elektromotor.mat“. Die dazugehörige Regeldatei ist unter einem anderen Präfix nämlich

„regeldatei/antrieb/motor/elektromotor.xml“, abgelegt. Durch den Aufruf „get_Rules()“ wird der Inhalt der entsprechenden Regeldatei zurückgegeben. In Abbildung 4.4 ist dies durch die Kante ,,`get_Rules()`‘‘ und dem Objekt `Entscheidungsregeln` dargestellt. Die Ablage in unterschiedlichen Dateien und darüber hinaus in unterschiedlichen Verzeichnissen birgt die Gefahr in sich, dass durch Namensänderung bei den Modulen und der Ordnerstruktur die eindeutige Beziehung zwischen Regeldatei und Simulationsmodul verloren geht.

Jede Regeldatei kann aus ein oder mehreren Regeln bestehen. Jede Regel definiert dabei einen Variabilitätspunkt. Die Regeln, die in den Regeldateien gespeichert sind, sind nach einem „if-then-else“ Schema aufgebaut, wie in Listing 4.1 dargestellt.

```
1 IF Bedingung THEN
2    Anweisung_1();
3 ELSE IF Bedingung_2 THEN  // ELSE IF kann beliebig oft vorkommen.
4    Anweisung_2();
5 ELSE
6    Anweisung_3();
7 END IF;
```

Listing 4.1: if-then-else Verzeweigung

Die bedingte Anweisung wird wie auch in diversen anderen Programmiersprachen ausgewertet. Wenn die Bedingung wahr ist, wird der anschließende Codeabschnitt ausgeführt. Trifft die Bedingung nicht zu, wird die nächste Bedingung (else if) ausgewertet und analog bearbeitet. Sollte keine Bedingung zutreffen, wird das „else“ Statement ausgeführt, sofern vorhanden. Die genaue XML-Syntax der Regeln wird in Kapitel 4.4 vorgestellt.

Die Bedingungen der Regeln enthalten Bezüge auf Features, die im Feature Set definiert sind. Die Anweisung zur jeweiligen Bedingung enthält Befehle zum Definieren von Parametern für die Simulationsmodule. Ein Anwendungsbeispiel ist der Fall, bei der die Regel prüft, ob es sich um ein Elektro-, ein Hybrid- oder ein Benzinfahrzeug handelt und bei einem Elektrofahrzeug einen Elektromotor und eine Batterie, bei einem Benzinfahrzeug einen Benzinmotor und einen Tank und bei einem Hybridfahrzeug sowohl einen Elektro- und Benzinmotor als auch eine Batterie und und einen Tank konfiguriert. Listing 4.2 stellt die Regeln für das vorgenannte Anwendungsbeispiel in einem „if-then-else“ Schema vor.

```
1  IF isset_Feature(Elektrofahrzeug) AND isset_Feature(Benzinfahrzeug) THEN
2     set_Parameter(Elektromotor,1);
3     set_Parameter(Batterie,1);
4     set_Parameter(Benzinmotor,1);
5     set_Parameter(Tank,1);
6  ELSE IF isset_Feature(Elektrofahrzeug) THEN
7     set_Parameter(Elektromotor,1);
8     set_Parameter(Batterie,1);
9  ELSE
10    set_Parameter(Benzinmotor,1);
11    set_Parameter(Tank,1);
```

```
12 END IF;
```

Listing 4.2: Mögliches „if-then-else“ Schema für Elektro- Hybrid oder Benzinfahrzeuge

Die Bedingung in der Zeile 1 prüft, ob im Feature Set ein „Benzin- und Elektrofahrzeug Feature“ konfiguriert sind. In Abbildung 4.4 ist dies durch die Kante ,,`get_Feature()`‘‘ und dem `Feature Set` dargestellt. Ist die Bedingung wahr, handelt es sich um ein Hybridfahrzeug (Elektro- und Benzinfahrzeug). In diesem Fall werden ein Elektromotor, eine Batterie, ein Benzinmotor und ein Tank benötigt. Diese werden im Parameter Set mit Namen und Wert gesetzt. Dies wird in den Zeilen 2 bis 5 des Listings 4.2 dargestellt. In Abbildung 4.4 ist dies rechts unten durch die Kante „lesen/schreiben“ dargestellt.
Wurde die erste Bedingung zu falsch ausgewertet, wird die zweite Bedingung, „isset_Feature(Elektrofahrzeug)“ geprüft. Ist das Ergebnis der zweiten Bedingung wahr, werden Elektromotor und Batterie mit Namen und Wert im Parameter Set gesetzt. Sollte es sich um kein Elektrofahrzeug handeln, wird die Bedingung in der Zeile 6 zu falsch ausgewertet und die Anweisungen in den Zeilen 10 und 11 werden ausgeführt.

Nach Abarbeitung der Regeldatei werden die angefragten Parameter aus dem Parameter Set an das jeweilige Simulationsmodul zurückgegeben. Sind die Parameter im Parameter Set nicht vorhanden, wird der jeweilige Standardwert, der im Simulationsmodul enthalten ist, gesetzt. Sofern kein Standardwert hinterlegt ist, wird eine Warnung ausgegeben. Sobald die Parameter an das Simulationsmodul zurückgegeben wurden, werden diese im Simulationsmodul gesetzt. Dies ist im Aktivitätsdiagramm in Abbildung 4.3 durch die Aktion `Simulationsmodul parameterisieren` dargestellt.

Die Vorgänge `Parameter abfragen` und `Simulationsmodul parameterisieren` werden während der Iteration für alle Simulationsmodule durchgeführt. Daraus entsteht das Gesamtmodell, das in Abbildung 4.3 durch den Objektknoten `Gesamtmodell` dargestellt ist. Dieses kann simuliert werden und ist in der Abbildung 4.3 durch die Aktivität `Simulation durchführen` dargestellt.

4.3 Architektur und Realisierung des Variabilitätsmanagements

In diesem Kapitel beschäftigten wir uns mit der Architektur des Variabilitätsmanagements und die Einordnung in das Umfeld. Die Realisierung des Variabilitätsmangements erfolgt objektorientiert. Um die Erweiterbarkeit und einfache Integration in ein bestehendes Simulationsframework zu ermöglichen, wurde komponentenbasiert entwickelt und darauf geachtet, dass vor allem eine lose Kopplung existiert. Abbildung 4.5 zeigt ein UML-Klassendiagramm mit der Architektur des Variabilitätsmanagements und das Umfeld, das in Verbindung mit Abbildung 4.4 bereits behandelt wurde.

Das Variabilitätsmanagement wird initial durch den „get_Parameter()„-Aufruf ausgeführt. Der Aufruf erfolgt an die Klasse `VariabilityManagement`, wie dies in der Abbildung 4.5 durch die eingehenden Kanten symbolisiert ist. Die Klasse `VariabilityManagement`

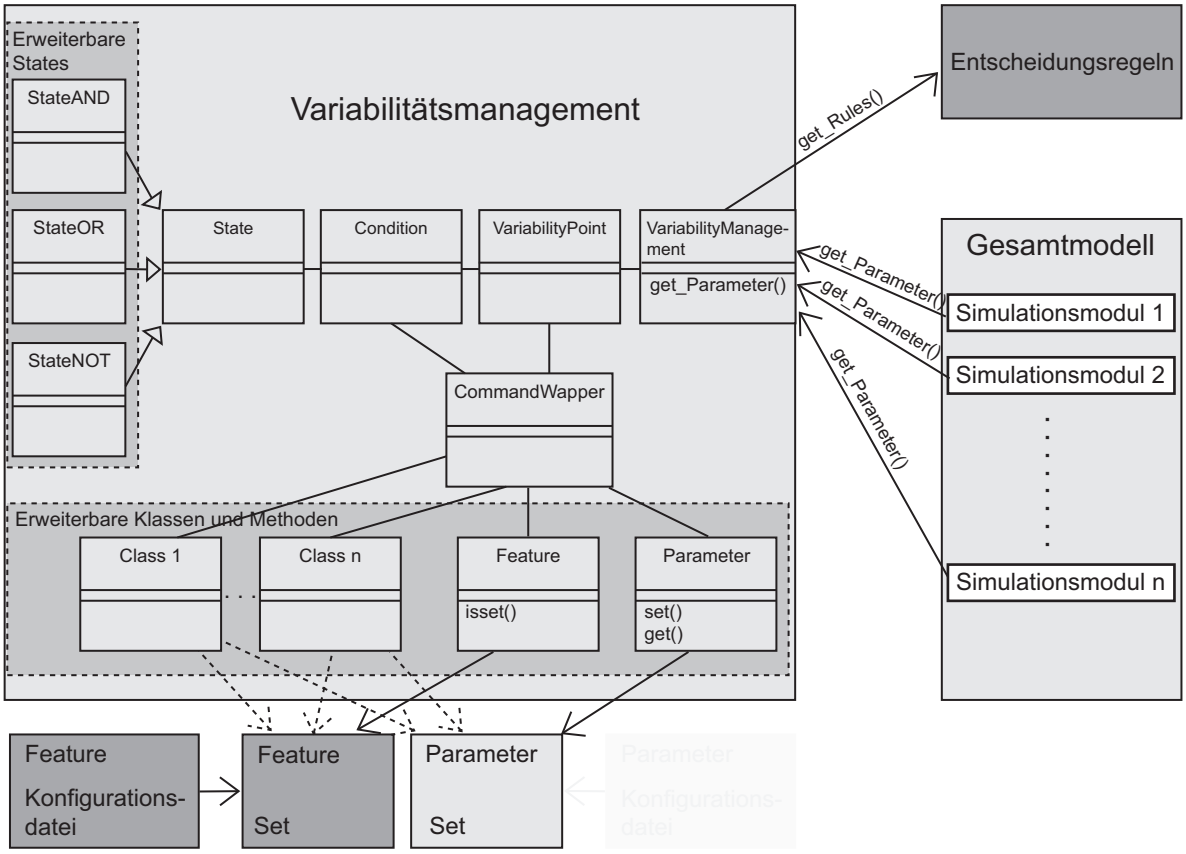

Abbildung 4.5: Architektur des Variabilitätsmanagement

öffnet die dazu gehörige Regeldatei und richtet sich dabei nach dem Namen des Simulationsmoduls. Die Regeldatei ist in XML definiert und kann einen oder mehrere Variabilitätspunkte, also Regeln enthalten. der in Matlab integrierte XML Parser teilt die Regeldatei in einzelne Regeln auf. In einer Schleife wird für jede Regel eine eigene Instanz der Klasse `VariabilityPoint` erzeugt. Die Regel wird jeweils als Parameter übergeben. In Abbildung 4.5 ist dies durch die Klasse `VariabilityPoint` sowie dessen Kante zur Klasse `VariabilityManagement` aufgezeigt.
`VariabilityPoint` steuert die Auswertung der jeweiligen Regel. Dazu wird die Regel durchlaufen, die Bedingungen zur Auswertung an die Klasse `Condition` und die Anweisungen an die Klasse `CommandWrapper` delegiert. Sofern eine Bedingung den Wahrheitswert „true“ zurückgibt, wird der dazu gehörige Anweisungsblock ausgeführt und es werden keine weiteren Bedingungen mehr geprüft. Wenn alle Bedingung zu falsch ausgewertet werden, wird die Anweisung, die auf „else“ folgt, ausgeführt. Grundsätzlich wäre alle Anweisungen von der Klasse `CommandWrapper` ausgeführt. Eine Anweisung entspricht immer einer Klasse, einer zugehörigen Methode sowie weiteren Übergabeparametern. Die genaue Syntax und deren Abbildung auf Klasse, Methode und Parameter wird in Kapitel 4.4 beschrieben.

Die Klasse `CommandWrapper` bildet den zentralen Zugriffspunkt, für den in der Abbildung 4.5 dunkelgrau unterlegten Bereich „Erweiterbare Klassen und Methoden“. In diesem Bereich befindet sich die Klasse `Parameter`, welche mit der Methode „set()“ die Möglichkeit bietet Parameter im `Parameter Set` zu definieren und die Klasse `Feature`, welches mit der Methode „isset(Feature)“ die Möglichkeit bietet abzufragen, ob ein bestimmtes Feature konfiguriert ist.
Die in der Abbildung 4.5 dargestellten Klassen `Class 1` bis `Class n` stehen symbolisch für erweiterbare Klassen. Diese können sowohl auf das „Feature oder Parameter Set“ als auch auf andere Komponenten des Simulationsframeworks zurückgreifen.

Um eine Anweisung auszuführen, werden dem `CommandWrapper` eine Klassenname, ein Methode und Parameter übergeben. Der `CommandWrapper` überprüft, ob die Klasse sowie Methode existiert. Sofern diese vorhanden ist, wird sie instanziiert und die Methode mit den entsprechenden übergebenen Parametern ausgeführt. Den Rückgabewert reicht der `CommandWrapper` zurück an die Klasse, die ihn instanziiert hat. Sofern die Methode nicht existiert, wird eine Fehlermeldung ausgegeben. Meistens bezieht sich eine Anweisung auf das Setzen oder Manipulieren eines Parameters im `Parameter Set`.

Die Überprüfung, ob eine Bedingung der Regel zutrifft, wird von der Klasse `VariabilityPoint` an die Klasse `Condition` delegiert, wie dies in Abbildung 4.5 dargestellt ist. Um die Bedingung korrekt auswerten zu können bedient sich die Klasse `Condition` der Aussagenlogik. Die Operatoren und Operanden sind ebenso in Klassen ausgelagert, um eine Erweiterbarkeit zu ermöglichen. Der dunkelgrau hinterlegte Bereich „Erweiterbare Operatoren“ in Abbildung 4.5 stellt eine Sammlung der Klassen dar, welche die boolschen Operatoren implementieren. Beispielhaft sind in der Abbildung „and“, „or“ und „not“ aufgeführt. Über das Elternobjekt „State“ der spezifizierten Operatoren wird der Klasse `Condition` der Zugriff auf die verfügbaren Operatoren gewährt. Jeder Operator wird als eigene Klasse modelliert. Dies ermöglicht die Erweiterbarkeit ohne Detailwissen über das gesamte Variabilitätsmanagement.

Die Operanden der Bedingung wurden, genauso wie die Anweisungen und Operatoren, in eigene Klassen und Methoden ausgelagert. Der Zugriff darauf erfolgt wieder zentral über die Klasse `CommandWrapper`, der als Übergabewert einen Klassen- und Methodennamen und Parameter erfordert. Um einen Operanden auszuwerten, wird wie bei einer Anweisung eine entsprechende Klassemethode ausgeführt und der Rückgabewert zurück an die Klasse `Condition` gegeben. Anhand der ausgewerteten Operanden, kann die Klasse `Condition` die Bedingung mit Hilfe der Operatoren vollständig auswerten. Das Ergebnis der Auswertung wird dann an die Klasse `VariablityPoint` zurückgegeben. Falls der Rückgabewert „false“ ist, wird die nächste Bedingung von `Variablity Point` an Condition delegiert. Ist die Bedingung wahr, werden die zugehörigen Anweisungen an `CommandWrapper` delegiert.

4.4 Bedingungsregeln

Die Abstraktionsschicht zwischen der direkten Konfiguration und der Feature Konfiguration entsteht durch die eingeführten Features und die Bedingungsregeln. Die Bedingungsregeln, die sich meist auf Features beziehen, ermöglichen ein Setzen und Manipulieren von Parametern im Parameter Set.

Die Syntax ist in XML definiert und wird in Kapitel 4.4.1 detailliert beschrieben. Die Auswertung und deren Realisierung, also die Auflösung der Bedingung in einen Wahrheitswert, wird in Kapitel 4.4.2 erläutert.

4.4.1 Syntax und Semantik

Die Syntax der Bedingungsregeln wurde mit der Extensible Markup Language (XML) definiert. XML ist eine Metasprache zum Definieren eigener Markup-Sprachen. In dieser kann die logische Bedeutung von Daten beschrieben werden, um diese in Software zu verarbeiten oder zwischen Applikationen auszutauschen [Geno3].

Elemente in XML werden mit Hilfe von sogenannten Start- und Endtags eingeschlossen, z.B. <automobil>Audi A3</automobil>. Durch Verschachtelung entstehen Hierarchien, die Zusammengehörigkeit beschreiben [Geno3]. Zusätzlich können sogenannte Attribute beim jeweiligen Element hinterlegt werden. Attribute definieren das Element genauer, bspw. <automobil farbe="schwarz">Audi A3</automobil>.

In einer vorgegebenen XML-Struktur liegt der Vorteil,, dass sie den Nutzer bei den Möglichkeiten zur Erstellung der Regeln einschränkt. XML ist zudem plattformunabhängig und die XML-Dokumente können auch anderweitig verarbeitet werden.
Nachteilig wäre, wenn man dem Nutzer eine abstrakte Klasse zur Verfügung stellen würde, welche er ableiten kann. Er hätte dann eine Vielzahl an Möglichkeiten in die Applikation einzugreifen. Außerdem müsste er die Entwicklungssprache Matlab beherrschen. Ein weiterer Nachteil ist, dass eine syntaktische Prüfung nicht möglich wäre und dass die erweiterte Klasse bei einem Syntaxfehler die Simulationssoftware zum Absturz bringt.

Die Regeldatei stellt, wie bereits in Kapitel 3.2 ausgeführt, die Verbindung zwischen den Variabilitätspunkten und den Features dar. Dabei prüft die Bedingung einer Regel ob bestimmte Features im Feature Set existieren. Beim Zutreffen der Bedingung wird dann eine Anweisung ausgeführt, die Parameter im Parameter Set setzt. Die Parameter werden dann zur Parameterisierung der Simulationsmodule zu verwendet.
Eine beispielhafte Regeldatei für Variabilität ist in Listing 4.3 dargestellt, das zugehörige XML Schema[3], in Listing 4.4. Dem XML Schema lässt sich der genaue Aufbau des XMLs

[3]http://www.edition-w3c.de/TR/2001/REC-xmlschema-0-20010502/

entnehmen. Auf die Syntax von einem XML Schema wird an dieser Stelle nicht eingegangen.

Listing 4.3: Regeldatei - Beispiel 1

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<variability>
    <vp>
        <if>
            <or>
                <key type="feature" operator="isset">FeatureName</key>
                <and>
                    <key type="feature" operator="isset">FeatureNameA</key>
                    <key type="feature" operator="isset">FeatureNameB</key>
                </and>
            </or>
        </if>
        <then>
            <key type="parameter" operator="set" value="5">ParName</key>
            <key type="parameter" operator="set" value="6">ParName2</key>
            <key type="parameter" operator="set" value="7">ParName3</key>
        </then>
        <else>
            <key type="parameter" operator="set" value="8">ParName3</key>
        </else>
    </vp>
</variability>
```

Listing 4.4: XML Schema für Regeldatei - Beispiel 1

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="variability" type="VariabilityPoint" />

    <xs:complexType name="VariabilityPoint">
        <xs:sequence>
            <xs:element name="vp" minOccurs="1" maxOccurs="unbounded">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="if" type="Condition" />
                    <xs:element name="then" type="Command" />
                    <xs:sequence minOccurs="0" maxOccurs="unbounded">
                        <xs:element name="elseif" type="Condition" minOccurs="1" maxOccurs="1"
                            />
                        <xs:element name="then" type="Command" minOccurs="1" maxOccurs="1" />
                    </xs:sequence>
                    <xs:element name="else" type="Command" minOccurs="0" maxOccurs="1" />
                </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>

     <xs:complexType name="Condition">
        <xs:choice>
            <xs:element name="and" type="OR_AND" />
            <xs:element name="or" type="OR_AND"/>
```

```
27            <xs:element name="not" type="NOT" />
28         </xs:choice>
29      </xs:complexType>
30
31       <xs:complexType name="OR_AND">
32           <xs:sequence>
33              <xs:choice>
34                  <xs:element name="key" type="Key "/>
35                  <xs:element name="and" type="OR_AND" />
36                  <xs:element name="or" type="OR_AND" />
37                  <xs:element name="not" type="NOT" />
38              </xs:choice>
39              <xs:choice>
40                  <xs:element name="key" type="Key" minOccurs="1" maxOccurs="unbounded"/>
41                  <xs:element name="and" type="OR_AND" minOccurs="1" maxOccurs="unbounded" />
42                  <xs:element name="or" type="OR_AND" minOccurs="1" maxOccurs="unbounded"/>
43                  <xs:element name="not" type="NOT" minOccurs="1" maxOccurs="unbounded" />
44              </xs:choice>
45          </xs:sequence>
46      </xs:complexType>
47
48       <xs:complexType name="NOT">
49           <xs:choice>
50              <xs:element name="key" type="Key" />
51              <xs:element name="and" type="OR_AND" />
52              <xs:element name="or" type="OR_AND"/>
53              <xs:element name="not" type="NOT" />
54          </xs:choice>
55      </xs:complexType>
56
57      <xs:complexType name="Command">
58          <xs:sequence>
59          <xs:element name="key" type="Key" minOccurs="1" maxOccurs="unbounded"/>
60          </xs:sequence>
61      </xs:complexType>
62
63      <xs:complexType name="Key">
64          <xs:simpleContent>
65              <xs:extension base="xs:string">
66                  <xs:attribute name="type" type="xs:string"/>
67                  <xs:attribute name="operator" type="xs:string"/>
68                  <xs:attribute name="value" type="xs:integer"/>
69              </xs:extension>
70          </xs:simpleContent>
71      </xs:complexType>
72
73  </xs:schema>
```

Das Listing 4.3 beginnt in der Zeile 1 mit der Angabe einer XML-Deklaration. Diese gibt an, dass es sich um ein XML-Dokument handelt. Hier können auch diverse Parameter, wie die XML-Version, Kodierungsdeklarationen oder andere Informationen in Attributen hinterlegt

werden. Die genaue Bedeutung und Möglichkeiten sind der W3C Recommendation[4] zu entnehmen.

Die Zeilen 2 und 22 des Listings 4.3 definieren den Start- bzw. End-Tag der Regeldatei. In diesem Beispiel mit **<variability>...</variability>** bezeichnet. Dieses wird auch als Wurzelelement oder Root-Element bezeichnet und stellt in jeder Regeldatei das äußerste Element dar. Entsprechend ist dies im XML Schema, Listing 4.4 in Zeile 3 dargestellt.

In den Zeilen 3 bis 21 des Listings 4.3 wird ein Variabilitätspunkt beschrieben. Diese beginnen mit **<vp>** und enden mit **</vp>**. Die Anzahl der Variabilitätspunkte sind unbegrenzt und können mehrfach hintereinander definiert werden. Der Variabilitätspunkt besteht aus einer oder mehreren Bedingungen. Die Bedingungen werden, wie bereits in Listing 4.3 dargestellt, mit einer „if-then-else“ Struktur definiert, diese in den Zeilen 4 bis 12 durch das **<if>-Element** formuliert. In den Zeilen 13 bis 17 (**<then>-Element**) stehen die Anweisungen, die ausgeführt werden, wenn die Bedingung zutrifft. In den Zeilen 18-20 (**<else>-Element**) stehen die Befehle, die ausgeführt werden wenn die Bedingung zum Wahrheitswerte „false“ ausgewertet wird. Zu jeder Bedingung muss es Anweisungen geben, die direkt auf die Bedingungen folgen. Hingegen ist ein **<else>...</else>** Block optional.

Im XML Schema ist der Variablitätspunkt durch den „complexType“ Variability Point in den Zeilen 5 bis 21 in Listing 4.3 dargestellt. Der Variabilitätspunkt mit Namen „vp“ muss im XML-Dokument mindestens einmal vorkommen. Dies ist in der Zeile 7 abgebildet. Jeder Variabilitätspunkt besteht aus den Elementen „if“, „then“, dem optionalen Tupel aus „elseif“ und „then“ sowie dem Element „else“. Die Elemente „if“ in Zeile 10 bzw. „elseif“ in Zeile 13 sind vom Typ „Condition“ in den Zeilen 23 bis 29 des Listing 4.4 definiert. Dieser bietet eine Auswahl (xs:choice) aus den Elementen „and“, „or“ und „not“. Bei Auswahl des Elements „and“ bzw. „or“ müssen immer mindestens zwei Kindelemente gewählt werden, nämlich wahlweise zwei oder mehrere Operatoren, einen Operator sowie mindestens einen Operanden oder alternativ einen Operanden und mindestens einen Operatoren. Dies ist im Listing 4.4 in den Zeilen 33-44 dargestellt. Bei Auswahl des Elements „not“ kann als Kindelement optional ein Operator oder ein Operand gewählt werden, da not ein unnärer Operator ist. Das ist im Listing 4.4 in den Zeilen 48 bis 55 dargestellt. Die Auswahl eines der Operatoren hat zur Folge, dass eine Rekursion so lange erfolgt, bis letztendlich nur noch Operanden gewählt wurden. Die Operanden haben den Elementnamen “key“ und sind vom gleichnamigen Typ „Key“. Elemente vom Typ „key“ (Zeilen 63-71, Listing 4.4) besitzen zusätzlich die Attribute „type“, „operator“ und „value“.

Die Elemente „then“ und „else“ des Variabilitätspunktes sind vom Typ „Command“. Diese haben als Kindelemente, genauso wie die Operanden, mindestens ein „key“ Element vom Typ „Key“. Dies ist in Zeile 57-61 in Listing 4.4 dargestellt.

<key>-Elemente bilden den Zugriff auf Klassen und Methoden ab, welche den „Er-

[4]http://www.edition-w3.de/TR/2000/REC-xml-20001006/

weiterbaren Klassen und Methoden“ durch den CommandWrapper (siehe Abbildung 4.5) aufgerufen werden können. **<key>-Elemente** findet man im Listing 4.3 in den Zeilen 6, 8, 9, 14, 15, 16 und 19. Das Attribut „type“ definiert den Namen der Klasse und das Attribut „operator“ die Methode. Der Inhalt des **<key>-Elements** gibt den Übergabeparameter an, mit dem die Klasse arbeiten soll. Abbildung 4.6 stellt den Zusammenhang zwischen dem **<key>-Element** und einer Klasse anhand eine Beispiels schaubildlich dar.

Das Beispiel bezieht sich auf die Zeile 6 des Listings 4.3. Im konkreten Fall wird die

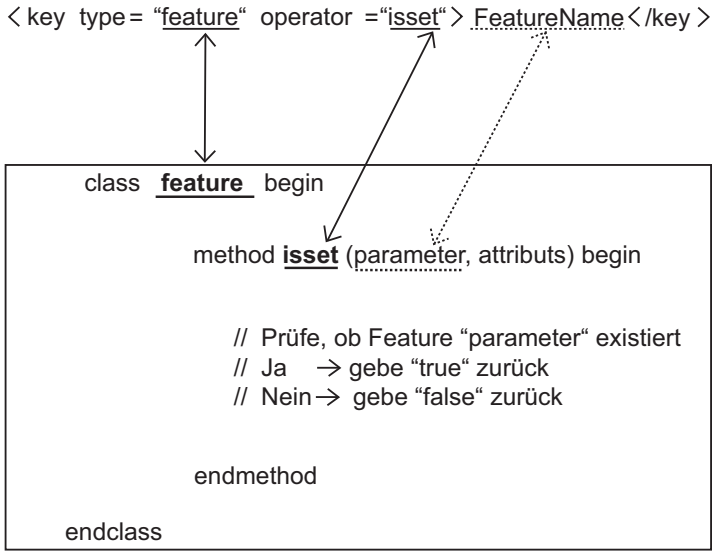

Abbildung 4.6: Klassen- und Methodenaufruf der Regeldatei von EVA

Methode „isset()“ der Klasse „feature“ aufgerufen. Als Parameter wird „FeatureName“ übergeben. Zusätzlich werden der Methode alle Attribute des XML-Elements als Array übergeben. In der Abbildung 4.6 ist dies durch den Parameter `attributs` dargestellt. Die Übergabe der Attribute des XML-Elements an eine Klasse könnte nützlich sein, um Vergleichsoperation durchzuführen. Beispielsweise ist das Element <key type=“mathematics“ operator=“between“ int1=“10“ int2=“40“>25</key> ein Aufruf, der prüft, ob der Wert 25 zwischen „int1“ und „int2“ liegt.

Der Vorteil dieses indirekten Zugriffs über den CommandWrapper auf die Klassen ist, dass der Nutzer tatsächlich nur die Klassen ansprechen kann, welche explizit vorgesehen wurden. Der CommandWrapper hat, wie bereits in Abbildung 4.5 dargestellt, nur Zugriff auf die Klassen des Bereichs „Erweiterbare Klassen und Methoden“. In der Praxis stellt dieser Bereich einen einzelnen Ordner im Dateisystem dar. Der CommandWrapper kontrolliert den Zugriff auf authorisierte Klassen in einem einzelnen Ordner. Damit ist ein Absturz zu erzwingen oder schadhaften Code zu programmieren nahezu unmöglich, sofern die angesprochenen Klassen gut implementiert sind. Zusätzlich wird dem Entwickler emöglicht diverse Klassen und Methoden zu implementieren ohne in das Variabilitätsmanagement direkt eingreifen zu müssen.

Während der Entwicklung wurde mit Repräsentanten der Firma Bosch GmbH weitere mögliche Syntax für den Zugriff auf die Klassen und Methoden besprochen. Eine weitere Darstellungsmöglichkeit des Zugriffs auf Klassen und Methoden findet sich in Listing 4.5.

Listing 4.5: Regeldatei - Beispiel 2

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<variability>
    <vp>
        <if>
            <or>
                <feature_isset>FeatureName</feature_isset>
                <and>
                    <feature_isset>FeatureNameA</feature_isset>
                    <feature_isset>FeatureNameB</feature_isset>
                </and>
            </or>
        </if>
        <then>
            <parameter_set value="5">ParName</parameter_set>
            <parameter_set value="6">ParName2</parameter_set>
            <parameter_set value="7">ParName3</parameter_set>
        </then>
        <else>
            <parameter_set value="8">ParName3</parameter_set>
        </else>
    </vp>
</variability>
```

Die Regeln die den Aufruf der Klasse und Methode jeweils in Attribute hinterlegt haben, sind hier als eigenes Element definiert - also <KLASSENNAME_METHODENNAME attribut="xyz">INHALT<KLASSENNAME_METHODENNAME>. Hieraus ergibt sich der Vorteil, dass die Regeldatei in kompakter Form darstellt wird und damit besser für den Menschen/Nutzer lesbar ist. Dies erhöht die Akzeptanz bei den Entwicklern. Eine kompaktere Regelform hat den Nachteil, dass die Elemente anhand der Zeichenkette in Klassenname und Methode zerlegt werden müssten. Im Gegensatz dazu ist die Syntax aus Listing 4.3 für die Machine besser lesbar. Sie kann direkt mit Hilfe eines XML-Parsers, wofür bisher in den meisten Programmiersprachen entsprechende Klassen existieren, geparst werden.
Die Definition eines XML Schemas ist nur dann sinnvoll, wenn die existierenden Elemente im Schema angegeben sind. Sonst muss beim Hinzufügen von neuen Klassen und Methoden das XML Schema angepasst werden. Wenn die Notation <KLASSENNAME_METHODENNAME attribut="xyz"> gewünscht wird, ist eine Transformation mit Hilfe Extensible Stylesheet Language (XSLT) zu empfehlen. Aufgrund der Komplexität wird an dieser Stelle nicht weiter auf XML Transformationen eingegangen, sondern auf die W3C Recommendation[5] verwiesen.

[5]http://www.w3.org/TR/xslt

In den Zeilen 5 und 11 bzw. 7 und 10 des Listings 4.3 befinden sich die Operatoren „OR“ und „AND“, dargestellt als XML Elemente. Mit Hilfe dieser können komplexe Bedingungen erzeugt werden. Die Kindelemente werden jeweils mit dem Elternknoten (also dem Operator) verknüpft. Um die aussagenlogische Formel zu bestimmen, beginnt man bei den Knoten mit der größten Tiefe und löst die Bedingung bis zur Wurzel auf. Die Klammerung wird aufgrund der gegebenen baumartigen Struktur von XML automatisch überflüssig.

Abbildung 4.7 stellt die Bedingung aus Listing 4.3 grafisch in einem Baum dar.
Um die Lesbarkeit in dieser Arbeit zu fördern wird feature->isset(EIN_FEATURE) im

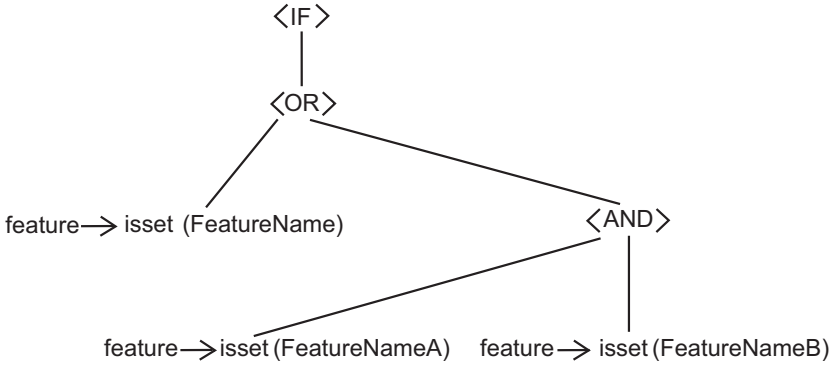

Abbildung 4.7: Bedingung einer Entscheidungsregel, dargestellt als XML Baum

Folgenden als EIN_FEATURE dargestellt. Durch den Operator <AND> werden „FeatureNameA“ und „FeatureNameB“ verknüpft. Danach wird das Resultat mit „FeatureName“ durch den Operator <OR> verknüpft. Die daraus resultierende Bedingung lautet:

(I) **Bedingung = FeatureName** $\vee$ **(FeatureNameA** $\wedge$ **FeatureNameB)**

Mit Hilfe dieser Notation ist es möglich, beliebige aussagenlogische Formeln zu definieren. Im Prototyp sind bislang nur die logischen Operatoren AND, OR und NOT definiert. Mit Hilfe dieser Gatter lassen sich andere Operationen wie zum Beispiel XOR, Äquivalenz oder Implikation erzeugen und müssen daher nicht implementiert werden. Dennoch wurde die Möglichkeit der Erweiterungsmöglichkeit dieser Operatoren im Variabilitätsmanagement berücksichtigt. Bei häufigem Vorkommen anderer Operatoren als AND, OR oder NOT kann durch Ableitung der abstrakten „State“-Klasse (Abbildung 4.5) ein neuer Operator erzeugt werden. Eine Änderung und Anpassung des Variabilitäsmanagements ist somit nicht nötig. Die Zuordnung des XML Element (<or>, <not>, <and>) zur jeweiligen Klasse, wird anhand des Namens vorgenommen. Der Klassenname besitzt zusätzlich das Präfix „State“. Das Element <and> wird also der Klasse „StateAND“ zugeordnet, das Element <or> der Klasse

„StateOR“ usw.
Zusätzlich befindet sich in Tabelle 4.1 eine Übersicht aller verfügbaren XML Elemente. Hieraus kann entnommen werden, welche Bedeutung die jeweiligen Elemente in der Regeldatei haben. Beispielsweise definiert das **<vp>-Element** einen Variablitätspunkt. Es kann 1...n mal vorkommen und besitzt keine Attribute.
Tabelle 4.2 gibt eine Übersicht aller bisher implementierten „Erweiterbaren Klassen und Methoden“ wieder. Die Klasse „Feature“ stellt beispielsweise einen Zugriff auf das Feature Set her. Dazu bietet es die Methoden „set()“ und „isset()“ zum Speichern eines Features oder zum Überprüfen, ob ein bestimmtes Feature vorhanden ist. In Tabelle 4.3 werden alle verfügbaren Operatoren für die Auswertung von Bedingungen aufgeführt. Jeder Operator ist in einer eigenen Klasse implementiert.

Element	Attribute	Kardinalität	Beschreibung
variability		1	Wurzelelement der Regeldatei
vp		1..n	Definiert einen Variabilitätspunkt
if		1..n	Enthält eine Bedingung
then		1..n	Enthält Anweisungen, die ausgeführt werden, wenn die vorherige Bedingung wahr ist. Element „then“ muss auf ein „if“ oder „elseif“ folgen.
else		0..n	Enthält Anweisungen, welche ausgeführt werden, wenn „if“ und „elseif“ Bedingungen zu falsch ausgewertet wurden.
elseif		0..n	Enthält eine Bedingung, welche geprüft wird, wenn „if“ und vorherige „elseif“ Bedingungen zu falsch ausgewertet werden.
key	type, operator	2..n	Ist die Schnittstelle zu den implementierten Klassen und dessen Methoden. Das Attribut „type“ enthält die Klasse und „operator“ die Methode.
and		0..n	Operator um zwei Elemente mit logisch UND zu verknüpfen.
or		0..n	Operator um zwei Elemente mit logisch ODER zu verknüpfen.
not		0..n	Operator um den Wahrheitswert eines Elements zu negieren.

Tabelle 4.1: Verfügbare XML Elemente

Klasse	Methode	Parameter	Beschreibung
Feature	isset	FeatureName	Prüft, ob ein entsprechendes Feature in der Feature Set vorhanden ist.
Feature	set	FeatureName	Entsprechendes Feature wird in das Feature Set aufgenommen.
Parameter	set	Wert, ParameterName	Setzt Parameter mit entsprechendem Wert im Parameter Set.
Parameter	add	Wert, ParameterName	Addiert Parameter zu vorhandenem Wert im Parameter Set.
Parameter	del	ParameterName	Löscht entsprechenden Parameter im Parameter Set.

Tabelle 4.2: Implementierte erweiterbare Klassen

Operator	Klasse	Neutrales Element	Beschreibung
and	StateAND	1	Logisches UND
or	StateOR	0	Logisches ODER
not	StateNOT		Logisches NOT

Tabelle 4.3: Verfügbare erweiterbare Operatoren

4.4.2 Realisierung der Auswertung

Die Architektur des Variabilitätsmanagements, welche bereits in Kapitel 4.4.2 besprochen wurde, enthält eine „Condition“ Klasse, die für die Auswertung der Bedingungen zuständig ist. Die Berechnung, die durchgeführt wird, wird anhand einer komplexeren Bedingung erläutert. Die Abbildung 4.8 zeigt die Baumstruktur von folgender Bedingung:

(II) **Bedingung = A $\wedge$ B $\wedge$ (C $\vee$ $\neg$ (D $\wedge$ E))**

Die Operanden A bis E stehen stellvertretend für Klassenmethoden, können als Feature- oder Parameterabfragen betrachtet werden, und geben bei Ausführung jeweils einen Wahrheitswert zurück. Für die Auswertung muss der Baum traversiert werden. Dies wird mit einem Tiefendurchlauf realisiert. Beginnend mit der Wurzel werden alle direkt erreichbaren Kinder besucht. Dazu folgt man zunächst immer der ersten abgehenden Kante. Falls keine Kante existiert, die zu einem unbesuchten Knoten führt, wird zum Vorgängerknoten zurückgegangen. Von dort aus wird die nächste Kante verfolgt. Jeder besuchte Knoten wird dabei in einer Liste gespeichert.

Geht man im Baum aus Abbildung 4.8 wie beschrieben vor, erhält man folgende

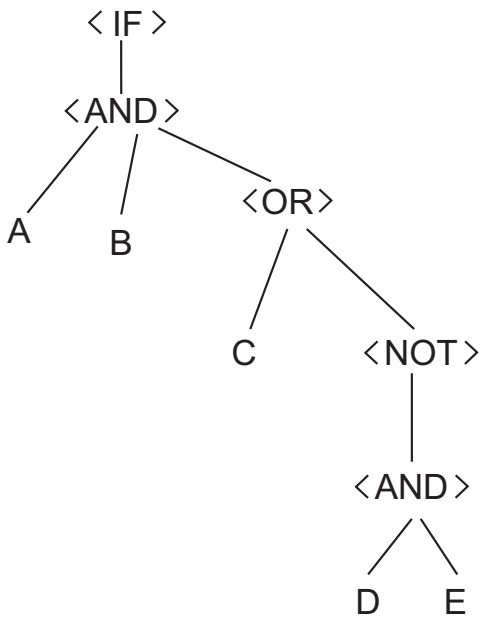

Abbildung 4.8: Bedingung II, dargestellt als XML Baum

abzuarbeitende Liste:

traversierter Baum: **<AND>, A, B, <OR>, C, <NOT>, <AND>, D, E**

Durch diese Liste wird iteriert. Wenn das nächste Element ein Operator ist, wird dieser inkl. dessen neutralem Element auf einem Stack abgelegt. Ein neutrales Element eines Operators kann beliebig oft mit dem bisherigen Ergebnis verknüpft werden ohne dabei den Wahrheitswert zu ändern. Ist der Operator beispielsweise OR, wird dieser mit seinem neutralem Element „0“ auf dem Stack abgelegt, also (OR, 0).
Sollte das nächste Element ein Operand sein, wird der Wert dieses Operanden mit dem obersten Element des Stacks sowie dessen Wert verknüpft. Das Ergebnis überschreibt dann den Wert des obersten Elements auf dem Stack.
Sobald kein Element mehr in der Liste vorhanden ist, wird der Wert des obersten Stack Elements mit dem zweitobersten Operanden und dessen Wert verknüpft. Dies wird solange wiederholt, bis der Stack leer ist. Der letzte Wert im Stack ist das Ergebnis.

Die Abbildung 4.9 demonstriert die Auswertung der Bedingung II. Für die beispielhafte Auswertung werden folgende Wahrheitswerte angenommen: A=1, B=1, C=0, D=1, E=0.

Die in der Abbildung 4.9 mit A-N bezeichneten offenen Rechtecke stellen den jeweiligen Zwischenstand des Stacks dar. Die Pfeile definieren jeweils eine Iteration. Zu Beginn der Berechnung ist der Stack leer (A). Beginnend mit der Iteration durch die Liste wird der Operator „AND“ als erstes Element der Liste sowie dessen neutrales Element „1“ auf den

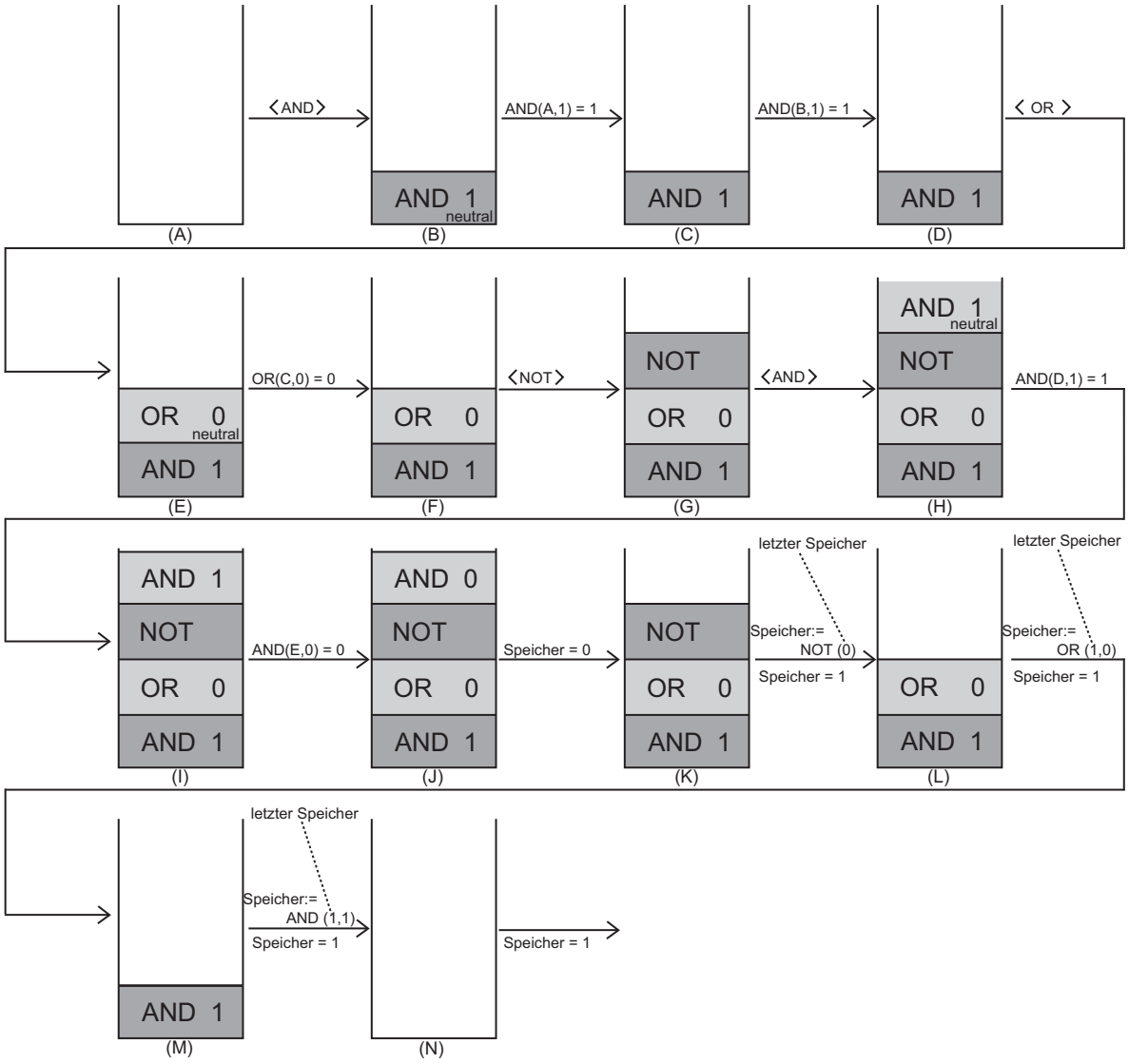

Abbildung 4.9: Die Abbildung demonstriert eine beispielhafte Auswertung einer Bedingung. Die Rechtecke stellen den jeweiligen Zwischenstand der Stacks dar. Die Pfeile zwischen den Stacks definieren die Iterationen.

Stack gelegt (B). Danach wird das zweite Element der Liste eingelesen, der Operand A. Um mit diesem Operanden rechnen zu können, wird das oberste Element, der Operator, vom Stack genommen. Der boolsche Wert des Operanden A wird dann mit dem obersten Element (Operator „AND“) und dessen Wert „1“ verknüpft (and(A,1)=1). Das daraus resultierende Ergebnis wird zurück auf den Stack geschrieben. Dies ist auch in Stack (C), Abbildung 4.9 dargestellt. Anschließend wird mit dem Operanden B, also dem nächsten Element der Liste, identisch verfahren (D). Als nächstes folgt in der Liste der Operator „OR“. Dieser wird wieder mit seinem neutralen Element auf den Stack geschrieben (E). Der darauf folgende Operand C wird dann erneut mit dem obersten Element, in diesem Fall „OR“ und dessem aktuellem Wert 0 verknüpft. Nach der Berechnung wird das Ergebnis wieder zurück auf den Stack geschrieben (F). Eine Besonderheit tritt bei dem nächsten Element der Liste, dem Operator „NOT“ auf. Da es sich hierbei um einen unären Operator ohne neutralem Element handelt, wird kein Wert zusätzlich gespeichert und nur der Operator selbst auf den Stack geschrieben. Bis Ende der Liste also bis zum Operator E wird dann identisch verfahren. Abbildung (J) stellt den Stack nach Abarbeitung des traversierten Baumes dar. Sobald kein Element mehr in der Liste vorhanden ist, wird das oberste Element vom Stack entfernt und dessen Wert zwischengespeichert (in der Abbildung mit „Speicher“ bezeichnet (K)). Der zuletzt gespeicherte Wert (Speicher) wird dann mit dem neuen obersten Wert und Operator des Stacks verknüpft. Im Beispiel durch Negation (NOT). Das Ergebnis wird dann wieder zwischengespeichert (Speicher=1) und das oberste Stackelement verworfen (L). Dieses Vorgehen wird durchgeführt, bis der Stack leer ist. Der dann aktuelle zwischengespeicherte Wert ist das Ergebnis. In diesem Fall und bei angenommener Belegung wird die Bedingung II zu „true“ ausgewertet.

Auch wurde der Fall berücksichtigt, dass sich der Wahrheitswert während der Auswertung einer Formel oder Teilformel nicht mehr ändern kann. Bei einer Verknüpfung mit dem „OR“ Operator kann die Bedingung nicht mehr zu „false“ ausgewertet werden, wenn mindestens ein Element zu „true“ ausgewertet wurde.

true = true $\vee$ X1 $\vee$ X2 $\vee$ X3 $\vee$ .... $\vee$ Xn

Auch kann eine „AND“ Verknüpfung nicht zu „true“ ausgewertet werden, wenn mindestens ein Element zu „false“ ausgewertet wurde.

false = false $\wedge$ X1 $\wedge$ X2 $\wedge$ X3 $\wedge$ .... $\wedge$ Xn

Bei Iteration durch die Liste wird geprüft, ob sich der Wahrheitswert des aktuellen obersten Stackelements noch ändern kann. Soll dies nicht der Fall sein, werden alle Blätter und Teilbäume des jeweiligen Operators nicht mehr geprüft. Betrachtet man erneut die Liste des traversierten Baumes mit anderen Wahrheitswerten (A=0, B=1, C=0, D=1, E=0) würde die Berechnung bereits nach dem Iterieren über A abbrechen, da der Operator AND bereits „false“ angenommen hat.

5 Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde ein Konzept sowie die Entwicklung eines Variabilitätsmanagements vorgestellt, das den Nutzer bei der Erstellung einer Fahrzeugsimulation unterstützen soll. Die Auflösung der Variabilitäten erfolgt dabei durch eine automatische Entscheidungsfindung anhand aussagenlogischer Regeln. Anschließend wurde eine Implementierung des Variabilitätsmanagement für ein bestehendes Simulationsframework der Firma Bosch GmbH beispielhaft vorgenommen. In diesem Kapitel fassen wir die Ergebnisse der vorliegenden Arbeit zusammen. Desweiteren geben wir einen Ausblick für weiterführende Arbeiten.

Die Konzeption des Variabilitätsmanagements beginnt mit der Domänenanalyse nach FODA um die Variabilitäten eines Systems zu erkennen und diese gegenüber der gemeinsamen Elemente abzugrenzen. Dabei wurden Bereiche auf dessen Zusammenhänge und Abläufe untersucht um wiederverwendbare Elemente zu gewinnen. Beginnend mit der Kontextanalyse, als Teilgebiet der Domänenanalyse, wurde die Problematik und die Anforderung der kompletten Domäne erfasst, dessen Umfang definiert sowie eine Eingliederung in deren Kontext durchgeführt.

In der anschließenden Domänenanalyse wurde das Gesamtsystem in logische Komponenten eingeteilt um diese in „gemeinsame“ und „variable“ Elemente einzuordnen. Die Trennung der Elemente muss so erfolgen, dass jedes Element eine eigene Zuständigkeit erhält und somit einen eigenen Aufgabenbereich abdeckt. Dieses Vorgehen ist nötig, damit die einzelnen Komponenten anwendungsneutral und dadurch innerhalb der Anwendung mehrfach einsetzbar sind. Für die Trennung des Gesamtmodells der Fahrzeugsimulation in einzelne Simulationsmodule nimmt man das reale Automobil zur Definition der Systemgrenzen als Vorbild, da für die Bestandteile eines Autos die Zuständigkeiten bereits definiert und bekannt sind. Ebenso musste entschieden werden, wie hoch die Granularität bei der Trennung der Zuständigkeiten sein soll. Dies ist abhängig von den einzelnen Komponenten und dem Einsatzzweck der Software. Nach Trennung des Gesamtsystems und Einteilung in „gemeinsame“ und „variable“ Elemente wurden die Abhängigkeiten analysiert und in einem sogenannten Feature-Modell dargestellt.

Des Weiteren wurde in der Arbeit eine featurebasierte Konfiguration vorgestellt und erläutert. Durch eine Konfiguration mit Hilfe von Features entsteht eine theoretische Abstraktionsschicht zwischen dem Nutzer und der direkten Konfiguration. Die Zuordnung der Simulationsmodule zu den Features, genauso wie die Abhängigkeiten der Simulationsmodule untereinander werden über Entscheidungsregeln abgebildet. Die Regeln werden durch die Variabilitätspunkte direkt mit dem Simulationsmodul verknüpft.

Folglich wird die direkte Konfiguration der Simulationsmodule unnötig und das benötigte Wissen um eine Konfiguration vornehmen zu können, wird für einen Nutzer dadurch verringert. Zusätzlich wurden die Abhängigkeiten der Simulationsmodule mit Hilfe der Entscheidungsregeln automatisch gelöst, was Fehlkonfigurationen vorbeugt. Das Detailwissen wird also vom Nutzer weg in die Entscheidungsregeln verlagert.

Jedem Simulationsmodul ist eine eigene Regeldatei zugeordnet. Bei Auswertung dieser Regeldatei werden durch die Bedingungen, die sich meist auf Features beziehen, dynamisch diverse Parameter im Simulationsmodul manipuliert. Die Variabilität erfolgt also implizit durch das Setzen und ändern von Parametern im Simulationsmodul. Im Gegensatz dazu wurde in diversen anderen Ansätzen, wie z.B. [ML08] und [MALP09] Variabilität explizit modelliert. Bei der expliziten Modellierung von Variabilität wird im Quelltext eine Methode eingebunden, welche den Variabilitätspunkt darstellt. Diese bindet explizit an dieser Stelle des Quellcodes eine Variante.

Die durch Entscheidungsregeln eingeführte Variabilität betrifft vor allem die „Variabilität in Features“, um den Funktionsumfang dynamisch an die Anforderung des Nutzers anzupassen. Dadurch sind vergleichende Simulationen inklusive automatischer Auflösung der Abhängigkeiten möglich. Des Weiteren können diverse andere Typen von Variabilität mit Hilfe der Entscheidungsregeln eingesetzt werden. Die Variabilität im Datenformat oder -umfang kann z.B. verwendet werden, um Datenmanipulationen vorzunehmen.

Ebenso wurden in dieser Arbeit die Herausforderungen eines Variabilitätsmanagements in der Fahrzeugsimulation heraus gearbeitet. Das Zerlegen des Gesamtsystems in einen zu hohen und unnötigen variablen Anteil erschwert das Verständnis und führt zu ineffizienter Nutzung des Systems. Des Weiteren muss die Grenze und Granularität eines Simulationsmoduls korrekt festgelegt werden, um die Anforderungen der Nutzer erfüllen zu können. Bei zu feiner und unnötiger Variabiltität hingegen wird die Handhabbarkeit schwieriger und überfordert den Nutzer.
Außerdem wächst mit jedem variablen Element oder Simulationsmodul gleichzeitig die Anzahl der Regeln, welche zur Entscheidungsfindung genutzt werden, um mögliche Abhängigkeiten aufzulösen. Auch die Entscheidungsregeln bilden großes Fehlerpotential, basierend auf menschlichen Wissens und können nicht vollständig automatisiert auf Korrektheit geprüft werden. Eine Überprüfung kann nur stichprobenartig erfolgen.
Schon bei der Entwicklung der Software verursacht eine große Anzahl variable Elemente, vor allem in Testphasen oder beim Debugging Probleme. Allein die Durchführung von Einzeltests reicht nicht aus um eine Zuverlässigkeit der Software zu erreichen. Die Kombination von unterschiedlichen Simulationsmodule können beispielsweise zu „Race Conditions“ führen.

Bei optionalen Features steigen die Kombinationsmöglichkeiten exponentiell an und bilden damit die Obergrenze. Jedes variable Element kann aktiviert oder deaktiviert werden. Die Grenze ist damit 2^n. Bei obligatorischen Features hingegen wächst die Anzahl der Möglichkeiten linear mit der Anzahl der variablen Elementen. Dies bildet die Untergrenze, da das Wachstum von n kleiner ist als das Wachstum von 2^n.

Bei der Entwicklung sollte also darauf geachtet werden, die Variabilität minimal zu gestalten, um die Anzahl der Kombinationsmöglichkeiten zu senken. Dennoch muss ein Mindestmaß an Variabilität vorhanden sein, damit die Anwendung seinen Zweck erfüllt und vom Nutzer akzeptiert wird. Bei sehr geringer Variabilität ist das Variabilitätsmanagementsystem unnötig, da es die Inititalkosten der Architektur nicht rechtfertigt. Hier wäre auf eine manuelle Konfiguration zurückzugreifen.
Im zweiten Teil der Arbeit wurde im Rahmen dieser Arbeit und in Kooperation mit der Firma Bosch GmbH ein prototypisches Variabilitätsmanagement für ein bestehendes Simulationsframework implementiert. Dazu wurde, um ein grundlegendes Verständnis zu erlangen, der ehemalige Ablauf des Simulationsframework EVA beschrieben. Anschließend wurden die Anforderungen an das Variabilitätsmanagement definiert und bei der Architektur und Implementierung berücksichtigt.
Das Variabilitätsmanagement wurde als eine Art „Black Box“ System entwickelt. Dadurch erfolgt eine einfache Integrationsmöglichkeit, in der erreicht wird, ohne detailliertes Wissen über genaue Funktionalitäten einzelner Objekte und Methoden des vorhandenen Systems das Variabilitätsmanagement zu integrieren. Zusätzlich wurde darauf geachtet, dass nur eine geringe Anzahl von Schnittstellen zu externen Objekten des Frameworks notwendig sind. Die bereits existierenden Technologien wurden weitgehend übernommen, um möglichst wenige der bisherigen Objekte und Schnittstellen anpassen zu müssen.
Weiterhin wurde versucht eine einfache, verständliche und leistungvollständige Syntax und Semantik der Regeln zu definieren. Dadurch soll die Akzeptanz beim Nutzer gesteigert werden und den Umgang mit der Software erleichtern. Ebenso soll die Syntax und Semantik dafür sorgen, dass alle Leistungen die von einer Software gefordert auch tatsächlich erbracht werden.
Durch Weiterentwicklungen oder Veränderungen kann es wichtig sein, die Regeln zu erweitern oder anzupassen. Daher bestand die Notwendigkeit die Syntax so zu definieren, dass die Bedingungen und Anweisungen erweiterbar sind. Die Bedingungen der Regeln sind so definiert und im Variabilitätsmanagement entwickelt, dass sowohl die Operatoren als auch die Operanden in eigenen Klassen ausgelagert sind. Beispielhaft wurden AND, OR und NOT implementiert, mit denen jegliche andere Operatoren modelliert werden können.
Weiterhin wurde ein Hauptaugenmerk auf die Zuverlässigkeit und Korrektheit des System gelegt. Dies betrifft vor allem die Konfiguration der Simulationsmodule. Im bisherigen Ansatz wurden diese manuell konfiguriert und ins Gesamtmodell eingebunden. Durch die Veränderung eines Simulationsmoduls wurden Abhängigkeiten verletzt. Dies wurde durch die featurebasierte Konfiguration verbessert und eine automatisierte Auflösung von Abhängigkeiten realisiert, was deutlich fehlertoleranter ist, als eine direkte Konfiguration.

Das Simulationsframework wurde entsprechend des Konzepts dahingehend geändert, dass die Konfiguration featurebasiert erfolgt. Die Zuordnung der Simulationsmodule zu den Features wird durch Entscheidungsregeln implementiert. Die Feature Konfigurationsdatei dient zur Konfiguration der Features und damit zur indirekten Konfiguration der Simulationsmodule.
Im Gegensatz zum bisherigen Ansatz wird der get_Parameter() Aufruf nicht mehr direkt an das Parameter Set gerichtet sondern an das Variabilitätsmanagement. Das Variabilitätsmanagement entscheidet anhand von den Simulationsmodulen zugehörigen Regeln und Features,

welche Parameter im Simulationsmodul gesetzt werden. Die Syntax der Bedingungsregeln wurde mit XML definiert. In einer vorgegebenen XML-Struktur liegt der Vorteil darin, dass sie den Nutzer bei den Möglichkeiten zur Erstellung der Regeln einschränkt. XML ist zudem plattformunabhängig und die XML-Dokumente können auch anderweitig verarbeitet werden.

Die Ergebnisse dieser Arbeit können auch vielfältig in weiterführenden Arbeiten verwendet werden. Einerseits können die Konzepte zur Variabilität in der Fahrzeugsimulation verfeinert werden. Andererseits bietet die Variabilität auch unabhängig der Fahrzeugsimulation die Möglichkeit, eine gezielte Wiederwendung zu betreiben um die Entwicklungszeit zu beschleunigen und die Kosten zu senken.

Die FODA Analyse bietet ein Konzept zum Zerlegen des Gesamtsystems in einzelne variable Artefakte. Die Festlegung der Granularität wird in FODA nicht betrachtet, sie stellt in der Entwicklung jedoch eine große Herausforderung dar. Darüber hinaus hat sie einen beträchtlichen Einfluss in der Akzeptanz beim Nutzer. Eine hohe und vor allem unnötige Variabilität erschwert das Verständnis und führt zu ineffizienter Nutzung des Systems.
Ein weiterer Aspekt, welcher beleuchtet werden sollte, ist die Herausforderung in Testphasen. Durch das automatisierte Einbinden von variablen Elementen entstehen eine hohe Anzahl an Kombinationen von Gesamtsimulationen. Die Durchführung von „Unit Tests“, versichert zwar die Funktionalität im Einzelnen, jedoch nicht im Kontext. Mögliche Verfahren und Ansätze könnten heraus gearbeitet werden um einen systematischen Test zu ermöglichen.
Der Weiteren basieren die Regeln auf menschlichem Wissen und sind daher schwer zu prüfen. Auch hier könnten Ansätze und Lösungen gefunden werden um die Konfiguration des impliziten Wissens zu unterstützen und um möglichen Fehlern entgegenzuwirken.

Literaturverzeichnis

[APD91] G. Arango, R. Prieto-Diaz. Domain Analysis and Software Systems Modeling. Technical report, IEEE Computer Society Press, 1991. (Zitiert auf Seite 18)

[BB01] F. Bachmann, L. Bass. Managing Variability in Software Architectures. Technical report, Carnegie Mellon University, 2001. (Zitiert auf Seite 31)

[BCC+10] J. K. Bergey, G. Chastek, S. Cohen, P. Donohoe, L. G. Jones, L. Northrop. Software Product Lines: Report of the 2010 U.S. Army Software Product Line Workshop. Technical report, Software Engineering Institute CarnegieMellon University, 2010. (Zitiert auf Seite 14)

[Beh02] S. Behl. Domain Analysis. Technical report, Universität Stuttgart, 2002. (Zitiert auf den Seiten 19, 23 und 24)

[BHP03] S. Bühne, G. Halmans, K. Pohl. Modelling Dependencies between Variation Points in Use Case Diagrams. Technical report, Universität Duisburg-Essen, 2003. (Zitiert auf Seite 25)

[BK04] A. Borusan, E. Kleinod. Software-Produktlinien. Technical report, Technische Universität Berlin, Computergestützte Informationssysteme, 2004. (Zitiert auf Seite 14)

[BKPS04] G. Böckle, P. Knauber, K. Pohl, K. Schmid. *Software-Produktlinien*. dpunkt.verlag, 2004. (Zitiert auf den Seiten 7, 11, 13, 14, 16, 25, 27, 31 und 42)

[Dum03] R. Dumke. *Software Engineering*. vieweg, 2003. (Zitiert auf Seite 28)

[Gen03] K. Gensthaler. *Definition und Integration einer XML-Schnittstelle für die Recherche von Medienrechten in einem heterogenen Archivverbund*. Master's thesis, Universität Karlsruhe, 2003. (Zitiert auf Seite 63)

[Har05] S. Hartmann. The World as a Process: Simulations in the Natural and Social Sciences. Technical report, University of Pittsburgh, 2005. (Zitiert auf Seite 6)

[JRH+04] M. Jeckle, C. Rupp, J. Hahn, B. Zengler, S. Queins. *UML2 glasklar*. Hanser, 2004. (Zitiert auf Seite 28)

[KCH+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson. Feature-Oriented Domain Analysis (FODA) - Feasibility Study. Technical report, Software Engineering Institute, Carnegie Mellon University Pittsburgh, 1990. (Zitiert auf Seite 19)

[LL07] J. Ludewig, H. Lichter. *Software Engineering*. dpunkt.verlag, 2007. (Zitiert auf den Seiten 10, 24, 37 und 49)

[MALP09] R. Mietzner, M. A, F. Leymann, K. Pohl. Variability Modeling to Support Customization and Deployment of Multi- Tenant-Aware Software as a Service Applications. Technical report, University of Stuttgart, 2009. (Zitiert auf den Seiten 47 und 76)

[Mar06] S. Marr. Feature-Diagramme und Variabilität. Technical report, Hassi-Plattner-Institut Potsdam, 2006. (Zitiert auf Seite 22)

[ML03] T. von der Maßen, H. Lichter. Modellierung von Variabilität mit UML Use Cases. Technical report, RWTH Aachen, 2003. (Zitiert auf Seite 28)

[ML08] R. Mietzner, F. Leymann. Generation of BPEL Customiziation of Processes for SaaS Applications from Variability Descriptors. Technical report, University of Stuttgart, Institute of Architecture of Application Systems, 2008. (Zitiert auf den Seiten 25, 47 und 76)

[ML09] R. Mietzner, F. Leymann. Concepts and Techniques for a generic Application Portal in the Cloud. Technical report, University of Stuttgart, 2009. (Zitiert auf Seite 25)

[PA08] K. Pohl, M. A. Variabiliätsmanagement in Software-Produktlinien. Technical report, Universität Duisburg-Essen, 2008. (Zitiert auf den Seiten 28 und 31)

[PBL05] K. Pohl, G. Böckle, F. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005. (Zitiert auf Seite 31)

[PH10] K. Pohl, G. Halmans. Modellierung der Variabilität einer Software Produktfamilie. Technical report, Universität Essen, 2010. (Zitiert auf Seite 31)

[Puk06] M. Pukall. FAME-DBMS: Entwurf ausgewaehlter Aspekte der Transaktionsverwaltung. Technical report, Otto-von-Guericke-Universitaet Magdeburg, 2006. (Zitiert auf Seite 14)

[Sch03] W. Schleicher. Domain Analysis und Scoping. Technical report, Universität Stuttgart, 2003. (Zitiert auf den Seiten 17 und 19)

[Tan03] A. S. Tanenbaum. *Moderne Betriebssysteme*. Pearson Eduction, 2003. (Zitiert auf Seite 48)

[Wüb10] A. Wübbeke. *Variabilitätsmanangement in Anforderungs- und Testfallspezifikation für Software-Produktlinien*. Ph.D. thesis, Universität Paderborn, 2010. (Zitiert auf den Seiten 31, 37 und 38)

Danksagung

Ich möchte mich herzlich bei all denen bedanken, die mich bei dieser Arbeit unterstützend begleitet haben.

Mein Dank geht an die Firma Bosch GmbH, die mir das interessante Projekt und einen Einblick in deren Abläufe ermöglicht hat. Insbesondere danke ich auch meinem Betreuer Dr. Uwe Scholz, der Firma Bosch GmbH, der mich in die Materie eingeführt hat und jederzeit für fachliche Fragen offen war.

Auch meiner Betreuerin Dipl. Inf. Katharina Görlach, der Universität Stuttgart, ein herzliches Dankeschön für das große Engagement, die unermüdliche Unterstützung und der hilfreichen Hinweise.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Michael Müller)