Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät II
Institut für Informatik

Durchgängiges Variantenmanagement von der Anforderungsspezifikation bis zum Test auf Basis von MERAN und CTE XL Professional

Diplomarbeit

Eingereicht von Christian Gebhardt
Berlin, den 17. November 2011

Gutachter:
Prof. Dr. sc. nat. Klaus Bothe
Dr. rer. nat. Joachim Wegener (Berner & Mattner Systemtechnik GmbH)

Danksagung

An dieser Stelle möchte ich all denen danken, die mich bei der Erstellung dieser Diplomarbeit unterstützt haben.

Mein erster Dank gilt Herrn Prof. Dr. sc. nat. Klaus Bothe für seine Bereitschaft zur Betreuung dieser Diplomarbeit. Seine zielführenden Anregungen waren mir beim Verfassen dieser Arbeit eine große Hilfe. Zudem hat mir seine Unterstützung bei der Suche eines geeigneten Themas erst die Chance eröffnet, diese Arbeit im Rahmen einer Diplomandentätigkeit bei der Berner & Mattner Systemtechnik GmbH schreiben zu können.

Des Weiteren möchte ich mich bei Herrn Dr. rer. nat. Joachim Wegener bedanken, der mich als Zweitgutachter und Betreuer in den vergangenen Monaten stets unterstützt und mir durch sein Vertrauen die Stelle als Diplomand bei der Berner & Mattner Systemtechnik GmbH erst ermöglicht hat. Seine kreativen Anregungen und konstruktiven Verbesserungsvorschläge haben maßgeblich zum Gelingen dieser Arbeit beigetragen.

Stellvertretend für meine Freunde möchte ich mich bei Manuel Hertlein bedanken, der mir nicht nur ein wertvoller Weggefährte während des Studiums, sondern mit all seinem Zuspruch auch immer wieder ein wichtiger Motivator war.

Ganz besonders möchte ich meinen Eltern und dem Rest meiner Familie danken, die nie aufgehört haben an mich zu glauben und mein Studium in all den Jahren mit viel Geduld und Verständnis unterstützt und begleitet haben. Danke für eure Ausdauer und euren Glauben an mich!

Mein größter Dank gebührt Heide, die mir durch ihre unendliche Geduld und ihre moralische Unterstützung den nötigen Rückhalt gegeben und mit ihren motivierenden Worten stets das Ziel vor Augen geführt hat. Ohne dich hätte ich es nicht geschafft!

Für die unzähligen Stunden des Korrekturlesens danke ich schließlich Heide und ihrem Vater Bernd Stache.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

> *„I will build a motor car for the great multitude. It will be large enough for the family but small enough for the individual to run and care for. It will be constructed of the best materials, by the best men to be hired, after the simplest designs that modern engineering can devise. But it will be so low in price that no man making a good salary will be unable to own one - and enjoy with his family the blessing of hours of pleasure in God's great open spaces.“*
>
> *Henry Ford*

Henry Ford entwickelte Anfang des 20. Jahrhunderts mit dem Modell T das erste Fahrzeug, das in Massenfertigung am Fließband hergestellt wurde. Zwischen 1908 und 1927 wurden auf diese Weise mehr als 15 Millionen Exemplare dieses Modells produziert – Ford machte das Automobil für die breite Masse zugänglich [Collins 2007].

Wollte Ford seinerzeit noch ein Auto für Jedermann bauen, so versuchen die Automobilhersteller heutzutage für jeden Kunden das passende Auto zu entwickeln. Auch in den anderen Bereichen der Konsumgüterindustrie wurden im Zuge der kundenindividuellen Massenfertigung die Vorteile der Massenproduktion mit denen der kundenindividuellen Einzelfertigung kombiniert. War es für Ford noch ein Alleinstellungsmerkmal Produkte in großer Stückzahl fertigen zu können, entscheiden heute innovative Produktfunktionen und eine große Produktpalette über das Bestehen am Markt. Das führt dazu, dass beispielsweise die aktuelle Serie der E-Klasse von Mercedes-Benz in der Karosserieform Limousine, kombiniert mit den Individualisierungsmerkmalen Motor, Designlinie, Sonderausstattungspaket, Farbe, Felgen und Polsterung, in über 100.000 wählbaren Fahrzeugvarianten lieferbar ist [Mercedes-Benz 2011].

Um die günstige, termingerechte und qualitativ hochwertige Fertigung einer derartigen Variantenvielfalt gewährleisten zu können, müssen Entwicklungs- und Produkti-

onsprozesse angepasst, sowie Organisationsstrukturen überarbeitet werden. Gleichzeitig stellt die Verwaltung verschiedener Produktvarianten für alle Phasen des Entwicklungsprozesses einen zusätzlichen Aufwand an Zeit und Kosten dar, den es zu minimieren gilt. Die große Anzahl an Produktvarianten ist erst durch den Einsatz von Software-Werkzeugen entlang des gesamten Entwicklungsprozesses beherrschbar. Dementsprechend müssen die existierenden Werkzeuge an die veränderten Rahmenbedingungen angepasst oder neue Werkzeuge entwickelt werden.

1.1. Problemstellung

Innerhalb der Produktionstechnik wurde der Variabilität mit Baukastensystemen, dem Plattformprinzip oder Variantenstücklisten begegnet. Da es sich heutzutage bei komplexen Produkten meist um softwareintensive Systeme handelt, wurden diese Ansätze auf den Bereich der Softwareentwicklung übertragen und mit der Software-Produktlinien-Entwicklung ein analoger Ansatz eingeführt.

Für die Entwicklung hochwertiger Produkte spielt neben derartigen systematischen Entwicklungsprozessen die nachhaltige Qualitätssicherung eine entscheidende Rolle. Ein grundlegendes Konzept zur Steigerung der Produktqualität stellt die Nachverfolgung von Anforderungen dar, die sich über den gesamten Entwicklungsprozess hinweg durch die Verlinkung der Anforderungen mit vorhergehenden oder nachfolgenden Artefakten der verschiedenen Entwicklungsphasen realisieren lässt. Die Verlinkung von Artefakten hat für die einzelnen Phasen und die beteiligten Akteure eine unterschiedliche Semantik ([Ramesh und Jarke 2001]). Werden Anforderungen beispielsweise mit den Softwarekomponenten der Implementationsphase, die sie erfüllen, verlinkt, lässt sich überprüfen, ob alle Anforderungen mit den Teilen des entstehenden Endprodukts abgedeckt sind. Wurden die Anforderungen dagegen mit den dazugehörigen Testfällen verlinkt, kann zum einen die Abdeckung der Anforderungen durch Testfälle analysiert werden und zum anderen können beim Auftreten von Fehlern während des Testens die Anforderungen identifiziert werden, die mit Implementierungsfehlern korrelieren.

Um die Verlinkung von Anforderungen mit Testfällen effizient zu unterstützen, hat die Berner & Mattner Systemtechnik GmbH (Berner & Mattner) den Klassifikationsbaum-Editor CTE XL Professional um eine Schnittstelle zum weit verbreiteten Spezifikationswerkzeug IBM® Rational® DOORS erweitert, die es ermöglicht, systema-

tisch erstellte Testfälle in CTE XL Professional mit Anforderungsspezifikationen aus DOORS zu verknüpfen.

Zur Unterstützung der variantenspezifischen Anforderungsanalyse wurde bei Berner & Mattner auf Basis von DOORS die Toolbox MERAN entwickelt, die die Generierung variantenspezifischer Anforderungsspezifikationen erlaubt und durch die Automatisierung einzelner Prozessschritte eine Kosten- und Zeitersparnis in den frühen Analyse- und Spezifikationsphasen ermöglicht. Dazu wird die modellbasierte Anforderungsanalyse mittels DOORS mit einem effizienten Variantenmanagement für die Anforderungsspezifikation durch MERAN verknüpft ([Robinson-Mallet und Wegener 2009]). Für die Software-Produktlinien-Entwicklung bietet MERAN somit eine wirkungsvolle Werkzeugunterstützung des Domain Requirement Engineering und des Application Requirement Engineering (vgl. Abbildung 1.1).

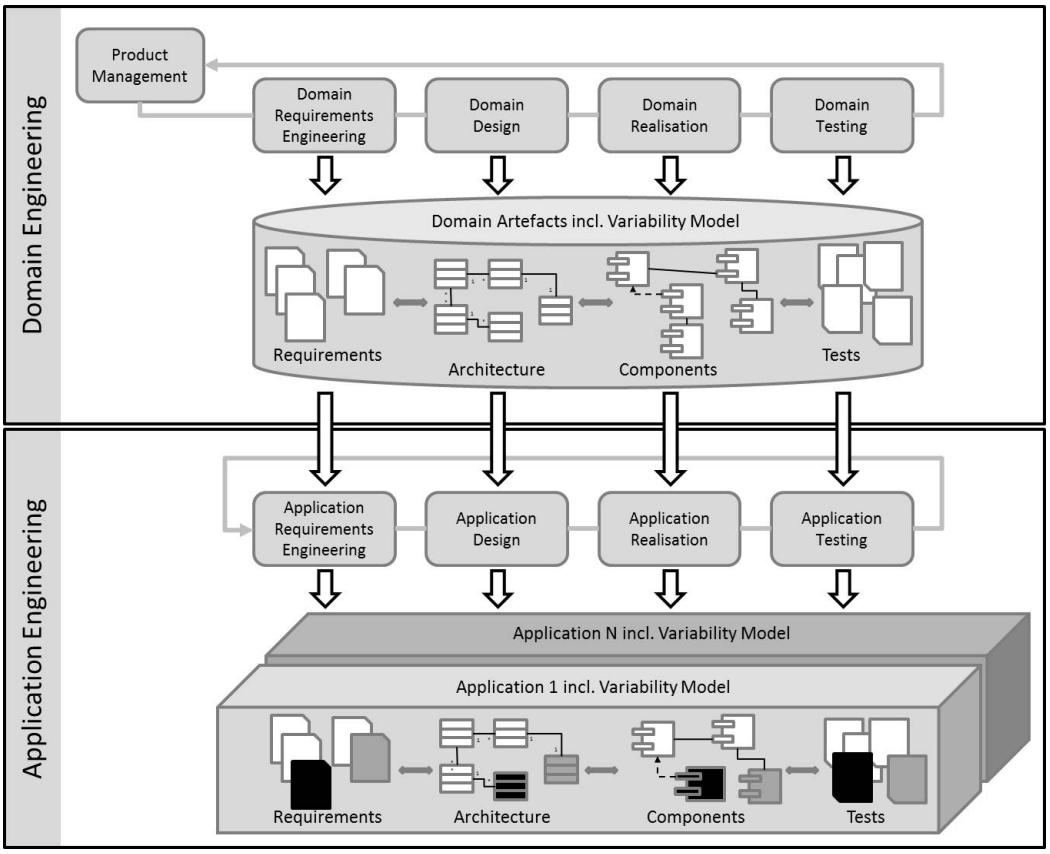

Abbildung 1.1.: Software-Produktlinien-Entwicklung (nach [Pohl u. a. 2005])

Ein Variantenmanagement für Anforderungen reicht jedoch allein nicht aus, um eine variantenspezifische Anforderungsnachverfolgung über den gesamten Entwicklungsprozess hinweg zu realisieren und so die effiziente Qualitätssicherung für die einzelnen Entwicklungsphasen zu gewährleisten. Vielmehr muss auch der CTE XL Professio-

nal und die ihm zugrundeliegende Klassifikationsbaum-Methode ([Grochtmann und Grimm 1993]) durch eine Unterscheidung von Varianten erweitert werden, um die durchgängige Verwendung von Varianten auch für die Testfallbestimmung zu unterstützen. Wie für die frühen Phasen des Entwicklungsprozesses, ergeben sich auch für die variantenspezifische Testphase signifikante Einsparpotentiale bezüglich der Zeit und der Kosten für ihre Erstellung und Nutzung. Zur Erschließung dieser Potentiale und zur Unterstützung der variantenspezifischen Anforderungsnachverfolgung ist ein Software-Werkzeug erforderlich, das die Erstellung variantenspezifischer Testfälle ermöglicht und deren Verlinkung mit den dazugehörigen Anforderungen gestattet. Durch eine derartige Werkzeugunterstützung wird die Variantenvielfalt beherrschbarer und ein weiterer Schritt hin zu einem durchgängigen Variantenmanagement entlang des gesamten Entwicklungsprozesses realisiert.

1.2. Zielsetzung der Arbeit

Ziel der vorliegenden Arbeit ist die Entwicklung eines **generischen Verfahrens** für ein Variantenmanagement während der Testfallbestimmung mithilfe der Klassifikationsbaum-Methode und dem CTE XL Professional.

Auf Basis des generischen Verfahrens soll ein **Software-Prototyp** in Form eines Plug-Ins für den CTE XL Professional konzipiert und implementiert werden. Hierbei soll es möglich sein, mit dem CTE XL Professional verschiedene Varianten eines Produktes anzulegen und zu verwalten. Darüber hinaus soll ein Import von Varianteninformationen aus DOORS und MERAN möglich sein, welche im CTE XL Professional zu übernehmen sind. Verlinkungen, die mithilfe der Schnittstelle zwischen DOORS und CTE XL Professional erzeugt wurden, sollen auch für die Varianten ausgegeben werden, um die variantenspezifische Anforderungsnachverfolgbarkeit zu unterstützen. Die Varianten sollen mittels eines Sichtenkonzepts im CTE XL Professional dargestellt werden, das die Umschaltung zwischen den verschiedenen Varianten erlaubt.

Bei der Betrachtung des in Abbildung 1.2 dargestellten Klassifikationsbaumes für einen Sensor zur Warnung bei unterschiedlichen Türzuständen im Pkw und unter Berücksichtigung der Fahrzeugvarianten Cabriolet, Limousine und Kombi, ist zu erkennen, dass für eine Variante immer nur bestimmte Elemente des Klassifikationsbaumes und spezielle Testfälle relevant sind. So besitzt ein Cabrio in der Regel hinten links und hinten rechts keine Türen. Damit müssen für die Klassifikationen

Tür hinten links und *Tür hinten rechts* auch keine Testfälle erstellt werden, da diese nicht getestet werden können. Somit ist die Anzeige in CTE XL Professional für den Test des Cabriolets nicht notwendig. Gleichzeitig müssen erstellte Testfälle, die die Klassifikationen einer Variante nicht abdecken, auch nicht angezeigt werden.

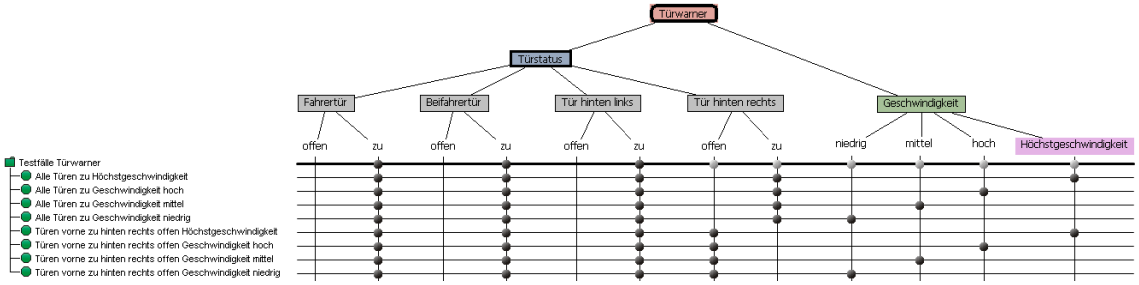

Abbildung 1.2.: Testfälle für einen Sensor zur Warnung bei unterschiedlichen Türzuständen im Pkw

Parameter sind wichtige Unterscheidungsmerkmale von Varianten, die auch bei der Testfallerstellung und Testdurchführung berücksichtigt werden müssen. Der in Abbildung 1.2 bereits in den Klassifikationsbaum als Klasse eingefügte Parameter *Höchstgeschwindigkeit* könnte zum Beispiel für die Variante Cabriolet 200 km/h und für die Variante Limousine 250 km/h betragen und so die unterschiedliche Höchstgeschwindigkeit der Fahrzeugvarianten beschreiben.

Im Rahmen dieser Arbeit sollen Parameter als zusätzliche Varianteninformation in den CTE XL Professional integriert werden. Wurden Parameter bereits mit den Werkzeugen DOORS und MERAN spezifiziert, soll deren Import nach CTE XL Professional möglich sein. Die Parameter, einschließlich der variantenspezifischen Parameterwerte, sollen im CTE XL Professional durch ein Sichtenkonzept angezeigt werden.

In einem weiteren Schritt soll eine **Verifikationsfunktion** in das Plug-In integriert werden, welche falsche oder fehlende Zusammenhänge aufzeigt. Wurde beispielsweise im obigen Fall die Variante Cabriolet fälschlicherweise mit der Klassifikation *Tür hinten links* im Klassifikationsbaum verknüpft, soll dies durch die Verifikationsfunktion aufgedeckt werden. Dazu muss ein Mechanismus entworfen werden, der solche Zusammenhänge erkennt und anzeigt. Ferner soll die Testfallüberdeckung von Varianten überprüft werden. Sind nicht alle Objekte der Variantenspezifikation mit Testfällen verknüpft, soll dies durch die Verifikationsfunktion ausgegeben werden. Umgekehrt sollen die Testfälle angezeigt werden, die noch nicht mit einer oder mehreren Varianten verknüpft sind.

Abschließend soll das Verfahren anhand der prototypischen Umsetzung und deren Anwendung auf ein bespielhaftes Nutzungsszenario bewertet werden.

Bei der Entwicklung der Schnittstelle ist eine möglichst generische Implementierung anzustreben, um die spätere Erweiterbarkeit, zum Beispiel die Anbindung an andere Requirement Tracing-Werkzeuge wie MKS, IBM® Rational® Quality Manager oder IRQA, zu gewährleisten.

1.3. Zusammenfassung der Ergebnisse und Aufbau der Arbeit

Nachdem in Kapitel 1 eine Einführung in die Thematik und die Motivation der Arbeit erfolgt sind, werden im weiteren Verlauf der Arbeit zunächst die theoretischen Grundlagen zur Entwicklung eines generischen Verfahrens für das Variantenmanagement bei der Testfallbestimmung mithilfe der Klassifikationsbaum-Methode und dem CTE XL Professional gelegt. Anschließend erfolgt die Herleitung des Verfahrens, für das im darauffolgenden Kapitel ein Konzept für eine prototypische Umsetzung entwickelt wird. Die abschließenden Kapitel der Arbeit befassen sich mit der Bewertung des Verfahrens und der Zusammenfassung der Ergebnisse. Im Detail gliedert sich die vorliegende Arbeit wie folgt:

Kapitel 2 erläutert die Grundlagen der Qualitätssicherung und führt die betrachteten Software-Werkzeuge ein. Dabei wird zuerst das Requirements Engineering als wichtiger qualitätssichernder Bestandteil der Entwicklungsprozesse erläutert und das Dynamic Object-Oriented Requirements System als Management Werkzeug für die Anforderungsspezifikation und -verwaltung eingeführt. Anschließend wird erläutert, welchen Beitrag die Anforderungsnachverfolgung zur Qualitätssicherung leistet. Im nächsten Schritt wird MERAN als Werkzeug zur Unterstützung des Variantenmanagements bei der Anforderungsspezifikation vorgestellt. Den Abschluss des Kapitels bilden eine Einführung in die Testmethodik, die Erklärung der Klassifikationsbaum-Methode und die Beschreibung des CTE XL Professional als Werkzeug zur Testfallbestimmung. In diesem Zusammenhang wird auch die Schnittstelle zwischen DOORS und CTE XL Professional beschrieben.

Als Grundlage für die Entwicklung eines generischen Verfahrens für ein Variantenmanagement bei der Testfallbestimmung führt Kapitel 3 zunächst kurz in die Thematik

der individuellen Massenfertigung ein und klärt, welche Ansätze dafür im Bereich der Software-Entwicklung entstanden sind. In diesem Zusammenhang wird die Software-Produktlinien-Entwicklung als wichtigster Ansatz für die Herstellung variantenreicher Software-Produkte und die systematische Wiederverwendung identifiziert und die Grundlagen einer solchen Entwicklung werden erklärt. Dabei wird das Software-Produktlinien-Framework nach [Pohl u. a. 2005] als Beispiel eines Vorgehensmodells zur Produktlinien-Entwicklung genauer erläutert, um im Anschluss daran eine Einordnung von DOORS, MERAN und CTE XL Professional innerhalb eines derartigen Vorgehens vorzunehmen. In einem weiteren Schritt werden verschiedene Ansätze für das Testen innerhalb der Software-Produktlinien-Entwicklung diskutiert, auf deren Grundlage abschließend das generische Verfahren für ein Variantenmanagement bei der Testfallbestimmung mithilfe der Klassifikationsbaum-Methode und dem CTE XL Professional entwickelt wird.

In Kapitel 4 werden für die Konzeption einer prototypischen Umsetzung die Anforderungen an das zu entwickelnde Variantenmanagement-Plug-Ing für CTE XL Professional ermittelt. Dabei wird zunächst eine allgemeine Zustandsbeschreibung des CTE XL Professional gegeben und im Anschluss daran werden die Anwendungsfälle und Anforderungen formuliert.

Zur Bewertung des entwickelten Verfahrens wird der Prototyp in Kapitel 5 auf Basis eines Bespielszenarios angewendet und analysiert. Anschließend werden die dabei gemachten Beobachtungen dokumentiert und bewertet.

Den Abschluss der Arbeit bildet das Kapitel 6 mit einer Zusammenfassung der gewonnenen Erkenntnisse und einem Ausblick auf zukünftige Arbeiten. Dabei werden sowohl technische Aspekte als auch theoretische Ansatzpunkte für die Weiterentwicklung des Variantenmanagements mithilfe der Klassifikationsbaum-Methode und dem CTE XL Professional angegeben.

2. Grundlagen der Qualitätssicherung

Eines der wichtigsten Verkaufsargumente für ein Produkt ist dessen Qualität. Da Software heutzutage in einer Vielzahl von Erzeugnissen komplexe Aufgaben übernimmt, ist insbesondere die Softwarequalität ein entscheidender Faktor für den Erfolg eines Produkts ([Spillner und Linz 2010]). Damit sind die stetige Qualitätsverbesserung und ein nachhaltiges Qualitätsmanagement für die Softwareentwicklung unerlässlich.

Es lassen sich zwei Arten des Qualitätsmanagement unterscheiden. Das produktorientierte Qualitätsmanagement beeinflusst die Qualität der Software und der Teilergebnisse des Entwicklungsprozesses direkt, wohingegen das prozessorientierte Qualitätsmanagement die Qualität der Software indirekt über die Qualität des Entwicklungsprozesses reguliert ([Balzert 2008]). Dabei kommen konstruktive und analytische Qualitätsmanagementmaßnahmen zum Einsatz. Zu den konstruktiven Maßnahmen zählen Methoden, Werkzeuge und Richtlinien, die sicherstellen, dass entstehende Artefakte beziehungsweise der Entwicklungsprozess an sich schon bei der Erstellung bestimmte Qualitätseigenschaften aufweisen. Im Gegensatz dazu messen die analytischen Maßnahmen das vorliegende Qualitätsniveau und identifizieren Defekte, was durch Rückkopplungen im Entwicklungsprozess zur Qualitätsverbesserung führt ([Balzert 2008]).

2.1. Requirements Engineering

Anforderungen legen die erwarteten Eigenschaften eines Softwaresystems fest ([Balzert 2009]). Falsche, missverständliche oder unvollständige Anforderungen behindern nicht nur den Entwicklungsprozess, sondern führen zu Fehlern und im ungünstigsten Fall sogar zum Projektabbruch.

Eine systematische standardisierte Anforderungsverarbeitung hilft Missverständnisse zu vermeiden, eine vollständige Spezifikation der Anforderungen zu erzeugen und somit die entstehenden Fehler zu minimieren. Die Aktivitäten der Anforderungsverarbeitung werden im englischen Begriff Requirements Engineering zusammengefasst und sind den konstruktiven Qualitätsmanagementmaßnahmen zuzuordnen. Dabei werden produktorientiertes und prozessorientiertes Vorgehen kombiniert. Abbildung 2.1 verdeutlicht die verschiedenen Aktivitäten und Artefakte des Requirements Engineering.

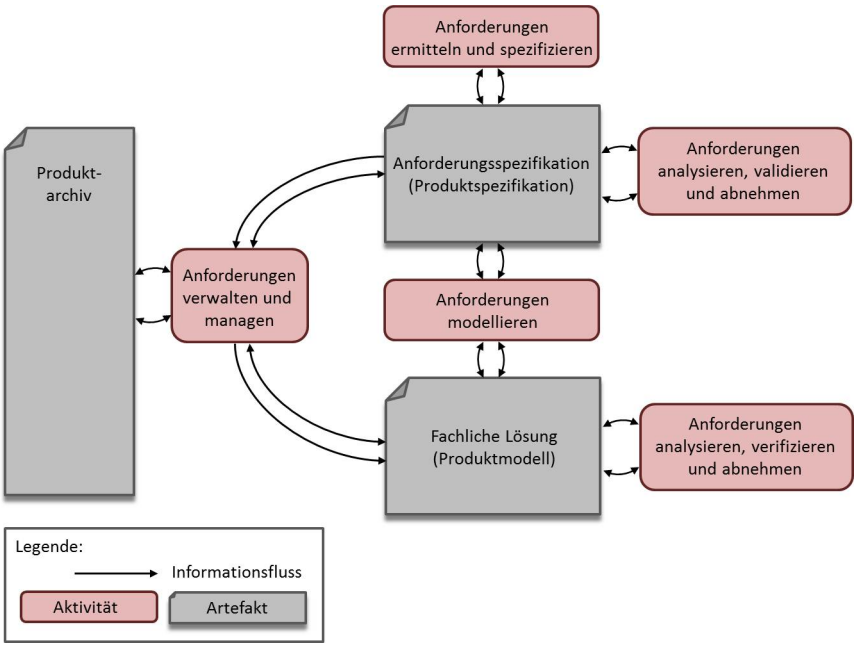

Abbildung 2.1.: Requirements Engineering (nach [Balzert 2009])

Den ersten Schritt bei der Erstellung einer Anforderungsspezifikation stellt die systematische Gewinnung der Anforderungen dar. Hierbei werden sowohl die Wünsche und Anforderungen der verschieden Akteure, als auch einzuhaltende Normen, Richtlinien und Standards berücksichtigt. Im nächsten Schritt werden die gesammelten Anforderungen mithilfe von Methoden, Richtlinien, Konventionen und Schablonen in eine standardisierte Form überführt. Dabei entsteht die Anforderungsspezifikation. Diese dient, nachdem sie auf Fehler und Missverständnisse überprüft wurde, als Grundlage für die Modellierung des Produktmodells. Das Produktmodell bildet die Anforderungsspezifikation in einer funktionalen, formalisierten Form ab und beinhaltet verschiedene Artefakte, wie zum Beispiel UML-Modelle [Object Management Group 1989] und Benutzeroberflächenkonzepte. Das entstandene Produktmodell wird in einem weiteren Schritt gegen die Spezifikation verifiziert um zu überprüfen, ob es mit dieser übereinstimmt. Alle Anforderungen werden im

Requirements Engineering durch ein Anforderungsmanagement verwaltet, welches Mechanismen zur Änderungsüberwachung, Nachverfolgbarkeit, Versionierung und für die Zugriffskontrolle bereitstellt ([Balzert 2009]).

2.1.1. Dynamic Object-Oriented Requirements System

Das Anforderungsmanagement wird häufig durch den Einsatz von Software-Werkzeugen unterstützt. Ein weit verbreiteter Vertreter dieser Klasse ist das Dynamic Object-Oriented Requirements System, das von der Firma Quality Systems and Software Ltd. erstmals 1993 veröffentlicht wurde und inzwischen von IBM unter dem Namen IBM® Rational® DOORS (DOORS) ([IBM 2011]) weiterentwickelt und vertrieben wird.

DOORS ist ein Client-Server-basiertes objektorientiertes Datenbanksystem, das einen eigenen Datenbankserver zur Speicherung von Anforderungen verwendet und zur Erweiterung und Anpassung des Funktionsumfanges die integrierte Skriptsprache DOORS eXtension Language (DXL) bereitstellt.

Anforderungen werden in DOORS als Objekte in sogenannten Modulen gespeichert. Neben formalen Modulen, die die Objekte enthalten, bietet DOORS Link-Module für die Verlinkung einzelner Objekte und beschreibende Module (*engl. Deskriptive Module*), die unstrukturierte Daten, wie zum Beispiel Notizen, E-Mails oder Interviews, aufnehmen. DOORS fasst Module in Ordnern und Projekten zusammen und bietet damit eine zusätzliche Strukturierungsebene. Abbildung 2.2 zeigt eine solche Projekt- und Ordnerstruktur im linken Teil des Fensters, sowie die darin enthaltenen Module im rechten Teil.

Innerhalb eines Moduls verleihen die Objekte der Anforderungsspezifikation ihre Struktur. Objekte erhalten in DOORS eine eindeutige Identifizierungsnummer (ID), die auch nach Löschen eines Objekts nicht wieder freigegeben wird. Dies ist Teil eines Backup- und Verwaltungsmechanismus. Objekte werden nicht endgültig gelöscht, sondern im Hintergrund weiter gespeichert. Bei der Wiederherstellung erhalten sie die alte ID zurück.

Abgesehen von vordefinierten Systemattributen wie der ID kann der Benutzer für die Objekte eines Moduls eigene Attribute mit verschiedenen Datentypen wie Integer, String, Text, etc. definieren und so beispielsweise auch UML-Diagramme und Grafiken einbinden. Damit können Schablonen, Vorlagen oder Standards des Requi-

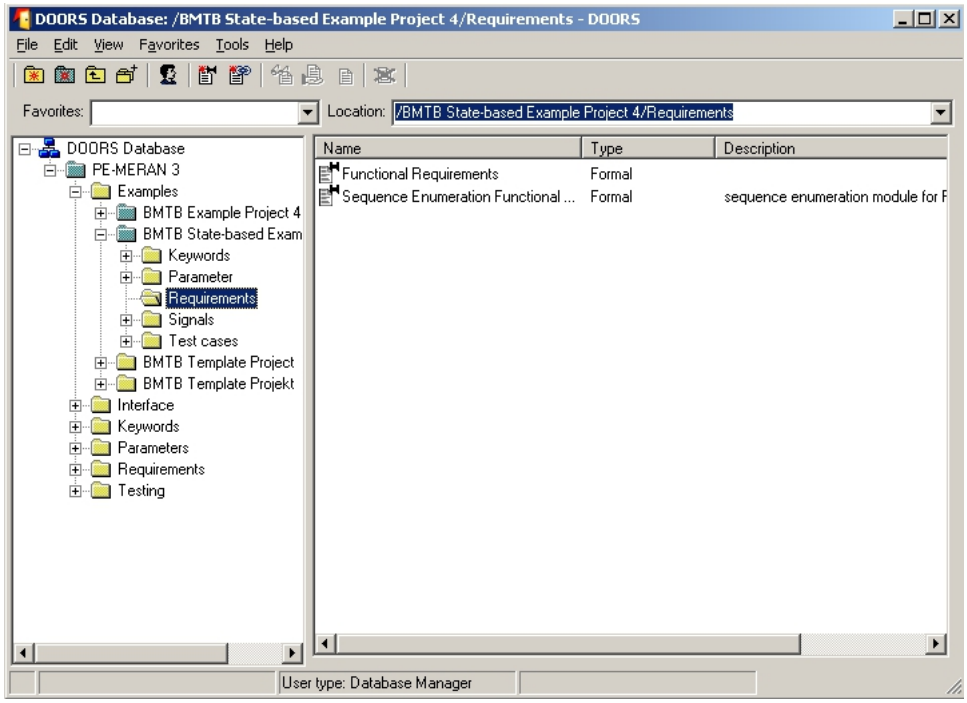

Abbildung 2.2.: DOORS Projekt- und Ordnerstruktur

rements Engineering durch individuelle Attributmengen abgebildet und die Anforderungsspezifikation richtlinienkonform gestaltet werden. Des Weiteren lassen sich auf diese Weise auch andere Arten von Spezifikationen, wie zum Beispiel Testfallspezifikationen, in einem formalen Test-Modul realisieren.

Abbildung 2.3 stellt einen Ausschnitt eines formalen Moduls in DOORS dar. Die Zeilen im rechten Teil des Fensters entsprechen den Objekten des Moduls, während die Spalten die einzelnen Attribute der Objekte widerspiegeln. DOORS verfügt über ein weitreichendes Rechtesystem, welches das Verwalten mehrerer Benutzer und Benutzergruppen erlaubt. Den verschiedenen Benutzern und Benutzergruppen können für die Projekte, Ordner, Module und Objekte unterschiedliche Zugriffsrechte zugeteilt werden. So lassen sich beispielsweise die Änderungsrechte von Benutzern für einzelne Objekte definieren.

Änderungen eines Objekts werden in einer History gespeichert. Eine aktuelle Version eines Moduls lässt sich als Baseline festhalten, die die History eines Moduls zurück bis zur letzten Baseline enthält. Dadurch können Unterschiede zwischen beliebigen Baselines dargestellt werden.

Um verschiedene Perspektiven auf die Objekte einer Anforderungsspezifikation zu erlauben, können in DOORS neben systemeigenen Views benutzerdefinierte Views

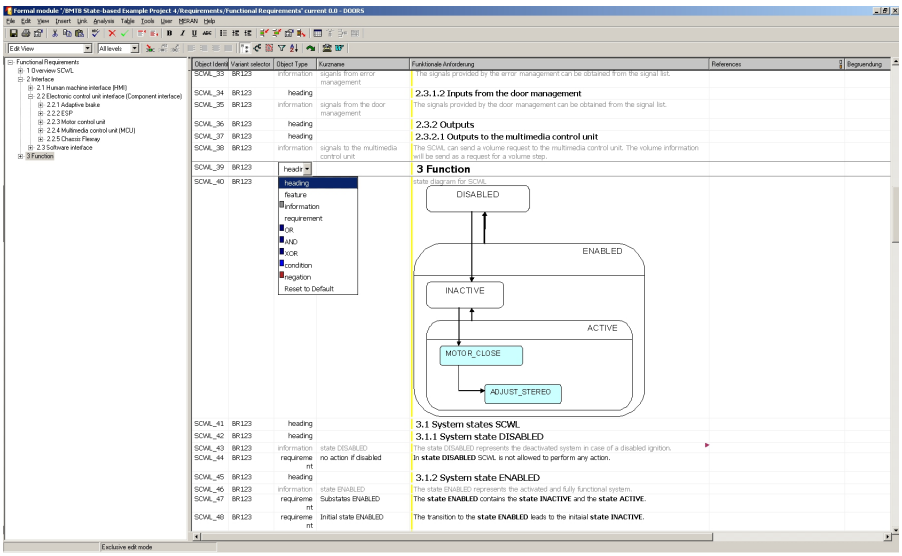

Abbildung 2.3.: Formales Modul in DOORS (siehe auch Abbildung B.1)

definiert werden. Während zum Beispiel der systemeigene View textitOutline die Module auf deren Überschriften verkürzt und alle anderen Objekte ausblendet, lassen sich in benutzerdefinierten Views auch Mengen von Attributen ausblenden, um eine übersichtliche und für jeden Benutzer angepasste Darstellung zu erreichen.

Eine der wichtigsten Funktionen in DOORS ist die Verlinkung zwischen Objekten. Dazu werden Start- und Zielobjekt definiert und die resultierende Verlinkung in Form von kleinen Pfeilen in der Zeile eines Objekts angezeigt. Verlinkungen zwischen zwei Modulen werden in einem Link-Modul gespeichert. Derartige Verlinkungen bilden die Grundlage für die Nachverfolgung von Anforderungen.

2.1.2. Anforderungsnachverfolgung

Anforderungen unterliegen über ihren gesamten Lebenszyklus hinweg ständigen Änderungen durch verschiedene Benutzergruppen. Meist sind Akteure unterschiedlicher Organisationen an einem Projekt beteiligt, die unterschiedliche Richtlinien und Vorgehensweisen zum Requirements Engineering verfolgen. Im Rahmen der Software-Produktlinien Entwicklung treten zudem mehrere Varianten des Endprodukts mit unterschiedlichen Anforderungen auf, was deren Verwaltung zusätzlich erschwert. Ein Maß für die Bewertung von Anforderungen und den Einfluss ihrer Änderungen auf spätere Artefakte im Entwicklungsprozess stellt die Nachverfolgbarkeit von Anforderungen (*engl. Requirements Traceability*) dar. Greenspan und McGowan [Greenspan und McGowan 1978] formulieren schon früh, dass die Verfolgung von Änderun-

gen einzelner Artefakte im Entwicklungsprozess eine wichtige Eigenschaft von Systembeschreibungstechniken darstellt. Die in diesem Zusammenhang am häufigsten zitierte Definition für Requirements Traceability geben [Gotel und Finkelstein 1994]:

> *„Requirements Traceability refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases).“*

[Ramesh und Jarke 2001] definieren Requirements Traceability als:

> *„. . . a characteristic of a system in which the requirements are clearly linked to their sources and to the artifacts created during the system development life cycle based on these requirements.“*

Beide Definitionen geben als Rahmen für die Anforderungsnachverfolgbarkeit den gesamten Entwicklungsprozess an, wobei [Gotel und Finkelstein 1994] die Richtung der Nachverfolgung einbeziehen und [Ramesh und Jarke 2001] die Verlinkung von Anforderungen mit anderen Artefakten des Entwicklungsprozesses als Grundlage für Nachverfolgbarkeit anführen.

Die Verlinkung von Anforderungen mit Artefakten und Objekten hat für die verschiedenen Entwicklungsphasen und Akteure eine unterschiedliche Semantik ([Ramesh und Jarke 2001]). So unterscheiden [Ramesh und Jarke 2001] die vier Linktypen:

- *Satisfaction Link*,
- *Evolution Link*,
- *Rationale Link* und
- *Dependency Link*

und beschreiben deren Eigenschaften.

Die Semantik von *Satisfaction Links* liegt in der Sicherstellung der Konsistenz zwischen den Endprodukten der verschiedenen Entwicklungsphasen. So führen Anforderungen beispielsweise zu Designentscheidungen. Mit *Satisfaction Links* lässt sich nun verfolgen, welche Designentscheidungen getroffen wurden, um eine Anforderung

zu erfüllen. Ein ähnliches Vorgehen gibt es für das Gesamtsystem, seine Subsysteme und Komponenten. Über *Satisfaction Links* können diese den Anforderungen zugeordnet werden, die sie erfüllen. Mithilfe von *Satisfaction Links* wird jedoch auch die Überdeckung kontrolliert. Dabei lässt sich analysieren, ob alle Anforderungen von Komponenten des Systems erfüllt werden, beziehungsweise alle Komponenten des Systems auch einer Anforderung zugeordnet sind. Bei der Verifikation von Anforderungen kann mithilfe von *Satisfaction Links* verfolgt werden, welche Testfälle für welche Anforderungen entwickelt wurden. So lässt sich anhand der Testergebnisse überprüfen, ob die Anforderungen wirklich erfüllt wurden.

Evolution Links werden zur Identifikation des Ursprungs von Artefakten und der darin enthaltenen Objekte verwendet. Damit soll das Verständnis für die Anforderungen und resultierende Artefakte und Objekte verbessert werden. Zudem wird die Historie von Artefakten verfolgt, das heißt, wer oder was für deren Erstellung, Änderung und Verfeinerung verantwortlich war. So kann zum Beispiel verfolgt werden, welche Designentscheidungen, ausgehend von Standards oder Richtlinien getroffen wurden, oder ob eine Anforderung aufgrund eines gescheiterten Test geändert wurde.

Die Erarbeitung, Modifikation oder Spezifikation von Artefakten und Objekten führt häufig zu Problemen oder Konflikten. Entscheidungen, die getroffen werden, um ein solches Problem beziehungsweise einen solchen Konflikt zu lösen, führen häufig zu Änderungen anderer Anforderungen, Artefakte und Objekte. Die Informationen über Entscheidungen, Annahmen und Interpretationen, die zur Problem- und Konfliktbeseitigung getroffen werden, lassen sich durch die Verwendung von *Rationale Links* festhalten. Mithilfe von *Rationale Links* lässt sich beispielsweise verfolgen, welche Anforderungen zu einem Konflikt geführt haben oder welche Alternativen bei der Entscheidungsfindung sich auf welches Problem, beziehungsweise welchen Konflikt beziehen.

Mit dem letzten Linktypen, den *Dependency Links*, werden Kompositionen, Hierarchien und Abhängigkeiten von Anforderungen, Artefakten und Objekten verfolgt. So kann unter anderem überprüft werden, ob eine Komponente Teil einer anderen Komponente ist oder von einem externen System abhängt.

2.1.3. MERAN

Ein Werkzeug, das die Generierung variantenspezifischer Anforderungsspezifikationen ermöglicht, ist das von Berner & Mattner entwickelte MERAN ([Berner&Mattner 2011]). MERAN erweitert die Funktionalität von DOORS durch einen Satz von DXL-Skripten. Wie in Abbildung 2.4 dargestellt, werden die in MERAN enthaltenen Funktionen über die DOORS-eigene Menüleiste bereitgestellt.

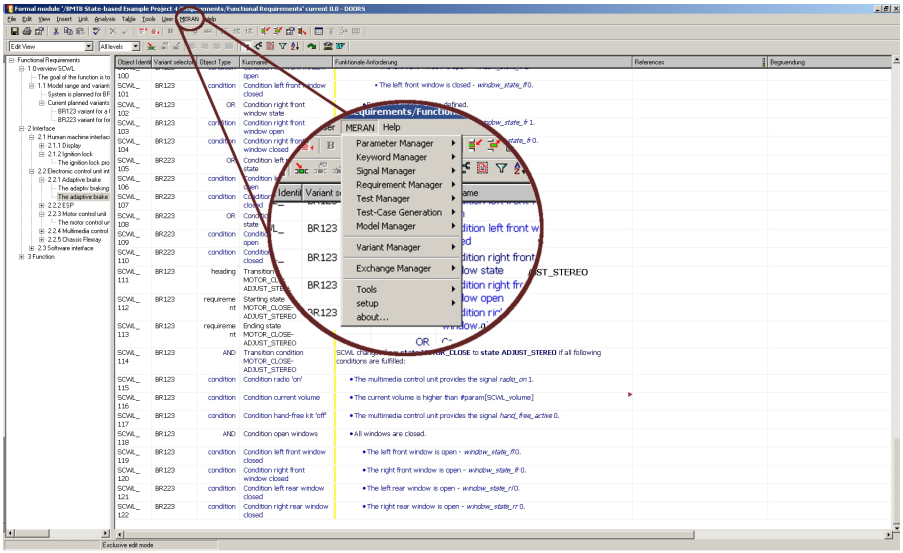

Abbildung 2.4.: MERAN-Einbindung in DOORS (siehe auch Abbildung B.2)

MERAN wurde zur Unterstützung des Variantenmanagements für das Requirements Engineering entwickelt und kann somit auch für das Domain Requirements Engineering innerhalb der Software-Produktlinien-Entwicklung eingesetzt werden. Durch die Automatisierung aufwendiger Prozessschritte wird eine Kosten- und Zeitersparnis für die frühen Analyse- und Spezifikationsphasen erreicht ([Meyer und Wegener 2010]). Dazu werden auf Basis einer generischen Anforderungsspezifikation mit den Mechanismen Selektion und Parameterersetzung variantenspezifische Teilspezifikationen abgeleitet. Zunächst spezifiziert der Benutzer alle relevanten Anforderungsobjekte in einem speziellen formalen Modul – dem generischen Modul. Dieses enthält sowohl Anforderungsobjekte, die alle Varianten gemeinsam haben, als auch Anforderungsobjekte, die nur für bestimmte Varianten zutreffen. Über ein besonderes Attribut, den Variantenselektor, legt der Benutzer die Zugehörigkeit eines Anforderungsobjekts zu einer Variante fest. Abbildung 2.5 stellt die beiden Möglichkeiten der Realisierung eines Variantenselektors dar.

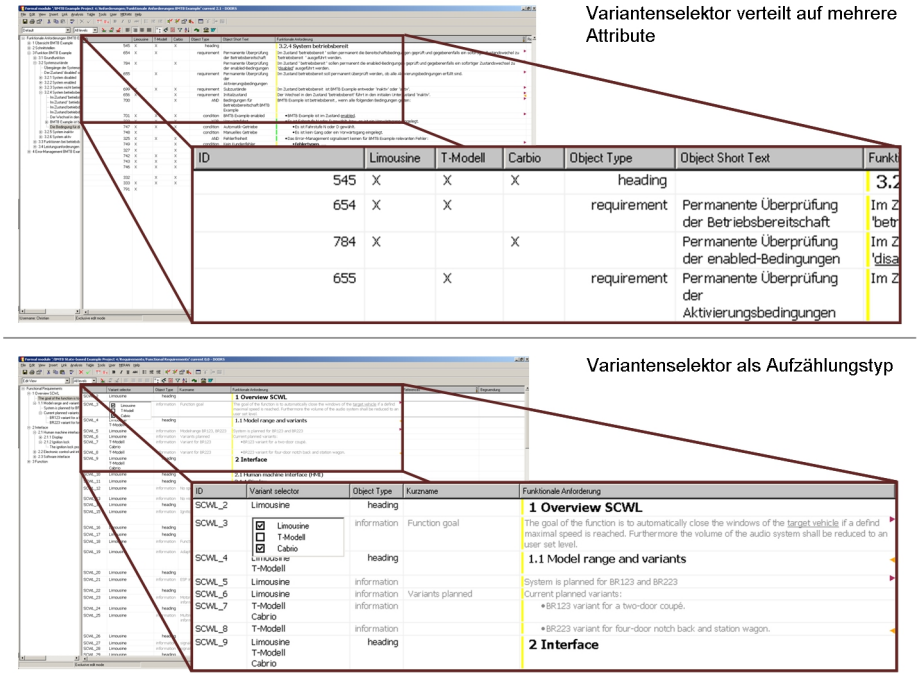

Abbildung 2.5.: Umsetzung eines Variantenselektors (siehe auch Abbildung B.3)

Im ersten Fall wird dem generischen Modul für jede Variante ein einstelliges Attribut hinzugefügt, welches die Zugehörigkeit zu der jeweiligen Variante anzeigt. Der zweite Fall benutzt als Variantenselektor ein Attribut des Aufzählungstypen *enum*, der gleichzeitig die Informationen über die Variantenzugehörigkeit zu allen Varianten festhält. In einem nächsten Schritt werden Parameter für die Anforderungsspezifikation definiert.

Parameter sind wichtige Unterscheidungskriterien für die Variantenerstellung. So lassen sich beispielsweise für die Software von Steuergeräten im Automobilbau die unterschiedlichen Höchstgeschwindigkeiten einzelner Fahrzeugvarianten mit dem Parameter *MAX Vsys* darstellen. Wie in Abbildung 2.6 abgebildet, werden Parameter mit ihren variantenspezifischen Werten von MERAN in einem formalen Parameter-Modul festgehalten und durch Platzhalter in den Anforderungsobjekten des generischen Moduls positioniert. Dabei können die Parameter neben numerischen Werten beliebige Zeichenketten, Bilddaten oder sogar OLE-Objekte als Parameterwert aufweisen.

Das so erstellte generische Modul und das Parameter-Modul bilden die Grundlage für die Generierung variantenspezifischer Teilspezifikationen. Diese werden in formalen Modulen für die einzelnen Varianten festgehalten. MERAN übernimmt dabei für jedes Varianten-Modul nur die Anforderungsobjekte, die der Variante mittels

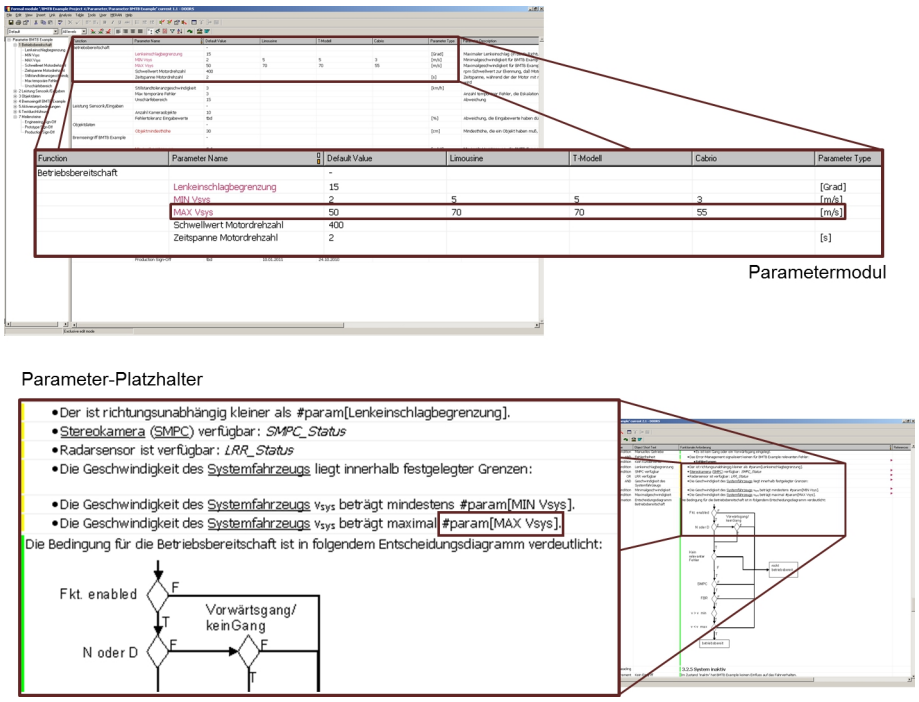

Abbildung 2.6.: Parametereinsatz in MERAN (siehe auch Abbildung B.4)

Variantenselektor zugewiesen wurden, und ersetzt die Parameter-Platzhalter durch die variantenspezifischen Parameterwert aus dem Parameter-Modul. Wurde kein Parameterwert für eine Variante spezifiziert, übernimmt MERAN einen festgelegten Default-Wert.

MERAN erlaubt so die zentrale Verwaltung und Pflege der Anforderungen und der dazugehörigen Parameter. So müssen Änderungen nur im generischen Modul oder im Parameter-Modul vorgenommen werden und können dann durch Synchronisation für alle Varianten-Module übernommen werden.

Weitere Funktionen von MERAN beinhalten die Überprüfung von bestehenden Verlinkungen zwischen Anforderungsobjekten und die Testfallgenerierung aus bestehenden Test-Modulen.

2.2. Testen und Methoden der Testfallermittlung

Unter den analytischen Qualitätsmanagementmaßnahmen für Software besitzt das dynamische Testen eine hohe praktische Bedeutung ([Liggesmeyer 2009]). Im Gegensatz zu statischen Testverfahren, die mithilfe von Reviews oder Programmtext-Analysatoren das Testobjekt überprüfen, bringt das dynamische Testen das Testobjekt unter Verwendung von Eingabedaten innerhalb einer Testumgebung zur Ausführung und vergleicht die erzielten Werte mit den erwarteten Werten. Da Testen eine komplexe und zeitaufwendige Aufgabe innerhalb des Entwicklungsprozesses darstellt, entfallen teilweise bis zu 50% des gesamten Entwicklungsaufwands auf Testaktivitäten ([Wegener 2001]).

Dynamische Testtechniken gehören zu den produktorientierten Verfahren und versuchen Fehler in fertiggestellten Softwareprodukten oder Teilprodukten zu identifizieren. Als Fehler werden hierbei Abweichung oder Nichterfüllung der spezifizierten Anforderungen sowie Inkonsistenzen innerhalb der Anforderungen bezeichnet ([Balzert 2008; Spillner und Linz 2010]). Eine Fehlerwirkung liegt vor, wenn bei der Ausführung der Software im Betrieb oder zu Testzwecken die Wirkung eines Fehlerzustands für den Anwender oder Tester sichtbar wird. Ein Fehlerzustand der Software liegt vor, wenn Teile des Programmcodes falsch beziehungsweise unvollständig sind und somit eine Fehlerwirkung hervorrufen. Darüber hinaus können menschliche Handlungen Fehlerwirkungen und Fehlerzustände verursachen, was als Fehlhandlung bezeichnet wird ([Spillner und Linz 2010]).

Verschiedene Vorgehensmodelle für die Softwareentwicklung messen dem Test unterschiedliche Bedeutung bei. Während beispielsweise das Wasserfallmodell ([Boehm 1981]) das Testen als abschließende Phase des Entwicklungsprozesses betrachtet, werden im V-Modell ([Boehm 1981]) die Testphasen Komponenten- beziehungsweise Modultest, Integrationstest, Systemtest und Abnahmetest unterschieden, die parallel zu den Entwicklungsphasen vorbereitet werden.

Wegener skizziert für jede Testphase des V-Modells die Aufteilung des Tests in einzelne Testaktivitäten als Grundlage für einen systematischen Test ([Wegener 2001]). Dazu werden die in Abbildung 2.7 dargestellten Aktivitäten durchgeführt.

Wegener unterscheidet hierbei die Kernaktivitäten *Testfallermittlung*, *Testdurchführung*, *Monitoring* und *Testauswertung* von den vorbereitenden und begleitenden Aktivitäten *Testplanung*, *Testorganisation* und *Testdokumentation*. Die *Testplanung*

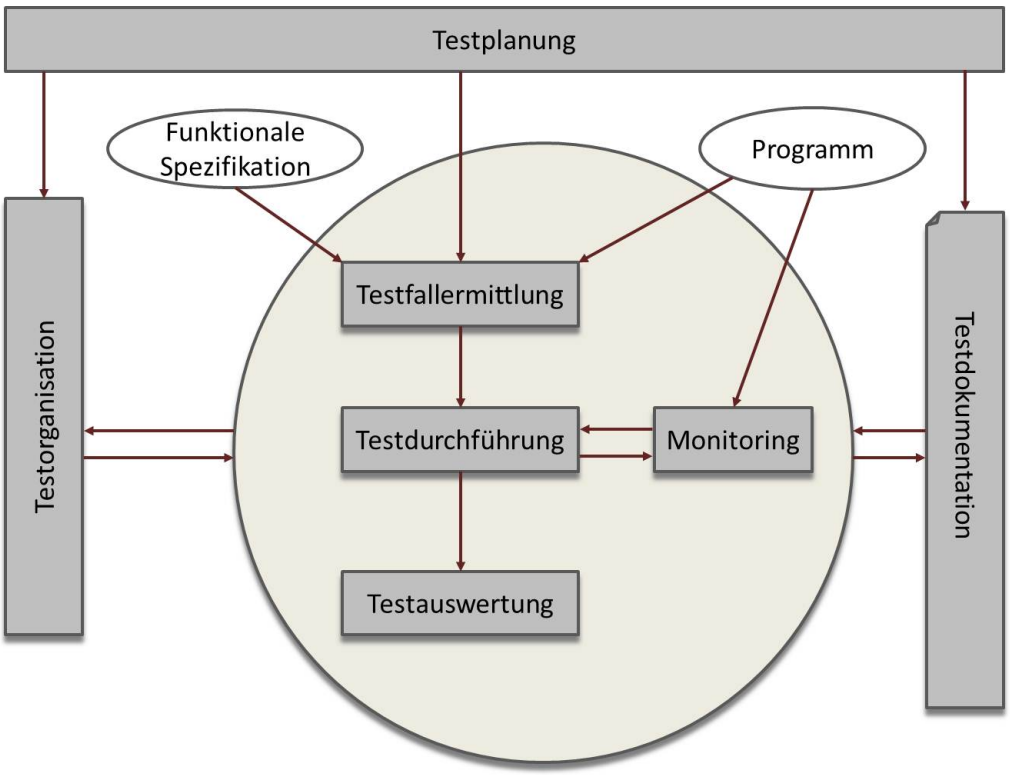

Abbildung 2.7.: Testaktivitäten (nach [Wegener 2001])

dient der Vorbereitung des Tests und legt die Vorgehensweise für den Test fest. Beispielsweise werden hier Kriterien für die Auswahl des Testobjektes definiert. Die *Testorganisation* umfasst neben der Datenverwaltung für Testobjekte, Testfälle und Testdaten die Bereitstellung der Testumgebung, in der das Testobjekt ausgeführt werden soll. Die Informationen und Ergebnisse, die während des Tests angefallen sind, werden mithilfe der *Testdokumentation* in eine verständliche und übersichtliche Form gebracht. Die *Testfallermittlung* widmet sich der Aufgabe, die Testfälle für die Prüfung eines Testobjekts zu definieren. Der Testfallermittlung ist die eigentliche *Testdurchführung* nachgelagert. Hierbei wird das Testobjekt mit auswählten Testdaten in einer Testumgebung zur Ausführung gebracht und die vom Testobjekt berechneten Werte und das Programmverhalten festgehalten. Während der *Testdurchführung* findet gleichzeitig das *Monitoring* zur Überwachung des Testablaufs und zur Kontrolle von Strukturtestkriterien statt, welches die Grundlage für den Ist-Soll-Abgleich innerhalb der *Testauswertung* bildet und Abweichungen im Zeitverhalten aufzeigt. Im letzten Schritt werden in der *Testauswertung* die erwarteten Werte mit den im Test erzielten Werten verglichen, sowie das Istverhalten und das Sollverhalten gegenübergestellt, um die Testergebnisse zu ermitteln. ([Wegener 2001])

Innerhalb dieser Testaktivitäten stellt der Testfall ein wesentliches Konzept dar. Der IEEE Standard 610 [IEEE 1990] definiert den Testfall als:

> *„A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.“*

[Spillner und Linz 2010] erweitern die IEEE Definition, indem sie:

- eine Menge von Vorbedingungen, als Beschreibung der Ausgangssituation,
- die Eingabedaten,
- eine Menge von Randbedingungen für die Testdurchführung,
- die erwarteten Ergebnisse und das erwartete Verhalten des Testobjekts und
- eine Menge von Nachbedingungen

als Bestandteile eines Testfalls definieren, wobei sie die mit dem Testfall verfolgte Zielsetzung vernachlässigen. Ein auf diese Weise spezifizierter Testfall abstrahiert von konkreten Eingabewerten und wird daher als logischer Testfall bezeichnet. Logische Testfälle werden in einem weiteren Schritt in konkrete Testfälle überführt, indem die tatsächlichen Eingabewerte festgelegt werden.

Die Bestimmung aller relevanten Testfälle stellt innerhalb der Testphase die wichtigste Aktivität dar. Ein vollständiger Test mit allen möglichen Testfällen ist in der Regel für moderne Softwaresysteme praktisch nicht durchführbar. Daher wird mit einem stichprobenartigen Vorgehen versucht, möglichst viele Fehlerwirkungen zu identifizieren und dabei alle Funktionen abzudecken. Ein systematisches Vorgehen zur Testfallermittlung bieten hierbei die Testfallentwurfsverfahren, die in Blackbox-, Whitebox- und Greybox-Verfahren unterteilt werden. Abbildung 2.8 stellt Blackbox- und Whitebox-Verfahren und deren Ablauf schematisch dar.

Beide Verfahren werden während des dynamischen Testens angewendet, das bedeutet, sie gehen von der Ausführung des Testobjektes auf Eingabedaten und der damit verbundenen Berechnung von Ausgabedaten aus. Der Unterschied liegt in der Ermittlung der Testfälle und den damit beabsichtigten Zielen. Bei der Verwendung von Blackbox-Verfahren basiert die Testfallermittlung auf einer Analyse der Spezifikation ([Spillner und Linz 2010]). Dabei wird von der tatsächlichen internen Struktur des Testobjekts abstrahiert und nur das äußere Verhalten und die Ausgabedaten des

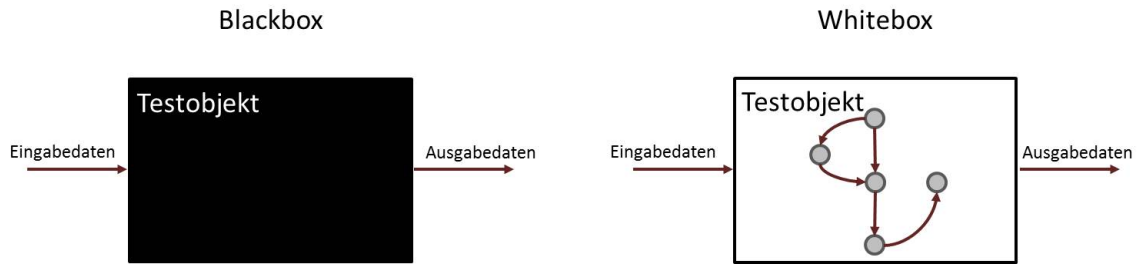

Abbildung 2.8.: Vergleich Blackbox- und Whitebox-Verfahren

Testobjekts geprüft. Hiermit lassen sich beispielsweise auf Ebene des Komponententests aufgetretene Fehlerwirkungen genau der getesteten Komponente zuordnen.

Im Gegensatz dazu legen Whitebox-Verfahren das Hauptaugenmerk auf die Analyse der internen Struktur des Testobjekts ([Spillner und Linz 2010]). Die Testfälle werden anhand der vorhandenen Programmstruktur festgelegt, um zum Beispiel in kontrollflussbezogenen Verfahren die Überdeckung aller möglichen Pfade im Kontrollflussgraphen zu überprüfen. Ein weiteres Beispiel für Blackbox-Verfahren stellen die datenflussbezogenen Verfahren dar, die das kontrollflussbezogene Verfahren um die Untersuchung des Variablenzugriffs in verschiedenen Ausführungszuständen erweitern.

Als Beispiele für die Klasse der Blackbox-Verfahren führen [Spillner und Linz 2010] unter anderem die Äquivalenzklassenbildung, den zustandsbezogenen Test, den anwendungsfallbasierten Test und die im nächsten Abschnitt näher erläuterte Klassifikationsbaum-Methode an.

Für eine vollständige Darstellung sei an dieser Stelle auch die Klasse der Greybox-Verfahren genannt, die eine partielle Kenntnis der internen Struktur des Testobjektes erfordert und somit eine Mischform der oben genannten Verfahren bildet ([Conrad 2004]).

2.2.1. Die Klassifikationsbaum-Methode

Der Eingabedatenraum eines Testobjekts kann selbst bei geringer Komplexität des Prüflings auf ein Maß anwachsen, das den vollständigen Test aller möglichen Eingabewerte unmöglich macht und die Testfallbestimmung zu einer äußerst komplexen Aufgabe anschwellen lässt. Beim dynamischen Testen wird daher häufig die Äquivalenzklassenbildung verwendet, um auf Basis der funktionalen Spezifikation

den Eingabedatenraum zu partitionieren und auf diese Weise die Anzahl der möglichen Testfälle zu beschränken. Hierzu werden die Eingabewerte in verschiedene Äquivalenzklassen aufgeteilt, deren Elemente das gleiche funktionale Verhalten des Testobjekts hervorrufen. Durch die spezifikationsbasierte Äquivalenzklassenbildung kann sichergestellt werden, dass der Test die spezifizierten Funktionen anhand der zugeordneten Äquivalenzklassen abdeckt. ([Liggesmeyer 2009])

Insbesondere für die Identifizierung von Fehlern wird davon ausgegangen, dass Fehlerwirkungen, die von einem Vertreter der Äquivalenzklasse aufgedeckt werden, auch von allen anderen Elementen der Klasse identifiziert werden ([Spillner und Linz 2010]). Somit genügt der Test mit einem Repräsentanten der Äquivalenzklasse um die Korrektheit der dazugehörigen Funktion für alle Elemente der Klasse sicherzustellen.

Die von [Ostrand und Balcer 1988] entwickelte Category-Partition Method verfeinert diesen Ansatz, indem sie ein systematisches Vorgehen für die Bildung der Äquivalenzklassen beschreibt und eine Sprache für die formale Beschreibung von Testfällen definiert. Grundlage für die Testfallbestimmung ist wiederum die funktionale Spezifikation, die in einem ersten Schritt in funktionale Einheiten zerlegt wird, die unabhängig voneinander getestet werden können. Daran anschließend werden Parameter und Umgebungseinflüsse identifiziert, die Auswirkung auf die Ausführung einer funktionalen Einheit haben. Diese werden anhand ihrer Eigenschaften in Kategorien (*categories*) unterteilt. Jede Kategorie wird in einem letzten Schritt in disjunkte Äquivalenzklassen (*choices*) zerlegt, die die möglichen Eingabewerte für diese Kategorie beinhalten.

Durch die Kombination von Vertretern der verschiedenen Äquivalenzklassen werden die Testfälle gebildet ([Ostrand und Balcer 1988]).

Da die *choices* verschiedener Kategorien häufig in Wechselwirkung miteinander stehen, muss beschrieben werden, welche Kombinationen von *choices* erlaubt, beziehungsweise sinnvoll sind. Dies geschieht durch die Definition von sogenannten *Constraints*, die beispielsweise angeben, dass eine *choice* nicht gemeinsam mit einer anderen *choice* innerhalb eines Testfalls auftreten darf ([Balcer 1988]). Die *Constraints* werden dabei durch einfache logische Formeln ausgedrückt.

Das systematische Vorgehen mithilfe der Category-Partition Method führt nicht zwangsläufig zur korrekten Zerlegung des Eingabedatenraums, da die einzelnen Dekompositionsschritte ein hohes Maß an Erfahrung des Testers erfordern.

Einen intuitiveren Ansatz, der auf der Category-Partition Method basiert, stellen [Grochtmann und Grimm 1993] mit der Klassifikationsbaum-Methode vor. Neben dem systematischen Testfallentwurf zeichnet sich die Klassifikationsbaum-Methode insbesondere durch eine kompakte und nachvollziehbare Darstellung des gesamten Testumfangs aus ([Lehmann und Wegener 2000]).

In einem schrittweisen und grafisch gestützten Vorgehen werden ähnlich zur Category-Partition Method Äquivalenzklassen gebildet, aus denen die Testfälle abgeleitet werden ([Wegener 2001]). Die Basis hierfür bildet die funktionale Spezifikation des Testobjekts, die vom Tester in einem ersten Schritt zur Identifikation testrelevanter Gesichtspunkte analysiert wird. Die identifizierten Gesichtpunkte dienen in einem nächsten Schritt der Zerlegung des Eingabedatenraums. Dabei klassifiziert jeder Gesichtspunkt einen bestimmten Bereich des Eingabedatenraums und bildet auf diese Weise eine Klassifikation. Die Klassifikationen teilen die Menge der möglichen Eingaben in disjunkte und vollständige Teilmengen (Klassen). Die Klassen können, falls es der Anwendungsfall erfordert, durch einen weiteren Klassifikationsschritt noch genauer unterteilt werden. Dieses Vorgehen lässt sich über mehrere Ebenen rekursiv wiederholen, wodurch sukzessive ein Klassifikationsbaum entsteht. Im Anschluss an den Klassifikationsbaum-Entwurf werden die Testfälle gebildet. Ein Testfall entsteht durch die Kombination von Klassen verschiedener Klassifikationen, wobei aus jeder Klassifikation genau eine Klasse berücksichtigt wird. Auf diese Weise kann eine vollständige Testfallmenge bezüglich der gewählten Klassifikationen effektiv entwickelt werden. ([Grochtmann und Grimm 1993; Wegener 2001])

Zum besseren Verständnis wird an dieser Stelle ein Beispiel zur Anwendung der Klassifikationsbaum-Methode aus dem Bereich des Automobilbaus erläutert. Die Domäne des Automobilbaus soll hierbei den praktischen Nutzen für industrielle Anwendungen verdeutlichen und zieht sich in Form von Beispielen auch durch die späteren Kapitel dieser Arbeit.

Im Folgenden soll ein Spurhalte-Assistenz-System für Fahrzeuge gestestet werden. Das Testobjekt wird als Wurzel in den Klassifikationsbaum aufgenommen. Der Wurzelknoten wird hierbei als Komposition der darunterliegenden testrelevanten Aspekte aufgefasst. Für den Test des Spurhalte-Assistenten wurden vier relevante Aspekte identifiziert:

- der Zustand der Fahrbahnmarkierung,
- der Straßenverlauf,
- der vom Fahrer getätigte Lenkeinschlag und
- die gefahrene Geschwindigkeit.

Diese vier Gesichtspunkte werden als Klassifikationen unterhalb des Wurzelknoten in den Klassifikationsbaum eingefügt. Den Klassifikationen werden im nächsten Schritt die zugehörigen Klassen zugeordnet. Dazu wird ein Repräsentant jeder Klasse als Blatt unterhalb der jeweiligen Klassifikation eingefügt. Anschließend lässt sich die Klasse *kurve* aus der Klassifikation *Straßenverlauf* durch die Klassifikation *Kurvenwinkel* und deren Klassen weiter verfeinern. Das Ergebnis ist der in Abbildung 2.9 dargestellte Klassifikationsbaum.

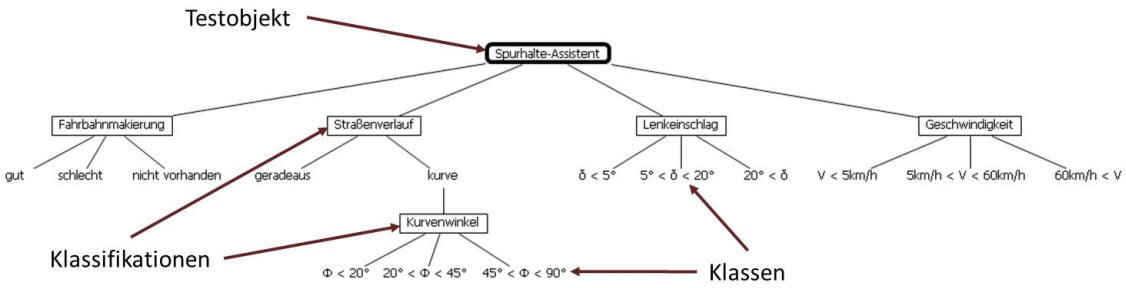

Abbildung 2.9.: Klassifikationsbaum des Spurhalte-Assistenten

Auf Basis dieses Klassifikationsbaumes werden nun die Testfälle mithilfe einer Kombinationstabelle gebildet. Jede Zeile der Tabelle stellt einen Testfall und jede Spalte eine nicht weiter verfeinerte Klasse dar (vgl. Abbildung 2.10). Der Testfall 1 kombiniert beispielsweise die Klasse gut, geradeaus, $\delta < 5°$ und V < 5km/h.

Die vier dargestellten Testfälle bilden gemeinsam das Minimalkriterium, das heißt jede Klasse wird in mindestens einem Testfall berücksichtigt. Die vollständige Kombination aller Klassen würde hingegen 108 Testfälle erzeugen. Die vollständige Kombination ist aber nicht in jedem Fall entscheidend für einen guten Test, vielmehr

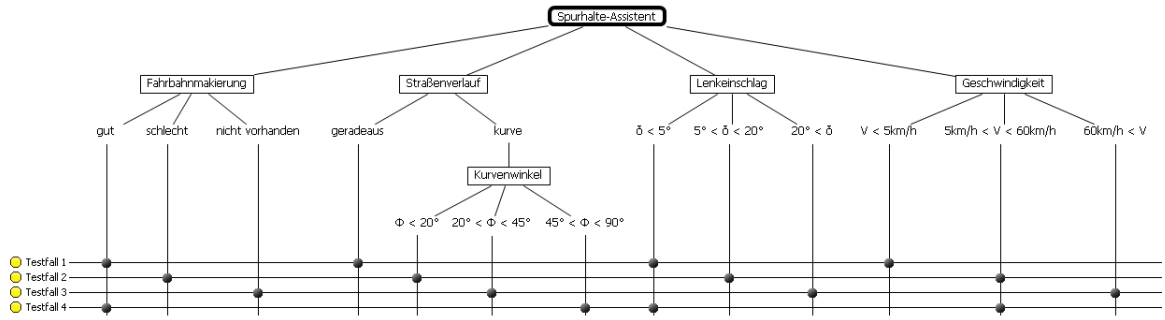

Abbildung 2.10.: Kombinationstabelle für das Beispiel Spurhalte-Assistent

versucht der Tester durch geschickte Kombination der Klassen solche Testmengen zu erzeugen, die möglichst fehlerträchtige Situationen enthalten ([Wegener 2001]).

Durch ihre Domänenunabhängigkeit hat die Klassifikationsbaum-Methode Einzug in viele Anwendungsbereiche gehalten. In Kombination mit dem Classification Tree Editor eXtended Logics als Werkzeugunterstützung konnte sie beispielsweise erfolgreich bei der Softwareentwicklung in den Bereichen Luftfahrt, Schienentransportwesen, Verteidigungselektronik und Automobilbau eingesetzt werden ([Lehmann und Wegener 2000]). Das systematische und strukturierte Vorgehen hilft bei der Vermeidung redundanter Testfälle und die grafische Darstellungsform erlaubt einen schnellen Überblick über die für das Testobjekt relevanten Testaspekte.

2.2.2. Classification Tree Editor eXtended Logics

Für die Softwareunterstützung der Klassifikationsbaum-Methode wurde in der Forschungsabteilung der Daimler-Benz AG 1993 die erste Version des Classification Tree Editor (CTE) ([Grochtmann und Grimm 1993]) entwickelt. In den Folgejahren wurde der CTE durch Mitarbeiter der DaimlerChrysler AG zum Classification Tree Editor eXtended Logics (CTE XL) ([Lehmann und Wegener 2000]) erweitert, der seit 2008 durch Berner & Mattner weiterentwickelt wird und inzwischen als Classification Tree Editor eXtended Logics Professional (CTE XL Prof.) erhältlich ist.

Der CTE XL Prof. wurde mithilfe des Eclipse Rich Client Plattform Frameworks ([Eclipse-Foundation 2011b]) realisiert, welches zur Entwicklung grafischer Softwaresysteme eingesetzt wird. Er bietet eine grafische Benutzeroberfläche, die die effiziente Erstellung und Verwaltung von Klassifikationsbäumen bereitstellt und sowohl die manuelle als auch die automatische Erzeugung von Testfällen erlaubt. Die auto-

matische Testfallerzeugung kann dabei mithilfe von Abhängigkeits- und Kombinationsregeln sinnvoll gesteuert werden.

Abhängigkeitsregeln definieren, welche Kombinationen von Klassen einen für das betrachtete Testobjekt gültigen Testfall bilden, und sind somit mit den Constraints aus der Category-Partition Methode gleichzusetzen. Hierbei werden implizite und explizite Abhängigkeitsregeln unterschieden. Eine implizite Regel stellt beispielsweise die in Abschnitt 2.2.1 erläuterte einmalige Berücksichtigung von Klassen für einen Testfall dar, welche innerhalb des Testfalls genau eine Klasse pro Klassifikation erlaubt. Derartige Regeln, die aus der Klassifikationsbaum-Methode abgeleitet wurden, sind im Quellcode des CTE XL Prof. fest verankert und werden somit bei dessen Ausführung immer angewendet. Demgegenüber stehen explizite Abhängigkeitsregeln, die der Benutzer während der Arbeit mit dem CTE XL Prof. in Form von logischen Ausdrücken im integrierten Abhängigkeitsregel-Editor definiert. Explizite Abhängigkeitsregeln bilden zusätzliches Domänenwissen bezüglich des Testobjekts ab und schränken somit die Menge der gültigen Testfälle weiter ein. So lässt sich beispielsweise für das Beispiel des Spurhalte-Assistenten die Regel:

$$45° < \sigma < 90° \Rightarrow V < 5km/h \; OR \; 5km/h < V < 60km/h$$

definieren, die bei einem Kurvenwinkel zwischen 45° und 90° nur Geschwindigkeit kleiner 5 km/h beziehungsweise kleiner 60 km/h erlaubt. Eine derartige Regel ist unter sicherheitskritischen Aspekten sinnvoll, kann aber auch auf Erfahrungen des Testers basieren, der in diesem Geschwindigkeitsbereich möglicherweise die meisten Fehler vermutet.

Innerhalb einer bereits erstellten Testfallmenge überprüft der CTE XL Prof. mithilfe des sogenannten Rule Checkers, welche Testfälle den definierten expliziten Abhängigkeitsregeln genügen. Hierbei werden ungültige Testfälle rot und gültige Testfälle grün markiert (vgl. Abbildung 2.11). Mit aktiviertem Rule Checker werden bei der automatischen Testfallgenerierung nur regelkonforme Testfälle erzeugt.

Einen weiteren Steuerungsmechanismus für die automatische Testfallgenerierung stellen Kombinationsregeln dar, die festlegen, welche Klassifikationen und Klassen bei der Testfallgenerierung berücksichtigt werden sollen und wie diese miteinander zu kombinieren sind. Neben der Option vollständige oder minimale Testfallmengen über den gewählten Baumelementen zu generieren, bietet der CTE XL Prof. auch die

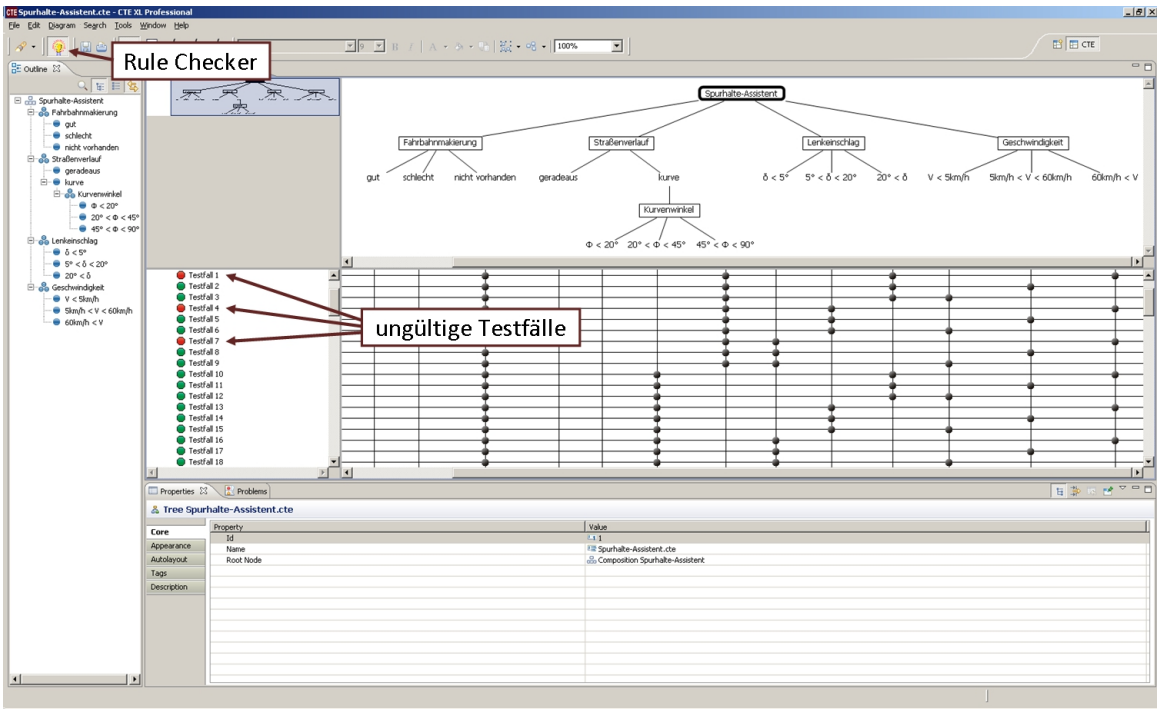

Abbildung 2.11.: Markierung ungültiger Testfälle

Möglichkeit nur eine Tupel, beziehungsweise ein Tripel aus der Menge der gewählten Baumelemente vollständig zu kombinieren, um die Testfallanzahl einzuschränken.

Als Ergänzung zu den Mitteln der automatischen Testfallgenerierung stellt der CTE XL Prof. weitere Werkzeuge zur Unterstützung des Testers bereit. So erlaubt ein integrierter Tag-Manager die Annotation aller Baum- und Testelemente mit zusätzlichen Meta-Informationen, eine Statistikfunktion listet die statistischen Merkmale des Baumes und der ermittelten Testfälle auf und eine Analysefunktion (der sogenannte Inspector) überprüft die Testfälle bezüglich vorhandener Duplikate und auf Vollständigkeit der Kombination innerhalb eines Testfalls. Erstellte Klassifikationsbäume und die dazugehörigen Testfälle können mithilfe einer Exportfunktion in andere Dateiformate überführt werden. Um die Interoperabilität zwischen den Werkzeugen des Entwicklungsprozesses zu gewährleisten, bietet der CTE XL Prof. sowohl eine Schnittstelle zu Quality Center von HP als auch zu DOORS.

2.3. DOORS-Schnittstelle des CTE XL Prof.

Zur Unterstützung der Anforderungsnachverfolgung von den Anforderungen in DOORS bis hin zu Testfällen im CTE XL Prof. wurde bei Berner & Mattner eine

Schnittstelle zwischen beiden Werkzeugen entwickelt und in den CTE XL Prof. integriert. Die Schnittstelle verwendet den DOORS-Client um eine Verbindung zu einem DOORS-Server herzustellen und mit diesem zu kommunizieren. Dazu wird im CTE XL Prof. die Requirements Interface View geöffnet, welche nach der Eingabe korrekter Log-In-Daten die Verbindung zu einem DOORS-Server herstellt. Nachdem die Verbindung erfolgreich aufgebaut wurde, zeigt die Requirements Interface View die auf dem DOORS-Server vorhandenen Module an, aus denen der Benutzer im nächsten Schritt das Modul auswählt, dessen Anforderungen mit Baumelementen oder Testfällen im CTE XL Prof. verlinkt werden sollen. Anschließend kann der Benutzer die Anforderungen dieses Moduls per Drag'n'Drop aus der Requirements Interface View auf die Baumelemente des Klassifikationsbaums beziehungsweise auf die Testfälle ziehen und so eine Verlinkung erstellen (vgl. Abbildung 2.12).

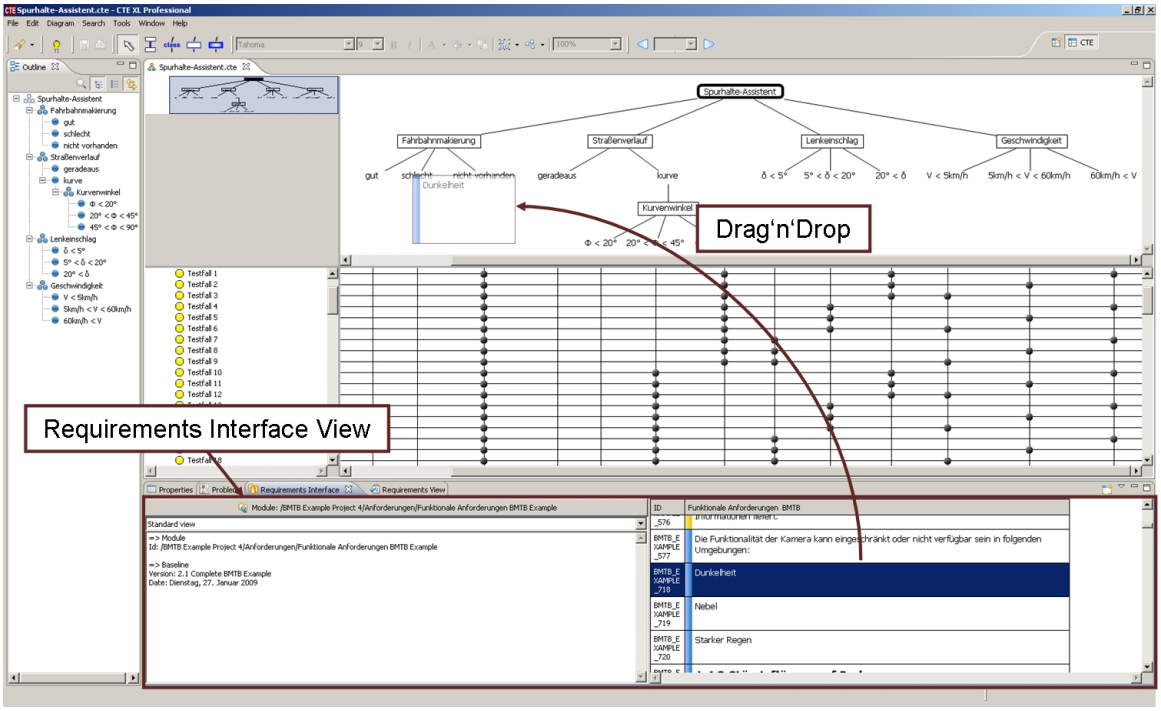

Abbildung 2.12.: Drag'n'Drop Verlinkung

Wie in Abschnitt 2.1.2 bereits erläutert, können derartige Verlinkungen verschiedene Bedeutungen für die Anforderungsnachverfolgung aufweisen. Wurden in DOORS beispielsweise schon Testfälle spezifiziert und mit Anforderungen verlinkt, handelt es sich dabei um Satisfaction Links. Wohingegen die Verlinkung eines derartigen DOORS-Testfalls mit einem Testfall im CTE XL Prof. einen Evolution Link darstellt. Enthält der DOORS-Server keine Testfallspezifikationen, lassen sich mit der DOORS-Schnittstelle direkt Satisfaction Links zwischen Anforderungen in DOORS und Testfällen im CTE XL Prof. erstellen. Außerdem bietet die Schnittstelle die

Möglichkeit Satisfaction Links, beziehungsweise Evolution Links zwischen Anforderungen in DOORS und Baumelementen im CTE XL Prof. zu definieren (vgl. Abbildung 2.13).

Nach der Erstellung werden die Verlinkungen in der Requirements View angezeigt. Diese bietet, neben Sortier- und Filterfunktionen für die Modifikation der Anzeige und einer grafischen Darstellung der Verlinkungen, die Möglichkeit die Anforderungen mit dem DOORS-Server zu synchronisieren. Während der Synchronisation wird überprüft, ob die in der Requirements View angezeigten Anforderungen noch mit denen in DOORS übereinstimmen. Wurden Anforderungen in DOORS geändert, zeigt die Requirements View die Änderungen an und überlässt dem Benutzer die Entscheidung, ob diese auch im CTE XL Prof. übernommen werden sollen. Bei der Änderung von verlinkten Anforderungen kann der Benutzer so überprüfen, ob auch die verlinkten Baumelemente oder Testfälle einer Änderung bedürfen und ob die Verlinkung immer noch sinnvoll ist. Die DOORS-Schnittstelle des CTE XL Prof. berücksichtigt nicht nur Anforderungsänderungen bei der Testfallermittlung, vielmehr vervollständigt sie durch die erstellten Verlinkungen das Gesamtbild der Anforderungsnachverfolgung innerhalb des Entwicklungsprozesses.

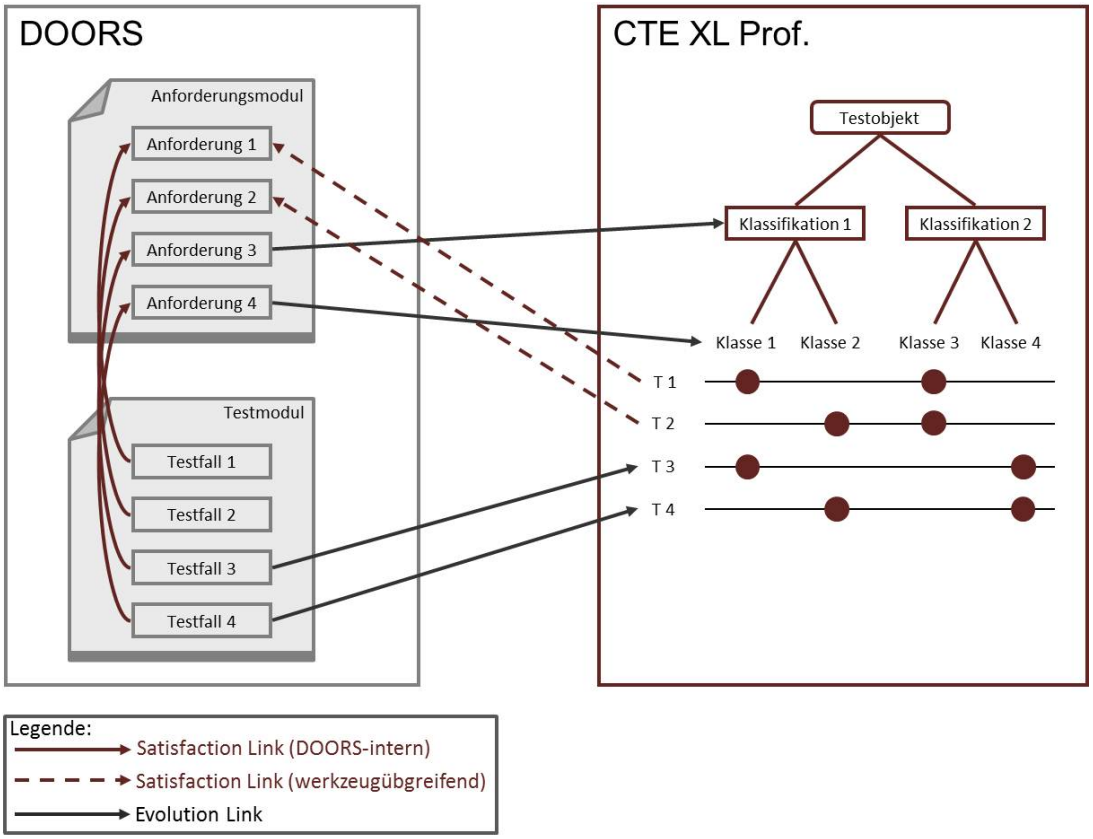

Abbildung 2.13.: Verlinkungen mit der CTE-DOORS-Schnittstelle

3. Variantenmanagement mit dem CTE XL Professional

„Any customer can have a car painted any colour that he wants so long as it is black.“

Das Zitat von Henry Ford ([Ford 2007]) aus dem Jahr 1909 spiegelt einen wesentlichen Nachteil der Massenfertigung wider. Hatte Ford mit der Serienfertigung des Modell T zwar die Grundlage geschaffen Produkte günstig und in großer Stückzahl herstellen zu können, so reduzierte sich damit gleichzeitig die Möglichkeit der Diversifikation.

Die gleiche Problematik lässt sich auch im Bereich der Software Herstellung beobachten. Während die Entwicklung von kundenspezifischer Individualsoftware mit sehr großem zeitlichem und finanziellem Aufwand behaftet ist, kann die günstige und schnelle Produktion von Standardsoftware nur ungenügend auf die Bedürfnisse des Kunden zugeschnitten werden ([Balzert 2008]).

Da die Bedürfnisse und Wünsche des Kunden heutzutage immer stärker in den Mittelpunkt rücken, wurden die Vorteile beider Produktionsformen kombiniert. Auf diese Weise entstand der Ansatz der individuellen Massenfertigung (*engl. Mass Customization*), die eine günstige Massenproduktion von Sachgütern unter der Berücksichtigung individueller Kundenwünsche und -bedürfnisse erlaubt. [Davis 1997].

Für die Unternehmen, die den Ansatz der Mass Customization verfolgen, entsteht jedoch zunächst ein höherer technologischer Entwicklungsaufwand, eine Umstellung der Entwicklungs- und Organisationsprozesse und somit niedrigere Profitmargen für individualisierte Produkte. Um derartige Nachteile zu vermeiden, haben viele Hersteller, insbesondere in der Automobilindustrie, das Plattformprinzip für die Fertigung kompletter Produktlinien eingeführt ([Pohl u. a. 2005]). Dabei werden Teile, die für mehrere Produktvarianten der Produktlinie verwendet werden können zu

einer Plattform zusammengefasst. Die Produktvarianten entstehen dabei durch die Kombination von Plattformteilen mit individualisierten variantenspezifischen Teilen.

Da der Anteil an Software gerade in komplexen Produkten immer mehr zunimmt und diese dadurch auch der wachsenden Variabilität ausgesetzt ist, wurden die Methoden der Produktionstechnik für den Bereich der Softwareentwicklung adaptiert. Dabei entstand der Ansatz der Software-Produktlinien für die Entwicklung softwareintensiver Systeme und klassischer Softwareprodukte ([Pohl u. a. 2005]).

Mit den in Kapitel 2 vorgestellten Werkzeugen DOORS, MERAN und CTE XL Prof. kann sowohl die produktionstechnische als auch die softwaretechnische Produktlinien-Entwicklung unterstützt werden. Aufgrund der fachlichen Ausrichtung dieser Arbeit wird der Fokus im Folgenden auf den Bereich der softwaretechnischen Produktlinien-Entwicklung gelegt.

Um DOORS, MERAN und CTE XL innerhalb dieses Kontextes besser einordnen zu können und eine Basis für den Entwurf eines Verfahrens zum Variantenmanagement mithilfe der Klassifikationsbaum-Methode und dem CTE XL Prof. zu schaffen, werden in Abschnitt 3.1 zunächst die Grundlagen der Software-Produktlinien-Entwicklung dargelegt. Im Anschluss daran wird das Software-Produktlinien-Framework nach [Pohl u. a. 2005] als Beispiel eines Vorgehensmodells zur Produktlinien-Entwicklung genauer erläutert, um in Abschnitt 3.3 die Einordnung von DOORS, MERAN und CTE XL Prof. innerhalb eines solchen Vorgehens vorzunehmen. In Abschnitt 3.4 werden in einem nächsten Schritt verschiedene Ansätze für das Testen innerhalb der Software-Produktlinien-Entwicklung diskutiert, auf deren Grundlage in Abschnitt 3.5 das generische Verfahren für ein Variantenmanagement bei der Testfallbestimmung mithilfe der Klassifikationsbaum-Methode und dem CTE XL Prof. entwickelt wird.

3.1. Grundlagen der Software-Produktlinien-Entwicklung

Die Grundlage für die Verkürzung der Entwicklungszeit, die Reduzierung der Entwicklungskosten und die Steigerung der Zuverlässigkeit von Software im Zuge der individuellen Massenfertigung ist das Prinzip der Wiederverwendung. Diese Vorteile können jedoch nur ausgeschöpft werden, wenn die Wiederverwendung systematisch

im Entwicklungsprozess geplant und organisiert wird, anderenfalls besteht das Risiko, dass die Entwicklungskosten mithilfe der Wiederverwendung die Kosten einer Neuentwicklung der Software übersteigen [Pohl u. a. 2005].

Wiederverwendung wurde als Prinzip für die effektive Software-Entwicklung schon früh erkannt (vgl. [Dijkstra 1972; Parnas 1976]). Mithilfe von Entwurfsmustern wurde dieses Prinzip beispielweise zur Lösung wiederkehrender Entwurfsprobleme eingesetzt und anhand von Referenzarchitekturen konnten generische Softwarearchitekturen entwickelt werden. Diese Ansätze beziehen sich jedoch nur auf Teilaspekte des Entwicklungsprozesses.

Software-Produktlinien bieten im Gegensatz dazu ein Konzept für die systematische Wiederverwendung entlang des gesamten Entwicklungsprozesses zur schnellen und günstigen Entwicklung qualitativ hochwertiger Softwareprodukte und stellen damit den wichtigsten Ansatz für eine systematische Wiederverwendung im Bereich der Software Entwicklung dar ([Pohl u. a. 2005]). Dabei werden Varianten eines Softwaresystems, die gemeinsame Merkmale aufweisen, für eine bestimmte Domäne durch die zielgerichtete Ableitung aus gemeinsamen Produktlinien-Artefakte entwickelt:

> *„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.“* [Clements und Northrop 2002]

Während bei traditionellen Software-Entwicklungsprozessen die Entwicklung eines spezifischen Softwaresystems innerhalb einer bestimmten Domain im Vordergrund steht, liegt der Fokus der Software-Produktlinien-Entwicklung auf der Erzeugung einer Softwareplattform, die die Gemeinsamkeiten aller Produkte der Produktlinie in Form von gemeinsamen Produktlinien-Artefakten zusammenfasst. Die Produktlinien-Artefakte umfassen dabei jegliche Art von Entwicklungsprozess-Artefakten wie zum Beispiel Anforderungen, Architekturbeschreibungen, Komponenten oder Testfälle. Eine derartige Softwareplattform bildet die gemeinsame Struktur für die effektive Entwicklung und Produktion der abzuleitenden Softwareprodukte ([Meyer und Lehnerd 1997]).

Der für die Schaffung einer Softwareplattform verantwortliche Prozess innerhalb der Software-Produktlinien-Entwicklung ist das Domain Engineering. Neben der Identifizierung und Zusammenfassung gemeinsamer Artfakte ist die Definition von Variabili-

tät in den Artefakten eine wichtige Kernaufgabe des Domain Engineering-Prozesses. Die Variabilität in den Artefakten bildet die Grundlage für die Ableitung konkreter Software-Produkte im anschließenden Application Engineering-Prozess. ([Weiss und Lai 1999])

Damit lassen sich die beiden Grundmerkmale der Software-Produktlinien-Entwicklung formulieren ([Weiss und Lai 1999; Böckle u. a. 2004; Pohl u. a. 2005]):

- die Trennung der beiden Prozesse Domain und Application Engineering und
- die Definition und Verwaltung von Variabilität innerhalb der Produktlinien.

In manchen Fällen wird anstelle des Begriffs Software-Produktlinie auch der Ausdruck Software-Produktfamilie verwendet, welcher synonym zu Ersterem verwendet werden kann ([Pohl u. a. 2005]). Für eine bessere Verständlichkeit wird in dieser Arbeit jedoch auf die Begriffe Software-Produktlinie, beziehungsweise Produktlinie zurückgegriffen.

3.2. Produktlinien-Framework nach [Pohl u. a. 2005]

Der Ansatz der Software-Produktlinien-Entwicklung stellt ein sehr vielversprechendes Konzept für die systematische Wiederverwendung dar. Dementsprechend existieren inzwischen eine Vielzahl von Forschungsprojekten und Anwendungsframeworks zu dieser Thematik [Heymans und Trigaux 2003].

[Heymans und Trigaux 2003] fassen neben verschiedenen Forschungsprojekten zum Thema Software-Produktlinien-Entwicklung unterschiedliche Frameworks für eine praktische Anwendung des Software-Produktlinie-Ansatzes zusammen und geben damit einen wertvollen Überblick über den Stand der Technik.

Bei der genaueren Betrachtung der untersuchten Frameworks, beziehungsweise Methoden, fällt jedoch auf, dass fast alle Ansätze die Qualitätssicherung mithilfe des Testens vernachlässigen. Einzig die *Product Line Software Engineering*-Methode (**PuLSE**) nach [Bayer u. a. 1999] beinhaltet einen Akzeptanztest am Ende des Entwicklungsprozesses, ohne jedoch auf den Aspekt der Variabilität bezüglich der Testartfakte einzugehen.

Die *Feature-Oriented Domain Analysis* (**FODA**) [Kang u. a. 1990] beschreibt, wie auftretende Varianten definiert und dokumentiert werden können, betrachtet dabei aber lediglich die Anforderungsanalyse.

Einen moderneren Ansatz für die Software-Produktlinien-Entwicklung, der den gesamten Entwicklungsprozess inklusive der Testphase betrachtet und ein eigenes phasenübergreifendes Variabilitätsmodell beschreibt, stellen [Pohl u. a. 2005] vor. Die Wurzeln dieses Ansatzes liegen in den europäischen Forschungsprojekten ESAPS ([ESAPS 2011]), CAFÉ ([CAFÉ 2011]) und FAMILIES ([Families 2011]).

Mit dem Konzept eines Produktlinien-Entwicklungsprozesses, der sowohl eine variantenspezifische Anforderungsanalyse, als auch eine variantendifferenzierende Testphase betrachtet, ist der Ansatz von [Pohl u. a. 2005] für die Entwicklung eines generischen Verfahrens zum Variantenmanagement bei der Testfallbestimmung von großer Bedeutung und wird aus diesem Grund im Folgenden detailliert erläutert.

Der von [Pohl u. a. 2005] beschriebene Ansatz kombiniert das Plattform-Prinzip als Methodik für Wiederverwendung mit dem Ansatz der Mass Customisation und beinhaltet, analog zur klassischen Software-Produktlinien-Entwicklung, die Trennung zwischen den Teilprozessen *Domain* und *Application Engineering*.

Mithilfe des *Domain Engineering* wird die Plattform der Produktlinie mit den wiederzuverwendenden Artefakten entwickelt. Dazu werden die Gemeinsamkeiten und Unterschiede innerhalb der Produktlinie zusammengetragen und die abzuleitenden Endprodukte spezifiziert. Die Plattform wird möglichst erweiterbar und modifizierbar gestaltet, um auch zukünftige Produktentwicklungen wie beispielsweise eine Erweiterung der Produktlinie um neue Produkte zu berücksichtigen. Mit diesem Wissen können die einzelnen Artefakte definiert und realisiert werden. Im Gegensatz zur Entwicklung von Individualsoftware zeichnet sich die Produktlinien-Entwicklung nach [Pohl u. a. 2005] durch Variabilität in den Artefakten aus. Diese Variabilität wird mit Hilfe eines Variabilitätsmodells definiert und dargestellt, welches durch die in Abbildung 3.1 dargestellten Phasen des *Domain Engineerig* immer weiter verfeinert und ergänzt wird.

Im *Application Enginnering* werden auf Basis der Produktlinien-Plattform die konkreten Endprodukte abgeleitet, wobei möglichst viele Artefakte wiederverwendet werden. Wie in Abbildung 3.1 ersichtlich, nutzen die einzelnen Entwicklungsphasen des *Application Engineering* die Artefakte der entsprechenden Phasen auf Seiten des *Domain Engineering* und binden die darin eingeführte Variabilität. Im bestmög-

lichen Fall können bei der Ableitung der Endprodukte ausschließlich Artefakte der Plattform verwendet werden. In der Regel müssen jedoch während des *Application Engineering* spezifische Artefakte für ein konkretes Endprodukt zusätzlich entwickelt werden. Diese Abweichungen von der Plattform werden als *Deltas* bezeichnet. Eine weitere wichtige Aufgabe des *Application Engineering* stellt somit die Ermittlung dieser *Deltas* und des damit verbundenen zusätzlichen Entwicklungsaufwands dar. Gleichzeitig wird gegenüber dem *Domain Engineering* eine Rückmeldung gegeben, welche Plattform-Artefakte anwendbar waren und welche *Deltas* aufgetreten sind. Anhand der Rückmeldungen kann die Plattform für künftige Ableitungen modifiziert werden.

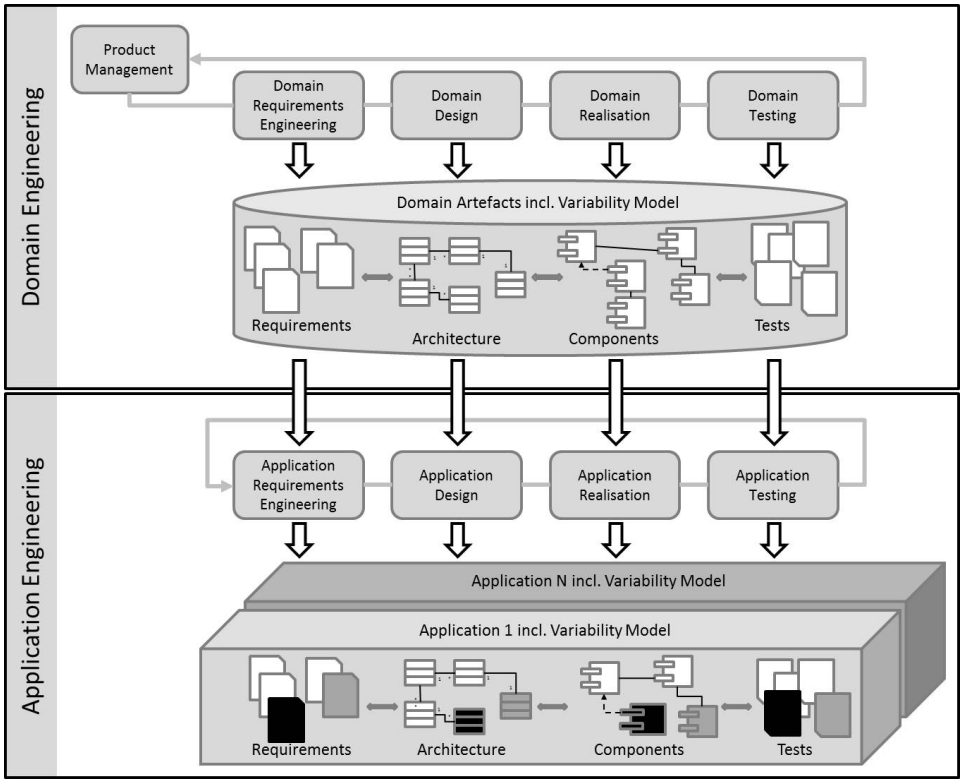

Abbildung 3.1.: Produktlinien-Entwicklungsprozess (vgl. [Pohl u. a. 2005])

Weder die Teilprozesse, noch die einzelnen Phasen innerhalb der Teilprozesse müssen sequentiell abgearbeitet werden. Dies wird durch den in Abbildung 3.1 dargestellten Schleifenpfeil verdeutlicht ([Pohl u. a. 2005]). Zudem bedeutet die Rückkopplung innerhalb des *Domain Engineering*, dass die gesammelten Erkenntnisse innerhalb der einzelnen Phasen wieder in das *Product Management* einfließen um die Produktlinie neu auszurichten.

3.2.1. Orthogonales Variabilitätsmodell

Eine Schlüsseltätigkeit, die sich durch alle Phasen des *Domain Engineering* hindurchzieht, ist die Definition und Dokumentation von Variabilität. Demgegenüber steht die Ausschöpfung der Variabilität innerhalb des *Application Engineering* ([Pohl u. a. 2005]).

Während andere Produktlinien-Ansätze Variabilität nur in einzelne Artefakte und Modelle des Entwicklungsprozesses einführen (siehe beispielsweise [Kang u. a. 1990]), stellen [Pohl u. a. 2005] ein prozessübergreifendes und zu den einzelnen Artefakt-Modellen orthogonales Variabilitätsmodell vor. Dieses hat den Vorteil, dass die Variabilität entlang des gesamten Entwicklungsprozesses unabhängig von einzelnen Artefakten und Prozessen konsistent modelliert werden kann ([Pohl u. a. 2005]). Die grundlegenden Variabilitätskonzepte können modellübergreifend definiert und dargestellt werden, wodurch sich Änderungen bezüglich der Variabilität besser nachverfolgen lassen und die einzelnen Artefakt-Modelle nicht verändert werden müssen. Die FODA-Methode ist zudem zur Modellierung spezieller Variabilitätsaspekte innerhalb von Produktlinien nicht ausdrucksstark genug. Beispielsweise können keine Gruppen von Features als optionale Bestandteile eines Endproduktes ausgewählt werden.

Das orthogonale Variabilitätsmodell nach [Pohl u. a. 2005] bietet neben einem Metamodell, das die grundlegenden Konzepte *Variationspunkt* und *Variante* in Beziehung setzt, eine grafische Notation für die Darstellung der Variabilität.

Wie in Abbildung 3.2 dargestellt, bildet ein *Variationspunkt* einen variablen Gegenstand der realen Welt innerhalb eines Artefakts ab. Analog dazu repräsentiert eine *Variante* ein Variabilitätsobjekt innerhalb eines Artefakts. ([Pohl u. a. 2005])

In einer Software-Produktlinie beschreiben *Variationspunkte* somit die Teile eines Artefaktes, in denen sich die einzelnen Endprodukte unterscheiden. Eine *Variante* hingegen stellt eine bestimmte Instanziierung des *Variationspunktes* dar.

Auf Basis dieser Schlüsselkonzepte wurde von [Pohl u. a. 2005] folgendes Metamodell in UML 2 Notation für das orthogonale Variabilitätsmodell entwickelt:

Die *Variationspunkte* wurden in dem in Abbildung 3.3 dargestellten Metamodell als abstrakte Klassen modelliert. Die Spezialisierung in die Klassen *interner Variationspunkt* und *externer Variationspunkt* spiegeln die interne und externe Variabilität der

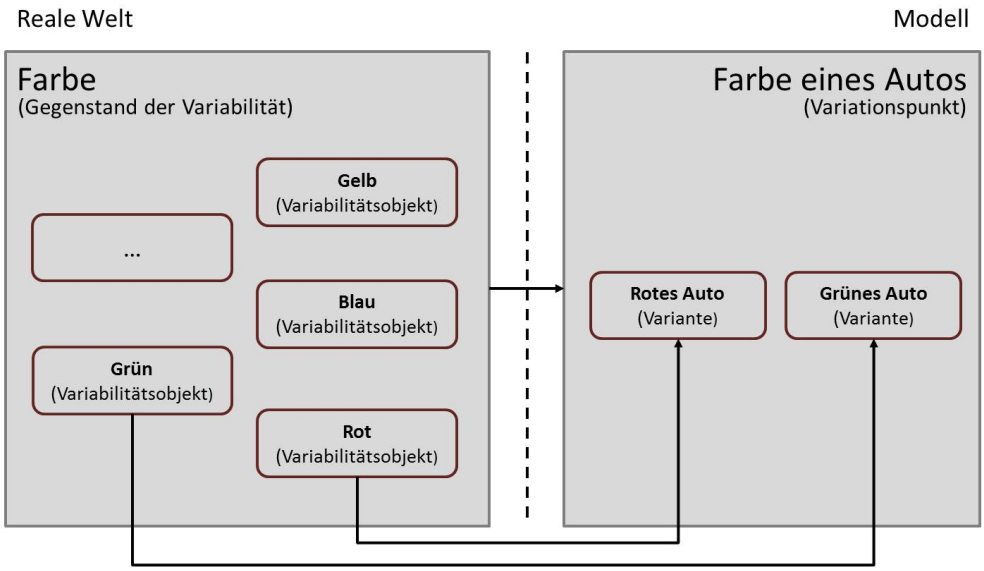

Abbildung 3.2.: Variabilität in der realen Welt und im Modell (vgl. [Pohl u. a. 2005])

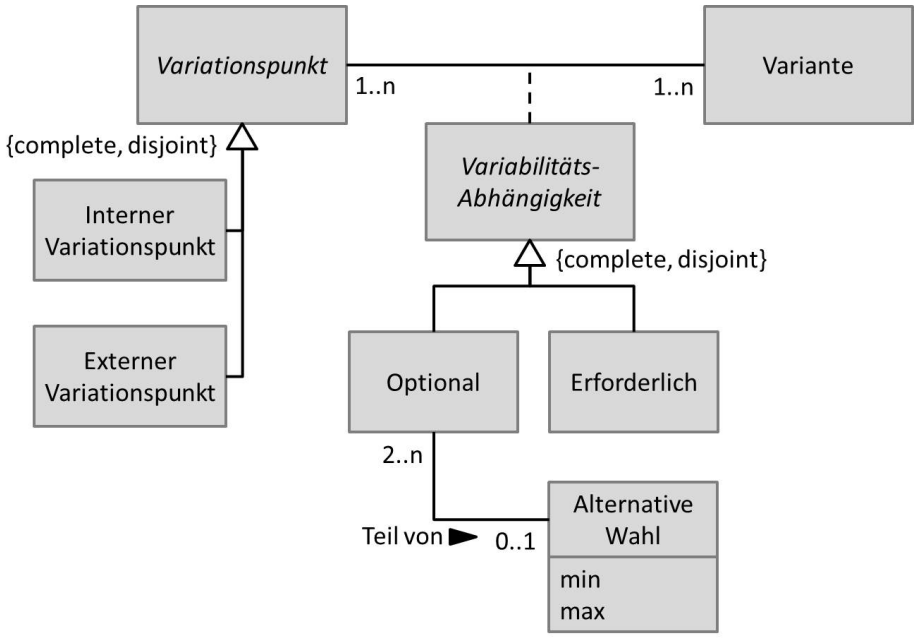

Abbildung 3.3.: Vereinfachtes Metamodell des orthogonalen Variabilitätsmodells (vgl. [Pohl u. a. 2005])

Artefakte wider. Die Varianten eines *externen Variationspunktes* sind sowohl für den Kunden, als auch für den Entwickler sichtbar, wohingegen *interne Variationspunkte* mit Varianten assoziiert sind, die nur der Entwickler sehen kann ([Pohl u. a. 2005]). Der Kunde kann dabei die Varianten wählen, die seine Bedürfnisse am besten erfüllen. In manchen Fällen wählt aber auch das Product Management die Varianten aus und definiert damit eine Menge der Endprodukte, aus denen der Kunden wählen kann. Die interne Variabilität ist im Gegensatz dazu die Variabilität, die für die Kunden nicht ersichtlich ist ([Pohl u. a. 2005]). Dabei handelt es sich um Varianten, die auf einem tieferen Abstraktionsniveau liegen und somit für den Kunden nicht von Interesse sind. Die Wahl zwischen verschiedenen Kommunikationsprotokollen ist ein Beispiel für innere Variabilität, da für den Kunden hierbei nicht relevant ist, auf welche Weise zwei Geräte kommunizieren. Der Entwickler hingegen muss in Abhängigkeit höherer Funktionsschichten das Protokoll auswählen, um beispielsweise eine gesicherte Verbindung zu realisieren. Dieses Beispiel verdeutlicht, dass externe Variabilität häufig auch interne Variabilität zur Folge hat, die es zu beherrschen gilt.

Die *Variabilitätsabhängigkeit* wurde im Metamodell von [Pohl u. a. 2005] als Assoziationsklasse modelliert, die die Eigenschaften der Assoziation zwischen den *Variationspunkten* und ihren *Varianten* beschreibt. Zudem handelt es sich dabei um eine abstrakte Klasse, die durch die Klassen *Optional* und *Erforderlich* spezialisiert wird. Eine *optional Variabilitätsabhängigkeit* bedeutet, dass eine zu einem *Variationspunkt* gehörende *Variante* Teil eines speziellen Endprodukts der Produktlinie sein kann, es aber nicht zwingend notwendig ist. Demgegenüber stehen *erforderliche Variabilitätsabhängigkeiten*, welche definieren, dass eine *Variante* für einen *Variationspunkt* erforderlich ist. Hierbei muss die *Variante* im Endprodukt enthalten sein, wenn der übergeordnete *Variationspunkt* im Endprodukt enthalten ist. Mithilfe der *Alternativen Auswahl* wird schließlich die Möglichkeit modelliert, die minimale und maximale Anzahl gewählter optionaler *Varianten* innerhalb eines *Variationspunktes* festzulegen. Damit lassen sich Gruppen von optionalen *Varianten* definieren.

[Pohl u. a. 2005] erweiterten das Metamodell, ähnlich zum Feature-Modell ([Kang u. a. 1990]), durch *requires*- und *excludes*-Constraints. Damit lässt sich beispielsweise beschreiben, ob eine *Variante* nur bei Enthaltensein einer anderen Variante gewählt werden darf, oder deren Abwesenheit fordert. Derartige Constraints wurden zwischen *Varianten*, zwischen *Varianten* und *Variationspunkten* und zwischen *Variationspunkten* definiert.

Um die definierten *Varianten* entlang des gesamten Entwicklungsprozesses den Entwicklungs-Artefakten zuordnen zu können, die sie verfeinern beziehungsweise realisieren, werden darüber hinaus Artefakte-Abhängigkeiten definiert.

Für die grafische Darstellung der im Metamodell definierten Variabilitätskonzepte wurde die in Abbildung 3.4 dargstellte Notation entwickelt ([Pohl u. a. 2005]).

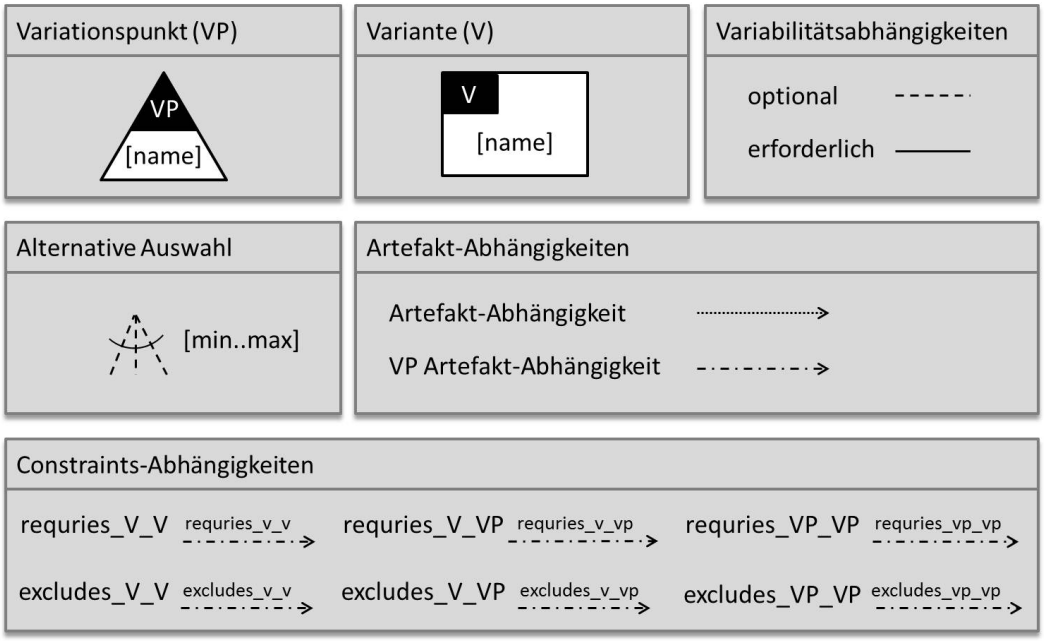

Abbildung 3.4.: Grafische Notation des orthogonalen Variabilitätsmodells (vgl. [Pohl u. a. 2005])

Nachdem deutlich wurde, wie der Prozess der Software-Produktlinien-Entwicklung nach [Pohl u. a. 2005] aufgebaut ist und wie Variabilität innerhalb dieses Prozesses modelliert wird, werden in den folgenden zwei Abschnitten die Phasen des Domain und des Application Engineering näher erläutert.

3.2.2. Phasen des Domain Engineering

Das *Product Management* bildet durch die Betrachtung der ökonomischen Aspekte der Produktlinie und die Entwicklung einer Marktstrategie die Grundlage für die weiteren Phasen des Domain Engineering. Die wichtigste Aufgabe stellt hierbei die Definition des Produktportfolios dar. Mithilfe von Scoping-Techniken werden dazu die Produkte festgelegt, die innerhalb der Produktlinie entwickelt werden sollen und die, die außerhalb liegen ([Pohl u. a. 2005]). Daraus resultiert eine Product Roadmap, die die gemeinsamen und variablen Features künftiger Produkte bestimmt und einen

Zeitplan für deren Realisierung definiert. Bei der Entwicklung der Product Roadmap werden vorausschauend die zukünftigen Änderungen von gesetzlichen Vorschriften und Standards sowie die Weiterentwicklung von Technologien und die damit verbundenen Veränderungen von Features berücksichtigt. In einem weiteren Schritt stellt das *Product Management* den weiteren Phasen eine Liste bereits existierender Produkte und Entwicklungsartfakte zur Verfügung, die bei der Erstellung der neuen Plattform wiederverwendet werden können ([Pohl u. a. 2005]).

Auf Basis der Product Roadmap und der schon existierenden Artefakte werden im *Domain Requirements Engineering* die gemeinsamen und variablen Anforderungen der Produktlinie entwickelt und dokumentiert. Wie in Abbildung 3.5 dargestellt, werden diese in Verbindung mit dem dazugehörigen Variabilitätsmodell an das Domain Design weitergeben.

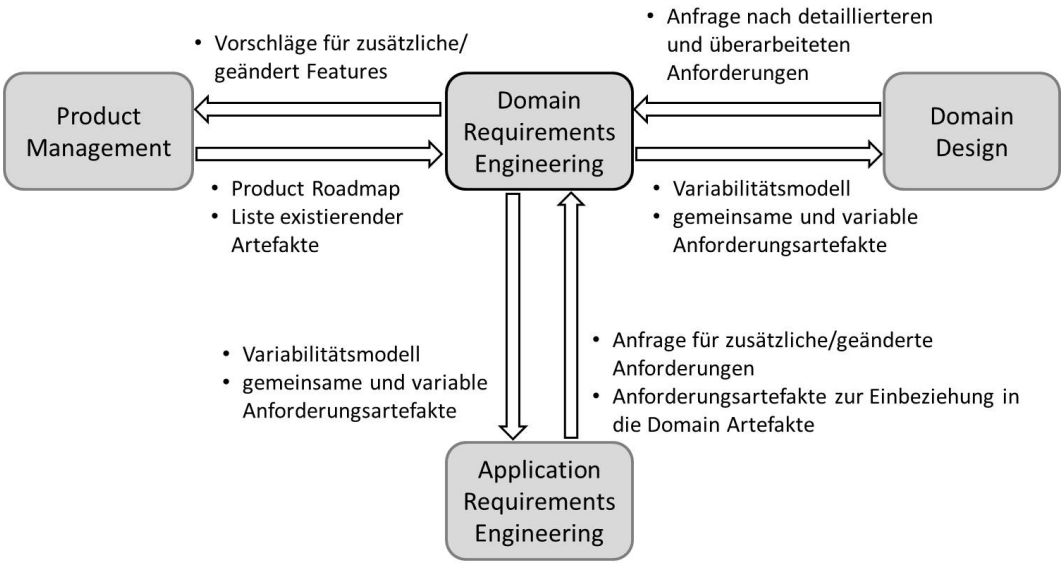

Abbildung 3.5.: Informationsfluss zwischen dem *Domain Requirements Engineering* und den angrenzenden Phasen (vgl. [Pohl u. a. 2005])

Fallen während des *Domain Requirement Engineerings* bei der Analyse der existierenden Produkte, der Kundenbedürfnisse, der Gesetze und Standards und der anderen Anforderungsquellen fehlende oder zu ändernde Anforderungen auf, werden diese an das *Product Management* zurückgemeldet. Das *Domain Design* richtet Anträge auf Hinzufügung oder Änderung von Anforderungen an das *Domain Requirements Engineering*, wenn diese beim Entwurf der Design-Artefakte als ungenaue oder fehlerhafte identifiziert werden. Schließlich stellt das *Domain Requirements Engineering* dem *Application Engineering* die vordefinierten gemeinsamen und variablen Anforderungen inklusive dem assoziierten Variabilitätsmodell zur Verfügung. Dabei

werden die Assoziationen zwischen dem Variabilitätsmodell und den Anforderungsartefakten mit Artefakt-Abhängigkeiten abgebildet. Die Abbildungen 3.6 und 3.7 stellen die Beschreibung der Variabilität mithilfe des Variabilitätsmodells für textuelle Anforderungen und ein Usecase-Diagramm für ein Fahrzeugschließsystem aus dem Automobilbereich dar.

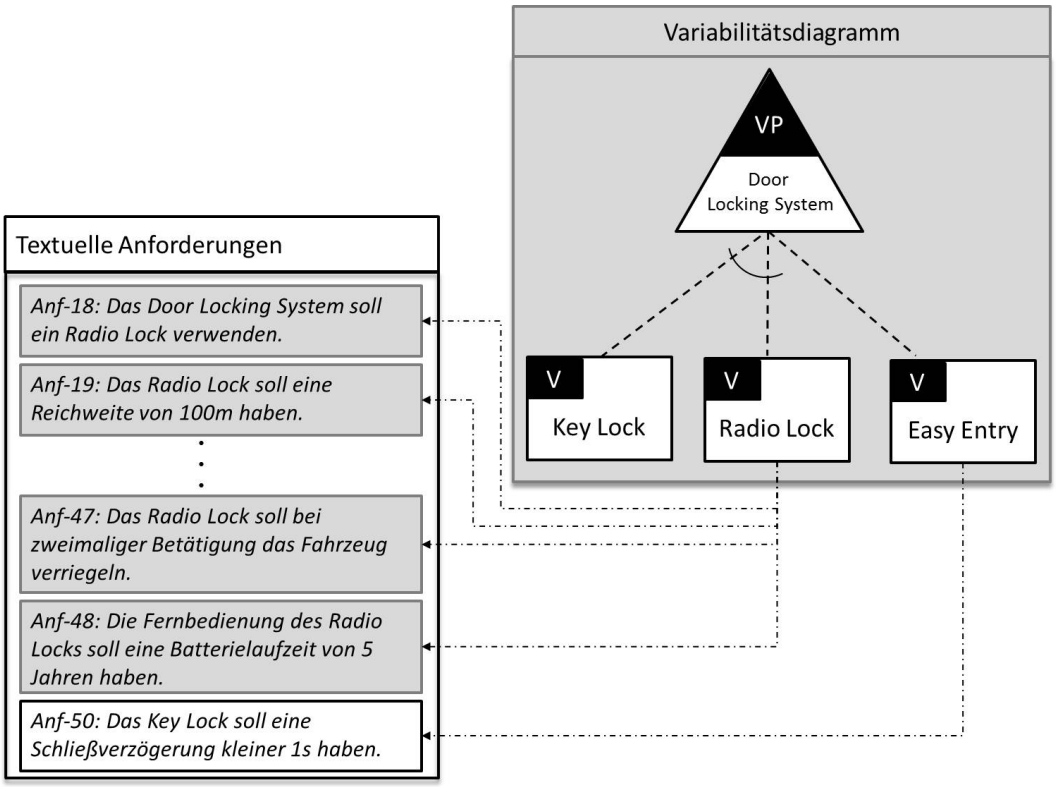

Abbildung 3.6.: Artefakt-Abhängigkeiten zwischen textuellen Anforderungen und dem Variabilitätsmodell

In Abbildung 3.6 werden die einzelnen Anforderungen den Varianten im Variabilitätsmodell zugeordnet. In Abbildung 3.7 sind es hingegen die Anwendungsfälle, die mit den Varianten des Variabilitätsmodells assoziiert werden. Auf diese Weise können Inkonsistenzen zwischen den Anforderungsartefakten mithilfe des Variabilitätsmodells identifiziert und korrigiert werden.

Auf Basis der gemeinsamen und variablen Anforderungen und dem Variabilitätsmodell aus dem *Domain Requirements Engineering* wird im *Domain Design* die Referenzarchitektur der Produktlinie entwickelt und der *Domain Realisation* und dem *Application Design* zur Verfügung gestellt. Dabei bildet das *Domain Design* die externe Variabilität der Anforderungen auf die interne Variabilität in der Referenzarchitektur ab. Diese enthält eine variable Struktur, die die Basis für die Struktur jedes Endprodukts der Produktlinie bildet, und beschreibt den Aufbau von wieder-

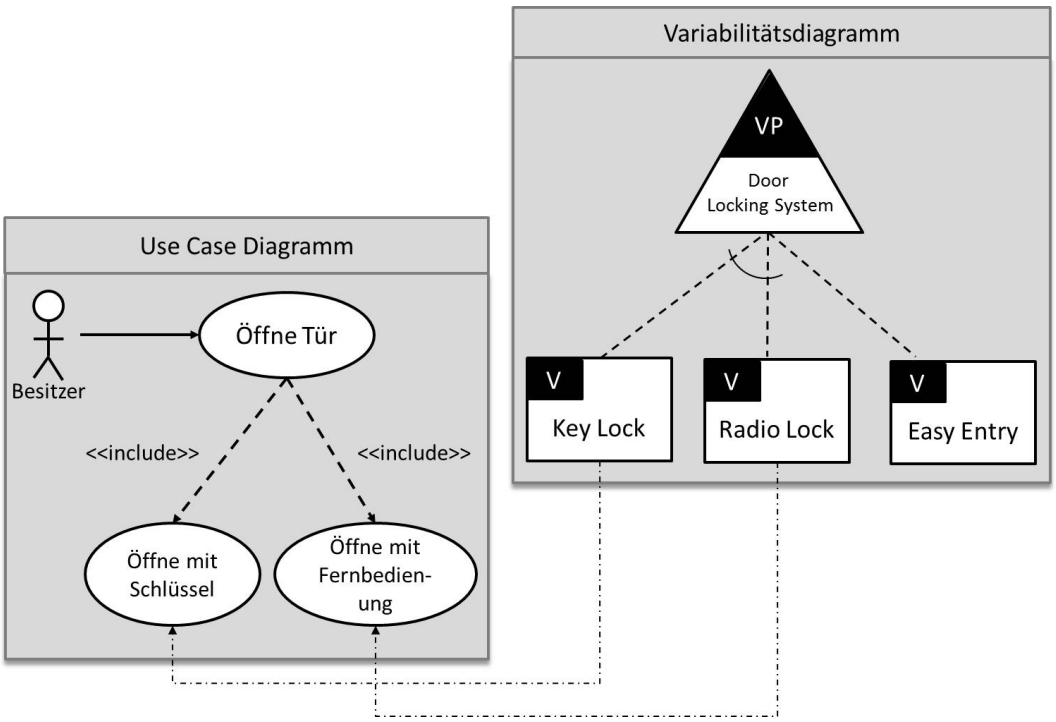

Abbildung 3.7.: Artefakt-Abhängigkeiten zwischen Usecase-Diagramm und Variabilitätsmodell

verwendbaren Komponenten und Schnittstellen ([Pohl u. a. 2005]). Neben der Referenzarchitektur wird eine Auswahl wiederverwendbarer Domain-Artefakte, die durch die *Domain Realisation* entwickelt werden müssen, weitergereicht. Komplikationen bei der Realisierung der Komponenten und Schnittstellen werden von der *Domain Realisation* an das *Domain Design* zurückgemeldet, um die entwickelte Architektur und ihre Bestandteile zu modifizieren.

Während der *Domain Realisation* werden das detaillierte Design entwickelt und die wiederverwendbaren Komponenten und Schnittstellen implementiert. Zusätzlich werden Konfigurationsmechanismen realisiert, die während der *Application Realisation* zur Auswahl der Varianten und somit zum Bau der einzelnen Endprodukte aus den Komponenten und Schnittstellen verwendet werden. Die wiederverwendbaren Komponenten und Schnittstellen, sowie die Schnittstellenbeschreibung werden an das *Domain Testing* weitergeben, welches die Testergebnisse, Problem Reports und Defekte in der Schnittstellenbeschreibung als Feedback an die *Domain Realisation* zurückgibt und somit zur Verbesserung der Software-Artefakte beiträgt.

Im *Domain Testing* werden schließlich die Ergebnisse der vorhergehenden Phasen validiert. Dabei liegt der Fokus auf der Validierung der Software-Artefakte der Rea-

lisierungsphase ([Pohl u. a. 2005]). Abbildung 3.8 zeigt, welche Informationen aus den anderen Phasen des Domain Engineering in das *Domain Testing* einfließen.

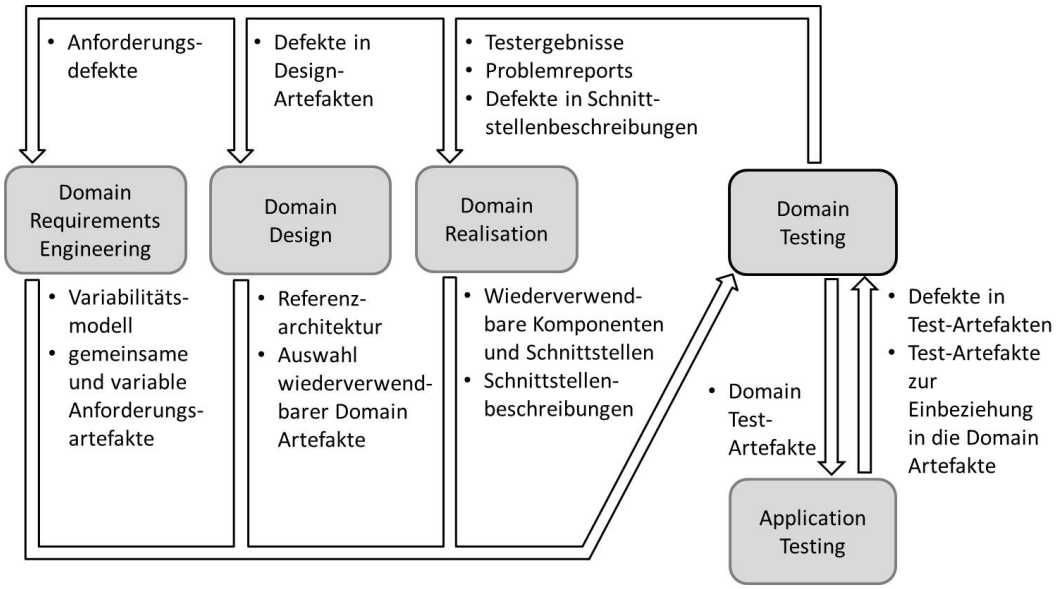

Abbildung 3.8.: Informationsfluss zwischen dem *Domain Testing* und den anderen Phasen des Domain Engineering (vgl. [Pohl u. a. 2005])

Die Anforderungsartefakte und das Variabilitätsmodell aus dem *Domain Requirements Engineering* werden für die Entwicklung des Systemtests verwendet. Da die Plattform im Gegensatz zu einem Endprodukt nur aus lose gekoppelten Komponenten besteht, kann der Systemtest nur Teilsysteme validieren, die eine Menge gemeinsamer Anforderungen realisieren und nicht von Variabilität betroffen sind. Hierbei wird das Variabilitätsmodell für die Ableitung von Testfällen verwendet. Das *Domain Testing* nutzt die Referenzarchitektur und die Auswahl wiederverwendbarer Software-Artefakte für die Durchführung von Integrationstests, die die Wechselwirkungen zwischen den einzelnen Komponenten validieren. Es lassen sich jedoch nicht alle Wechselwirkungen untersuchen. Insbesondere die Interaktionen mit im Application Engineering realisierten variablen Komponenten können vom *Domain Testing* nicht zufriedenstellend überprüft werden ([Pohl u. a. 2005]). Auf Basis der in der *Domain Realisation* implementierten Komponenten und Schnittstellen werden im *Domain Testing* Unittests gegen die Schnittstellenbeschreibungen durchgeführt.

Wurde die Testfallbestimmung durch unklare oder unvollständige Anforderungen verhindert, wird dies an das *Domain Requirements Engineering* zurückgemeldet. Bei Defekten in den Design-Artfakten, die die Definition von Test-Artefakten unmöglich machen, wird ein Feedback an das *Domain Design* gegeben und fehlerhafte Komponenten oder Schnittstellen werden der *Domain Realisation* übermittelt. Mithilfe

dieses Feedbacks werden die Domain-Artefakte überarbeitet, womit das *Domain Testing* zur Verbesserung der Produktlinien-Plattform beiträgt.

Eine weitere wichtige Aufgabe des *Domain Testing* besteht in der Bereitstellung wiederverwendbarer Test-Artefakte, wie zum Beispiel von Testfällen oder Testszenarien, für das *Application Testing*. Dabei werden alle Testfälle, auch die, die schon während des *Domain Testings* ausgeführt wurden, an das *Application Testing* übergeben. Abbildung 3.9 verdeutlicht, wie Variabilität für ein gemeinsames Testszenario mithilfe des Variabilitätsmodells abgebildet werden kann.

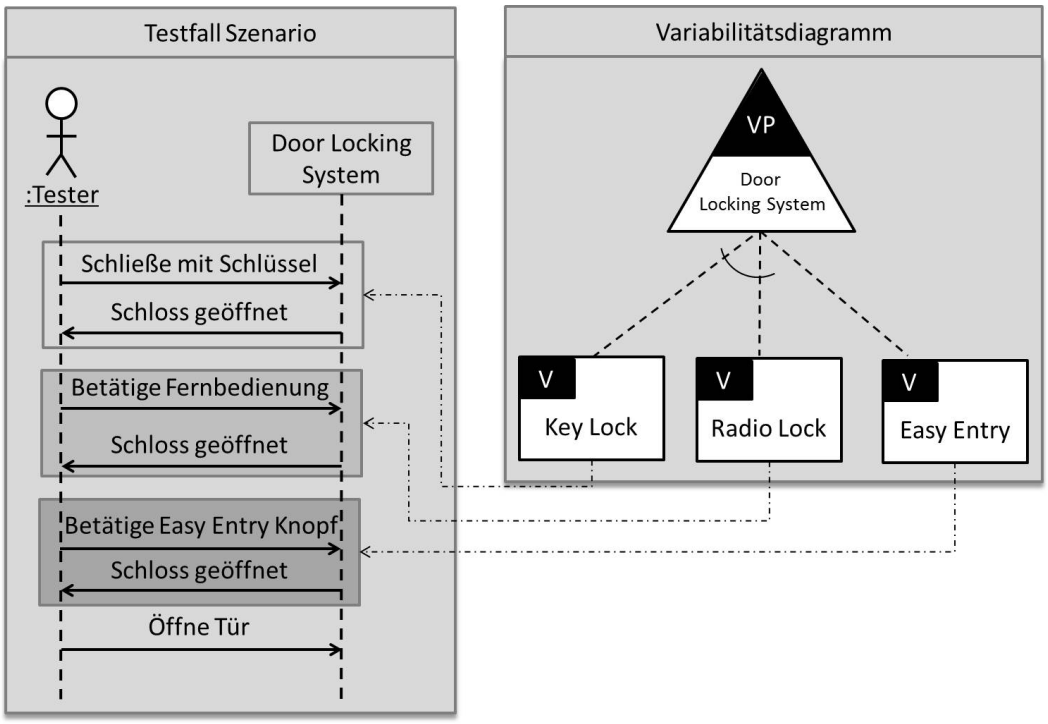

Abbildung 3.9.: Variabilität im *Domain Testing* (vgl. [Pohl u. a. 2005])

3.2.3. Phasen des Application Engineering

Während im *Domain Requirements Engineering* Anforderungsartefakte für die gesamte Software-Produktlinie festgelegt wurden, wendet sich das *Application Requirements Engineering* der Aufgabe zu, Anforderungen für die einzelnen Endprodukte zu wählen und zu dokumentieren. Dabei wird das Ziel verfolgt so viele Domain Anforderungsartefakte wie möglich wiederzuverwendet.

Die Features der einzelnen Produkte der Produktlinie werden im *Product Management* festgelegt. Werden während des *Application Requirements Engineering* neue

oder fehlerhafte Features identifiziert, können diese dem *Product Management* zum Hinzufügen zur Plattform oder zum Ändern vorgeschlagen werden.

Das *Application Requirements Engineering* verwendet die gemeinsamen und variablen Anforderungen und das Variabilitätsmodell des *Domain Engineerings* um die Varianten zu bestimmten und die für die Endprodukte wiederzuverwendenden Anforderungsartefakte zu identifizieren. Stellt sich dabei heraus, dass Anforderungsartefakte vergessen wurden oder Fehler aufweisen, wird dem *Domain Requirements Engineering* eine Anfrage für zusätzliche oder zu ändernde Anforderungsartefakte übermittelt, welche in die Plattform aufgenommen oder in ihr verbessert werden sollen. Zudem werden anwendungsspezifische Anforderungen, die die Bedürfnisse der Produktlinie erfüllen, für künftige Produkte in die Plattform übernommen.

Die Hauptaufgabe des *Application Designs* besteht in der Entwicklung der Anwendungsarchitektur als Spezialisierung der Referenzarchitektur. Hierzu wird die architektonische Variabilität innerhalb der Referenzarchitektur an den Variationspunkten gebunden. Neben der Nutzung wiederverwendbarer Domain Artefakte werden neue Architekturelemente auf Basis der anwendungsspezifischen Anforderungsspezifikation entwickelt. Als Grundlage dafür dienen die vom *Application Requirements Engineering* übergebene anwendungsindividuelle Anforderungsspezifikation und das Variabilitätsmodell der Anwendung. Die Anforderungsspezifikation setzt sich hierbei sowohl aus wiederverwendeten Domain Anforderungen, als auch aus neu bestimmten anwendungsspezifischen Anforderungen zusammen. Das Variabilitätsmodell der Anwendung wird aus dem globalen Variabilitätsmodell der Produktlinie abgeleitet, enthält Nachverfolgbarkeitsverlinkungen zu den verwendeten Domain Artefakten und bildet die Beziehungen innerhalb der gewählten Variante ab. Das *Application Design* identifiziert auf Basis der eingegangenen Informationen und Artefakte die Abweichungen beziehungsweise *Deltas* zwischen den Domain Anforderungsartefakten und den Anforderungsartefakten des Endproduktes. Daraufhin wird eine Aufwandsabschätzung für die Umsetzung der *Deltas* erstellt und für eine Entscheidungsfindung an das *Application Requirements Engineering* zurückgemeldet.

Bei einer Entscheidung für die Umsetzung der identifizierten *Deltas* wird die Architektur des Endprodukts entwickelt und an die *Application Realisation* weitergegeben. Darüber hinaus gibt das *Application Design* Feedback an das Domain Design zurück. Dabei werden neu entwickelte Architekturelemente zur Plattform hinzugefügt und Änderungswünsche innerhalb der Referenzarchitektur geäußert.

Die *Application Realisation* entwickelt auf Basis der Anwendungsarchitektur und unter Verwendung der wiederverwendbaren Software-Artefakte und Konfigurationsmechanismen aus der *Domain Realisation* ein detailliertes Design und implementiert das konkrete Endprodukt. Die dabei neu entwickelten Software-Artefakte werden der *Domain Realisation* für die Hinzufügung zur Plattform übergeben; darüber hinaus werden Anfragen zur Änderung von existierenden Software-Artefakten gestellt. Bei Implementierungsproblemen, die auf Designfehler zurückzuführen sind, wird ein Feedback an das *Application Design* übermittelt, um diese zu beheben.

Die *Application Realisation* übergibt schließlich dem *Application Testing* das fertige Endprodukt und die Schnittstellenbeschreibung. Das *Application Testing* zielt im Gegensatz zum *Domain Testing* auf den Test des Endproduktes ab. Wie in Abbildung 3.10 dargestellt, werden dazu neben den wiederverwendbaren Domain Test-Artefakten, die zu anwendungsspezifischen Testfällen abgeleitet werden, die Teilergebnisse aus den einzelnen Phasen des Application Engineering berücksichtigt. Die Ableitung der anwendungsspezifischen Testfälle wird durch das Binden der Variabilität in den Testfällen realisiert (vgl. Abbildgung 3.11).

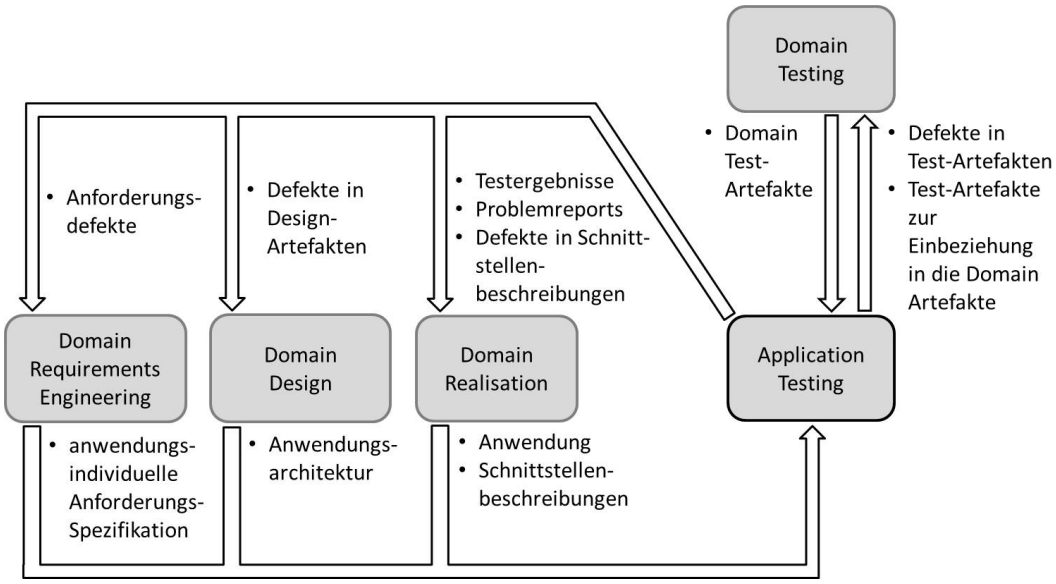

Abbildung 3.10.: Informationsfluss zwischen dem *Application Testing* und den anderen Phasen des Application Engineering (vgl. [Pohl u. a. 2005])

Für einen Unittest müssen die fertigen Software-Komponenten der *Application Realisation* vorliegen. Um einen Integrationstest durchführen zu können, wird die Anwendungsarchitektur aus dem *Application Design* benötigt, und um das System gegen die Anforderungen testen zu können, muss die anwendungsindividuelle Anforderungsspezifikation zur Verfügung stehen. Wurde der Test durchgeführt, gibt das

Application Testing den anderen Phasen ein Feedback über aufgetretene Fehler. Dabei werden Fehler in den Anforderungen, im Design und in den Komponenten und Schnittstellen zurückgemeldet. Zusätzlich werden dem *Domain Testing* fehlerhafte Domain Test-Artefakte gemeldet und neue Test-Artefakte für die Einbeziehung in die Plattform übergeben.

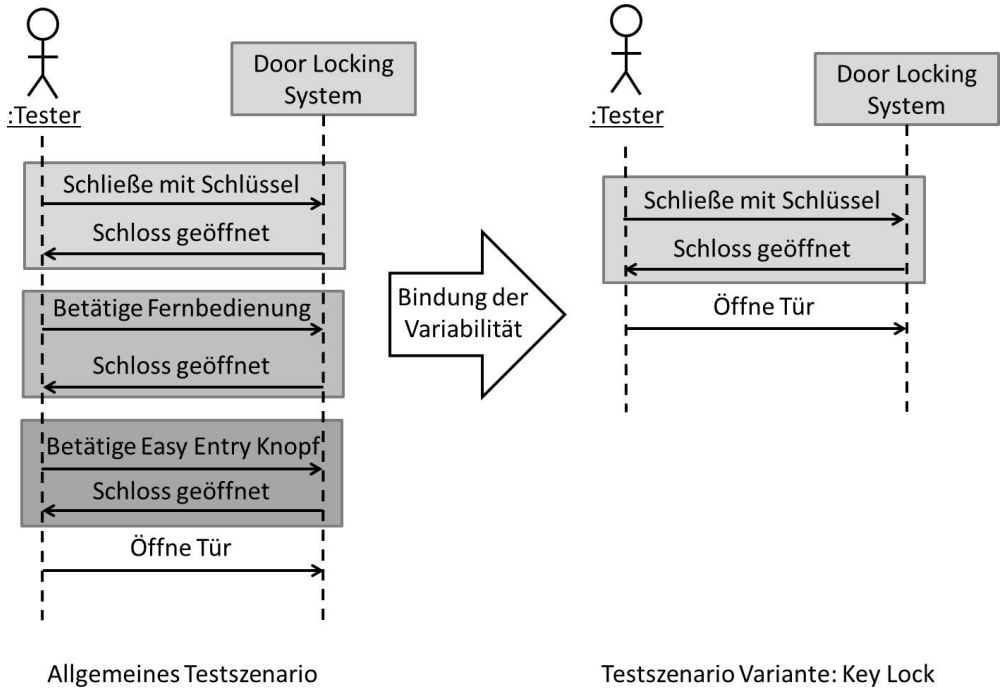

Abbildung 3.11.: Bindung der Variabilität im *Application Testing* (vgl. [Pohl u. a. 2005])

Zusammenfassend lässt sich sagen, dass die Phasen des Domain Engineering den Phasen des Application Engineering die wiederzuverwendenden Artefakte der Plattform und das verfeinerte Variabilitätsmodell zur Verfügung stellen. In der entgegengesetzten Richtung werden die neu entwickelten Artefakte der einzelnen Produkte den Phasen des Domain Engineering übergeben und in die Plattform aufgenommen. Zudem werden Änderungen der Plattform beantragt, falls diese in ihrer ursprünglichen Form nicht für das konkrete Produkt anwendbar ist.

3.3. Einordnung der Werkzeuge DOORS, MERAN und CTE XL Prof.

Innerhalb der Software-Produktlinien-Entwicklung können DOORS, MERAN und der CTE XL. zur Unterstützung unterschiedlicher Phasen eingesetzt werden. So lassen sich mit DOORS und MERAN die gemeinsamen und variablen Domain-Anforderungen des Domain Requirements Engineering dokumentieren und verwalten. Zudem bietet MERAN eine einfache Generierung von variantenspezifischen Anforderungsspezifikationen, die im Application Engineering zur Spezifikation der Endprodukte verwendet werden können. MERAN bindet im Sinne der Produktlinien-Entwicklung die Variabilität innerhalb des generischen Moduls. Der verwendete Variantenselektor und die Parameterersetzung bilden dabei die Variationspunkte ab.

Die Verlinkung von Anforderungen mit anderen Artefakten des Entwicklungsprozesses und die damit verbundene Nachverfolgbarkeit der Anforderungen hat auch für die Software-Produktlinien-Entwicklung große Bedeutung. Die derzeitige Werkzeugkombination aus DOORS, MERAN und CTE XL Prof. erlaubt zwar die Nachverfolgung einzelner Anforderungen bis hin zur Testfallermittlung, die Informationen über die Variantenzugehörigkeit der Anforderungen wird dabei jedoch nicht bis in den CTE XL Prof. übertragen (vgl. Abbildung 3.12).

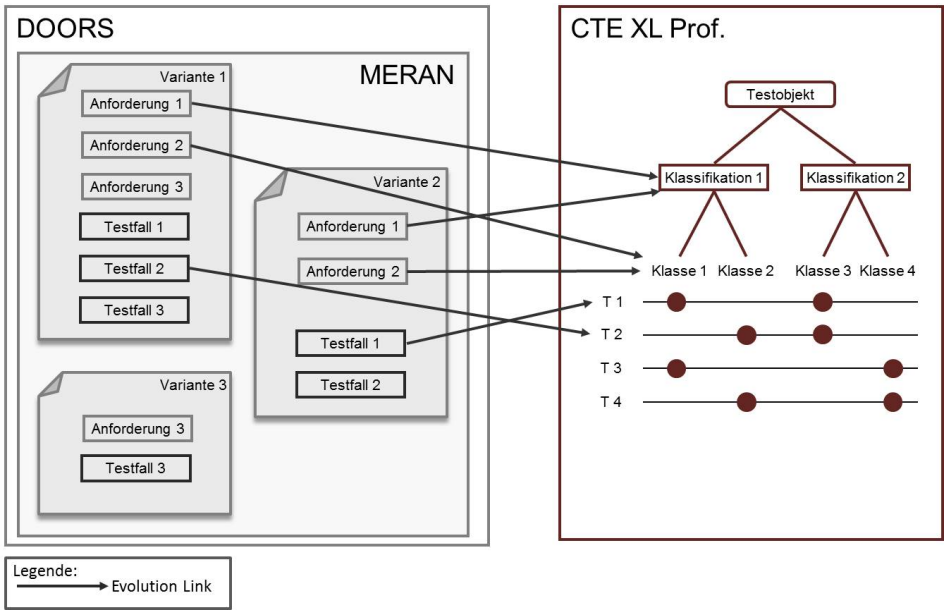

Abbildung 3.12.: Verlorene Varianteninformationen

Für die effiziente Wiederverwendung von Testfällen ist die Suche nach den anwendbaren Domain Testfällen für ein Endprodukt eine essentielle Aufgabe, die nur mithilfe von Nachverfolgbarkeitsverlinkungen realisiert werden kann. Abbildung 3.13 verdeutlicht diesen Vorgang innerhalb des Software-Produktlinien-Entwicklungsprozesses.

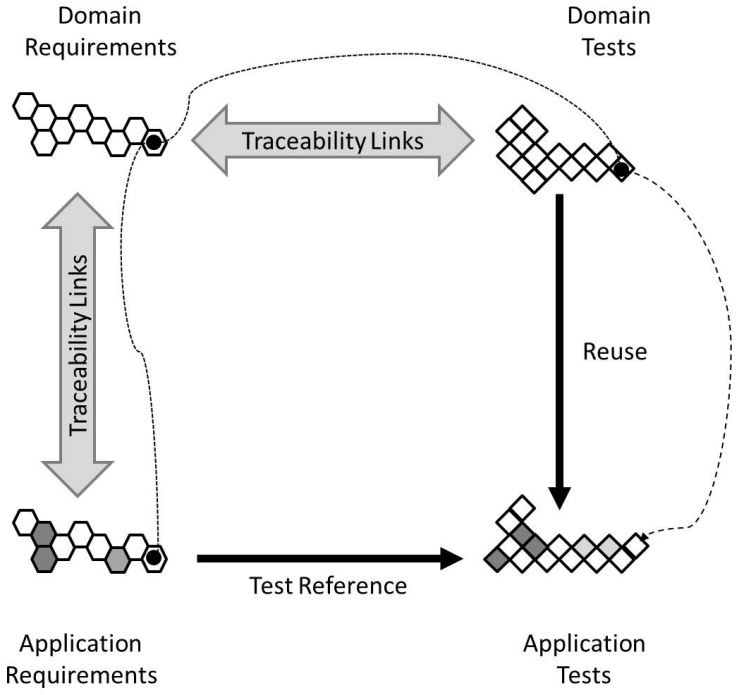

Abbildung 3.13.: Suche von Domain Testfällen (nach [Pohl u. a. 2005])

Die in dieser Arbeit zu entwickelnde Erweiterung des CTE XL Prof. kann somit nicht nur bei der Bestimmung von gemeinsamen und variablen Testfällen der Produktlinien im Domain Testing und zur Ableitung der variantenspezifischen Testfälle für das Application Engineering verwendet werden, sie lässt sich auch bei der Nachverfolgung zur Suche von Domain Testfällen einsetzen. Auf diese Weise wird nicht nur die Qualität des Endprodukts sichergestellt, sondern auch der Entwicklungsprozess verbessert.

3.4. Ansätze für das Testen in der Software-Produktlinien-Entwicklung

Auch wenn die Qualitätssicherung in den überwiegenden Frameworks zur Software-Produktlinien-Entwicklung vernachlässigt, beziehungsweise nur oberflächlich behandelt wird, gibt es inzwischen eine Vielzahl von Forschungsansätzen, die sich dem

Thema des Testens innerhalb der Software-Produktlinien-Entwicklung widmen. [Lamancha u. a. 2009] geben einen Überblick über die entstandenen Arbeiten und klassifizieren diese anhand verschiedener Gesichtspunkte. Die wichtigsten Kategorien für die Entwicklung eines Variantenmanagements für die Klassifikationsbaum-Methode und den CTE XL Prof. stellen dabei die verwendete Testart und die eingesetzten Testtechnologien dar.

Da die Klassifikationsbaum-Methode im Bereich des funktionalen Testens angesiedelt ist, werden im Folgenden nur die Arbeiten betrachtet, die dieser Art des Testens angehören. Hierbei werden insbesondere die Verfahren identifiziert, die aufgrund der angewendeten Testtechnologien einen Erkenntnisgewinn für die Entwicklung eines Verfahrens zum Variantenmanagement für die Testfallbestimmung in Aussicht stellen. Der Test gegen die funktionale Anforderungsspezifikation ist dabei von besonderem Interesse.

Ein Großteil der von [Lamancha u. a. 2009] betrachteten Ansätze sind der Kategorie des funktionalen Testens zuzuordnen. Dabei konzentrieren sich [McGregor 2001], [Bertolino und Gnesi 2004], [Nebut u. a. 2003a], [Olimpiew und Gomaa 2005] und [Reuys u. a. 2005] auf die Ableitung von Testfällen auf Basis modifizierter Anwendungsfälle, welche die Variabilität der Produktlinie abbilden ([Lamancha u. a. 2009]).

In einem weiteren Ansatz verwenden [Nebut u. a. 2003b] Sequenzdiagramme als Grundlage der Testfall-Ableitung und präsentieren eine werkzeuggestützte Methode zur automatischen Generierung von Testfällen. [Kang u. a. 2007] hingegen erweitern die Notation von Sequenzdiagrammen, so dass damit Variabilität in Anwendungsszenarios abgebildet werden kann. Mithilfe des orthogonalen Variabilitätsmodells und von Sequenzdiagrammen definieren [Cohen u. a. 2006] ein Überdeckungs-Test-Modell für die Software-Produktlinien-Entwicklung.

Weitere Ansätze definieren Meta-Modelle für das Testen ([Dueñas u. a. 2004],[Baerisch 2007]), versuchen das funktionale Testen zu automatisieren [Ardis u. a. 2000] oder generalisieren existierende und neue Testfälle um eine generische Testfallmenge für Legacysysteme zu erzeugen ([Geppert u. a. 2004]).

Insbesondere die Arbeiten von [McGregor 2001], [Dueñas u. a. 2004] und [Bertolino und Gnesi 2004] können nutzbringend bei der Entwicklung eines Variantenmanagements für die Klassifikationsbaum-Methode und den CTE XL Prof. eingesetzt werden. Daher werden diese im Folgenden näher erläutert.

[McGregor 2001] beschreibt ein allgemeines Verfahren zum Testen innerhalb der Software-Produktlinien-Entwicklung. Dazu werden sowohl die Testfallbestimmung im Domain Engineering, als auch das Ableiten von Testfällen im Application Engineering betrachtet. Während der Fokus im Domain Engineering auf dem Testen von Artefakten und deren Korrektheit und Vollständigkeit liegt, wird im Application Engineering die Erfüllung der Anforderungen in den Mittelpunkt gestellt ([McGregor 2001]).

Analog zum Software-Produktlinien-Framework von [Pohl u. a. 2005] geht [McGregor 2001] davon aus, dass die gemeinsamen Testfälle der Plattform für den Test des Endprodukts wiederverwendet werden und produktspezifische Testfälle neu entwickelt werden. Der Prozess der Testfallbestimmung beginnt demnach mit dem Prozess des Domain Requirement Engineering für die Produktlinie. Damit ergibt sich die in Abbildung 3.14 dargestellte hierarchische Struktur der Artefakte für die Erstellung von Testfällen.

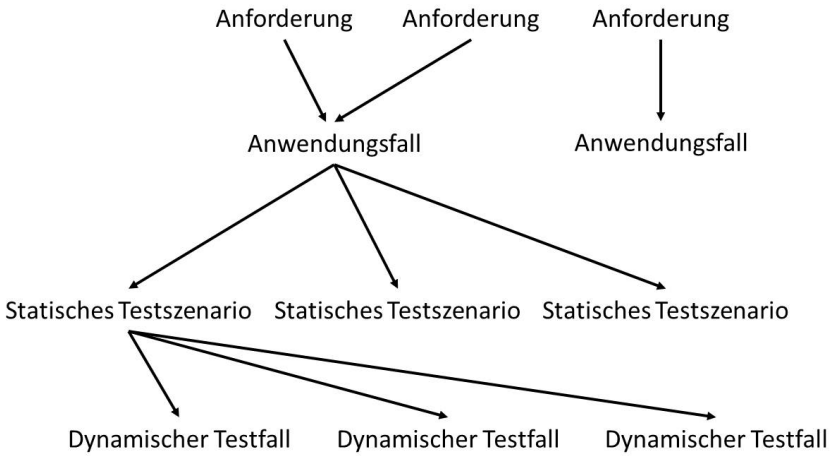

Abbildung 3.14.: Ableiten von Testfällen (vgl. [McGregor 2001])

Dabei fasst jeder Anwendungsfall (*engl. Usecase*) mehrere Anforderungen zusammen, die das Endprodukt erfüllen muss. Eine Usecase-Beschreibung umfasst neben dem Szenario für den Hauptablauf die Szenarien für Alternativ-, Erweiterungs- und Ausnahmeabläufe. Im Ansatz von [McGregor 2001] werden generelle Szenarien für die Testfallbestimmung des Produktes zu produktspezifischen Szenarien spezialisiert. Dabei werden unbekannte Werte im generellen Szenario durch spezifische Werte im Produktszenario ersetzt, wobei jeder unbekannte Wert mehrere produktspezifische Werte für die verschiedenen Produktvarianten annehmen kann.

Tabelle 3.1 veranschaulicht ein generelles Hauptszenario eines Usecases für Zielorteingabe eines Fahrzeugnavigationssystems. Dabei ist die Vorbedingung, dass noch Speicherplätze für persönliche Zielorteingaben frei sind, erfüllt. Das dazugehörige Testszenario wird in Tabelle 3.2 dargestellt.

Benutzereingabe	**Reaktion des Systems**
Auswahl der Funktion „Zielort hinzufügen“	1. Initialisierung eines leeren Eintrags 2. Anzeige des leeren Eintrags auf dem Display
Eingabe der benötigten Informationen	1. Speicherung des Zielorts im System 2. Anzeige einer Eingabebestätigung auf dem Display

Tabelle 3.1.: Generelles Szenario für Usecase *Zielorteingabe* (vgl. [McGregor 2001])

Benutzereingabe	**Reaktion des Systems**
Auswahl der Funktion „Zielort hinzufügen“	1. Initialisierung eines leeren Eintrags 2. Anzeige des leeren Eintrags auf dem Display
Eingabe „Berlin, Bahnhofsstraße 1“	1. Speicherung des Zielorts im System 2. Anzeige der Nachricht „Zielort hinzugefügt“ auf dem Display

Tabelle 3.2.: Testszenario für Usecase *Zielorteingabe* (vgl. [McGregor 2001])

Der dargestellte generelle Usecase wird innerhalb der Produktlinien-Entwicklung wie in Abbildung 3.15 zu produktspezifischen Usecases spezialisiert.

Der resultierende Usecase *Zielort per Spracheingabe hinzufügen* und der dazugehörige Testfall werden in den Tabellen 3.3 und 3.4 veranschaulicht:

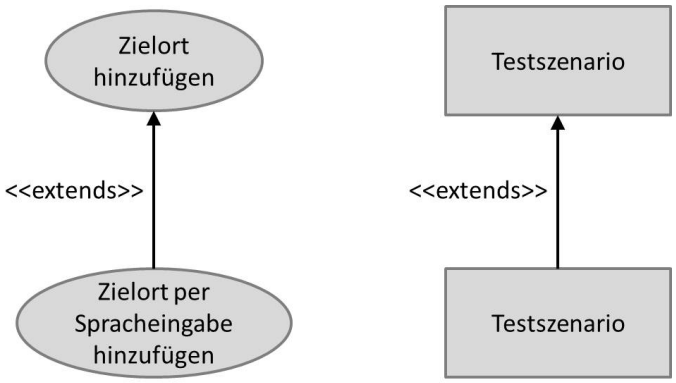

Abbildung 3.15.: Schema Spezialisierung von Usecases (vgl. [McGregor 2001])

Benutzereingabe	**Reaktion des Systems**
Auswahl der Funktion „Zielort per Spracheingabe hinzufügen“	1. Initialisierung der Spracheingabe 2. Anzeige des leeren Eintrags und der Nachricht „Bitte sprechen Sie jetzt“ auf dem Display
Eingabe der benötigten Informationen	1. Speicherung des Zielorts im System 2. Anzeige einer Eingabebestätigung auf dem Display

Tabelle 3.3.: Szenario für Usecase *Zielorteingabe per Spracheingabe* (vgl. [McGregor 2001])

Benutzereingabe	**Reaktion des Systems**
Auswahl der Funktion „Zielort per Spracheeingabe hinzufügen“	1. Initialisierung der Spracheingabe 2. Anzeige des Leeren Eintrags und der Nachricht „Bitte sprechen Sie jetzt“ auf dem Display
Spracheingabe „Berlin, Bahnhofsstraße 1“	1. Speicherung des Zielorts im System 2. Anzeige der Nachricht „Zielort hinzugefügt“ auf dem Display

Tabelle 3.4.: Testszenario für Usecase *Zielorteingabe per Spracheingabe* (vgl. [McGregor 2001])

Die Produkttestfälle werden auf dieselbe Weise zerlegt, beziehungsweise zusammengefasst, wie die Usecases strukturiert werden (vgl. Abbildung 3.16).

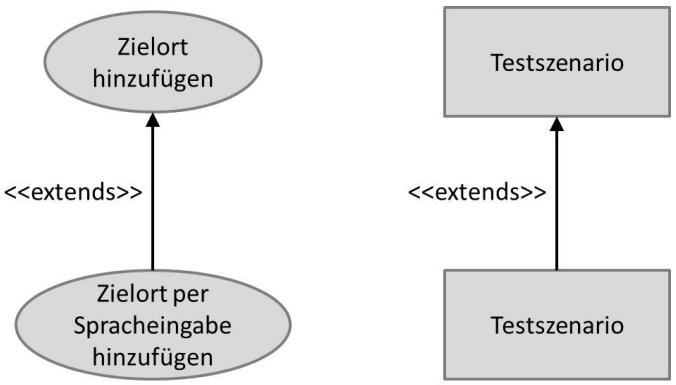

Abbildung 3.16.: Strukturierung von Usecases und Testfällen (nach [McGregor 2001])

[Dueñas u. a. 2004] übertragen in ihrem Ansatz die grundlegenden Prinzipien der modellgetriebenen Entwicklung und der modellgetriebenen Architektur auf den Bereich der Software Produktlinien. Dabei unterscheiden sie wie [McGregor 2001] zwischen den zwei Aspekten des Testens, dem Test der Plattform sowie dem Produkttest, und definieren ein Metamodell in UML-Notation für das Testen innerhalb der Software-Produktlinien-Entwicklung.

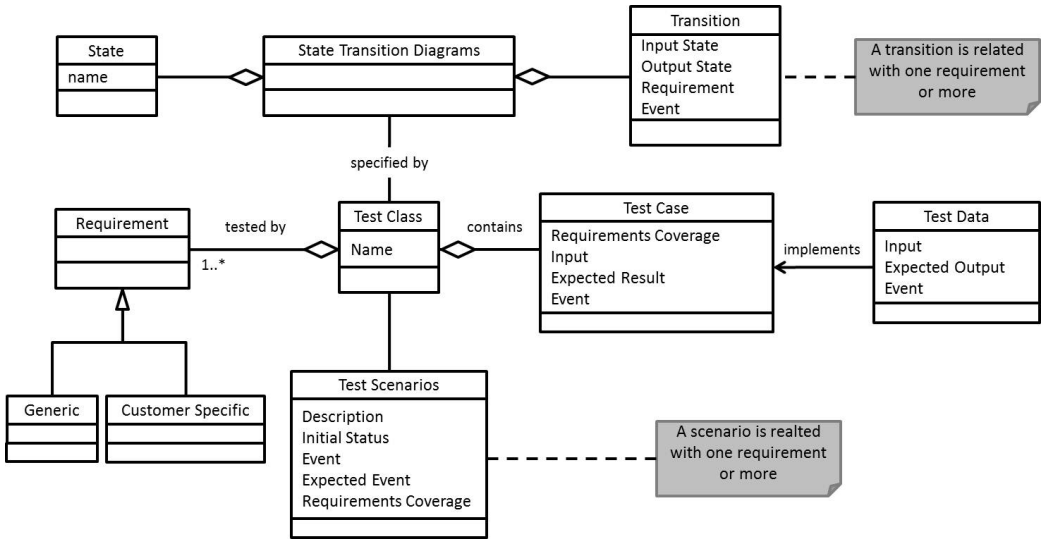

Abbildung 3.17.: Metamodell (nach [Dueñas u. a. 2004])

Aus dem in Abbildung 3.17 dargestellten Metamodell geht hervor, dass sowohl generische Anforderungen als auch produktspezifische Anforderungen das Testen in der Software-Produktlinien-Entwicklung antreiben (citepDu04). Eine Testklasse ist dabei eine Gruppe von Testfällen, die einer bestimmten Systementität zugeordnet

sind. Basierend auf den Zustandsübergangsdiagrammen und Testszenarien wird ein Modell für das Verhalten des Systems entwickelt, aus dem schließlich die Testklassen abgebildet werden. Die Testfälle enthalten Nachverfolgungsinformation bezüglich der Anforderungen, wodurch die Überdeckung der Anforderungen bewertet werden kann. [Dueñas u. a. 2004] erweitern das Metamodell um das Konzept der Variationspunkte und beschreiben damit die Variabilität innerhalb der Testfälle.

Der Ansatz, der der Testfallbestimmung mithilfe der Klassifikationsbaum-Methode am meisten ähnelt, ist die von [Bertolino und Gnesi 2004] vorgestellte *Product Line Use case Test Optimization* (PLUTO). Die Arbeit stellt eine Methode vor, die produktspezifische Testfälle aus erweiterten Usescases ableitet, um den Test von Endprodukten der Produktlinie zu unterstützen.

Im Sinne der Software-Produktlinien-Entwicklung gehen [Bertolino und Gnesi 2004] davon aus, dass die Anforderungen einer Produktlinie aus festen und variablen Bestandteilen bestehen. Der feste Anteil spiegelt die Anforderungen wider, die die gemeinsamen Features oder Funktionen der Produktlinie beschreiben, wohingegen der variable Teil die Funktionalitäten beschreibt, die die einzelnen Produkte voneinander unterscheiden. Die Anforderungsspezifikation der Produktlinie setzt sich somit aus dem festen Teil $\mathbf{R}$, der die gemeinsamen Anforderungen der Produktlinie enthält und dem variablen Teil $\mathbf{R_{var}}$, der die Variationspunkte beschreibt und so die verschiedenen Produkte unterscheidet. Damit ergibt sich für die Anforderungsmenge $\mathbf{Reqs_{PL}}$ der Produktlinie:

$$\mathbf{Reqs_{PL}} = \mathbf{R} + \mathbf{R_{var}}$$

Während des Application Engineering werden die Produkte nicht von Grund auf neu entwickelt, vielmehr wird der variable Anteil in den Anforderungen instanziiert. Damit entsteht die produktspezifische Anforderungsmenge $\mathbf{Reqs_{Pr}}$, die neben den festen Anforderungen eine Menge von instanziierten Anforderungen aufweist:

$$\mathbf{Reqs_{Pr}} = \mathbf{R} + \mathbf{R_{ist}}$$

Auf dieser Grundlage stellen [Bertolino und Gnesi 2004] Regeln zur Wiederverwendung von Artefakten für die Aktivitäten der Testplanung und des Testmanagements auf. Dazu betrachten die Autoren die Menge der Testfälle innerhalb des Domain und des Application Engineering. Auf Ebene der Domain können Testfälle beschrieben

werden, die die gesamte Produktlinie betreffen und so für den Test aller Endprodukte erforderlich sind. Die Menge der erforderlichen Testfälle **T** ist dabei unmittelbar den festen, beziehungsweise erforderlichen Anforderungen in **R** zugeordnet, während der variable Teil $\mathbf{T_{var}}$ die Testfälle enthält, die die variablen Anforderungen $\mathbf{R_{var}}$ abdecken und im Application Engineering instanziiert werden müssen. Damit ergibt sich für die Menge der Produktlinien-Testfälle:

$$\mathbf{Tests_{PL}} = \mathbf{R} + \mathbf{R_{var}}$$

Auf Produktebene lässt sich analog zu den Anforderungen die Menge der Produkttestfälle definieren, die aus dem festen Teil **T** und den instanziierten Variablen Testfällen $\mathbf{T_{var}}$ besteht:

$$\mathbf{Tests_{Pr}} = \mathbf{T} + \mathbf{T_{ist}}$$

Alle beschriebenen Testfälle werden als logische Testfälle aufgefasst. Nachdem [Bertolino und Gnesi 2004] die grundlegenden Überlegungen für Anforderungs- und Testfallmengen angestellt haben, definieren sie eine Erweiterung des Anwendungsfallkonzepts. Das Ergebnis sind *Product Line Use Cases* (PLUCs), die Anforderungen bezüglich der Benutzerziele strukturieren und eine textuelle Beschreibung des Systemverhaltens in Tabellenform bereitstellen. Analog zu normalen Usecases werden Szenarien für die verschiedenen Abläufe innerhalb der PLUCs definiert. Die Variabilität innerhalb der Produktlinie wird innerhalb der PLUCs mithilfe von speziellen Annotationen in Form von Tags modelliert. Dabei werden drei Arten von Tags unterschieden:

- *alternative tags* drücken die Möglichkeit aus, Anforderungen durch die Auswahl einer Instanz aus einer vordefinierten Menge von Wahlmöglichkeiten zu instanziieren, wobei die Auswahl unabhängig von anderen Variationspunkten ist.

- Die Instanziierung von *parametric tags* ist mit dem aktuellen Parameterwert innerhalb einer Anforderung verbunden, der für ein spezifisches Produkt gilt. Jedes dieser Tags ist abhängig vom Auftreten einer Bedingung.

- *optional tags* werden instanziiert, falls ein dazugehöriges optionales Feature im abgeleiteten Endprodukt enthalten ist.

Mithilfe dieser Tags modifizieren [Bertolino und Gnesi 2004] die Category-Partition Method, um darin die Variabilität der Produktlinien handhaben zu können und die produktspezifischen Testfälle zu instanziieren. Die *choices* werden innerhalb der Category-Partition Method mit Constraints annotiert, um redundante und sinnlose Kombinationen zu vermeiden und so die Testfallmenge zu reduzieren. Die Liste der identifizierten *choices* aller Kategorien und die zugewiesenen Constraints bilden dabei die Testspezifikation.

Die von [Bertolino und Gnesi 2004] vorgestellten Variabilitäts-Tags werden analog zur Verwendung von Constraints einer bestimmten Menge von *choices* zugeordnet und bei der Ableitung der Testfälle derartig mit Tagwerten gefüllt, dass nur Kombinationen, die für das spezifische Produkt relevant sind, entstehen. Im Falle des optionalen Tags wird das korrespondierende Feature berücksichtigt, falls es im Endprodukt vorhanden ist. Bei einem alternativen Tag wird ein relevantes Feature ausgewählt; bei Parameter-Tags wird das Feature im Zusammenspiel mit einem bestimmten Wert berücksichtigt. Da die Testfälle aus Usecases abgeleitet wurden, enthält die Testspezifikation eine Kategorie *Szenarios*, die alle Szenarios auflistet. Da die Usecases in natürlicher Sprache formuliert sind, wird auch die von [Bertolino und Gnesi 2004] vorgestellte Erweiterung der Category-Partition Method von Hand ausgeführt.

Zwischen den Szenarien der Usecase einer Produktlinie existieren Abhängigkeiten, die als sogenannte *cross-cutting features* bezeichnet werden. Um solche Abhängigkeiten zu beherrschen, werden die PLUCs mit der Anmerkung *„See another PLUC“* annotiert. Besteht eine derartige Abhängigkeit zwischen zwei PLUCs, werden die Testfälle durch Kombination der mit den PLUCs assoziierten relevanten *choices* realisiert.

Die Testfallmengen für die Produktlinie und die einzelnen Produkte können nun bezüglich der Variabilitäts-Tags beschrieben werden. Dabei enthält die Menge $\mathbf{T}$ der erforderlichen Testfälle der Produktlinie nur solche Testfälle, die keine Variabilitäts-Tags beinhalten, das bedeutet nur die Kombinationen von *choices*, die allen Produkten der Produktlinie gemein sind. Andererseits formen alle möglichen Kombinationen von *choices*, die Tags enthalten, die Menge $\mathbf{T_{var}}$ der variablen Testfälle. [Bertolino und Gnesi 2004] heben dabei hervor, dass für die Produktlinie nicht die gesamte Menge $\mathbf{T_{PL}}$ der Testfälle abgeleitet wird, sondern lediglich die nicht entfaltete Produktlinien-Testspezifikation, aus der Testfälle abgeleitet werden können. Die Ableitung der Testfälle geschieht jedoch erst während des Application Engineering,

wenn ein spezielles Produkt entwickelt wird. Dazu werden in jedem PLUC die Tags mit entsprechenden Werten instanziiert.

Abbildung 3.18 stellt einen PLUC für die verschiedenen Varianten eines Infotainmentsystems aus dem Automobilbereich dar. Die Varianten gehören der selben Produktlinie an. Das Infotainmentsystem verfügt über eine Navigationsfunktion, wobei sich der mitgelieferte Kartenumfang und die verfügbaren Kommunikationsschnittstellen unterscheiden.

Produktlinien USE CASE Navigation

Ziel: Navigieren zu einem Ziel mithilfe eines [V0] Infotainmentsystems und lade Staumeldungen
Scope: Das [V0] Infotainmentsystem
Vorbedingung: Das [V0] Infotainmentsystem ist eingeschaltet, d.h. die Zündung ist an
Auslösendes Ereignis: Die Funktion Navigation wurde im Hauptmenü ausgewählt
Primärer Akteur: Fahrer des Fahrzeugs (der Fahrer)
Sekundärer Akteur: das [V0] Infotainmentsystem (das System)
der Anbieter des Kartenmaterials
Hauptszenario:
1. Das System zeigt die Liste {[V1] verfügbarer} Karten
2. Der Fahrer wählt eine Karte
3. Der Fahrer wählt ein Ziel
4. Der Fahrer startet die Navigation {[V2] und das System lädt die aktuellen Staumeldungen via UMTS}

Erweiterungen:
1a. Keine Karten verfügbar:
1a1. Rückkehr zum Hauptmenü
3a. Der Fahrer startet die Navigation, die läuft bis ein Anruf eingeht

Varianten:
V0: Alternativ:
0. Modell0
1. Modell1
2. Modell2

V1: Parameter:
if V0 = 0 dann zeige Nachricht *„Keine Karten verfügbar“* else
if V0 = 1 dann Karte *Deutschland* else
if V0 = 2 dann Karte *Deutschland* oder *Europa*

V2: Optional
wenn V0 = 2

Abbildung 3.18.: Beispiel eines Usecases in PLUC Notation (vgl. Bertolino und Gnesi [2004])

[Bertolino und Gnesi 2004] führen geschweifte Klammern und die Tags [V0], [V1] und [V2] zur Identifizierung der Variationspunkte innerhalb des Anwendungsfalls ein.

Die Instanziierungsoptionen der variablen Teile werden dabei im Variationsbereich des PLUCs festgelegt. Bei der Anwendung der PLUTO-Methode lassen sich die Kategorien *Infotainment System Modell*, *Karten*, *Ziele*, *Szenarios* und *Verbindung* identifizieren. Mithilfe dieser Kategorien können alle Anforderungen bezüglich der Funktion Navigation validiert werden.

Im nächsten Schritt werden die Kategorien in die relevanten *choices* zerlegt. Dabei ist zu erkennen, dass einige *choices* für alle Produkte der Produktlinie verfügbar sind, wohingegen beispielsweise die Kategorie *Verbindung* und die dazugehörigen *choices* von der jeweiligen Produktvariante abhängen und somit nicht für alle Varianten getestet werden müssen. Dieser Aspekt wird durch das optionale Tag V2 abgebildet und analog zu den Constraint der Category Partition Method bei der Ableitung produktspezifischer Testfälle verwendet. Dazu wird, wie in Abbildung 3.19 dargestellt, den *choices* der Kategorie *Verbindung* eine IF-Bedingung hinzugefügt, die festlegt, dass diese *choices* nur von Interesse sind, wenn die Eigenschaft P2 erfüllt ist, welche für die *choice Modell2* definiert wurde.

PLUC Navigation Testspezifikation

[V0]: Infotainment System Modell:
0. Modell0 [Property P0]
1. Modell1
2. Modell2 [Property P2]

Karten:
Keine Karten [if P0]
Deutschland [if NOT P0]
Europa [if NOT P0] [if P2]

Ziele: [if NOT P0]
Berlin, Bahnhofsstr.1
Hamburg, Am Fischmarkt 2

Szenarios
Hauptszenario [if NOT P0]
Erweiterung: keine Karten verfügbar [if P0]
Erweiterung: Anruf geht ein „seeAnrufBeantwortung“

[V2]: Verbindung:
UMTS Verbindung an [if P2]
UMTS Verbindung aus [if P2]

Abbildung 3.19.: Beispiel einer Testspezifikation den Usecase *Navigation*

Die Generierung aller möglichen Kombinationen für die in Abbildung 3.19 dargestellte Testspezifikation erzeugt die Menge $\mathbf{Tests_{PL}}$ aller Testfälle der Produktlinie, welche alle Testfälle relativ zu dem verwendeten PLUC enthält. Zudem kann direkt die Menge $\mathbf{Tests_{Pr}}$ für ein spezielles Produkt abgeleitet werden, die Werte der Tags innerhalb der Testspezifikation instanziiert werden. Für die Ableitung der Testfälle des *Modell2* würde es beispielsweise genügen die Eigenschaft P2 auf wahr zu setzen.

Mithilfe der Annotation *seeAnrufBeantwortung* wurde die Abhängigkeit zu einem anderen PLUC gekennzeichnet. Für die Ableitung der Testfälle werden somit auch die relevanten *choices* des PLUC *Anrufbeantwortung* für die Kombination berücksichtigt.

3.5. Entwurf eines generischen Verfahrens zum Variantenmanagement

Auf Basis der vorgestellten Frameworks und Forschungsansätze lässt sich ein generisches Verfahren für ein Variantenmanagement bei der Testfallbestimmung mit der Klassifikationsbaum-Methode herleiten.

Bei der Anwendung der Klassifikationsbaum-Methode werden die Testfälle anhand der funktionalen Spezifikation bestimmt. Nicht in jedem Fall ist dabei eine Strukturierung der Anforderungen mithilfe von Usecases gegeben. Mit DOORS und MERAN lassen sich Anforderungen beispielsweise auch durch die Verwendung von Überschriften für die Generierung von Lastenheften strukturieren. Daher werden im Rahmen dieser Arbeit die Anforderungen als Grundlage der Testfallbestimmung betrachtet.

[McGregor 2001] beschreibt ein allgemeines Vorgehen zur Dokumentation von Variabilität in den Anforderungen. Mithilfe eines abstrakten Textersetzungsmechanismus werden hierbei die generellen Szenarien innerhalb der Usecases zu produktspezifischen Szenarien spezialisiert. Da sich der Ansatz jedoch auf einer sehr allgemeinen und abstrakten Ebene bewegt, wird nicht geklärt, wie eine solche Textersetzung im konkreten Fall zu realisieren ist.

MERAN führt, wie in Abschnitt 2.1.3 beschrieben, mit der Benutzung eines speziellen Variantenselektor-Attributs und der Verwendung von Parameterplatzhaltern zwei Konzepte für die Beschreibung von Variabilität innerhalb der Anforderungen

ein. Im Gegensatz zu [McGregor 2001] werden dazu jedoch konkrete Mechanismen für eine praktische Umsetzung definiert. In MERAN werden die Anforderungen mit speziellen Informationen annotiert, die zum einen die Selektion und damit die Zuordnung der Anforderungen zu den einzeln Varianten ermöglichen und zum anderen die Parameterersetzung und die damit verbundene produktspezifische Anpassung der Anforderungen realisieren.

Der Ansatz von [Bertolino und Gnesi 2004] formuliert mit der Verwendung von Variabilitäts-Tags ein sehr ähnliches Konzept zur Darstellung der Variabilität in den Anforderungen, welches auch auf der Annotation der Anforderung mithilfe zusätzlicher Varianteninformationen basiert. Zudem stellen [Bertolino und Gnesi 2004] eine Methode vor, die diese Informationen auch auf die Testfallbestimmung überträgt und somit die Ableitung variantenspezifischer Testfälle erlaubt. Da der Ansatz von [Bertolino und Gnesi 2004] auf der Category-Partition Method beruht, lassen sich die Ergebnisse auch auf die Klassifikationsbaum-Methode übertragen, die denselben Ursprung aufweist.

Die Anreicherung der Anforderungen mit zusätzlichen Varianteninformationen stellt das grundlegende Konzept für die Ableitung variantenspezifischer Testfälle mithilfe der Klassifikationsbaum-Methode dar. Abbildung 3.20 verdeutlicht, wie die von [Bertolino und Gnesi 2004] eingeführten Variabilitäts-Tags auch für die Beschreibung von Variationspunkten in den Anforderungen verwendet werden können.

Die Anforderungsmengen $\mathbf{Reqs_{PL}}$ für die Produktlinie sowie $\mathbf{Reqs_{Pr}}$ für die einzelnen Produkte werden dabei analog zum Ansatz von [Bertolino und Gnesi 2004] auch für die Klassifikationsbaum-Methode definiert.

Auf die in Abbildung 3.20 dargestellten Anforderungen wird in einem ersten Schritt die Klassifikationsbaum-Methode angewendet. Durch die Partitionierung des Testdatenraums entstehen die Klassifikationen *Infotainment System Modell*, *Karten*, *Ziele* und *Verbindung*. Auf die Klassifikation *Szenarios* kann an dieser Stelle verzichtet werden, da die funktionalen Zusammenhänge der Szenarios implizit in den Anforderungen enthalten sind. Analog zu der von [Bertolino und Gnesi 2004] vorgenommenen Auswahl von *choices* werden die Klassen für die einzelnen Klassifikationen gebildet. Abbildung 3.21 präsentiert den auf diese Weise entstandenen Klassifikationsbaum.

Produktlinien Anforderungen Navigation

Anf-01: *Mithilfe eines [V0] Infotainmentsystems soll die Navigation zu einem gewünschten Ziel möglich sein.*

Anf-02: *Aktuelle Staumeldungen sollen für die Navigation zum Ziel berücksichtigt werden.*

Anf-03: *Den Anwendungsbereich soll das [V0] Infotainmentsystem bilden.*

Anf-04: *Das [V0] Infotainmentsystem muss für die Benutzung Navigationsfunktion eingeschaltet sein, d.h. die Zündung muss an sein.*

Anf-05: *Die Navigationsfunktion soll über ein Hauptmenüeintrag ausgewählt werden können.*

Anf-06: *Die primäre Rolle ist der Fahrer des Fahrzeugs (der Fahrer). Sekundäre Rollen stellen das [V0] Infotainmentsystem (das System) und der Anbieter des Kartenmaterials dar.*

Anf-07: *Nach Auswahl der Navigationsfunktion im Hauptmenü soll das System eine Liste {[V1] verfügbarer} Karten anzeigen.*

Anf-08: *Fahrer soll eine Karte und ein Ziel wählen können.*

Anf-09: *Nachdem der Fahrer Karte und Ziel gewählt hat, soll er die Navigation starten können {[V2] und während der Navigation soll das System aktuelle Staumeldung via UMTS herunterladen und bei der Navigation berücksichtigen}.*

Anf-10: *Wenn keine Karte verfügbar ist, soll das [V0] Infotainmentsystem in Hauptmenü zurückspringen.*

Anf-11: *Die Navigation soll von einem eingehenden Anruf unterbrochen werden.*

Varianten:

V0: Alternativ:
0. Modell0
1. Modell1
2. Modell2

V1: Parameter:
wenn V0 = 0 dann zeige Nachricht *„Keine Karten verfügbar“* sonst
wenn V0 = 1 dann Karte *Deutschland* sonst
wenn V0 = 2 dann Karte *Deutschland* oder *Europa*

V2: Optional
wenn V0 = 2

Abbildung 3.20.: Anforderungen für die Funktion *Navigation* des Infotainmentsystems

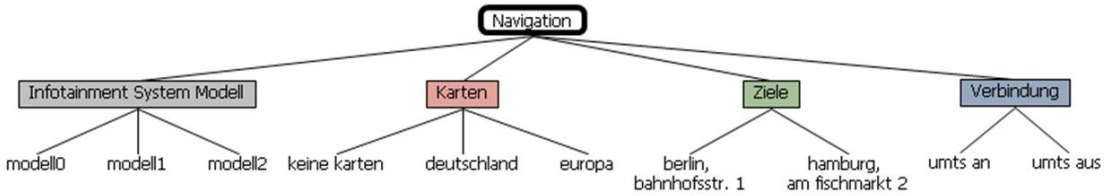

Abbildung 3.21.: Klassifikationsbaum für das Beispiel des Infotainmentsystems

Analog zu [Bertolino und Gnesi 2004] werden in einem nächsten Schritt die Constraints in Form von Abhängigkeitsregeln auf die Klassifikationen und Klassen übertragen, womit sich folgende Abhängigkeitsregeln ergeben:

1. *keine karten* $\Longrightarrow$ *modell0*
2. *deutschland* $\Longrightarrow$ *NOT modell0*
3. *europa* $\Longrightarrow$ *NOT modell0 AND modell2*
4. *ziele* $\Longrightarrow$ *NOT modell0*
5. *umts an* $\Longrightarrow$ *modell2*
6. *umts aus* $\Longrightarrow$ *modell2*

Die Testfallmengen für die Produktlinie und Produkte lassen sich damit auch bezüglich der Klassen im Klassifikationsbaum beschreiben. Dabei enthält die Menge $\mathbf{T}$ der generellen, beziehungsweise erforderlichen Testfälle der Produktlinie, nur derartige Testfälle, die keine Variabilitäts-Tags enthalten, das bedeutet nur die Kombinationen von Klassen, die allen Produkten der Produktlinie gemein sind. Die Menge $\mathbf{T_{var}}$ der variablen Testfälle wird hingegen von allen mit Tags behafteten Kombinationen gebildet.

Der entwickelte Klassifikationsbaum stellt in Verbindung mit den definierten Abhängigkeitsregeln die nicht entfaltete Testspezifikation dar. Analog zu [Bertolino und Gnesi 2004] wird im Domain Engineering nicht die gesamte Testfallmenge $\mathbf{Tests_{PL}}$ der Produktlinie gebildet. Dies geschieht erst bei der Entwicklung der einzelnen Produktvarianten im Application Engineering, wobei durch Instanziierung der Variabilitäts-Tags und die Kombination der Klassen die Menge $\mathbf{Tests_{Pr}}$ der produktspezifischen Testfälle abgeleitet wird. Abbildung 3.22 stellt die auf diese Weise abgeleiteten Testfälle für die Produktvariante *modell2* dar.

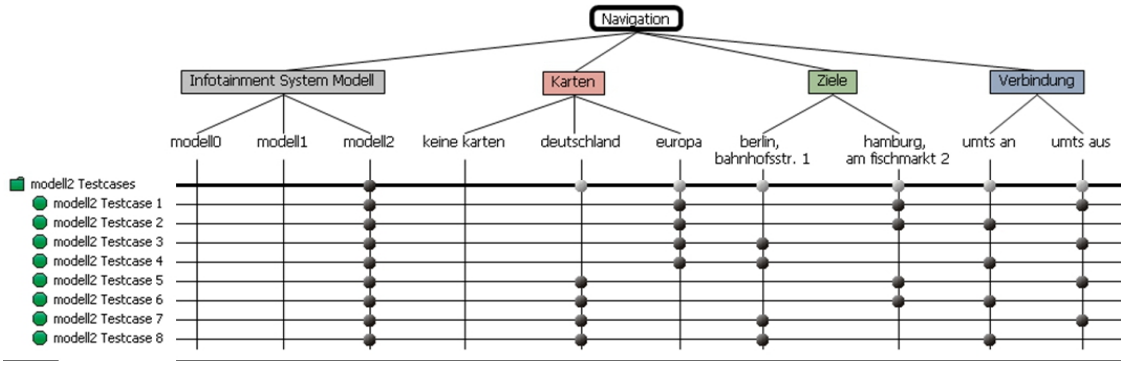

Abbildung 3.22.: Klassifikationsbaum und Testfallmenge für die Variante *modell2*

Aufgrund der Abhängigkeitsregeln werden nur sinnvolle und für das konkrete Produkt relevante Testfälle abgeleitet. Zudem werden auch nur die Klassen kombiniert, für die Features im Produkt enthalten sind.

Die definierten Parameter-Werte wurden als Klassen der Klassifikation *Karten* in den Baum hinzugefügt. Parameter stellen ein wichtiges Differenzierungsmerkmal von Produktvarianten dar. Auch wenn die Parameterwerte auf Ebene der domainübergreifenden Testspezifikation noch nicht zur Differenzierung einzelnen Produktvarianten herangezogen werden können, da diese noch nicht instanziiert sind ([Bertolino und Gnesi 2004]), identifizieren sie die Produktvarianten bei der Entwicklung der spezifischen Produkte.

Neben den im Beispiel des Infotainmentsystems definierten Zeichenketten-Parametern, sind insbesondere numerische Parameterwerte für die Produktenentwicklung von großer Bedeutung. Diese können analog zu den Zeichenketten-Parametern mithilfe der Parameter-Tags definiert werden. Das Beispiel des Infotainmentsystems lässt sich so um eine maximale Anzahl von Zwischenhalten erweitern. Abbildung 3.23 stellt die erweiterte Testspezifikation für das Infotainmentsystem dar.

Dabei wurde die Menge der Anforderungen um die Anforderung *Anf-12* erweitert, die die maximale Anzahl verwendbarer Zwischenziele im System spezifiziert. Zusätzlich wurde das Parameter-Tag **V3** eingeführt, welches anhand von Bedingungen die Instanziierung von Parameterwerten steuert. Auf Basis dieser Anforderungsspezifikation werden wiederum mithilfe der Klassifikationsbaum-Methode die Klassifikationen und Klassen gebildet. Für den neu eingeführten Parameter wird eine neue Klassifikation *Anzahl* identifiziert, die daraufhin in verschiedene Klassen zerlegt wird. Wie in Abbildung 3.24 veranschaulicht, werden dabei die Parameterwerte als eigenständige zu testende Klassen zur Klassifikation *Anzahl* hinzugefügt.

Produktlinien Anforderungen Navigation

...

...

Anf-12: *Die Navigation soll die Eingabe von maximal {[V3]} Zwischenzielen unterstützen.*

Varianten:

V0: Alternativ:
- 0. Modell0
- 1. Modell1
- 2. Modell2

V1: Parameter:
- wenn V0 = 0 dann zeige Nachricht *„Keine Karten verfügbar“* else
- wenn V0 = 1 dann Karte *Deutschland* else
- wenn V0 = 2 dann Karte *Deutschland* oder *Europa*

V2: Optional
- wenn V0 = 2

V3: Parameter:
- wenn V0 = 1 dann V3 = 10
- wenn V0 = 2 dann V3 = 100

Abbildung 3.23.: Erweiterte Anforderungsspezifikation

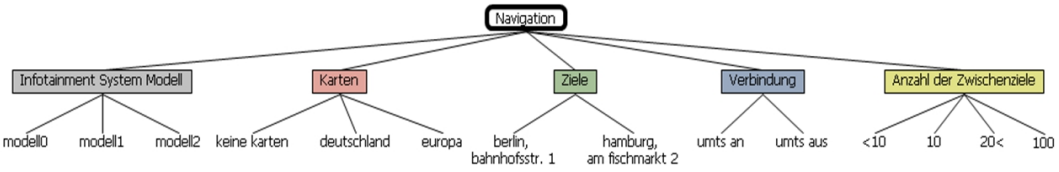

Abbildung 3.24.: Erweiterung des Beispiels eines Infotainmentsystems um Parameter Anzahl der Zwischenziele

Die Testfallspezifikation enthält somit für die Parameter-Klassifikation *Anzahl* neben den durch die Klassifikationsbaum-Methode identifizierten Klassen verschiedene Parameterklassen, die über Abhängigkeitsregeln den einzelnen Produktvarianten zugeordnet werden. Die Menge der Abhängigkeitsregeln muss dazu um die Regeln:

1. $100 \implies modell2$
2. $10 \implies modell1$

erweitert werden.

4. Konzept für die prototypische Umsetzung

Ziel dieses Kapitels ist die Ermittlung und Formulierung der Anforderungen an das zu entwickelnde Varianten-Management-Plug-In. Dabei wird zunächst eine allgemeine Beschreibung des CTE XL Professional und seiner Schnittstellen zu den anderen Werkzeugen gegeben. Im Anschluss daran werden die Anwendungsfälle für das Plug-In formuliert und sowohl funktionale als auch nicht-funktionale Anforderungen in textueller Form definiert.

4.1. Allgmeine Systembeschreibung

Wie in Abschnitt 2.2.2 beschrieben, ist der CTE XL Prof. ein grafischer Editor zur Unterstützung der Klassifikationsbaum-Methode, dessen technische Realisierung auf der Eclipse Rich-Client-Platform ([Eclipse-Foundation 2011b]) basiert. Die Rich-Client-Platform stellt ein Framework für die Entwicklung grafischer Plug-in-basierter Applikationen dar. Plug-Ins sind einzelne Software-Module, die frei miteinander kombiniert werden können. Die zentrale Steuerung übernimmt dabei eine Laufzeitumgebungs-Komponente, die die benötigten Plug-Ins läd und diese zur Laufzeit hinzufügen beziehungsweise entfernen kann. Wie in Abbildung 4.1 dargestellt, definiert eine Plug-In eine Reihe von sogenannten Extension Points, mit deren Hilfe bestimmte Funktionalitäten anderen Plug-Ins zur Verfügung gestellt werden können. Der CTE XL Prof. wird dementsprechend durch eine Menge von Plug-Ins realisiert, die miteinander interagieren und von denen jedes einzelne eine bestimmte Funktionalität implementiert. Eine derartige Architektur kann durch Hinzufügen oder Entfernen einzelner Plug-Ins leicht modifiziert werden.

Ein Beispiel für die Realisierung einer Funktionalität mithilfe von Plug-Ins stellt die Schnittstelle zwischen CTE XL Prof. und DOORS, beziehungsweise MERAN dar.

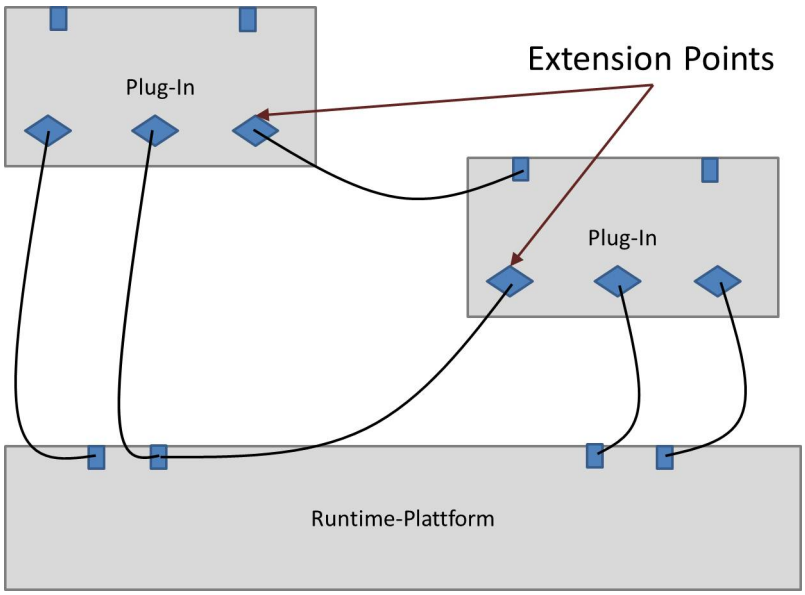

Abbildung 4.1.: Extension Point Konzept (vgl. Eclipse-Foundation [2011b])

Diese wird durch drei verschiedene Plug-Ins realisiert, die unterschiedliche Aufgaben wie den Zugriff auf das Datenmodell, die Kommunikation mit dem DOORS-Server oder die Präsentation mittels Benutzerschnittstelle implementieren und miteinander in Wechselwirkung stehen. Die Kommunikation mit DOORS wird dabei über einen Kommandozeilenaufruf des installierten DOORS-Clients realisiert. Dieser stellt mithilfe von DXL-Skripten Anfragen an den Server, um die gewünschten Ergebnisse geliefert zu bekommen.

Das dem CTE XL Prof. zugrundeliegende Datenmodell wurde auf Basis des Eclipse Modeling Frameworks (EMF) ([Eclipse-Foundation 2011a]) entwickelt und enthält die in Abbildung 4.2 dargestellten Datentypen.

Der *GobalSuperType* bildet den grundlegenden Datentyp, von dem alle anderen Typen abgeleitet werden. Jeder *GlobalSuperType* enthält ein Objekt des Typs *Tag*. Damit wird sämtlichen Objekten innerhalb des Datenmodells ein *Tag* zugeordnet und somit können alle Daten mit zusätzlichen Informationen annotiert werden.

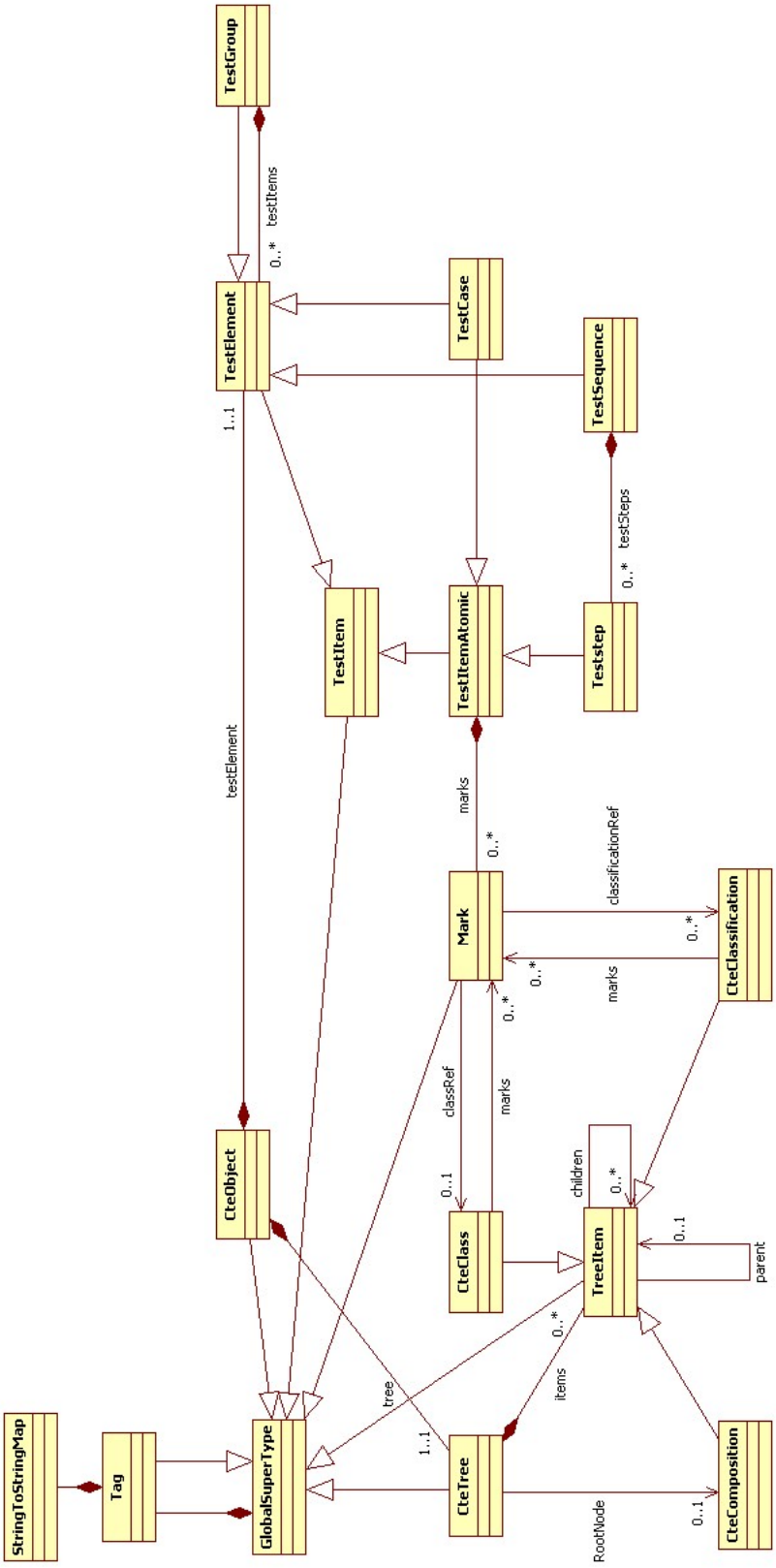

Abbildung 4.2.: Datenmodell des CTE XL Prof.

4.2. Anwendungsfälle

Für das zu entwickelnde Varianten-Management-Plug-In wurden die in Abbildung 4.3 dargestellten Anwendungsfälle identifiziert.

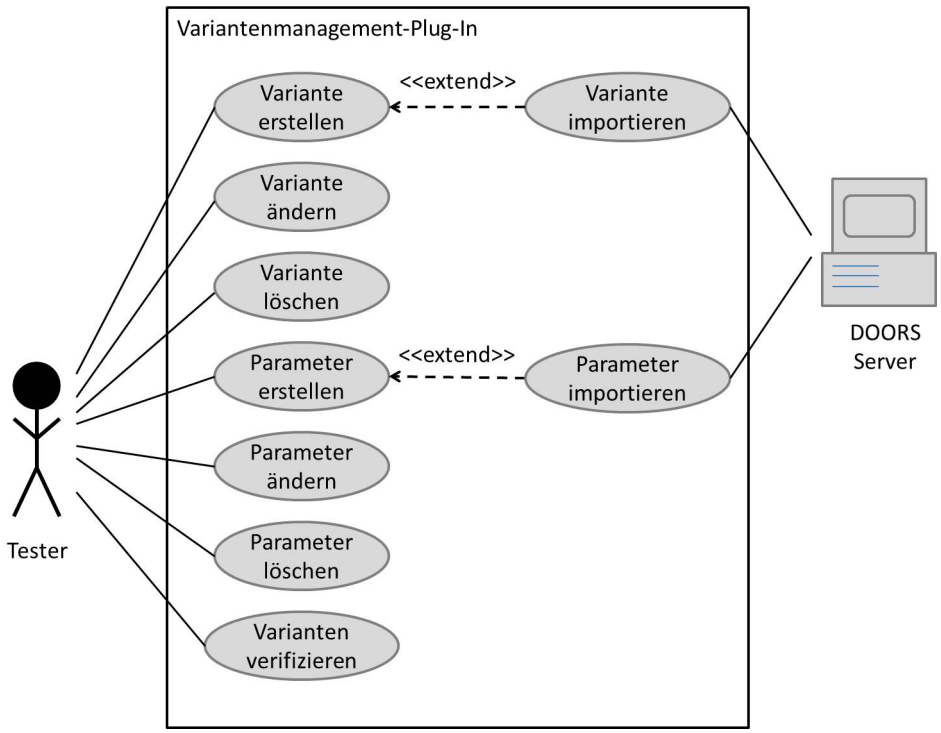

Abbildung 4.3.: Usecase-Diagramm für Variantenmanagement-Plug-In

4.2.1. Variante erstellen

Ziel

Der Tester erstellt eine Variante auf einem existierenden Klassifikationsbaum.

Vorbedingung

- Ein nicht leerer Klassifikationsbaum ist im CTE XL Prof. geöffnet.

Nachbedingung

- Es liegt ein Klassifikationsbaum mit Varianten vor.

Standardablauf

1. Eingabe eines neuen Variantennamens.
2. Prüfung, ob der Variantenname bereits verwendet wird.
3. Zuordnung von Baumelementen zur Variante.
4. Erstellung von Testfällen.

Alternativabläufe

Tester gibt bereits verwendeten Variantennamen ein:

1. Der Tester wird informiert, dass der Variantenname schon von einer anderen Variante verwendet wird.
2. Eingabe eines alternativen Variantennamens.

Tester gibt ungültigen Variantennamen ein:

1. Der Tester wird informiert, dass der Variantenname ungültig ist.
2. Eingabe eines alternativen Variantennamens ein.

4.2.2. Variante ändern

Ziel

Der Tester ändert eine Variante auf einem existierenden Klassifikationsbaum.

Vorbedingung

- Ein nicht leerer Klassifikationsbaum mit Varianten ist im CTE XL Prof. geöffnet.

Nachbedingung

- Es liegt ein Klassifikationsbaum mit geänderten Varianten vor.

Standardablauf

1. Auswahl der zu ändernden Variante.
2. Änderung des Variantennamens geändert.
3. Prüfung, ob der Variantenname bereits verwendet wird.

Alternativabläufe

Der Tester ändert ein Varianten-Baumelement:

1. Auswahl des zu ändernden Varianten-Baumelements.
2. Name des Varianten-Baumelements wird geändert.

Der Tester ändert ein Varianten-Testelement:

1. Auswahl des zu ändernden Varianten-Testelements.
2. Name eines Varianten-Testelements wird geändert.

Der Tester erstellt ein neues Varianten-Baumelement:

1. Neues Varianten-Baumelement wird erstellt.

Der Tester erstellt ein neues Varianten-Testelement:

1. Neues Varianten-Testelement wird erstellt.

Der Tester fügt ein weiteres generisches Baumelement zur Variante hinzu:

1. Auswahl des hinzuzufügenden generischen Baumelements.
2. Zusätzliches generisches Baumelement wird Variante zugeordnet.

Der Tester entfernt ein Baumelement aus der Variante:

1. Auswahl des zu entfernenden Varianten-Baumelements.
2. Baumelement wird aus Variante entfernt.

Der Tester entfernt ein Testelement aus der Variante:

1. Auswahl des zu entfernenden Varianten-Testelements.
2. Testelement wird aus Variante entfernt.

4.2.3. Variante löschen

Ziel

Der Tester löscht eine Variante aus einem existierenden Klassifikationsbaum.

Vorbedingung

- Ein nicht leerer Klassifikationsbaum mit Varianten ist im CTE XL Prof. geöffnet.

Nachbedingung

- Es liegt ein Klassifikationsbaum ohne die gelöschte Variante vor.

Standardablauf

1. Auswahl der zu löschenden Variante.
2. Variante wird inklusive ihrer Baum- und Testelemente gelöscht.

Alternativabläufe

4.2.4. Parameter erstellen

Ziel

Der Tester erstellt einen Parameter für einen existierenden Klassifikationsbaum.

Vorbedingung

- Ein nicht leerer Klassifikationsbaum mit Varianten ist im CTE XL Prof. geöffnet.

Nachbedingung

- Es liegt ein Klassifikationsbaum mit Varianten und Parametern vor.

Standardablauf

1. Eingabe eines neuen Parameternamens.
2. Prüfung, ob der Parametername bereits verwendet wird.
3. Eingabe der Varianten-Parameterwerte und des default-Parameterwerts.
4. Zuweisung des Parameters zu einer Klassifikation.

Alternativabläufe

Der Tester gibt einen bereits vorhandenen Parameternamen ein:

1. Der Tester wird informiert, dass der Parametername schon von einem anderen Parameter verwendet wird.
2. Eingabe eines alternativen Parameternamens.

Der Tester gibt einen ungültigen Parameternamen ein:

1. Der Tester wird informiert, dass der Parametername ungültig ist.

2. Eingabe eines alternativen Parameternamens.

4.2.5. Parameter ändern

Ziel

Der Tester ändert einen Parameter für einen existierenden Klassifikationsbaum.

Vorbedingung

- Ein nicht leerer Klassifikationsbaum mit Varianten und Parametern ist im CTE XL Prof. geöffnet.

Nachbedingung

- Es liegt ein Klassifikationsbaum mit Varianten und geänderten Parametern vor.

Standardablauf

1. Auswahl des zu ändernden Parameters.
2. Eingabe des geänderten Parameternamens.
3. Prüfung, ob der Parametername bereits verwendet wird.
4. Änderung des Parameternamens.

Alternativabläufe

Der Tester ändert den Parameterwert:

1. Änderung eines Parameterwertes.

Tester gibt bereits vorhandenen Parameternamen ein:

1. Der Tester wird informiert, dass der Parametername schon von einem anderen Parameter verwendet wird.
2. Eingabe eines alternativen Parameternamens.

4.2.6. Parameter löschen

Ziel

Der Tester löscht einen Parameter eines existierenden Klassifikationsbaumes.

Vorbedingung

- Ein nicht leerer Klassifikationsbaum mit Varianten und Parametern ist im CTE XL Prof. geöffnet.

Nachbedingung

- Es liegt ein Klassifikationsbaum mit Varianten und ohne den gelöschten Parameter vor.

Standardablauf

1. Auswahl des zu löschenden Parameters.
2. Parameter wird gelöscht.

Alternativabläufe

4.2.7. Variante importieren

Ziel

Der Tester importiert eine Variante aus einer Anforderungsspezifikation in einen Klassifikationsbaum.

Vorbedingung

- Ein nicht leerer Klassifikationsbaum ist in CTE XL Prof. geöffnet. Eine Verbindung zu einem Spezifikationswerkzeug lässt sich herstellen.

Nachbedingung

- Es liegt ein Klassifikationsbaum mit Varianten vor.

Nachbedingung Fehlschlag

- Der Tester wird darüber benachrichtigt, dass der Import der gewählten Variante nicht durchgeführt werden konnte.

Standardablauf

1. Auswahl einer Datenquelle.
2. Auswahl eines Spezifikationsmoduls.
3. Auswahl der Varianten-Selektor-Spalten.
4. Varianten-Namen werden importiert.
5. Der Tester ordnet selbst Baum- und Testelemente zu.

Alternativabläufe

Der Tester wählt eine nicht erreichbare Datenquelle aus:

1. Der Tester wird informiert, dass die Datenquelle nicht erreichbar ist.
2. Auswahl einer alternativen Datenquelle.

Es wurde eine Datenquelle ohne Module geladen:

1. Der Tester wird informiert, dass die Module nicht verfügbar sind.
2. Auswahl einer alternativen Datenquelle.

Der Tester wählt ein ungültiges Variantenselektor-Attribut:

1. Der Tester wird informiert, dass die Varianten-Selektor-Spalte keine Varianten enthält.
2. Wahl eines alternativen Attributes.

Der Tester wählt eine Variante, deren Name im CTE XL Prof. schon enthalten ist:

1. Der Tester wird informiert, dass der Varianten Name schon enthalten ist.
2. Eingabe eines alternativen Variantennamens.

Der Tester wählt Variante deren Name im CTE XL Prof. schon enthalten ist:

1. Der Tester überschreibt die bestehende Variante.

Der Tester wählt eine Variante, deren Name im CTE XL Prof. schon enthalten ist:

1. Der Tester behält die bestehende Variante bei.

4.2.8. Parameter importieren

Ziel

Der Tester importiert einen Parameter aus einer Anforderungsspezifikation in einen Klassifikationsbaum.

Vorbedingung

- Ein nicht leerer Klassifikationsbaum mit Varianten ist im CTE XL Prof. geöffnet. Eine Verbindung zu einem Spezifikationswerkzeug lässt sich herstellen.

Nachbedingung

- Es liegt ein Klassifikationsbaum mit Varianten und Parametern vor.

Nachbedingung Fehlschlag

- Der Tester wird darüber benachrichtigt, dass der Import des gewählten Parameters nicht durchgeführt werden konnte.

Standardablauf

1. Auswahl einer Datenquelle.
2. Auswahl eines Parametermoduls.
3. Auswahl des Parameternamen-Attributes.
4. Parametername und Parameterwerte für die Varianten werden übernommen.

Alternativabläufe

Der Tester wählt eine nicht erreichbare Datenquelle aus:

1. Der Tester wird informiert, dass die Datenquelle nicht erreichbar ist.
2. Auswahl einer alternativen Datenquelle.

Der Tester wählt eine Datenquelle ohne Parametermodul:

1. Der Tester wird informiert, dass das Parametermodul nicht verfügbar ist.
2. Auswahl einer alternativen Datenquelle.

Der Tester wählt ein Attribut, welches keine Parameternamen enthält:

1. Der Tester wird informiert, dass ein Parameternamen-Attribut keine Parameternamen enthält.
2. Auswahl eines alternativen Attributes

Der Tester wählt einen Parameter, dessen Name schon in CTE XL Prof. enthalten ist:

1. Der Tester wird informiert, dass der Parametername schon enthalten ist.

Der Tester wählt einen Parameter, dessen Name schon in CTE XL Prof. enthalten ist:

1. Der Tester wird informiert, dass der Parametername schon enthalten ist.
2. Der Tester überschreibt den bestehenden Parameter.

Der Tester wählt einen Parameter, dessen Name schon in CTE XL Prof. enthalten ist:

1. Der Tester wird informiert, dass der Parametername schon enthalten ist.
2. Der Tester behält den bestehenden Parameter bei.

4.2.9. Varianten verifizieren

Ziel

Der Tester nutzt die Verifikationsfunktion um fehlende bzw. fehlerhafte Sachverhalte aufzudecken.

Vorbedingung

- Ein nicht leerer Klassifikationsbaum mit Varianten ist im CTE XL Prof. geöffnet. Eine Verbindung zu einem Spezifikationswerkzeug lässt sich herstellen.

Nachbedingung

- Fehlende bzw. fehlerhafte Sachverhalte werden hervorgehoben.

Standardablauf

1. Der Tester aktiviert die Verifikationsfunktion.
2. Fehlende bzw. fehlerhafte Sachverhalte werden hervorgehoben.

Alternativabläufe

Es sind keine fehlerhaften oder falschen Sachverhalte enthalten:

1. Der Tester wird informiert, dass keine fehlenden bzw. falschen Sachverhalte zur Anzeige vorhanden sind.

4.3. Anforderungen

Nicht-funktionale Anforderungen

Das Varianten Plug-In interpretiert einen mit CTE XL Prof. erstellten Klassifikationsbaum, der noch keine Varianten enthält, als generischen Baum.

Die Parametererstellung findet in einer gesonderten Parameter View statt. Ein Parametername ist innerhalb einer Klassifikation eindeutig.

Funktionale Anforderungen

Auf Basis des generischen Baumes werden die Zuordnungen von Baumelementen zu den einzelnen Varianten realisiert. Im generischen Baum wird analog zum in Abschnitt 3.5 entwickelten generischen Verfahren eine Klassifikation angezeigt, die die Variantennamen als Klassen enthält. Zudem werden hier alle Testfälle aller Varianten angezeigt.

Enthält die geöffnete .cte-Datei mindestens eine Variante, kann der Tester mittels der Variantenschaltflächen im CTE XL Prof. zwischen den einzelnen Varianten und dem generischen Baum hin- und herschalten. Beim Umschalten zwischen den Varianten werden immer nur die Baum- und Testelemente angezeigt, die zu der jeweiligen Variante gehören.

Änderungen an Parametern in der Parameter View werden sofort in den Baum übernommen.

4.3.1. Variante erstellen

- Den Varianten werden sowohl Baum- als auch Testelemente zugeordnet.
- Die Zuordnung wird mittels Drag'n'Drop realisiert.
- Bei der Zuordnung einer Klassifikation zu einer Variante werden die dazugehörigen Klassen auch der Variante zugeordnet.
- Bei der Zuordnung einer einzelnen Klasse zu einer Variante wird auch die darüber liegende Klassifikation der Variante zugeordnet.
- Die Informationen über erstellten Varianten werden im .cte-File gespeichert.
- Bei der Zuordnung einer Klasse zu einer Variante wird eine Abhängigkeitsregel der Form: *Klasse* $\Longrightarrow$ *Variante* erzeugt.

4.3.2. Variante ändern

- Es lassen sich der neben Variantennamen, Baumelementbezeichnern und Testfallnamen auch die Menge der zugeordneten Varianten-Baumelemente und die Menge der Varianten-Testelemente ändern.
- Es können Baum- und Testelemente für die zu ändernde Variante neu erstellt werden.
- Der zu ändernden Variante können generische Baumelemente hinzugefügt oder entfernt werden.

- Elemente, die für einzelne Varianten erstellt wurden, werden auch im generischen Baum angezeigt und sind initial nur dieser Variante zugeordnet. Später können die auf diese Weise angelegten Baumelemente auch anderen Varianten zugeordnet werden.

- Werden der Bezeichner der Baum- und Testelemente oder die Markierungen in den Testfällen für eine Variante geändert, wird die Änderung auch in den anderen Varianten und im generischen Baum übernommen.

4.3.3. Variante löschen

- Das Löschen der Varianten ist nur über einen Varianten-Wizard und nach Bestätigung möglich.

- Beim Löschen werden die Varianten-Zuordnungen zu Baumelementen, die Varianten-Testfälle und die Variantenparameterwerte in der Parameterview entfernt.

- Das endgültige Löschen von Baumelementen kann nur im generischen Baum vorgenommen werden.

- Die dazugehörigen Informationen werden aus dem .cte-File gelöscht.

4.3.4. Parameter erstellen

- Die Parameter View lässt sich nur bei geöffnetem Klassifikationsbaum mit Varianten öffnen.

- Das Variantenmanagement-Plug-In erkennt die erstellten Varianten und bietet deren Namen für die Parameter-Wert-Zuweisung an.

- Wurde nur ein default-Wert für einen Parameter gesetzt, wird dieser für alle Varianten ausgegeben.

- Bei der Erstellung eines neuen Parameters wird dieser nach Eingabe des Parameternamens per Drag'n'Drop einer Klassifikation zugeordnet. Der Klassifikation wird dabei automatisch eine neue Klasse hinzugefügt. Im generischen Baum wird als Bezeichner für diese Klasse der Parametername angezeigt, in

der Variantenansicht wird dieser durch den variantenspezifischen Parameterwert ersetzt.

- Parameter können nur Klassifikationen zugeordnet werden.
- Die Zuordnung kann entweder in den Varianten stattfinden oder im generischen Baum.
- Beim Anklicken eines Baumelements werden die dazugehörigen Werte in der Parameter View hervorgehoben, umgekehrt werden beim Anklicken eines Parameters die dazugehörigen Baumelemente hervorgehoben beziehungsweise selektiert.
- Im generischen Baum werden auf diese Weise alle Parameterwerte hervorgehoben.
- Wird im CTE XL Prof. eine neue Variante erzeugt, wird deren Name automatisch in der Parametertabelle als Spalte hinzugefügt.

4.3.5. Parameter ändern

- Parameter-Namen und Werte lassen sich über die Parameter View ändern.

4.3.6. Parameter löschen

- Beim Löschen eines Parameters werden dessen variantenspezifischen Werte aus der Parameter-View und die angelegte Klasse im Klassifikationsbaum gelöscht.

4.3.7. Varianten importieren

- Beim Import von Varianten aus DOORS oder MERAN können bereits angelegte Verlinkungen zwischen Anforderungen und Baum- beziehungsweise Testelementen sowie die Zuordnung von Anforderungen zu Varianten für eine Vorselektion der zur Variante gehörenden Baum- bzw. Testelemente genutzt werden.
- Nach der Bestätigung durch den Nutzer werden diese Baum- bzw. Testelemente auch im CTE XL Prof. der dazugehörigen Variante zugeordnet.

4.3.8. Parameter importieren

- Beim Parameter Import muss sichergestellt werden, dass das Parametermodul zum ausgewählten Anforderungsmodul passt.

4.3.9. Varianten verifizieren

- Bei der Verifizierung soll erkannt werden, wenn durch Abwesenheit eines Baumelements eine Abhängigkeitsregel unbrauchbar wird. Die beteiligten Elemente sollen grafisch hervorgehoben werden.

- Für die Prüfung auf Überdeckung kann weiterhin die Requirements View des CTE XL Prof. verwendet werden, da diese die Überdeckungsprüfung auch für die Varianten durchführt.

5. Evaluation des Verfahrens

Ohne geeignete Mechanismen und Verfahren müsste die Klassifikationsbaum-Methode bei der Anwendung auf variantenreiche Systeme innerhalb der Produktlinien-Entwicklung prinzipiell für jede Produktvariante einzeln angewendet werden. Abbildung 5.1 verdeutlicht dies am Beispiel der Fahrzeugvarianten Limousine, Cabriolet und Kombi der Mercedes E-Klasse ([Mercedes-Benz 2011]).

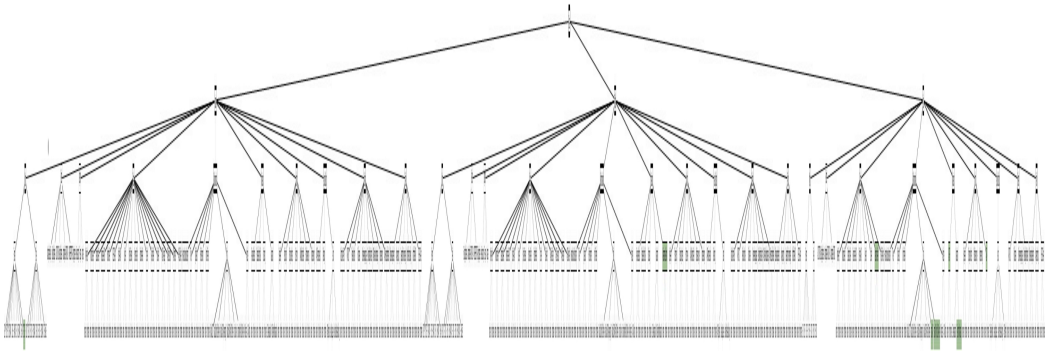

Abbildung 5.1.: Klassifikationsbaum für die drei Fahrzeugvarianten Cabrio, Limousine und Kombi der Mercedes E-Klasse ([Mercedes-Benz 2011])

Der dargestellte Klassifikationsbaum enthält die im Mercedes-Benz-Konfigurator ([Mercedes-Benz 2011]) wählbaren Features der drei Fahrzeugvarianten als Aspekte des Testobjektes E-Klasse und könnte somit beispielsweise für die Konfiguration von Testfahrzeugen herangezogen werden. Bei der Betrachtung des Baumes lassen sich die drei Teilbäume der Varianten erkennen, die zudem einen hohen Anteil gemeinsamer Klassifikationen und Klassen aufweisen. Für eine derartige Modellierung lassen sich zwei wesentliche Nachteile formulieren. Zum einen werden die entstehenden Bäume so groß, dass sie sich nur noch schlecht mit den Scrollfunktionen des CTE XL Prof. handhaben lassen, auf der anderen Seite wurden viele Aspekte redundant modelliert, das bedeutet für die Änderung eines Aspektes, dass sie im schlechtesten Fall in allen drei Teilbäumen vollzogen werden muss. Die Kombination beider Nachteile behindert die Arbeit des Testers signifikant.

Im Gegensatz dazu stellt die Abbildung 5.2 einen Klassifikationsbaum dar, der für die gleichen Varianten jedoch unter Verwendung des in Abschnitt 3.5 entwickelten generischen Verfahrens bestimmt wurde.

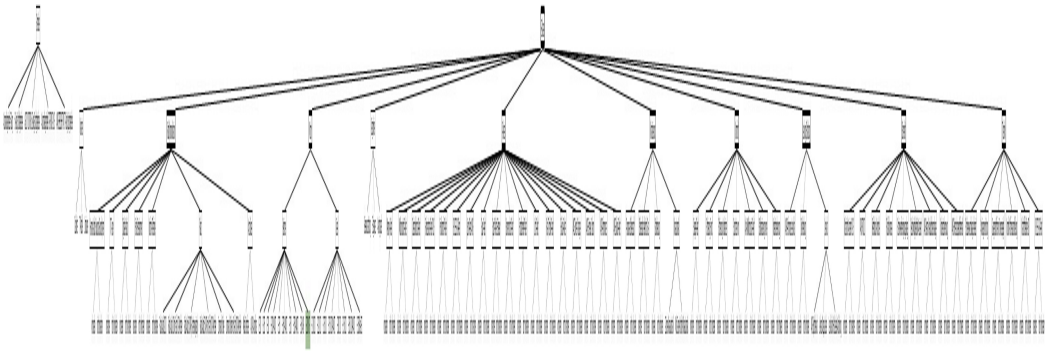

Abbildung 5.2.: Klassifikationsbaum für die drei Fahrzeugvarianten Cabrio, Limousine und Kombi der Mercedes E-Klasse ([Mercedes-Benz 2011])

Mithilfe des generischen Verfahrens konnten die drei redundanten Teilbäume auf einen Baum reduziert werden. Damit fällt die redundante Pflege derselben Testaspekte in unterschiedlichen Teilbäumen weg und die Größe des Baumes konnte auf ein Drittel des Ausgangsbaumes reduziert werden. In Abhängigkeit von der Variantenanzahl ergeben sich somit Einsparungen bezüglich der Größe der Klassifikationsbäume und bei der benötigten Bearbeitungszeit.

5.1. Beispielszenario

Da die Baumgröße des E-Klasse-Beispiels, selbst unter Anwendung des entwickelten generischen Verfahrens, eine leichte und übersichtliche Betrachtung behindert, wird zur Bewertung des Verfahrens anhand der prototypischen Umsetzung, wieder das Beispiel des Infotainmentsystems aus Abschnitt 3.5 herangezogen.

Als Szenario wird die Entwicklung einer Navigationsfunktion innerhalb des Infotainmentsystems angenommen. In diesem Zusammenhang sollen die variantenspezifischen Testfälle für die drei Modell-Varianten abgeleitet werden.

5.2. Anwendung und Bewertung

Die Entwicklung beginnt von Grund auf, das bedeutet, es muss zunächst einmal der generische Baum entwickelt werden. Abbildung 5.3

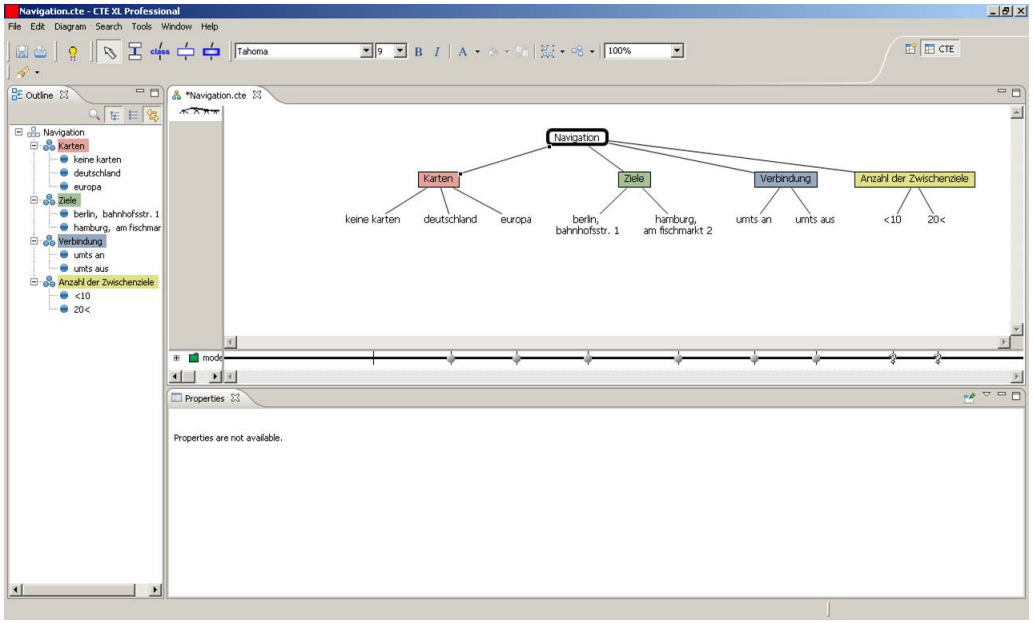

Abbildung 5.3.: Generischer Baum im Ausgangszustand

In einem nächsten Schritt werden mithilfe des Variantenmanagement-Wizards die Varianten *modell0*, *modell1* und *modell2*angelegt und mithilfe der *Modify Variant Elements*-Funktionen die dazugehörigen Klassen per Drag'n'Drop zugeordnet. Abbildung 5.4 fasst diese Schritte zusammen.

Mit der Drag'n'Drop-Zuordnung der Klassen zu den Varianten werden gleichzeitig die Abhängigkeitsregeln angelegt (vgl.Abbildung 5.5).

Nach dem die Varianten angelegt sind, wird das Sichtenkonzept aktiv und man kann zwischen den Varianten und dem generischen Baum umherschalten. Nun sind auch die Voraussetzungen erfüllt, um die Parameter anzulegen. Diese können wie in Abbildung 5.6 dargestellt, direkt per Drag'n'Drop auf die Klassen gezogen werden.

Damit ist der Vorgang der Variabilitätsmodellierung abgeschlossen und mithilfe der Verifikationsfunktion könnte nun noch überprüft werden, ob beim Anlegen der Varianten einzelne Abhängigkeitsregeln zerstört wurden. Die Verlinkung mit Anforderungen kann schließlich in der Requirements View überprüft werden.

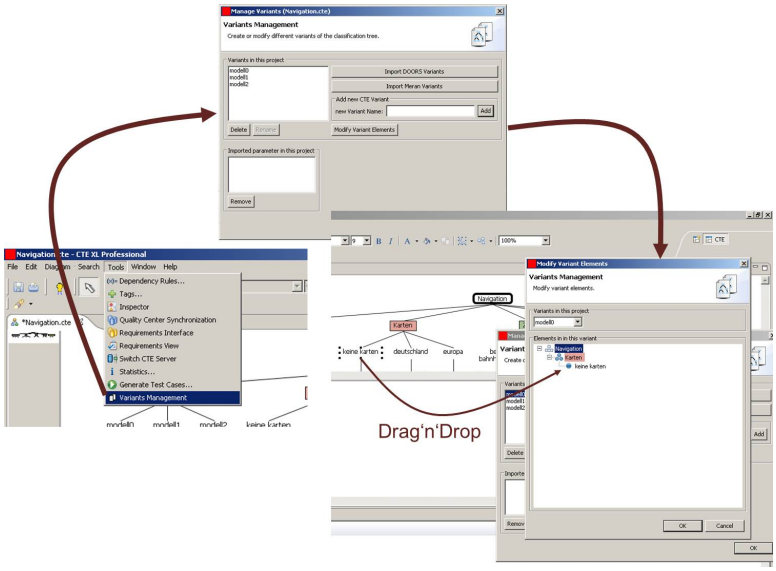

Abbildung 5.4.: Workflow zum Erstellen von Varianten

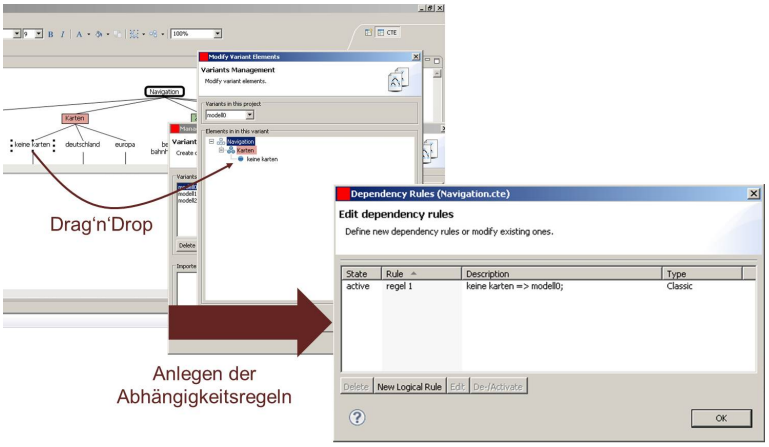

Abbildung 5.5.: Workflow zum Erstellen von Varianten

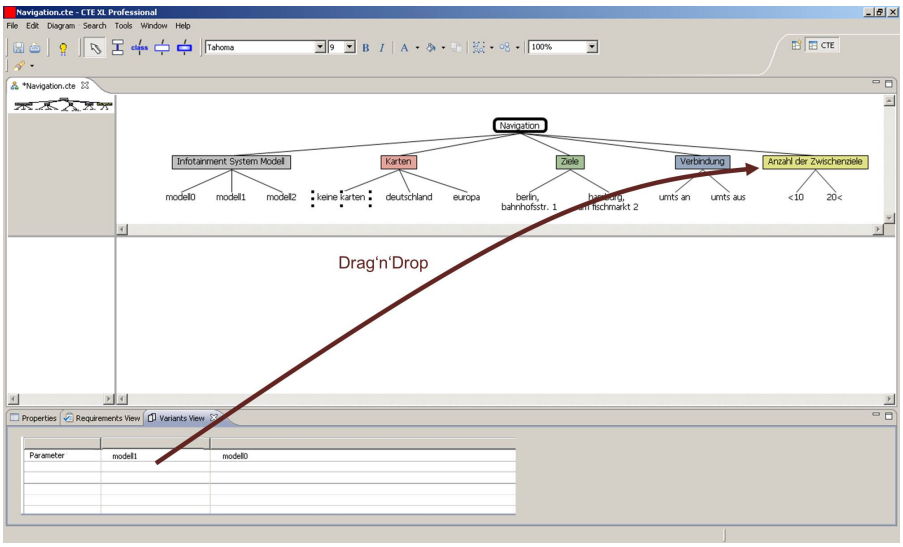

Abbildung 5.6.: Zuordnen von Parametern

Im nächsten Schritt können nun für die einzelnen Varianten die Testfälle abgeleitet werden. Bei aktiviertem Rule Checker werden dabei für jede Variante nur die durch die Abhängigkeitsregeln erlaubten Testfälle erstellt. Das Plug-In unterstützt jedoch auch das manuelle Anlegen von Testfällen, wobei im Nachgang mit dem Rule Checker die Korrektheit der Testfälle nachgewiesen werden kann.

Mit dem generischen Verfahren und der prototypischen Umsetzung kann die Erstellung von Varianten innerhalb der Klassifikationsbaum-Methode effizient und systematisch unterstützt werden. Das generische Verfahren leitet den Test dabei, so dass Fehler bei der Variantenerstellung vermieden werden. Zudem bietet das Variantenmanagement-Plug-In eine intuitive Benutzerführung. Damit kann sich der Tester die Varianten schnell und komfortabel zusammenklicken.

Mit dem generischen Verfahren lassen sich sowohl die Entwicklung der Plattform im Domain Engineering unterstützen, es können aber auch die variantenspezifischen Testfälle für einzelne Endprodukte innerhalb des Application Engineerings abgeleitet werden. Ein weiterer Einsatzbereich ist die Nachverfolgung der Variabilität innerhalb der Produktlinien-Entwicklung, die durch das entwickelte Variantenmanagement-Plug-In in Kombination mit dem im CTE XL Prof. bereits integrierten Requirements View unterstützt werden kann.

Insgesamt führt der Einsatz des Variantenmanagement-Plug-Ins zu Kosten- und Zeitersparnis bei der Erstellung und Pflege von variantenspezifischen Testfällen.

6. Zusammenfassung und Ausblick

6.1. Zusammenfassung

Die veränderten Anforderungen im Zuge der individuellen Massenfertigung stellen die Hersteller der Konsumgüterindustrie vor die Herausforderung ihre Prozesse für Produktion und Entwicklung adaptieren zu müssen. In diesem Zusammenhang beabsichtigt die Berner & Mattner Systemtechnik GmbH ihren Klassifikationsbaum-Editor CTE XL Prof. um ein Variantenmanagement für die Testfallbestimmung zu erweitern.

Die Ziele der vorliegenden Arbeit waren die Entwicklung eines generischen Verfahrens für ein Variantenmanagement zur Testfallbestimmung mithilfe der Klassifikationsbaum-Methode und dem CTE XL Prof., sowie dessen prototypische Umsetzung und Bewertung. Dabei sollte ein Konzept für die Verwendung von Parametern und den Einsatz einer Verifikationsfunktion zur Aufdeckung fehlerhafter Zusammenhänge erarbeitet werden.

Als Basis für die Entwicklung des generischen Verfahrens wurden zunächst die Grundlagen der Qualitätssicherung und der Begriff der Software-Produktlinien-Entwicklung erläutert. Im nächsten Schritt wurden ein modernes Software-Produktlinien-Framework und der Stand der Technik im Bereich des Testens von Software-Produklinien analysiert, um die für die Entwicklung des generischen Verfahrens notwendigen Erkenntnisse zu gewinnen.

Auf diesem theoretischen Fundament wurde die Klassifikationsbaum-Methode um das generische Verfahren für ein Variantenmanagement erweitert. Dabei wurden Parameter als wichtige Differenzierungsmerkmale von Varianten identifiziert und in das Verfahren integriert.

Anschließend wurde, basierend auf diesem Verfahren, eine prototypische Umsetzung konzipiert, die in einem weiteren Schritt anhand von Anwendungsszenarien bewertet wurde.

Die Evaluation ergab, dass durch die Integration eines Variantenmanagements in den CTE XL Prof. die Tätigkeit des Testens im Rahmen der Produktlinien-Entwicklung effizient unterstützt werden kann. Zudem wurde auf diese Weise die Grundlage geschaffen, generische Testspezifikationen für das Domain Engineering und produktspezifische Testfallmengen für das Application Engineering innerhalb der Software-Produktlinien-Entwicklung zu bestimmen. Durch die Anbindung des um ein Variantenmanagement erweiterten CTE XL Prof. an DOORS und MERAN konnten die dort verfügbaren Varianteninformationen sinnvoll auf die Testfallbestimmung übertragen werden. Darüber hinaus wird durch die Verlinkung von Anforderungen aus DOORS und MERAN mit den variantenspezifischen Baumelementen und Testfällen im CTE XL Prof. nicht nur die Qualitätssicherung des Entwicklungsprozesses verbessert, sondern auch die Nachverfolgbarkeit von Varianteninformationen für die Software-Produktlinien-Entwicklung gesteigert.

6.2. Ausblick

Im Rahmen der Weiterentwicklung des CTE XL Prof. bei der Berner & Mattner Systemtechnik GmbH werden die in dieser Arbeit gewonnenen Ergebnisse und das Variantenmanagement-Plug-In für die Herstellung eines marktreifen Releases in den CTE XL Prof. übernommen. Auch wenn das Variantenmanagement-Plug-In aufgrund der Verwendung der in CTE XL Prof. bereits integrierten Mechanismen ähnliche Eigenschaften bezüglich der Skalierbarkeit aufweisen sollte, muss dies durch den industriellen Einsatz bei der Entwicklung großer Produktlinien mit vielen Produktvarianten, wie sie beispielsweise bei der Steuergeräte-Entwicklung in der Automobilindustrie gängig sind, noch verifiziert werden. Die exemplarische Anwendung innerhalb von Beispiel-Szenarien reicht dafür allein nicht aus.

Zudem bleibt zu überprüfen, ob das entwickelte generische Verfahren auch für die produktionstechnische Produktentwicklung anwendbar ist. Aufgrund der Analogien zwischen der klassischen Produktlinien-Entwicklung und der Software-Produktlinien-Entwicklung scheint dies wahrscheinlich.

Einen weiteren Ansatzpunkt bietet die Produktlinien-übergreifende Wiederverwendung. Hierbei wäre zu klären, ob mit dem in dieser Arbeit entwickelten Verfahren auch die Wiederverwendung von Artefakten zwischen den Produkten unterschiedlicher Produktlinien realisiert werden kann.

Generell ist die Integration der verwendeten Werkzeuge DOORS, MERAN, CTE XL Prof. sowie anderer Entwicklungswerkzeuge zu einer prozessübergreifenden Gesamtlösung mit mehreren Sichten auf den Entwicklungsprozess anzustreben. Dadurch ließe sich ein durchgängiges Variantenmanagement, wie es beispielsweise die Software-Produktlinien-Entwicklung mithilfe von Variabilitätsmodellen fordert, noch genauer abbilden und die Probleme an den Schnittstellen der Werkzeuge könnten reduziert werden. Mithilfe eines derartigen Entwicklungswerkzeuges könnte zudem die Nachverfolgung von Anforderungen und anderen Artefakten des Entwicklungsprozesses weiter verbessert werden; durch eine integrierende grafische Darstellung ließen sich die komplexen prozessübergreifenden Zusammenhänge für die beteiligten Akteure deutlicher veranschaulichen.

Aufgrund des vom CTE XL Prof. verwendeten XML-basierten maschinenlesbaren Dateiformats könnte des Weiteren durch automatische Folgerungsalgorithmen, wie sie beispielsweise im Zusammenhang mit den Technologien des Semantic Web entwickelt wurden, implizites Wissen aus den gespeicherten Daten gefolgert werden. Auf diese Weise ließe sich die entwickelte Verifikationsfunktion weiter verfeinern und zusätzliche Informationen für die Nachverfolgung von Anforderungen könnten generiert werden.

Die Schnittstelle zwischen DOORS, beziehungsweise MERAN, und dem CTE XL Prof. könnte auf Basis des Feature-Modells ([Kang u. a. 1990]) weiterentwickelt werden, um noch weitere Informationen, wie zum Beispiel die strukturellen Merkmale der Varianten, aus den Anforderungen auf die Testfälle zu übertragen. So könnten beispielsweise Fehler in der Varianten-Baumelement-Zuordnung aufgedeckt werden, bei denen Baumelemente irrtümlicher Weise einer Variante zugeordnet sind.

Bei die Erstellung von Parametern im CTE XL Prof. wäre zudem eine *Matching*-Funktion denkbar, die die Parameternamen auch in anderen Klassen einer Klassifikation erkennt und diese bei der Anzeige der Varianten automatisch durch die variantenspezifischen Parameterwerte ersetzt.

Literaturverzeichnis

[Ardis u. a. 2000] Ardis, Mark ; Daley, Nigel ; Hoffman, Daniel ; Siy, Harvey ; Weiss, David: Software product lines: a case study. In: Software: Practice and Experience 30 (2000), Nr. 7, S. 825 – 847. – ISSN 1097-024X

[Baerisch 2007] Baerisch, S.: Model-driven test-case construction. In: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. New York, NY, USA, 2007 (ESEC-FSE '07), S. 587 – 590. – ISBN 978-1-59593-811-4

[Balzert 2008] Balzert, H.: Lehrbuch Der Softwaretechnik: Softwaremanagement. Spektrum Akademischer Verlag, 2008 (Spektrum Lehrbücher der Informatik). – ISBN 9783827411617

[Balzert 2009] Balzert, H.: Lehrbuch Der Softwaretechnik: Basiskonzepte Und Requirements Engineering. Spektrum Akademischer Verlag, 2009 (Spektrum Lehrbücher der Informatik). – ISBN 9783827417053

[Bayer u. a. 1999] Bayer, J. ; Flege, O. ; Knauber, P. ; Laqua, R. ; Muthig, D. ; Schmid, K. ; Widen, T. ; DeBaud, J.-M.: PuLSE: a methodology to develop software product lines. In: Proceedings of the 1999 symposium on Software reusability. New York, NY, USA : ACM, 1999 (SSR '99), S. 122 – 131. – ISBN 1-58113-101-1

[Berner&Mattner 2011] Berner&Mattner: MERAN - Spezifikation variantenreicher Systeme. 2011. – URL `http://www.berner-mattner.com/de/berner-mattner-home/produkte/meran/index.html`. – Zugriffsdatum: 03.08.2011

[Bertolino und Gnesi 2004] Bertolino, A. ; Gnesi, S.: PLUTO: A Test Methodology for Product Families. In: Lecture Notes in Computer Science. SpringerVerlag Heidelberg. 3014, Springer, 2004, S. 181 – 197

[Böckle u. a. 2004] Böckle, G. ; Knauber, P. ; Pohl, K. ; Schmid, K.: Software-Produktlinien: Methoden, Einführung und Praxis. Dpunkt.Verlag GmbH, 2004. – ISBN 9783898642576

[Boehm 1981] Boehm, B.W.: Software engineering economics. Prentice-Hall, 1981 (Prentice-Hall advances in computing science and technology series). – ISBN 9780138221225

[CAFÉ 2011] CAFÉ: CAFÉ - From Concepts to Application in System-Family Engineering. 2011. – URL `http://www.esi.es/Cafe`. – Zugriffsdatum: 26.08.2011

[Clements und Northrop 2002] Clements, P. ; Northrop, L.: Software product lines: practices and patterns. Addison-Wesley, 2002 (The SEI series in software engineering). – ISBN 9780201703320

[Cohen u. a. 2006] Cohen, M. B. ; Dwyer, M. B. ; Shi, J.: Coverage and adequacy in software product line testing. In: Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis. New York, NY, USA : ACM, 2006 (ROSATEA '06), S. 53 – 63. – ISBN 1-59593-459-6

[Collins 2007] Collins, T.: The Legendary Model T Ford: The Ultimate History of America's First Great Automobile. Krause Publications, 2007. – ISBN 9780896895607

[Conrad 2004] Conrad, M.: Modell-basierter Test eingebetteter Software im Automobil. Deutscher Universitäts-Verlag, 2004. – ISBN 9783824421886

[Davis 1997] Davis, S.M.: Future perfect. Addison-Wesley Publishing, 1997. – ISBN 9780201590456

[Dijkstra 1972] Dijkstra, E. W.: Structured programming. London, UK, UK : Academic Press Ltd., 1972, Kap. Chapter I: Notes on structured programming, S. 1 – 82. – ISBN 0-12-200550-3

[Dueñas u. a. 2004] Dueñas, J. C. ; Mellado, J. ; Cerón, R. ; Arciniegas, J. L. ; Ruiz, J. L. ; Capilla, R.: Model driven testing in product family context. In: University of Twente, 2004

[Eclipse-Foundation 2011a] Eclipse-Foundation: Eclipse Modeling Framework. 2011. – URL `http://eclipse.org/modeling/emf`. – Zugriffsdatum: 12.09.2011

[Eclipse-Foundation 2011b] Eclipse-Foundation: Eclipse Rich Client Platform. 2011. – URL `http://wiki.eclipse.org/index.php/Rich_Client_Platform`. – Zugriffsdatum: 12.09.2011

[ESAPS 2011] ESAPS: ESAPS - Engineering Software Architectures, Processes and Platforms for System-Families. 2011. – URL `http://www.esi.es/esaps/`. – Zugriffsdatum: 26.08.2011

[Families 2011] Families: Families - FAct-based Maturity through Institutionalisation Lessons-learned and Involved Exploration of System-family engineering. 2011. – URL `http://www.esi.es/Families/`. – Zugriffsdatum: 26.08.2011

[Ford 2007] Ford, H.: My Life and Work - An Autobiography of Henry Ford. NuVision Publications, 2007. – ISBN 9781595478757

[Geppert u. a. 2004] Geppert, B. ; Li, J. ; Robler, F. ; Weiss, D.: Towards Generating Acceptance Tests for Product Lines. In: Bosch, J. (Hrsg.) ; Krueger, C. (Hrsg.): Software Reuse: Methods, Techniques, and Tools Bd. 3107. Springer Berlin / Heidelberg, 2004, S. 35 – 48. – ISBN 978-3-540-22335-1

[Gotel und Finkelstein 1994] Gotel, O. C. Z. ; Finkelstein, A. C. W.: An Analysis of the Requirements Traceability Problem, 1994, S. 94 – 101

[Greenspan und McGowan 1978] Greenspan, S. J. ; McGowan, C. L.: Structuring software development for reliability. In: Microelectronics Reliability 17 (1978), Nr. 1, S. 75 – 83. – ISSN 0026-2714

[Grochtmann und Grimm 1993] Grochtmann, M. ; Grimm, K.: Classification trees for partition testing. In: Software Testing, Verification and Reliability 3 (1993), Nr. 2, S. 63 – 82. – ISSN 1099-1689

[Heymans und Trigaux 2003] Heymans, P. ; Trigaux, J. C.: Software Product Lines: State of the art / Institut d'Informatique FUNDP. September 2003. – Technical Report

[IBM 2011] IBM: IBM® Rational® DOORS. 2011. – URL `http://www-01.ibm.com/software/awdtools/doors/`. – Zugriffsdatum: 25.06.2011

[IEEE 1990] IEEE: IEEE Standard Glossary of Software Engineering Terminology. In: IEEE Std 610.12-1990 (1990), S. 1+

[Kang u. a. 1990] Kang, K. C. ; Cohen, S. G. ; Hess, J. A. ; Novak, W. E. ; Peterson, A. S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study / Carnegie-Mellon University Software Engineering Institute. November 1990. – Forschungsbericht

[Kang u. a. 2007] Kang, S. ; Lee, J. ; Kim, M. ; Lee, W.: Towards a Formal Framework for Product Line Test Development. In: Proceedings of the 7th IEEE International Conference on Computer and Information Technology. Washington, DC, USA : IEEE Computer Society, 2007, S. 921 – 926. – ISBN 0-7695-2983-6

[Lamancha u. a. 2009] Lamancha, B. P. ; Usaola, M. P. ; Velthius, M. P.: Software Product Line Testing - A Systematic Review. In: Shishkov, B. (Hrsg.) ; Cordeiro, J. (Hrsg.) ; Ranchordas, A. (Hrsg.): ICSOFT (1), INSTICC Press, 2009, S. 23 – 30. – ISBN 978-989-674-009-2

[Lehmann und Wegener 2000] Lehmann, E. ; Wegener, J.: Test Case Design by Means of the CTE XL. In: Proceedings of the 8th European International Conference on Software Testing, Analysis Review (EuroSTAR 2000), Kopenhagen, Denmark, December (2000)

[Liggesmeyer 2009] Liggesmeyer, P.: Software-Qualität: Testen, Analysieren und Verifizieren von Software. Spektrum Akademischer Verlag, 2009. – ISBN 9783827420565

[McGregor 2001] McGregor, J.D.: Testing a software product line. Carnegie Mellon University, Software Engineering Institute, 2001 (Technical report)

[Mercedes-Benz 2011] Mercedes-Benz: Mercedes Benz Konfigurator. 2011. – URL `http://www.mercedes-benz.de/content/germany/mpc/mpc_germany_website/de/home_mpc/passengercars/home/new_cars/configurator_showroom.flash.html`. – Zugriffsdatum: 25.06.2011

[Meyer und Wegener 2010] Meyer, J. ; Wegener, J.: Durchgängiges Variantenmanagement - von der Spezifikation bis zum HiL-Prüfstand. In: ATZ extra - Automotive Engineering Partners 5 (2010), S. 2 – 3

[Meyer und Lehnerd 1997] Meyer, M.H. ; Lehnerd, A.P.: The power of product platforms: building value and cost leadership. Free Press, 1997. – ISBN 9780684825809

[Nebut u. a. 2003a] Nebut, C. ; Fleurey, F. ; Le Traon, Y. ; Jézéquel, J.-M.: A Requirement-Based Approach to Test Product Families. In: Proc. Fifth Workshop Product Families Eng, Springer Verlag, 2003, S. 198 – 210

[Nebut u. a. 2003b] Nebut, C. ; Pickin, S. ; Le Traon, Y. ; Jézéquel, J.-M.: Automated Requirements-based Generation of Test Cases for Product Families. In: ASE'03, 2003, S. 263 – 266

[Object Management Group 1989] Object Management Group: UML. 1989. – URL `http://www.uml.org`. – Zugriffsdatum: 21.08.2011

[Olimpiew und Gomaa 2005] Olimpiew, Erika M. ; Gomaa, Hassan: Model-based testing for applications derived from software product lines. In: Proceedings of the 1st international workshop on Advances in model-based testing. New York, NY, USA : ACM, 2005 (A-MOST '05), S. 1 – 7. – ISBN 1-59593-115-5

[Ostrand und Balcer 1988] Ostrand, T. J. ; Balcer, M. J.: The category-partition method for specifying and generating fuctional tests. In: Commun. ACM 31 (1988), June, S. 676 – 686. – ISSN 0001-0782

[Parnas 1976] Parnas, D. L.: On the Design and Development of Program Families. In: IEEE Trans. Softw. Eng. 2 (1976), January, S. 1 – 9. – ISSN 0098-5589

[Pohl u. a. 2005] Pohl, K. ; Böckle, G. ; Linden, F.: Software product line engineering: foundations, principles, and techniques. Springer Verlag, 2005. – ISBN 9783540243724

[Ramesh und Jarke 2001] Ramesh, B. ; Jarke, M.: Toward Reference Models for Requirements Traceability. In: IEEE Transactions on Software Engineering 27 (2001), S. 58 – 93. – ISSN 0098-5589

[Reuys u. a. 2005] Reuys, A. ; Kamsties, E. ; Pohl, K. ; Reis, S.: Model-Based System Testing of Software Product Families. In: Pastor, O. (Hrsg.) ; Cunha, J. Falcao e (Hrsg.): Advanced Information Systems Engineering Bd. 3520. Springer Berlin / Heidelberg, 2005, S. 379 – 380. – ISBN 978-3-540-26095-0

[Robinson-Mallet und Wegener 2009] Robinson-Mallet, C. ; Wegener, J.: Die Vielfalt beherrschen - Software erleichtert das Handling zahlreicher Modellvarianten in frühen Spezifikationsphasen. In: AutomobilKonstruktion 4 (2009), S. 35

[Spillner und Linz 2010] Spillner, A. ; Linz, T.: Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester- Foundation Level nach ISTQB-Standard. Dpunkt.Verlag GmbH, 2010. – ISBN 9783898646420

[Wegener 2001] Wegener, J.: Evolutionärer Test des Zeitverhaltens von Realzeit-Systemen. Shaker Verlag, 2001. – ISBN 9783826592607

[Weiss und Lai 1999] Weiss, D.M. ; Lai, C.T.R.: Software product-line engineering: a family-based software development process. Addison-Wesley, 1999. – ISBN 9780201694383

A. Klassifikationsbäume

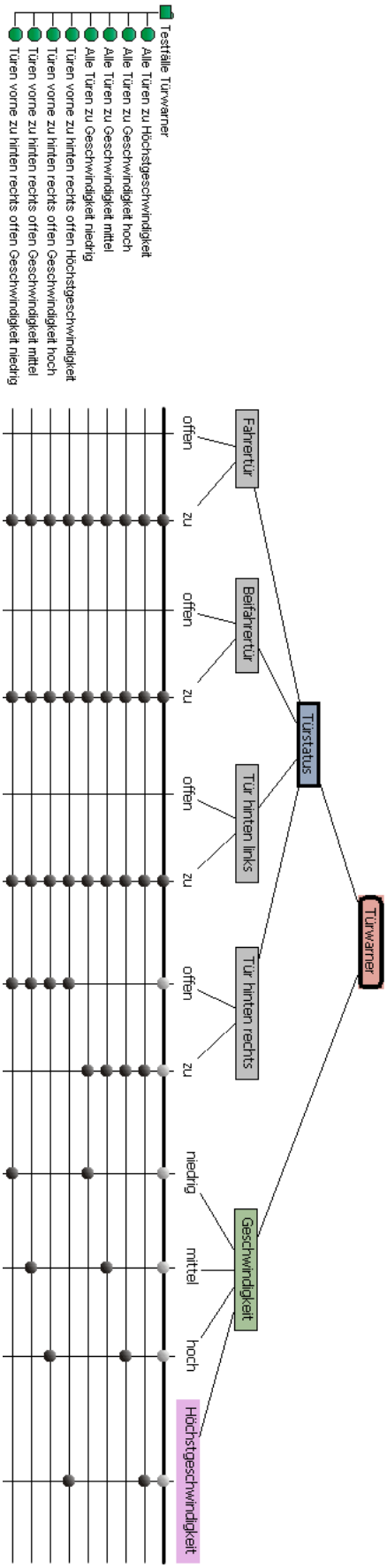

Abbildung A.1.: Testfälle für einen Sensor zur Warnung bei unterschiedlichen Türzuständen im Pkw

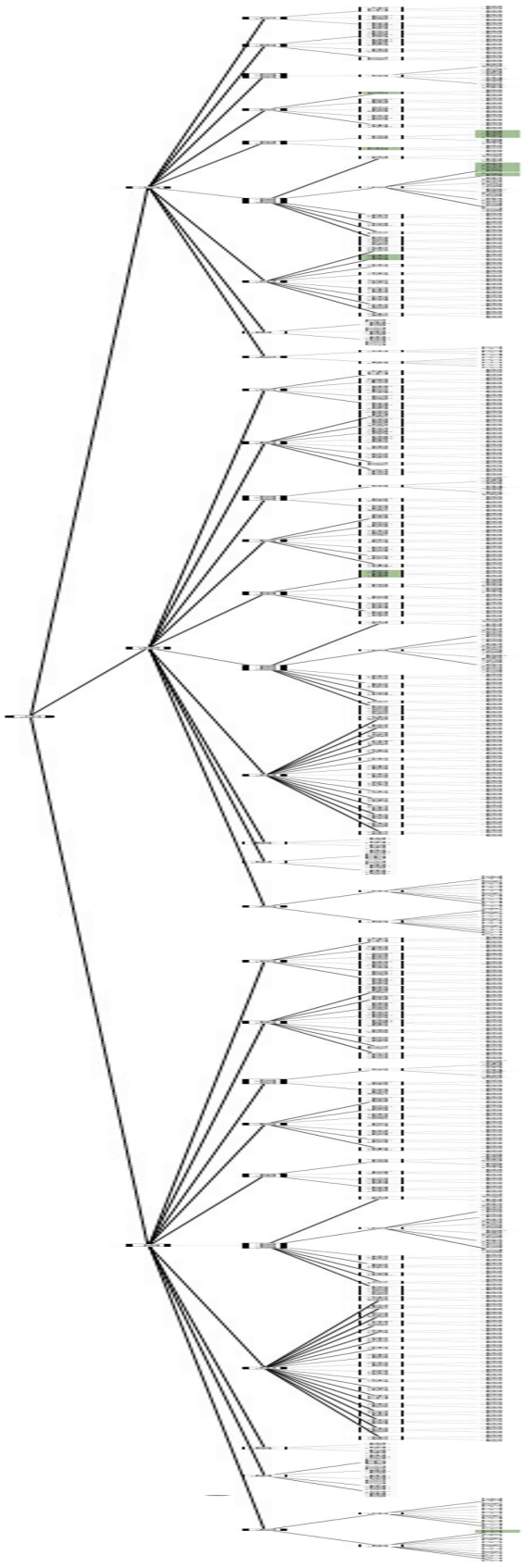

Abbildung A.2.: Klassifikationsbaum für die drei Fahrzeugvarianten Cabrio, Limousine und Kombi der Mercedes E-Klasse ([Mercedes-Benz 2011])

B. Screenshots DOORS und MERAN

Abbildung B.1.: Formales Modul in DOORS

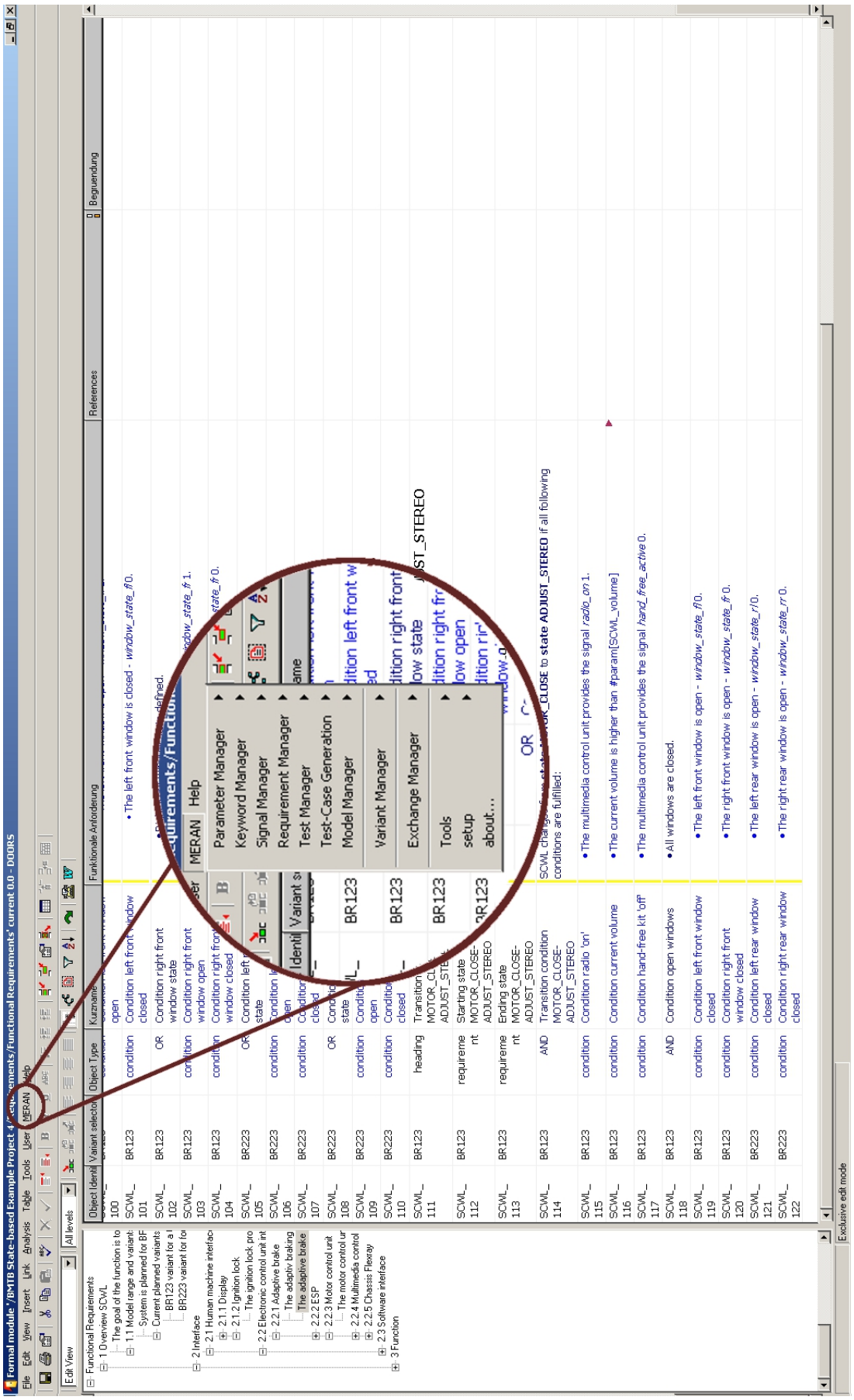

Abbildung B.2.: MERAN-Einbindung in DOORS

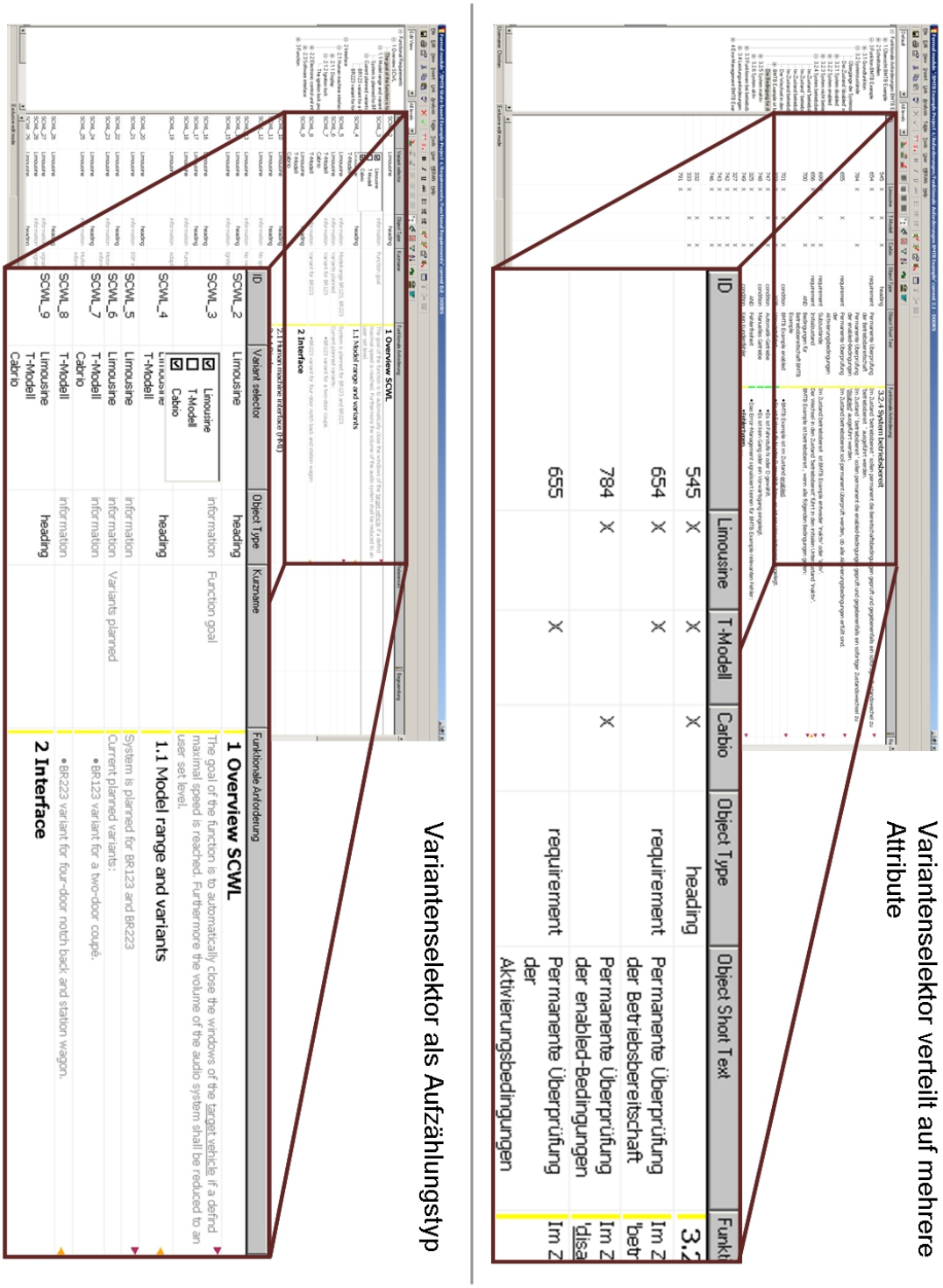

Abbildung B.3.: Umsetzung eines Variantenselektors

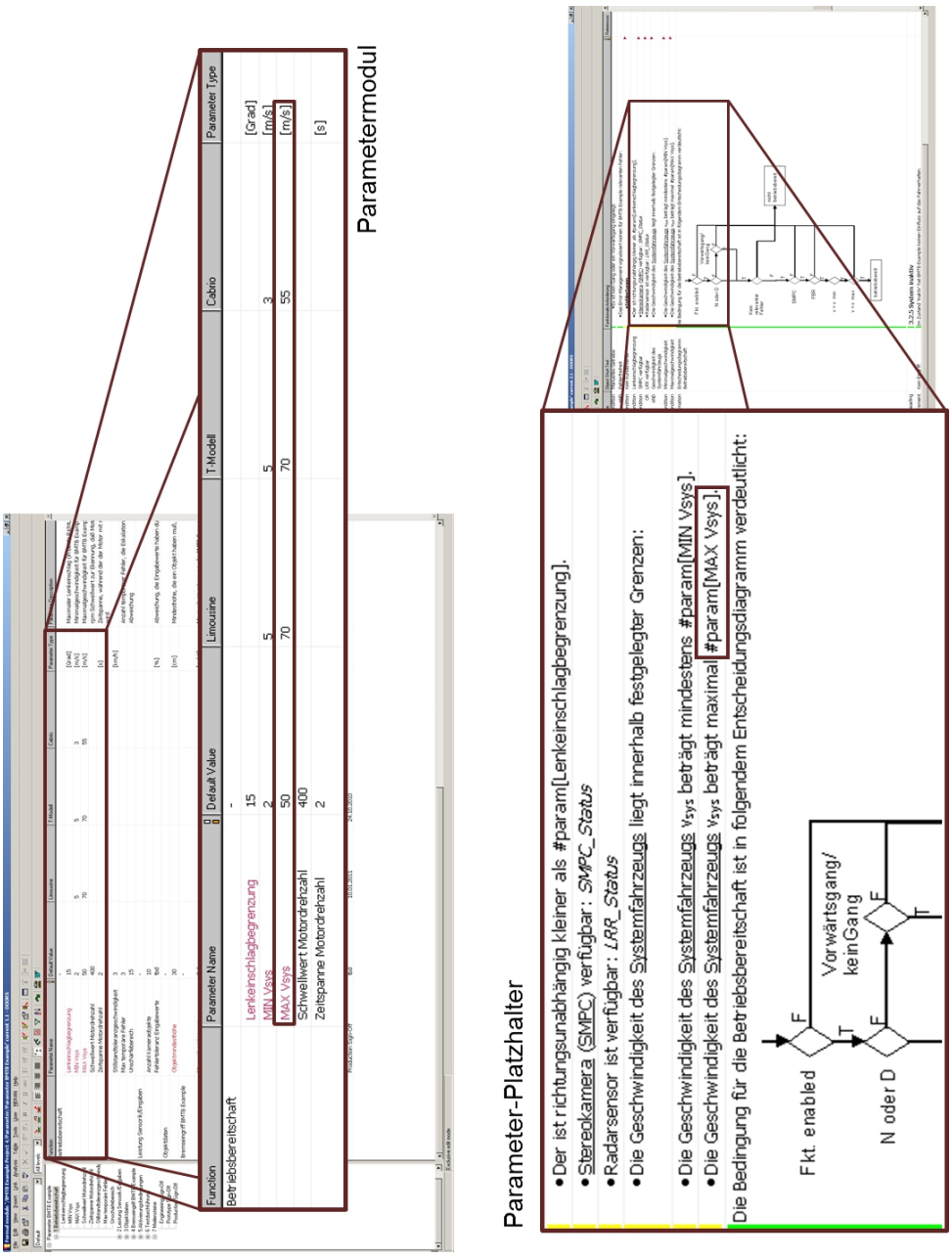

Abbildung B.4.: Parametereinsatz in MERAN

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit mit dem Titel „Durchgängiges Variantenmanagement von der Anforderungsspezifikation bis zum Test auf Basis von MERAN und CTE XL Professional“ selbständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Berlin, den 17. November 2011

Christian Gebhardt

Einverständniserklärung

Ich bin damit einverstanden, dass die vorliegende Arbeit mit dem Titel „Durchgängiges Variantenmanagement von der Anforderungsspezifikation bis zum Test auf Basis von MERAN und CTE XL Professional“ in der Bibliothek des Instituts für Informatik der Humboldt-Universität zu Berlin ausgelegt wird.

Berlin, den 17. November 2011

Christian Gebhardt