Beherrschung dynamischer Variantenmodelle für eingebettete Fahrzeugsysteme

Dissertation
der Mathematisch-Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Hannes Holdschick
aus Jena

Tübingen
2014

Gedruckt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Eberhard Karls Universität Tübingen.

Tag der mündlichen Qualifikation:	26.02.2015
Dekan:	Prof. Dr. Wolfgang Rosenstiel
1. Berichterstatter:	Prof. Dr. Herbert Klaeren
2. Berichterstatter:	Prof. Dr. Wolfgang Rosenstiel

Zusammenfassung

In der Automobilindustrie führt die gestiegene Fahrzeug- und Funktionsvielfalt zu einer immer höheren Anzahl von Modellvarianten. Zudem wächst die Bedeutung von Software für die Fahrzeugindustrie weiter, sodass eine steigende Variantenkomplexität in zukünftigen Software-Systemen zu erwarten ist. Der Produktlinienansatz mit merkmalbasierter Variantenmodellierung ist eine Möglichkeit, dieser Herausforderung zu begegnen, indem Software-Varianten extern dokumentiert und automatisch konfiguriert werden können.

Ein initial angelegtes Variantenmodell ist jedoch nur eine Momentaufnahme der Variabilität in der Implementierung. Wie es für Automotive-Software üblich ist, entwickelt sich diese ständig weiter - und damit auch die variablen Bestandteile darin. Daher ist die Wartbarkeit für Variantenmodelle im industriellen Entwicklungsprozess eine zentrale Eigenschaft.

Die vorliegende Arbeit präsentiert ein Konzept, um die Evolution dynamischer Variantenmodelle für eingebettete Fahrzeugsysteme zu beherrschen. Dafür wird zunächst eine formale Modellbeschreibung erarbeitet, die auf Aussagenlogik basiert. Auf dieser Grundlage werden Änderungsprozesse als konstruktiver Teil des Ansatzes erstellt. Diese fassen elementare Anpassungen zusammen, um häufige Evolutionsmuster des Variantenmodells abbilden zu können. Der analytische Teil des Ansatzes besteht aus der Definition von Konsistenzbedingungen. Damit ist es möglich, Inkonsistenzen zu lokalisieren, die entweder ein Risiko für den Konfigurationsprozess darstellen oder zumindest kritisch für die Wartbarkeit des Modells sind. Auch diese Analysen werden in die Sprache der Aussagenlogik übersetzt und lassen sich zum Teil auf die Erfüllbarkeit einer Formel zurückführen. Änderungsprozesse und Konsistenzbedingungen werden zudem in einem Prototyp realisiert, sodass sowohl Manipulationen als auch Konsistenzprüfungen des Variantenmodells automatisiert werden können. Die in dieser Arbeit vorgestellten Konzepte sorgen somit für eine sichere Evolution des Variantenmodells, indem es mit wenig Aufwand und geringem Risiko einer Inkonsistenz an Änderungen in der Implementierung angepasst werden kann.

Mithilfe einer Befragung von Experten wird der Ansatz evaluiert. Zudem kann der Prototyp in Fallstudien der Daimler AG zeigen, dass auch Variantenmodelle realistischer Größe effizient bearbeitet werden können.

Danksagung

Ganz besonders danken möchte ich meinem Doktorvater Prof. Dr. Herbert Klaeren. Er hat sich stets die Zeit genommen, meine Ideen mit mir zu diskutieren. Sein ehrliches und wertvolles Feedback zu diversen Zwischenständen meiner Arbeit war essentiell für die Fertigstellung dieser Dissertation. Außerdem danke ich Prof. Dr. Wolfgang Rosenstiel für die Übernahme des Zweitgutachtens.

Ein besonderer Dank geht auch an die ehemaligen Kollegen der Daimler AG. Insbesondere Peter Manhart und Dr. Michael Himsolt haben stets großes Interesse an meiner Arbeit gezeigt. Unsere zahlreichen fachlichen Diskussionen in sehr offener und konstruktiver Atmosphäre haben mir Impulse gegeben, ohne die diese Promotion nicht möglich gewesen wäre. Auch die Gespräche mit anderen Doktoranden waren nicht nur im fachlichen Sinne wichtig für meine Arbeit, wobei ich besonders Jan Scheible, Philipp Legrum und Christian Manz hervorheben möchte. Zudem bin ich allen Kollegen (innerhalb und außerhalb der Daimler AG) zum Dank verpflichtet, die sich bereit erklärt haben, an meiner Expertenbefragung teilzunehmen.

Marco Winkelbauer war, zunächst zusammen mit seinen Kommilitonen in einem Studentenprojekt der Hochschule Karlsruhe und anschließend im Rahmen seiner Bachelorarbeit in unserem Team, eine große Unterstützung bei der Implementierung des Prototypen. Neben ihm möchte ich mich auch bei Özgür Akcasoy bedanken, mit dem ich in den ersten Monaten bei Daimler das Büro teilen durfte. Er hat mir durch viele Gespräche und Diskussionen nicht nur den fachlichen Einstieg erleichtert.

Die Erstellung dieser Arbeit wäre nicht möglich gewesen ohne die Hilfe meiner Freundin Steffi. Ihr Verständnis für Abende und Wochenenden voller Arbeit und das Korrekturlesen diverser Versionen dieser Dissertation waren eine riesige Unterstützung für mich. Natürlich haben auch viele Freunde und Familienmitglieder mehr oder weniger bewusst zur Fertigstellung beigetragen. Vielen Dank dafür!

Ein besonderer Dank gilt dabei meinen Eltern, die mich stets bei allen Vorhaben und Plänen unterstützt haben.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Dieses Kapitel motiviert zunächst die Problemstellung der Arbeit, aus der im Anschluss die Zielstellung abgeleitet wird. Der letzte Abschnitt gibt einen Überblick über den Aufbau und die Inhalte der einzelnen Kapitel.

1.1. Problemstellung und Motivation

Eine breite Fahrzeugpalette, unterschiedliche Produktanforderungen aus verschiedensten globalen Märkten und eine Vielzahl softwarebasierter Ausstattungsmerkmale: In der Automobilindustrie führt die gestiegene Fahrzeug- und Funktionsvielfalt zu einer immer höheren Anzahl von Modellvarianten. So wäre es ohne Weiteres möglich, dass von den 330.000 Einheiten eines Mittelklasse-Fahrzeugs, die im Jahr 2006 gefertigt wurden, keine zwei identisch sind [Wei08]. Darüber hinaus spielt Software in der Fahrzeugindustrie eine immer größere Rolle. So geht man davon aus, dass 90 % aller zukünftigen Innovationen im Fahrzeug von E/E-Systemen inklusive deren Software getrieben werden [Bee07]. Diese Zahlen lassen erahnen, mit welcher Variantenkomplexität in den Software-Systemen zu rechnen ist.

Der Fokus dieser Arbeit liegt auf Software-Variabilität in der modellbasierten Funktionsentwicklung. Dafür existieren unterschiedliche Methoden, um verschiedene Varianten eines Systems generieren zu können. Beispielsweise kann ein Funktionsmodell erstellt werden, welches die Funktionalitäten aller vorgesehenen Varianten implementiert. Durch bestehende Variabilitätsmechanismen ist es dem Entwickler möglich, daraus das Funktionsmodell für eine gewünschte Variante zu konfigurieren. Dieses kann anschließend für die Simulation bzw. die Codegenerierung genutzt werden.

Erreicht die Variabilität in einem Funktionsmodell eine kritische Komplexität, so muss sie dokumentiert werden. In dem merkmalbasierten Ansatz für die Variantenmodellierung werden einerseits die gemeinsamen und die unterschiedlichen Eigenschaften der Software-Varianten in einem Merkmalmodell festgehalten [KCH+90]. Man kann dabei zwischen funktionalen (z. B. Limiter, Tempomat, Bremsassistent, etc.) und nicht-funktionalen Merkmalen (z. B. Markt, Baureihe, etc.) unterscheiden. Soll andererseits anhand dieser Merkmale auch die Konfiguration des Funktionsmodells erfolgen, muss zusätzlich eine

Abbildung zwischen diesen beiden Artefakten erstellt werden; zu diesem Zweck kann das Konfigurationsmodell genutzt werden.

Die Variantenmodellierung besteht damit aus Merkmal- und Konfigurationsmodell. Ein initial angelegtes Variantenmodell ist jedoch nur eine Momentaufnahme der Variabilität im Funktionsmodell. Wie es für Automotive-Software üblich ist, entwickelt sich dieses Funktionsmodell ständig weiter - und damit auch die variablen Bestandteile darin. Dadurch entsteht die Notwendigkeit, die Konsistenz zwischen Varianten- und Funktionsmodell zu gewährleisten, welche bereits in anderen Arbeiten als Herausforderung dargestellt wurde [LSB+10].

Man steht damit vor der Aufgabe, jede für die Variabilität relevante Änderung in der Funktionssoftware schnellstmöglich und korrekt im Variantenmodell nachzubilden. Potentielle Inkonsistenzen, die dabei entstehen können, sollten vermieden oder zumindest lokalisiert werden können. Wird beispielsweise eine neue Funktion implementiert, deren Auftreten im System von nicht-funktionalen Eigenschaften wie dem Markt, der Baureihe oder ähnlichem abhängt, muss im einfachsten Fall ein neuer Variationspunkt erstellt und dessen Abhängigkeit von den existierenden Merkmalen definiert werden. Ist die Konfiguration der neuen Funktion mithilfe der bestehenden Merkmale nicht möglich, wird darüber hinaus ein neues Merkmal im Merkmalmodell benötigt.

In vielen Fällen reicht das jedoch nicht aus. Die neue Funktion kann in der Funktionssoftware auch zu einer Umstrukturierung führen, sodass etwa größere Module in kleinere Teilfunktionen zergliedert werden oder umgekehrt aus mehreren kleineren Funktionen ein Gesamtmodul entsteht. Unter Umständen werden vorher gebündelte Funktionalitäten voneinander getrennt und eine neue Modularisierung eingeführt. Zudem können vertriebliche Vorgaben, neue Gesetze in bestimmten Märkten oder technische Innovationen dazu führen, dass sich vorher formulierte Abhängigkeiten zwischen Merkmalen ändern. Dadurch muss in der Regel auch das Variantenmodell neu strukturiert werden, um die Konsistenz zur Funktionssoftware zu gewährleisten. Unter Umständen werden im Merkmalmodell neue Merkmale oder Abhängigkeiten benötigt, die jedoch bestehende Systemkonfigurationen nicht beeinflussen dürfen.

Da Merkmal- und Konfigurationsmodell auf aussagenlogischer Basis miteinander verbunden sind, betreffen diese Änderungen meist beide Teilmodelle und ziehen damit einen höheren Aufwand nach sich. Dabei ist keinesfalls davon auszugehen, dass sich alle Modifikationen in der Software eins zu eins in das Variantenmodell übertragen lassen. Es muss ein Weg gefunden werden, aus diesen Modifikationen die richtigen Variantenmodellanpassungen abzuleiten. Dass dies nicht nur ein Problem in der Automobilindustrie ist, zeigen Arbeiten aus anderen Bereichen, die sich ebenfalls mit den Themen Evolutionsmuster in Variantenmodellen bzw. der Konsistenz der beteiligten Artefakte beschäftigen [PCW12, PCA+13, SHA12].

Je größer das Variantenmodell wird, umso unübersichtlicher werden die Zusammenhänge zwischen den Teilmodellen und umso aufwändiger ist es, Änderungen mit der Gewissheit durchzuführen, dass

1. das Variantenmodell die geänderte Variabilität der Funktionsmodellierung korrekt abbildet und

2. die Änderungen an Merkmal- und Konfigurationsmodell nicht zu Inkonsistenzen geführt haben und noch alle bisher definierten Konfigurationen gültig sind.

Mit steigender Komplexität der Variantenmodellierung erhöht sich die Wahrscheinlichkeit, dass die Entwickler den Überblick verlieren und von weiteren Änderungen Abstand nehmen, um die noch funktionierenden Konfigurationen nicht zu gefährden.

Beispiele aus anderen Domänen zeigen, dass solche Effekte in Modellen realistischer Größe auftreten. So wird auch die Variabilität des Linux Kernel nach dem merkmalbasierten Ansatz dokumentiert und konfiguriert. Lotufo et al. haben in ihrer Arbeit die Entwicklung dieses Variantenmodells im Laufe der Zeit verfolgt [LSB+10] und zitieren dabei die folgenden Änderungskommentare der Entwickler:

- „we do a select of SPARSEMEM_VMEMMAP ... because ... without SPARSEMEM_VMEMMAP gives us a hell of broken dependencies that I don´t want to fix“ bzw.

- „it´s a nightmare working out why CONFIG_PM keeps getting set“

Der erste Kommentar zeigt, dass die Abhängigkeiten in dem Variantenmodell so komplex geworden sind, dass die - zumindest für bestimmte Fälle - gewünschte Deaktivierung des Merkmals *SPARSEMEM_VMEMMAP* unterlassen wird, weil sich ansonsten ein erheblicher Mehraufwand ergeben würde. Auch der zweite Kommentar ist ein deutliches Zeichen dafür, dass die Komplexität aufgrund von Abhängigkeiten für die Entwickler nicht mehr beherrschbar ist. Das Merkmal *CONFIG_PM* wird durch Constraints automatisch aktiviert, obwohl dieses Verhalten nicht gewünscht ist. Der Grund dafür scheint aus der bestehenden Variantenmodellierung nicht unmittelbar hervorzugehen.

Um solche Probleme von Beginn an zu vermeiden, ist die Wartbarkeit des Variantenmodells aus unserer Sicht eine wichtige Eigenschaft. Was Wartbarkeit im Kontext des Software-Variantenmanagements bzw. der merkmalbasierten Variantenmodellierung bedeutet, wird im nächsten Abschnitt beschrieben.

1.2. Zielstellung Wartbarkeit

Wartbarkeit ist in der Softwaretechnik ein wichtiges Kriterium für die Qualität eines Systems. Sochos et al. [SRP05] schreiben dazu:

> „Softwaresysteme sind nur so lange nutzbar, wie sie an geänderte Anforderungen angepasst werden können. Da bei der Entwicklung zukünftige Veränderungen nicht vorhersehbar sind, muss die Komplexität der Software so gering wie möglich gehalten werden, um evolutionäre Weiterentwicklung zu vereinfachen, was durch das Qualitätsmerkmal Wartbarkeit ausgedrückt wird."

Auch wenn dabei von der Wartbarkeit eines Software-Systems die Rede ist, lässt sich diese Erklärung auf die Wartbarkeit von Variantenmodellen übertragen. Vor allem die Beschreibung als Unterstützung der evolutionären Weiterentwicklung prägt auch unsere Definition des Begriffes *Wartbarkeit*. Sie ist angelehnt an die ISO-Richtlinie 25010 über Softwarequalität[1]:

Definition 1.1. *Die* ***Wartbarkeit eines Variantenmodells*** *beschreibt dessen Fähigkeit, modifiziert zu werden. Dazu gehören Korrekturen, Verbesserungen oder Anpassungen des Modells an Änderungen der Umgebung.*

Fasst man die Variabilität in der Implementierung als Umgebung auf, so beschreibt diese Definition in passender Weise unsere Anforderung an das Variantenmodell: Es soll bestmöglich den Evolutionsprozess unterstützen. Die folgenden Kriterien für Wartbarkeit spezifizieren diese Anforderung weiter:

- **Modifizierbarkeit:** Nach einer Änderung der Umgebung (also des Funktionsmodells) soll die Anpassung des Variantenmodells, gemessen an deren Umfang, möglichst wenig Aufwand erfordern. Einfach erkennbare Muster bei der Modellierung und nachvollziehbare Strukturen für Beziehungen und Abhängigkeiten können diesen Prozess unterstützen.

- **Stabilität:** Nach einer Änderung des Variantenmodells soll das Risiko möglichst gering sein, dass es zu Fehlmodellierungen und damit ggf. zu einem inkonsistenten Modellstand gekommen ist. Eine genaue Kenntnis darüber, wo und wie sich Änderungen am Modell auswirken, hilft dabei, dieses Kriterium zu optimieren.

[1] In der ISO-Richtlinie [ISO11] heißt es: „maintainability: degree of effectiveness and efficiency with which a product can be modified [...]. Modifications can include corrections, improvements or adaption of the software to changes in environment, and in requirements and functional specifications. [...]"

- **Testbarkeit:** Das Variantenmodell auf kritische Modellierungen bzw. Inkonsistenzen zu testen, soll effizient sein und möglichst wenig Aufwand erfordern. Fehler sollten dabei genau lokalisiert werden können.

Das Ziel ist somit ein wart- und anpassbares Variantenmodell. Wird dieser Aspekt vernachlässigt, so steigt das Risiko, dass die Komplexität des Variantenmodells Änderungen schwer bis unmöglich macht, wie es auch im vorherigen Abschnitt beschrieben wurde. Auch für Projektumgebungen, in denen verschiedene Entwickler an demselben Modell arbeiten, muss sichergestellt werden, dass Änderungen mit angemessenem Aufwand durchgeführt werden können und Fehlmodellierungen auffindbar sind.

In welchem Umfang die oben genannten Eigenschaften zur Wartbarkeit des Variantenmodells beitragen, hängt davon ab, welche Art von Anpassungen am Modell häufig sind bzw. als kompliziert aufgefasst werden. Daher ist besonders relevant, welche variabilitätsrelevanten Änderungen im Funktionsmodell auftreten, die anschließend im Variantenmodell nachgebildet werden müssen und wie man Fehler lokalisiert, die dabei auftreten können. Der hier erarbeitete Lösungsansatz besteht dementsprechend aus zwei Teilen:

1. Es wurden **Änderungsprozesse** für das Variantenmodell definiert, welche die Reaktion auf Änderungen im Funktionsmodell darstellen. Diese Änderungsprozesse fassen mehrere elementare Änderungsschritte zusammen, wodurch die syntaktisch korrekte Änderung des Variantenmodells sicher gestellt wird und inkonsistente Zwischenstände vermieden werden. Durch diesen Teil der Lösung sollen die Eigenschaften Modifizierbarkeit und Stabilität positiv beeinflusst werden. Eine erste Sammlung von Änderungsprozessen mit der zugehörigen Adaption des Variantenmodells wurde bereits in einer früheren Arbeit veröffentlicht [Hol12]. In Kapitel 4 wird näher auf die Änderungsprozesse eingegangen.

2. Um ein bestehendes Variantenmodell zu prüfen, wurden die **Konsistenzbedingungen** definiert. Dadurch lassen sich einerseits Fehlmodellierungen lokalisieren. Andererseits wird der Entwickler auch auf syntaktisch korrekte Modellzustände hingewiesen, die für die Wartbarkeit des Modells kritisch bzw. unter Umständen ungewollt sind. Diese Maßnahme soll zu Stabilität und Testbarkeit des Variantenmodells beitragen. In Kapitel 5 wird näher auf die Konsistenzbedingungen eingegangen, von denen ein Teil bereits in einer früheren Arbeit vorgestellt wurde [HC13].

Der Lösungsansatz für die bereits beschriebene Herausforderung der Wartbarkeit kombiniert daher mit den Änderungsprozessen einen konstruktiven und mit den Konsistenzbedingungen einen analytischen Bestandteil. Es wird für eine sichere Evolution des Variantenmodells gesorgt, indem es mit wenig Aufwand und geringem Risiko einer Fehlmodellierung an Änderungen in der Implementierung angepasst werden kann. In Abb. 1.1 ist schematisch dargestellt, auf welche Kriterien für Wartbarkeit unsere Lösungskonzepte

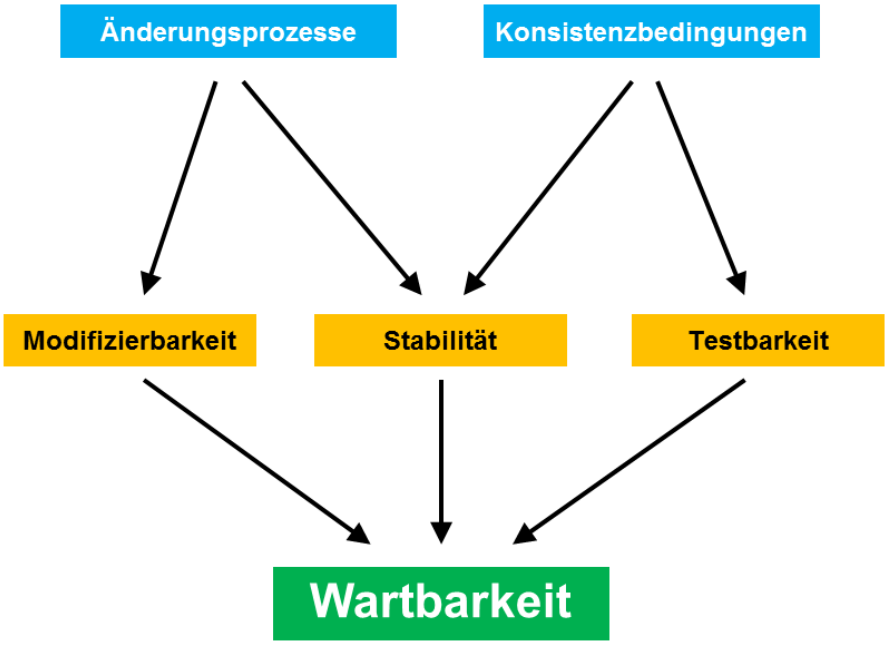


Abbildung 1.1.: Abbildung von Lösungsbestandteilen auf die Kriterien für Wartbarkeit

jeweils Einfluss nehmen.

Für den praktischen Nutzen dieser Konzepte wurden beide Teile des Lösungsansatzes als Erweiterung des Eclipse Plug-Ins pure::variants ([Sys14]) prototypisch umgesetzt, welches bei der Daimler AG unter anderem für die merkmalbasierte Variantenmodellierung und -konfiguration von Simulink / Targetlink-Modellen oder DOORS-Lastenheften verwendet wird[2]. Mithilfe der Implementierung können typische Evolutionsprozesse des Variantenmodells mit möglichst geringer Benutzer-Interaktion durchgeführt werden und es besteht außerdem die Möglichkeit, ein gegebenes Modell auf die beschriebenen Konsistenzbedingungen hin zu überprüfen.

1.3. Inhalt der Arbeit

Dieser Abschnitt geht anhand der Übersicht in Abb. 1.2 auf den Aufbau des vorliegenden Dokumentes ein. Nachdem im ersten Kapitel das Ziel der Arbeit aus der Problemstellung abgeleitet wird, legt Kapitel 2 die Grundlagen für das weitere Vorgehen. Anschließend bereitet die Formalisierung des Variantenmodells in Kapitel 3 die Beschreibung der Lö-

[2] Auch im weiteren industriellen Umfeld wird pure::variants häufig für die Variabilitätsmodellierung eingesetzt [BRN+13].

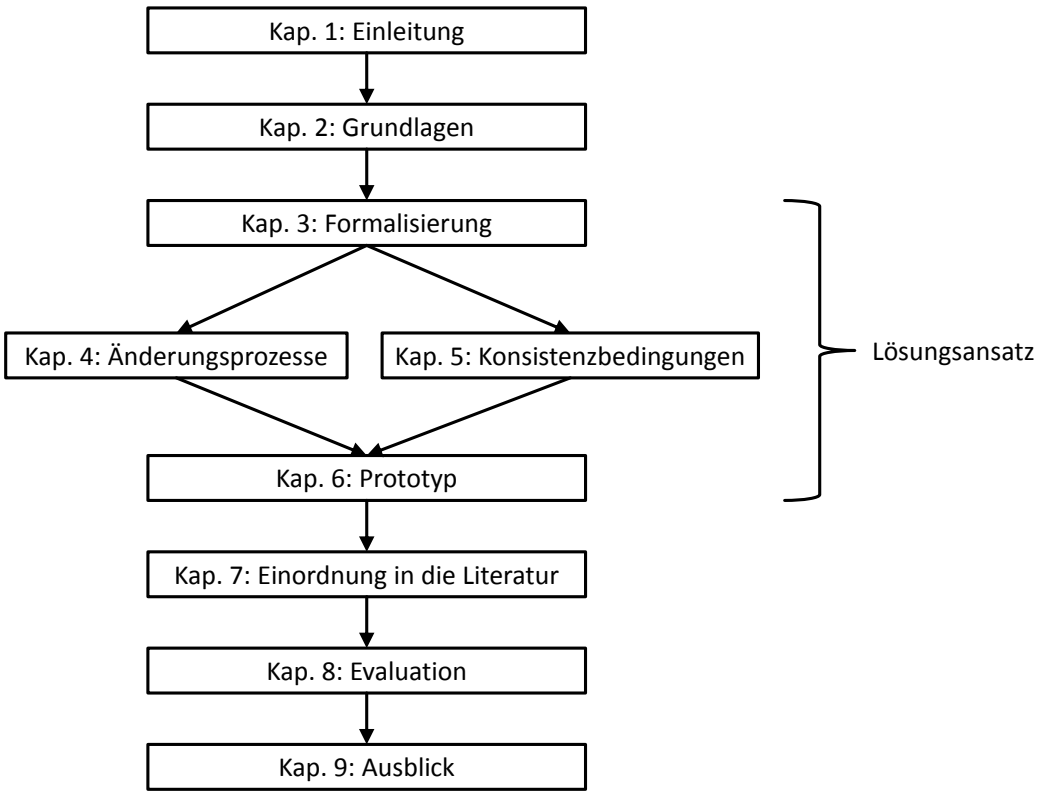


Abbildung 1.2.: Inhalt der Arbeit

sungskonzepte vor: Die Kapitel 4 und 5 stellen die Änderungsprozesse und die Konsistenzbedingungen dar. Danach wird deren prototypische Umsetzung erläutert, bevor in Kapitel 7 die hier beschriebenen Ansätze zu benachbarten Arbeiten abgegrenzt werden. Die Evaluation der Ergebnisse beinhaltet Kapitel 8. Abschließend wird in Kapitel 9 ein Ausblick über weiterführende Themen gegeben.

2. Grundlagen

In diesem Kapitel werden die für die Arbeit wichtigen Grundlagen erläutert. Nach einer kurzen Einführung über Software-Produktlinien folgt eine Beschreibung von Variabilitätsstrukturen in Funktionsmodellen. Anschließend werden die wichtigsten Aspekte des Software-Variantenmanagements dargelegt.

2.1. Software-Produktlinien

Die Methodik der Produktlinien unterstützt den Entwicklungsprozess von mehreren unterschiedlichen Produkten, die aus einem gemeinsamen Kern entstehen. Handelt es sich bei diesen Produkten um Software, so spricht man von Software-Produktlinien [CN01]. Dabei besteht der Kern einer Produktlinie aus einer Menge von Grundfunktionalitäten, die Teil jeder Systemvariante sind. Die unterschiedlichen Eigenschaften dieser Systemvarianten entstehen durch die Konfiguration bzw. das Hinzufügen von weiteren Komponenten. Bewährt hat sich dabei die Trennung zwischen *Domain Engineering* und *Application Engineering* [CE00], also der Konstruktion der Plattform einerseits und der Erstellung der konkreten Lösung andererseits. In [BKPS04] heißt es dazu:

> „Im *Domain Engineering* werden die gemeinsamen und variablen Artefakte, die Bestandteile der Plattform werden, für eine oder mehrere Domänen entwickelt. Im *Application Engineering* werden einzelne Produkte der Produktlinie entwickelt bzw. abgeleitet."

In Abb. 2.1 ist dargestellt, wie orthogonal dazu zwischen Problem- und Lösungsraum unterschieden wird. Im Problemraum werden die funktionalen Eigenschaften der Domäne und aller Produktvarianten spezifiziert und Regeln für deren Konfiguration angegeben. Dafür werden Merkmal- bzw. Konfigurationsmodelle verwendet, welche in Kapitel 3 formal beschrieben sind. Ein einzelnes Element des Problemraumes ist die Konfiguration eines bestimmten Systems, die in der Merkmalselektion abgebildet wird. Die tatsächliche Realisierung der Produktlinie, z. B. in Form einer Implementierung der Architektur und verschiedener Software-Bausteine, ist dagegen Teil des Lösungsraumes. Dieser stellt wiederum die Gesamtheit der Softwarevarianten dar, die beispielsweise spezifisch für verschiedene Baureihen oder Märkte sind.

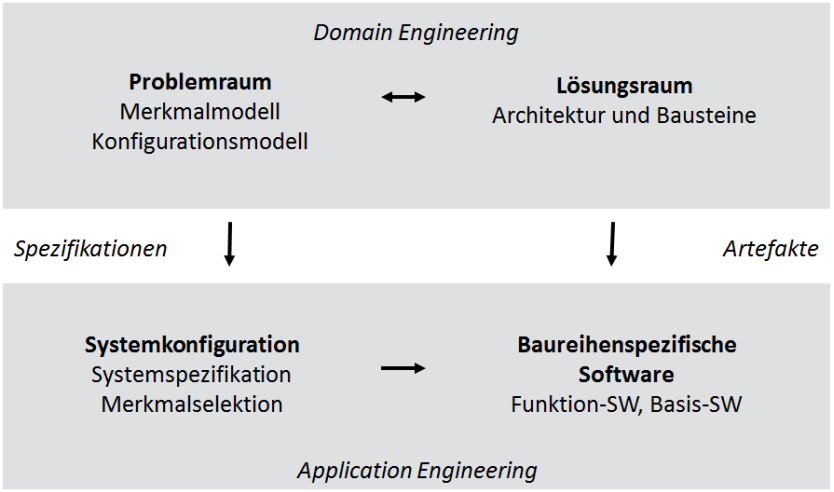


Abbildung 2.1.: Software-Produktlinienentwicklung

Ein zentraler Bestandteil des Produktlinienansatzes ist die explizite Modellierung der Variabilität des Lösungsraumes. Im nächsten Abschnitt wird genauer beschrieben, welche Art von Lösungsraum der Arbeit zugrunde liegt und welche variable Strukturen man dort vorfindet.

2.2. Variabilität in Funktionsmodellen

Dieser Abschnitt führt aus, wie in einem Implementierungsartefakt Variabilität eingeführt werden kann. Da in der eingebetteten Softwareentwicklung der Automobilindustrie der modellbasierte Entwicklungsansatz sehr weit verbreitet ist, konzentriert sich die vorliegende Arbeit auf Variabilität in Funktionsmodellen. Werden diese beispielsweise mit MATLAB / Simulink erstellt, wie es in weiten Teilen der Automobilbranche üblich ist, können Code-Generatoren wie TargetLink oder Real Time Workshop Embedded Coder verwendet werden, um aus solchen Funktionsmodellen C-Code zu generieren. Dieser Code wird anschließend von einem Compiler in Binärcode für den Zielprozessor auf dem Steuergerät übersetzt.

In Abb. 2.2 ist dargestellt, welche Aspekte des Funktionsmodells für uns relevant sind. Zunächst enthält es eine beliebige Anzahl von Komponenten.

Definition 2.1. *Eine **Komponente** ist eine funktionale Einheit des Funktionsmodells, welche aus Eingangssignalen, deren Verarbeitung und den daraus entstehenden Ausgangssignalen besteht.*

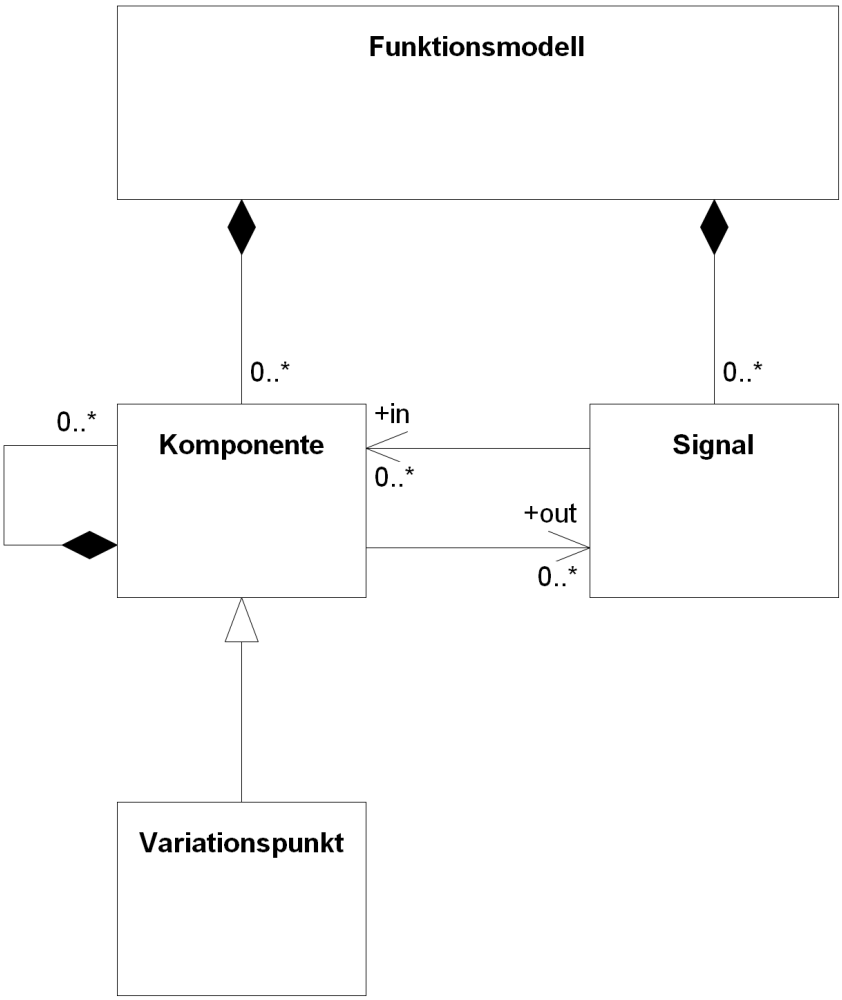


Abbildung 2.2.: Das Funktionsmodell

Das Modell ist dabei hierarchisch aufgebaut, sodass eine Komponente weitere Komponenten enthalten kann. Diese sind neben Kompositionsbeziehungen auch durch Signale miteinander verbunden. Ein Signal hat eine eindeutige Startkomponente und kann zu mehreren Zielkomponenten führen. Start- und Zielkomponente müssen dabei nicht zwingend in der gleichen Hierarchieebene liegen. Für die weiteren Betrachtungen ist zudem wichtig, dass ein Funktionsmodell Variationspunkte enthalten kann. Im Folgenden wird genauer auf diese spezialisierten Komponenten eingegangen.

Um in einem Funktionsmodell Variabilität einzuführen, existieren unterschiedliche Mechanismen, welche den Ansätzen auf Code-Ebene ähnlich sind [GA01]. Die Methode der positiven Variabilität [VG07] sieht vor, dass zu einem minimalen Modellkern über Erweiterungsschnittstellen die entsprechenden Komponenten hinzugefügt werden, um eine bestimmte Variante zu erzeugen. Dagegen werden bei der Methode der negativen Variabilität [BPK09] zunächst die Bausteine aller geplanten Varianten in einem Modell vorgehalten, dem sogenannten 150%-Modell. Um eine bestimmte Modellvariante zu konfigurieren, werden anschließend die nicht benötigten Komponenten deaktiviert. Die Delta-Modellierung stellt eine weitere Methode dar, bei der eine Variante durch eine Menge von Änderungen (sogenannte Deltas) beschrieben ist, die auf ein Ausgangsmodell angewendet werden [SBDT10]. Alle Methoden haben jedoch gemeinsam, dass Variationspunkte im Modell ausgezeichnet werden müssen. In [BKPS04] wird dieser Begriff im allgemeinen Kontext der Software-Produktlinien wie folgt definiert:

> „Ein **Variationspunkt** ist eine Stelle im Artefakt, an der die Auswahl einer oder mehrerer Ausprägungen für unterschiedliche Produktvarianten möglich ist. [...] Formal gesehen stellt ein Variationspunkt eine hinausgezögerte Designentscheidung dar."

Bei den weiteren Betrachtungen spielt das Funktionsmodell die Rolle des hier erwähnten Artefaktes. Angelehnt daran ist ein Variationspunkt in unserem Verständnis eine Komponente im Funktionsmodell, die Variabilität beinhaltet. Dadurch können an einer Stelle in diesem Modell - abhängig von der gewünschten Systemvariante - unterschiedliche Funktionalitäten bereitgestellt werden. Diese unterschiedlichen Funktionalitäten werden durch die Ausprägungen des Variationspunktes beschrieben:

Definition 2.2. *Jede **Ausprägung** an einem Variationspunkt steht für eine Variante, wie dieser ausgestaltet werden kann.*

Dabei hat ein echter Variationspunkt mindestens zwei Ausprägungen[1]. Eine Systemvariante lässt sich als Auswahl von genau einer Ausprägung je Variationspunkt beschreiben. Diese steht nach der Bindung fest:

[1] In Ausnahmefällen kann ein Variationspunkt auch nur eine Ausprägung haben: Falls entweder eine zweite (und dritte etc.) Ausprägung bekannt, aber noch nicht modelliert ist oder der Variationspunkt mehrere Ausprägungen hatte, von denen gerade alle bis auf eine gelöscht wurden.

Definition 2.3. *Mit der **Bindung** eines Variationspunktes wird der Prozess bezeichnet, in dem dessen Variabilität aufgelöst und die Ausprägung ermittelt wird.*

Grundsätzlich ist das in dieser Arbeit vorgestellte Vorgehen für alle Methoden anwendbar, bei denen ein merkmalbasiertes Variantenmodell erstellt wird. Da der Ansatz der negativen Variabilität in der modellbasierten Funktionsentwicklung im Automotive-Bereich die breiteste Anwendung findet, werden hier zwei wesentliche Konzepte betrachtet, um darin einen Variationspunkt zu definieren:

1. Variabilität durch eine **optionale Komponente**: Die implementierte Funktionalität in einer optionalen Komponente kann, abhängig von einer Bedingung, aktiviert bzw. deaktiviert werden. Nach der Auswertung der Bedingung wird ein zweiwertiger Varianten-Parameter gesetzt, der über die Ausprägung der Komponente entscheidet. Die genaue Struktur dieser Bedingungen wird in Kapitel 3 über die Formalisierung des Variantenmodells beschrieben.
   Beispiele für eine optionale Komponente sind:

   - Ein Abstandsregeltempomat lässt sich als optionale Komponente realisieren. Anschließend kann in jedem Fahrzeug, welches auch die benötigten Sensoren bereitstellt, diese Funktion für das Fahrerassistenz-Paket aktiviert werden.

   - Die Anzeige von aufbereiteten Online-Informationen in der Head-Unit, wie z. B. die Wetter-Prognose für den nächsten Tag oder Hinweise zu Sehenswürdigkeiten in der näheren Umgebung der aktuellen Fahrzeugposition, ist ein weiterer Anwendungsfall für die Implementierung einer optionalen Komponente. Diese Funktionalität könnte somit je nach Wunsch in das Modell integriert werden, etwa wenn eine Luxus-Variante der Head-Unit erzeugt werden soll.

2. Variabilität durch **alternative Komponenten**: Bei dieser Art eines Variationspunktes existiert eine Komponente, für die eine endliche Anzahl an alternativen Funktionalitäten implementiert wurde. Genau eine dieser Alternativen wird letztendlich in die Komponente eingefügt. Für jede Alternative gibt es eine Bedingung und auch hier wird nach der Auswertung dieser Bedingungen ein Varianten-Parameter gesetzt, dessen Wert die gewünschte Alternative repräsentiert.
   Beispiele für alternative Komponenten sind:

   - Für die Anzeige in der Head-Unit können existierende Farb- und Aufteilungskonzepte in alternativen Komponenten realisiert werden. Auf diese Weise ist es möglich, auf unterschiedliche gesetzliche Vorgaben bzw. Vorlieben der Käufer in den verschiedenen Märkten einzugehen.

   - Die Steuerung des Scheibenwischers lässt sich ebenfalls mithilfe alternati-

ver Komponenten implementieren. Verfügt das entsprechende Fahrzeug über einen Regensensor, so kann sich das Wischintervall der jeweiligen Wetterverhältnisse anpassen. Ist kein Sensor vorhanden, wird der Wischer alternativ mit festen Intervallzeiten gesteuert.

Optionale und alternative Komponenten werden im Allgemeinen als *variabel* bezeichnet. Darüber hinaus existieren auch obligatorische Komponenten, die nicht variabel sind und daher in jeder Systemvariante auftreten. Diese werden relevant, wenn sie z. B. im Laufe der Evolution der Produktlinie variabel werden und dadurch ein Variationspunkt entsteht.

Sind alle Variationspunkte durch optionale bzw. alternative Komponenten gebunden, kann das Funktionsmodell nur noch Variabilität durch Komponenten enthalten, deren Funktionalität mittels Parametrierung angepasst wird. Beispielsweise kann bei einer Automatikschaltung durch ein Kennfeld auf dem Getriebesteuergerät beeinflusst werden, ob die Fahrweise sportlich oder eher komfortabel ist. Entsprechend der Parameterwerte findet dann der Schaltvorgang später bzw. früher statt. Ein Parameter kann auch einen Grenzwert beschreiben. So lässt sich z. B. festlegen, ab welchem Abstand zum vorausfahrenden Fahrzeug ein Warnsignal ertönt. Diese Art der Variabilität wird jedoch im Entwicklungsprozess zu späteren Zeitpunkten gebunden, etwa nach dem Flashen der Software auf dem Steuergerät oder sogar erst nach dem Start-up des Fahrzeugs und wird daher üblicherweise nicht im Variantenmodell abgebildet. Da sich diese Arbeit mit Änderungen in einem Funktionsmodell beschäftigt, die direkten Einfluss auf das Variantenmodell haben, wird auf die Parametervariabilität nicht eingegangen.

Für die Realisierung der oben beschriebenen Arten von Variationspunkten existieren z. B. in Matlab / Simulink diverse Mechanismen [GW11, Wei08, Akc11]. Hier sollen in einer kurzen Übersicht nur die wichtigsten dargestellt werden:

- **Enabled Subsystem:** Mit dem *Enabled Subsystem*-Block kann man ein Subsystem aktivieren bzw. deaktivieren. Durch ein Kontrollsignal wird entschieden, ob die Funktionalität des Subsystems ausgeführt wird. Dieser Mechanismus eignet sich daher sehr gut, um eine optionale Auswahl einer Funktion zu modellieren. Grundsätzlich ist es auch möglich, mit dem *Enabled Subsystem*-Block die Auswahlmöglichkeit zwischen alternativen Funktionsmodulen abzubilden. Dazu müssen jedoch mehrere *Enabled Subsystem*-Blöcke verschachtelt werden, was die Modellierung unnötig unübersichtlich macht. Ein großer Vorteil des *Enabled Subsystem*-Mechanismus ist dagegen, dass damit durch die entsprechende Konfiguration des Kontrollblocks alle Bindungszeiten abgedeckt werden können. Die Bindung der Variabilität ist somit entweder bereits zur Erstellung des Simulink-Modells oder auch zu einem späteren Zeitpunkt möglich, beispielsweise zum Start-up des Steuergerätes. In Abb. 2.3 ist ein *Enabled Subsystem*-Block abgebildet, der durch den *Constant*-Block (*VAR_Schalter*) gesteuert wird.

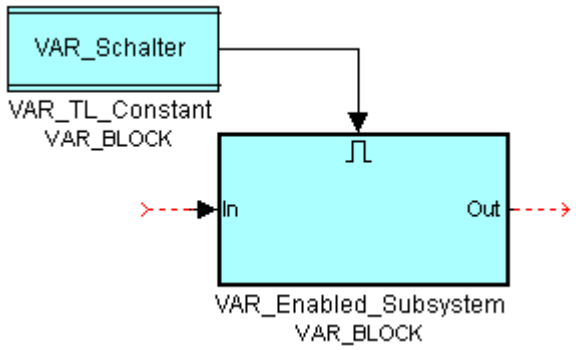


Abbildung 2.3.: Ein *Enabled Subsystem*-Block

- **Variant Subsystem:** Dieser Mechanismus eignet sich für die Auswahl zwischen alternativen Blockvarianten. Innerhalb des *Variant Subsystem*-Blocks werden die Alternativen als Subsysteme modelliert. Über ein variantenspezifisches Datenobjekt, das sogenannte *Variant Object*, entscheidet sich, welche Modellvariante ausgewählt wird. Dabei ist jedem *Variant Object* eine Bedingung zugeordnet. Ist diese Bedingung erfüllt, so wird die zugehörige Modellvariante aktiviert. Bei der Modellierung ist darauf zu achten, dass die Schnittstellen zwischen dem *Variant Subsystem*-Block und seinen Varianten identisch sind.

- **Switch- bzw. Multiport-Switch-Blöcke:** Für die Modellierung von Variabilität auf Signalebene eignen sich besonders *Switch-* bzw. *Multiport-Switch*-Blöcke. Beide Blocktypen wählen aus mehreren Eingangssignalen eine Variante und geben diese als Ausgangssignal aus. Die Auswahl des gewünschten Eingangssignals wird mithilfe des Kontrollsignals gesteuert, welches dafür mit einem konfigurierbaren Schwellwert verglichen wird. Der *Switch*-Block wählt dabei aus zwei Eingangssignalen aus, wohingegen bei dem *Multiport-Switch*-Block die Anzahl der Eingangssignale konfiguriert werden kann. Dadurch eignen sich beide Blocktypen besonders gut für die Modellierung einer Auswahl an alternativen Signalen. In Abb. 2.4 ist auf der linken Seite ein *Switch*-Block und auf der rechten Seite ein *Multiport-Switch*-Block dargestellt, die jeweils über ein Kontrollsignal aus dem *VAR_Schalter*-Block gesteuert werden.

- **Logische Gatter:** Eine weitere Möglichkeit, Variabilität auf Signalebene abzubilden, sind logische Gatter. Durch einen *Logical Operator*-Block wird dabei aus mehreren Eingangssignalen ein Ausgangssignal ermittelt. Dafür werden die Eingangssignale als Wahrheitswerte (*true* bzw. *false*) interpretiert und daraus das Ausgangssignal entsprechend des jeweiligen Operators (z. B. AND, OR etc.) bestimmt. Interpretiert man eines der Eingangssignale als Kontrollsignal, so lassen sich diese Blöcke als Variabilitätsmechanismen verwenden. Liegt beispielsweise auf dem Kontrollsignal eines logischen AND-Blocks der Wert 0, welcher als *false* inter-

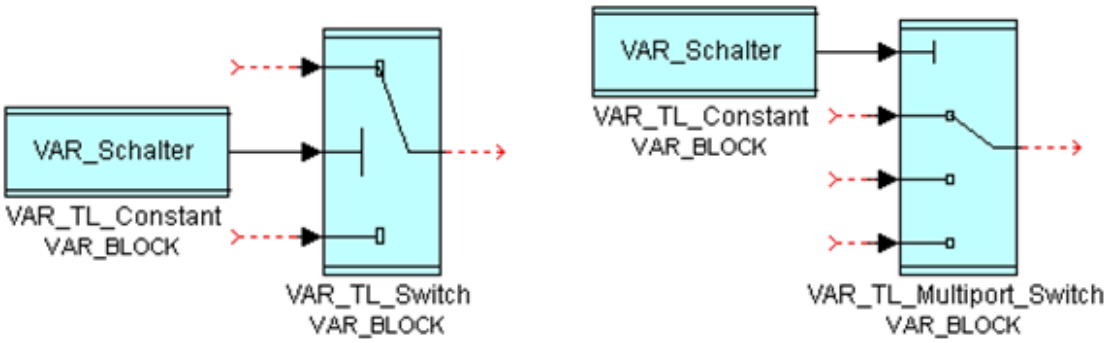


Abbildung 2.4.: Ein *Switch*- bzw. *Multiport-Switch*-Block

pretiert wird, so haben die übrigen Eingangssignale keinen Einfluss auf den Output. Ist der Wert dagegen ungleich Null, was als *true* interpretiert wird, so hängt das Ausgangssignal tatsächlich von den weiteren Eingangssignalen ab.

2.3. Software-Variantenmanagement

Die Basis für die Beherrschung variantenreicher Softwaresysteme ist ein Variantenmodell, welches von der technischen Umsetzung der enthaltenen Funktionalitäten abstrahiert und sich auf die Darstellung der unterschiedlichen Systemvarianten und die Methodik der Konfiguration beschränkt. Unabhängig von der eingesetzten Technik werden bei der Variantenmodellierung unter anderen die nachstehenden Ziele verfolgt:

- Unter der **Variabilitätsanalyse** versteht man die reaktive Untersuchung eines bestehenden Systems auf existierende Variantenstrukturen. Dabei stehen die Varianten des Systems mit deren gemeinsamen und unterschiedlichen funktionalen Eigenschaften im Vordergrund.

- Nach bzw. während der Analyse wird die Variabilität eines Systems in der Regel dokumentiert. Bei der **Variabilitätsdokumentation** geht es um die Extrahierung und Konzentration des Variabilitätswissens aus dem System in ein dafür vorgesehenes Modell. Implizites Wissen über Varianten und deren Abhängigkeiten soll dabei aus den Artefakten der Implementierung herausgezogen und an zentraler Stelle dokumentiert werden.

- Bei der **Variabilitätskontrolle** als Folgeaktivität der Variabilitätsanalyse und -dokumentation wird durch Regeln bzw. Bedingungen die Variantenvielfalt eingeschränkt und sowohl die bestehenden als auch die geplanten Varianten auf Konsistenz untersucht.

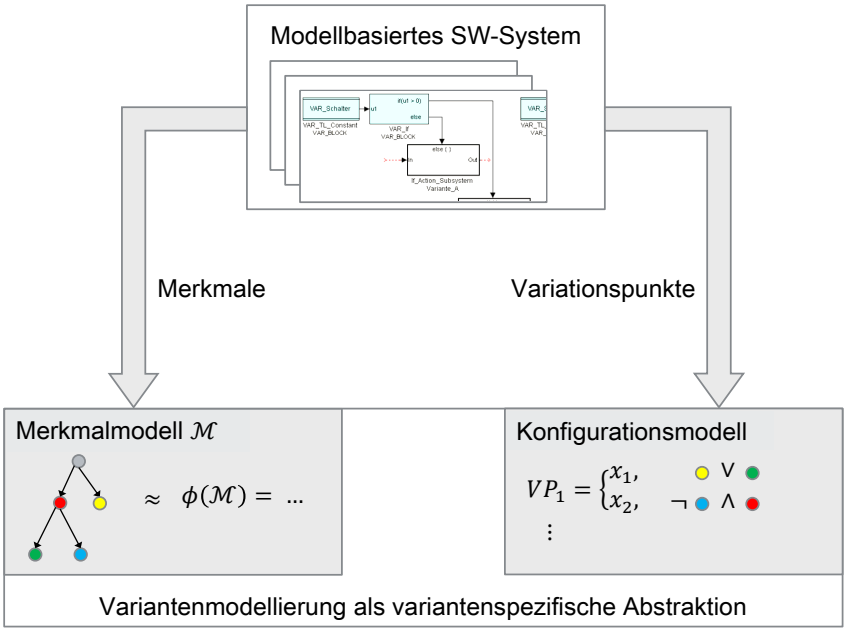


Abbildung 2.5.: Modellübersicht

- Die **Variabilitätsplanung** umfasst eine proaktive Variantenmodellierung vor dem Systementwurf und unterstützt damit den Entwickler bei dem Aufbau einer Produktlinie. Hierbei wird zunächst der Produktkern definiert und anschließend die Systemvarianten festgelegt. Auch für eine bestehende Produktlinie kann das Variantenmodell die Entscheidungsgrundlage für die Planung neuer Produktmerkmale auf funktionaler Ebene sein.

- Wurde die Variabilität des Softwaresystems modelliert, sorgen die Maßnahmen der **Variabilitätssteuerung** dafür, dass aus diesem Variantenmodell heraus eine automatische Variantenkonfiguration von Entwicklungsartefakten erfolgen kann. Dabei liefert das Variantenmodell die benötigten Informationen, um aus einem variantenreichen Artefakt eine Systemvariante abzuleiten.

Um diese Ziele zu erreichen, wurden in der Vergangenheit verschiedene Ansätze zur Modellierung von Software-Variabilität entwickelt. Neben der orthogonalen Variabilitätsmodellierung [MHP+07] hat sich auch die merkmalbasierte Variabilitätsmodellierung etabliert (s. z. B. [KCH+90]). Das dabei entstehende Merkmalmodell ist innerhalb der industriellen Praxis das am weitesten verbreitete Mittel, um Variabilität von Software zu modellieren [BRN+13]. Da das ebenso für das Automotive-Umfeld gilt, steht die Merkmalmodellierung im Fokus dieser Arbeit.

Eine Übersicht des Prozesses der Variabilitätsmodellierung zeigt Abb. 2.5. Wie im vorherigen Abschnitt beschrieben, wird in dieser Arbeit von einem modellbasierten Funktionsmodell ausgegangen, welches beispielsweise aus einem oder mehreren Simulink / Targetlink-Modellen bestehen kann (in der Abb. oben).

Daraus wird ein Variantenmodell erarbeitet, welches eine variantenspezifische Abstraktion des Softwaresystems darstellt. Dieses Variantenmodell besteht aus zwei Teilmodellen: dem Merkmalmodell und dem Konfigurationsmodell.

Das Merkmalmodell (in Abb. 2.5 unten links) fasst Eigenschaften des Funktionsmodells in einer Baumstruktur zusammen, welche als Merkmale bezeichnet werden. In dieser Baumstruktur lassen sich mehrere Variationstypen, also verschiedene Arten von Beziehungen zwischen Vater- und Kindmerkmalen unterscheiden. Dafür stehen die in der Literatur üblichen Variationstypen *obligatorisch*, *optional*, *alternativ* und *disjunktiv* zur Verfügung (s. z. B. [CE00]). Außerdem können dort auch Beziehungen zwischen Merkmalen definiert werden, die in der Hierarchie nicht unmittelbar benachbart sind. Bekannte Beispiele dafür sind die *requires*-Beziehung sowie die *conflicts*-Beziehung[2]. Das Merkmalmodell lässt sich außerdem in eine aussagenlogische Formel überführen (in der Abb. $\phi(\mathcal{M})$), worauf im nächsten Kapitel näher eingegangen wird.

Das Konfigurationsmodell (in Abb. 2.5 unten rechts) bildet die in der Software enthaltenen variablen Systemteile, also die in Abschnitt 2.2 beschriebenen Variationspunkte, und die für deren Konfiguration relevanten Informationen ab (in Abb. 2.5 dargestellt durch $VP_1, ...$). Eine detailliertere und auch formalisierte Beschreibung dieser beiden Teilmodelle beinhaltet Kapitel 3.

[2]K. Czarnecki spricht hierbei von „composition rules“ [Cza98].

3. Formalisierung

Im vorherigen Abschnitt 2.3 ist bereits von einem Variantenmodell die Rede, welches Merkmale und Variationspunkte enthält. Das vorliegende Kapitel beinhaltet eine Formalisierung dieser Elemente. Für den weiteren Verlauf der Arbeit stellt das ein wesentliches Ergebnis dar, weil dadurch

1. einerseits beliebige Änderungen an den Elementen des Variantenmodells formal beschrieben werden können. Das wird vor allem für die Änderungsprozesse aus Kapitel 4 von Bedeutung sein.

2. Andererseits ermöglicht die Formalisierung, bestimmte Modelleigenschaften nicht nur formal zu definieren, sondern auch maschinell zu analysieren. Die Konsistenzbedingungen aus Kapitel 5 können somit automatisch (z. B. mithilfe von SAT-Solvern) geprüft werden.

Da die Formalisierung auf Konzepte der Aussagenlogik zurückgreift, werden im ersten Abschnitt zunächst die relevanten Begriffe aus diesem Gebiet zusammengestellt. Das vorherige Kapitel hat gezeigt, dass das Variantenmodell aus einem Merkmal- und einem Konfigurationsmodell besteht, welche Inhalt der darauf folgenden Abschnitte sind. Es wird in diesem Kapitel außerdem auf Selektionen als Formalisierung von Varianten eingegangen. Diese bilden die Grundlage für den Konfigurationsprozess, welcher im letzten Abschnitt beschrieben wird.

3.1. Begriffe der Aussagenlogik

Dieser Abschnitt enthält einige grundlegende Definitionen der Aussagenlogik, die im Verlauf der Arbeit benötigt werden. Es wird davon ausgegangen, dass die Begriffe *aussagenlogische Variable* bzw. *aussagenlogische Formel* bekannt sind[1].

Definition 3.1. *Sei X eine beliebige Menge von aussagenlogischen Variablen und ϕ eine aussagenlogische Formel.*

[1] Für eine Einführung in die Aussagenlogik siehe z. B. [Ass83].

Dann bezeichnen wir mit

- $V(\phi) \subset X$ *die Menge der aussagenlogischen Variablen in* ϕ*, mit*
- $F(X) = \{\phi \mid V(\phi) \subset X\}$ *die Menge aller aussagenlogischen Formeln mit Variablen aus* X *und mit*
- $f : X \to \{true, false\}$ *eine Belegung von* X *mit Wahrheitswerten, wobei*
- $B(X) = \{f \mid f : X \to \{true, false\}\}$ *die Menge der Belegungen von* X *und*
- $f(\phi)$ *den Wahrheitswert der Formel* ϕ *unter der Belegung* f *beschreibt.*

Eine aussagenlogische Formel kann nun verschiedene Eigenschaften haben. Die nachfolgende Definition beschreibt, welche davon für unsere weiteren Betrachtungen relevant sind.

Definition 3.2. *Sei* $\phi \in F(X)$ *eine Formel.*

- ϕ *heißt* ***tautologisch****, falls gilt:* $f(\phi) = true$ *für alle* $f \in B(X)$*.*
- ϕ *heißt* ***kontradiktorisch****, falls gilt:* $f(\phi) = false$ *für alle* $f \in B(X)$*.*
- ϕ *heißt* ***erfüllbar****, falls gilt:* $f(\phi) = true$ *für mindestens ein* $f \in B(X)$*.*

Der nächste Abschnitt wird zeigen, dass sich das Merkmalmodell durch Konzepte der mathematischen Logik beschreiben lässt. Ein zentraler Begriff aus diesem Bereich ist die *Struktur*. Wie wir später sehen werden, stellt diese eine Verallgemeinerung des Merkmalmodells dar.

Definition 3.3. *Seien* I, J *und* K *beliebige Indexmengen. Eine* ***Struktur*** $\mathcal{A}$ *ist ein 4-Tupel*

$$\mathcal{A} = (A, (R_i^{\mathcal{A}} \mid i \in I), (f_j^{\mathcal{A}} \mid j \in J), (c_k^{\mathcal{A}} \mid k \in K))$$

wobei für jedes $i \in I, j \in J, k \in K$ *gilt:*

1. $A \neq \emptyset$ *ist der Träger der Struktur* $\mathcal{A}$
2. $R_i^{\mathcal{A}} \subset A^{n_i}$ *ist eine* n_i*-stellige Relation auf* A
3. $f_j^{\mathcal{A}} : A^{m_j} \to A$ *ist eine* m_j*-stellige Funktion auf* A
4. $c_k^{\mathcal{A}} \in A$ *ist ein Element (eine Konstante) von* A

Das Tripel $((n_i \mid i \in I), (m_j \mid j \in J), K)$ *wird dabei als Signatur bzw. Typ der Struktur bezeichnet.*

Die Menge der Merkmale kann später als Träger unserer Struktur fungieren. Darüber hinaus werden vor allem die oben erwähnten Relationen eine große Rolle bei der Formalisierung des Variantenmodells spielen. Wir werden sehen, dass sich diverse Beziehungen zwischen Merkmalen als zweistellige Relationen beschreiben lassen. Außerdem wird die Wurzel unseres Merkmalbaumes die einzige Konstante unserer Struktur darstellen.

Einer Struktur liegt immer eine formale Sprache zugrunde, welche beispielsweise die Zeichen für Relationen oder Funktionen bereitstellt. In dieser Sprache kann man nun auch Formeln innerhalb der Struktur formulieren. Für eine gegebene Formel ist dann unter anderem von Interesse, ob diese in der Struktur wahr ist.

Definition 3.4. *Sei nun* $\mathcal{A}$ *eine Struktur,* A *deren Träger,* $(R_i^{\mathcal{A}} \mid i \in I)$ *die Relationen der Struktur und* J *eine endliche Indexmenge. Sei außerdem* ϕ *eine Formel mit der Gestalt*

$$\phi = \bigwedge_{j \in J} R_j^{\mathcal{A}}(x_{j_1}, \ldots, x_{j_n}).$$

Gilt dann

$$R_j^{\mathcal{A}}(x_{j_1}, \ldots, x_{j_n}) \text{ für alle } j \in J \text{ und für alle } x_{j_1}, \ldots, x_{j_n} \in A,$$

so ist die Formel ϕ ***wahr in*** $\mathcal{A}$*. Wir sagen auch: die Struktur* $\mathcal{A}$ *ist ein* ***Modell der Formel*** ϕ*.*

Diese Eigenschaft lässt sich für aussagenlogische Formeln beliebiger Gestalt definieren. Wir haben uns hier auf diese Form beschränkt, da sie uns für die Formalisierung des Merkmalmodells im nächsten Abschnitt ausreichen wird. Auch andere Aspekte auf dem Weg zu dieser Definition, die in der mathematischen Logik üblicherweise detaillierter beschrieben werden, haben wir vernachlässigt, da sie keinen Beitrag zur Variantenmodellierung leisten. Hier sollten lediglich die für unsere Anwendung benötigten Begriffe aus dieser Theorie zusammengefasst werden.

3.2. Das Merkmalmodell

Dieser Abschnitt geht darauf ein, wie sich das Merkmalmodell mithilfe des Konzepts der Struktur beschreiben lässt, welches im vorherigen Abschnitt definiert wurde. Daher leiten wir zunächst die aussagenlogische Formel ϕ ab, welche das Merkmalmodell repräsentiert. Anschließend werden wir sehen, dass die Struktur des Merkmalmodells dadurch zu einem Modell für die Formel ϕ wird.

Definition 3.5. *Sei zunächst* $\mathcal{MB} = (M, (obl, opt, alt, dis), \emptyset, r)$ *eine Struktur mit vier zweistelligen Relationen, keiner Funktion und einer Konstante. Sei außerdem*

$$H := obl \ \dot\cup \ opt \ \dot\cup \ alt \ \dot\cup \ dis \quad \subset M \times M$$

Wir nennen $\mathcal{MB}$ *einen* ***Merkmalbaum****, falls gilt:*

(M1) *Für jedes* $m \in M \setminus \{r\}$ *existiert genau ein* $x \in M$ *mit* $(x, m) \in H$

(M2) *Es existiert kein* $x \in M$ *mit* $(x, r) \in H$

(M3) *Für jedes* $x \in M$ *existieren* $x_1, \ldots, x_n \in M$ *mit* $(r, x_1), \ldots, (x_n, x) \in H$

Mit anderen Worten: (M, H) ist ein Baum. Die Menge H definiert damit eine Hierarchie auf der Menge M aller Merkmale. Daher wird im Folgenden für $(x, y) \in H$ auch die Rede davon sein, dass x das Vatermerkmal von y bzw. y das Kindmerkmal von x ist.

Für spätere Betrachtungen, wie z. B. die Definition einer Produktvariante oder die Frage, ob eine solche Variante gültig ist, benötigen wir eine formale Beschreibung der Auswahl eines Merkmales. Dafür werden wir im weiteren Verlauf die Merkmale als aussagenlogische Variablen auffassen:

$$M = \{m_1, \ldots, m_n \mid m_i \in \mathbb{B} = \{true, false\}\}$$

Das hat die folgende Bedeutung: Hat ein Merkmal den Wert *true*, so ist es in dem entsprechenden Kontext ausgewählt, hat es den Wert *false*, ist es nicht ausgewählt. Alle aussagenlogische Formeln, die sich mithilfe der Merkmale als Variablen formulieren lassen, sind demnach in der Menge $F(M)$ zusammengefasst (vgl. Abschnitt 3.1), welche im weiteren Verlauf der Arbeit noch eine wichtige Rolle spielen wird.

Mit der oben beschriebenen Bedeutung der Merkmale können wir nun das Merkmalmodell definieren, indem wir unter anderem etwas genauer auf die hierarchischen Beziehungen darin eingehen. In Abschnitt 2.3 war die Rede von vier verschiedenen Variationstypen, die in einem Merkmalbaum auftreten können. Diese Variationstypen bestimmen das Verhältnis zwischen Vater- und Kindmerkmal in der Merkmalhierarchie.

Analog zu den hierarchischen Beziehungen H wollen wir zwischen den Merkmalen auch weitere Constraints

$$C := req \ \cup \ con \quad \subset M \times M$$

zulassen[2]. Wir konzentrieren uns dabei auf die beiden Beziehungen *requires* (*req*) und *conflicts* (*con*), da sie in der Praxis am häufigsten auftreten:

[2] In der Literatur werden diese Beziehungen auch als *cross-tree-constraints* bezeichnet (s. z. B. [BCTS06]).

1. Es besteht einerseits die Möglichkeit, eine sogenannte *requires*-Beziehung zwischen den Merkmalen $x, y \in M$ zu definieren ($(x, y) \in req \subset C$). Dadurch ist jede Merkmalauswahl ungültig, die x, aber nicht y enthält. Für eine requires-Beziehung gibt es verschiedene Anwendungsfälle. Man kann beispielsweise modellieren, dass eine Funktion einen bestimmten Sensor zusammen mit dem Modul benötigt, welches die Sensordaten verarbeitet. Aber auch Abhängigkeiten zwischen funktionalen und nicht-funktionalen Merkmalen lassen sich durch die requires-Beziehung ausdrücken. Ein typischer Anwendungsfall wäre, falls in einem bestimmten Markt aus gesetzlichen Gründen stets eine gewisse Funktion benötigt wird. In vielen Ländern ist z. B. Tagfahrlicht für PKW und LKW vorgeschrieben.

2. Mit der sogenannten *conflicts*-Beziehung kann andererseits der gegenseitige Ausschluss zweier Merkmale $x, y \in M$ definiert werden ($(x, y) \in con \subset C$). Dadurch dürfen x und y in einer Merkmalauswahl nicht zusammen auftreten. Gibt es beispielsweise für ein Modul zwei verschiedene Implementierungen, kann mit dieser Beziehung sichergestellt werden, dass nur eine davon selektiert wird.

Um diese beiden Constraints abbilden zu können, erweitern wir nun die Struktur des Merkmalbaumes um die Relationen *requires* bzw. *conflicts* und erhalten so das Merkmalmodell. Welche Semantik dabei die hierarchischen Beziehungen $H \subset M \times M$ und die Constraints $C \subset M \times M$ haben, zeigt die folgende Definition:

Definition 3.6. *Sei $\mathcal{MB} = (M, (obl, opt, alt, dis), \emptyset, r)$ ein Merkmalbaum und seien $x, y, y_1, \ldots, y_k \in M$.*

Wir nennen $\mathcal{M} = (M, (obl, opt, alt, dis, req, con), \emptyset, r)$ ein ***Merkmalmodell****, falls gilt:*

(H1) *Für $(x, y) \in obl$ gilt: $x \leftrightarrow y$.*
Dabei nennen wir y ein ***obligatorisches Merkmal****.*

(H2) *Für $(x, y) \in opt$ gilt: $y \rightarrow x$.*
Dabei nennen wir y ein ***optionales Merkmal****.*

(H3) *Für $(x, y_1), \ldots, (x, y_k) \in alt$ gilt:*[3]

$$(x \leftrightarrow (y_1 \vee \cdots \vee y_k)) \wedge \bigwedge_{i<j} \neg(y_i \wedge y_j).$$

Dabei nennen wir $y_1, \ldots, y_k$ ***alternative Merkmale****.*

(H4) *Für $(x, y_1), \ldots, (x, y_k) \in dis$ gilt:*

$$x \leftrightarrow (y_1 \vee \cdots \vee y_k).$$

[3] In der Literatur finden sich unterschiedliche (logisch äquivalente) Formalisierungen für alternative Merkmale. Diese stammt in abgewandelter Form aus [TKES11].

Dabei nennen wir $y_1, \ldots, y_k$ ***disjunktive Merkmale****.*

(C1) *Für* $(x, y) \in req$ *gilt:* $x \rightarrow y$*.*

(C2) *Für* $(x, y) \in con$ *gilt:* $\neg(x \wedge y)$*.*

Betrachtet man diese Definitionen vor dem Hintergrund der oben beschriebenen Semantik des Wahrheitswertes eines Merkmals, so lässt sich die Bedeutung der Variationstypen auch folgendermaßen formulieren:

- Ein obligatorisches Merkmal *muss* ausgewählt werden, falls dessen Vatermerkmal ausgewählt wurde.
- Ein optionales Merkmal *kann* ausgewählt werden, falls dessen Vatermerkmal ausgewählt wurde.
- Aus einer Gruppe alternativer Merkmale muss *genau ein* Merkmal ausgewählt werden, falls das Vatermerkmal der Gruppe ausgewählt wurde.
- Aus einer Gruppe disjunktiver Merkmale muss *mindestens ein* Merkmal ausgewählt werden, falls das Vatermerkmal der Gruppe ausgewählt wurde.

Alle nicht obligatorischen Merkmale werden wir auch als *variabel* bezeichnen. Bei den obligatorischen Merkmalen ist für manche der weiteren Betrachtungen die folgende Unterscheidung relevant:

Definition 3.7. *Sei* $\mathcal{M} = (M, (obl, opt, alt, dis, req, con), \emptyset, r)$ *ein Merkmalmodell mit* n *Merkmalen, wobei wir ohne Einschränkung* $m_n = r$ *annehmen. Sei außerdem* $m_i \in M$ *ein obligatorisches Merkmal mit* $(m_n = r, m_{n-1}), \ldots, (m_{i+1}, m_i) \in H$*. Dann bezeichnen wir* m_i *als* ***echt obligatorisch****, falls gilt:*

$$(m_n = r, m_{n-1}), \ldots, (m_{i+1}, m_i) \in obl \subset H.$$

Andernfalls sprechen wir davon, dass das Merkmal ***bedingt obligatorisch*** *ist.*

Ein echt obligatorisches Merkmal hat somit in der Merkmalhierarchie nur obligatorische Merkmale über sich, sodass es in jeder Merkmalauswahl vorhanden sein muss. Ist m_i dagegen bedingt obligatorisch, so befindet es sich in der Hierarchie unter mindestens einem Merkmal, das optional, alternativ oder disjunktiv ist. Dieses Merkmal wird im weiteren Verlauf als der *variable Vater* von m_i bezeichnet:

Definition 3.8. *Sei* $m_i \in M$ *ein bedingt obligatorisches Merkmal in* $\mathcal{M}$ *mit*

$$(m_n = r, m_{n-1}), \ldots, (m_{i+1}, m_i) \in H.$$

*Das erste nicht-obligatorische Merkmal in der Folge $m_{i+1}, \ldots, m_{n-1}, m_n = r$ nennen wir den **variablen Vater** von m_i.*

Durch die obige Definition des Merkmalmodells wird

$$\mathcal{M} = (M, (obl, opt, alt, dis, req, con), \emptyset, r)$$

zu einer Struktur mit den folgenden Eigenschaften:

1. Der Träger der Struktur ist die Menge aller Merkmale M.

2. Es existieren die Relationen $obl, opt, alt, dis \subset M \times M$, um die bereits in Abschnitt 2.3 erwähnten Variationstypen zu beschreiben. Diese müssen disjunkt sein, da für jedes Merkmal der Variationstyp eindeutig ist. Die Menge H umfasst damit alle hierarchischen Beziehungen im Modell. Außerdem gibt es zwei weitere Relationen, um Constraints zwischen beliebigen Merkmalen auszudrücken: $req, con \subset M \times M$. Diese werden in der Menge C zusammengefasst.

3. Die Struktur enthält keine Funktionen.

4. Das Wurzelmerkmal $r \in M$ ist die einzige Konstante der Struktur $\mathcal{M}$.

Ein Merkmalmodell ist durch die Angabe der Menge M aller Merkmale, der Menge H der hierarchischen Beziehungen, der Menge C der Constraints und des Wurzelmerkmals $r \in M$ eindeutig festgelegt. Da die Struktur des Merkmalmodells nach obiger Definition keine Funktionen enthält, werden wir es im Folgenden zu Gunsten der Übersichtlichkeit mit $\mathcal{M} = (M, H \cup C, r)$ bezeichnen, anstatt mit $\mathcal{M} = (M, (obl, opt, alt, dis, req, con), \emptyset, r)$.

Beispiel. *In Abb. 3.1 ist ein Merkmalmodell für die Fahrerassistenzsysteme dargestellt[4]. Es enthält das Wurzelmerkmal „Fahrerassistenzsysteme“ und außerdem*

- *die obligatorischen Merkmale „Limiter“ und „Radarsysteme“, die in der Grafik durch ein gelbes Ausrufezeichen vor dem Merkmalnamen gekennzeichnet sind,*

- *die optionalen Merkmale „Tempomat“, „Bremsassistent“, „Nahbereichsradar“ und „Fernbereichsradar“, die in der Grafik durch ein rotes Fragezeichen vor dem Merkmalnamen gekennzeichnet sind,*

- *eine Gruppe alternativer Merkmale, welche aus den alternativen Merkmalen „einstufiger_Tempomathebel“ und „zweistufiger_Tempomathebel“ besteht, die in der Grafik durch einen grünen Doppelpfeil vor dem Merkmalnamen gekennzeichnet sind*

[4]Die in dieser Arbeit gezeigten Variantenmodelle wurden mit dem bereits erwähnten Eclipse-Plug-In pure::variants [Sys14] erstellt.

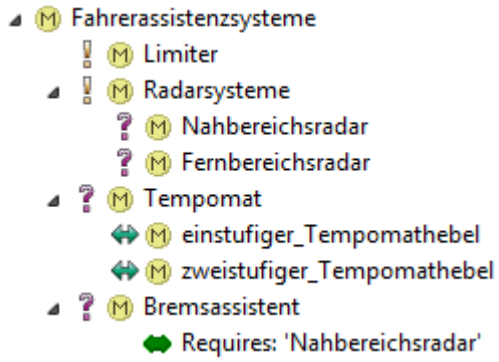


Abbildung 3.1.: Das Merkmalmodell

und keine disjunktiven Merkmale[5]*. Darüber hinaus wurde mithilfe eines Constraints die Beziehung formuliert, dass der Bremsassistent den Nahbereichsradar benötigt. In Abb. 3.1 wird diese Beziehung durch den grünen Doppelpfeil unter dem Merkmal Bremsassistent und das Schlüsselwort „Requires“ dargestellt. Der zugehörige Constraint in Aussagenlogik lautet in dem Fall:*

$$Bremsassistent \rightarrow Nahbereichsradar$$

Die Struktur $\mathcal{M} = (M, (obl, opt, alt, dis, req, con), \emptyset, r)$ *dieses Merkmalmodells hat dementsprechend die folgende Gestalt:*

- M = *{Fahrerassistenzsysteme, Limiter, Radarsysteme, Nahbereichsradar, Fernbereichsradar, Tempomat, einstufiger_Tempomathebel, zweistufiger_Tempomathebel, Bremsassistent}*
- obl = *{(Fahrerassistenzsysteme, Limiter), (Fahrerassistenzsysteme, Radarsysteme)}*
- opt = *{(Fahrerassistenzsysteme, Tempomat), (Fahrerassistenzsysteme, Bremsassistent), (Radarsysteme, Nahbereichsradar), (Radarsysteme, Fernbereichsradar)}*
- alt = *{(Tempomat, einstufiger_Tempomathebel), (Tempomat, zweistufiger_Tempomathebel)}*
- req = *{(Bremsassistent, Nahbereichsradar)}*
- $dis = con = \emptyset$

[5] Generell spielen disjunktive Merkmale in der industriellen Praxis eher eine untergeordnete Rolle. Sie sind aus Gründen der Vollständigkeit in die Formalisierung integriert, werden jedoch wegen der wenigen praktischen Anwendungsfälle im weiteren Verlauf der Arbeit nur am Rande erwähnt.

- $r = \textit{Fahrerassistenzsysteme}$

Ausgehend von der obigen Definition des Merkmalmodells lässt sich nun dessen aussagenlogische Repräsentation erstellen. Erstmals wurde die Übersetzung der Variantenmodellierung innerhalb einer Produktlinie in Aussagenlogik von Mannion [Man02] beschrieben. In weiteren Arbeiten wurden SAT-Solver vorgeschlagen, um mithilfe dieser Darstellung Eigenschaften von Merkmalmodellen zu prüfen [Bat05] bzw. die Extraktion von Merkmalmodellen aus aussagenlogischen Formeln betrachtet [CW07].

Definition 3.9. *Sei $\mathcal{M} = (M, H \cup C, r)$ ein Merkmalmodell mit $H = \{obl, opt, alt, dis\}$ und $C = \{req, con\}$. Dann bezeichnen wir mit $\phi(\mathcal{M}) \in F(M)$ die* ***Formel des Merkmalmodells*** *$\mathcal{M}$. Sie ist definiert durch[6]:*

$$\phi(\mathcal{M}) := \bigwedge_{(x,y)\in\ obl} (x \leftrightarrow y) \tag{3.1}$$

$$\wedge \bigwedge_{(x,y)\in\ opt} (y \rightarrow x) \tag{3.2}$$

$$\wedge \bigwedge_{\substack{x\in M \\ (x,y_1),\ldots,(x,y_k)\in\ alt}} (x \leftrightarrow (y_1 \vee \cdots \vee y_k)) \wedge \bigwedge_{i<j} \neg(y_i \wedge y_j) \tag{3.3}$$

$$\wedge \bigwedge_{\substack{x\in M \\ (x,y_1),\ldots,(x,y_k)\in\ dis}} (x \leftrightarrow (y_1 \vee \cdots \vee y_k)) \tag{3.4}$$

$$\wedge \bigwedge_{(x,y)\in\ req} (x \rightarrow y) \tag{3.5}$$

$$\wedge \bigwedge_{(x,y)\in\ con} (\neg x \vee \neg y) \tag{3.6}$$

Erklärung:

1. In Zeile 3.1 wird die Forderung aus der Definition für alle obligatorischen Merkmale ausgedrückt.

2. Zeile 3.2 bildet die Eigenschaft der optionalen Merkmale ab.

3. Zeile 3.3 repräsentiert die Eigenschaft der alternativen Merkmalgruppen.

4. Zeile 3.4 repräsentiert die Eigenschaft der disjunktiven Merkmalgruppen.

[6] Die Struktur der Formel ist angelehnt an die Darstellung in [SLB+11].

5. In Zeile 3.5 wird die Eigenschaft der *requires*-Constraints ausgedrückt.

6. In der letzten Zeile werden alle *conflicts*-Constraints des Merkmalmodells formuliert.

Die Formel $\phi(\mathcal{M})$ besteht aus einer Konjunktion von Ausdrücken, welche für jede existierende Relation deren Definition beschreibt. Sie wird damit von allen Elementen des Trägers unserer Merkmalmodellstruktur erfüllt, womit diese Struktur ein Modell der obigen Formel ist.

Darüber hinaus wird $\phi(\mathcal{M})$ im weiteren Verlauf der Arbeit noch an mehreren Stellen benötigt, z. B. für die formale Definition einer gültigen Variante; Außerdem wird die Formel im Kapitel über die Konsistenzbedingungen als aussagenlogische Darstellung des Merkmalmodells dabei behilflich sein, Inkonsistenzen zu definieren.

Beispiel. *Sei $\mathcal{M}$ das Merkmalmodell aus Abb. 3.1. Dann ist*

$$
\begin{aligned}
\phi(\mathcal{M}) = {} & (Fahrerassistenzsysteme \leftrightarrow Limiter) && (3.7)\\
& \wedge (Fahrerassistenzsysteme \leftrightarrow Radarsysteme) && (3.8)\\
& \wedge (Tempomat \rightarrow Fahrerassistenzsysteme) && (3.9)\\
& \wedge (Bremsassistent \rightarrow Fahrerassistenzsysteme) && (3.10)\\
& \wedge (Nahbereichsradar \rightarrow Radarsysteme) && (3.11)\\
& \wedge (Fernbereichsradar \rightarrow Radarsysteme) && (3.12)\\
& \wedge (Tempomat \leftrightarrow && (3.13)\\
& \quad (einstufiger\_Tempomathebel \vee zweistufiger\_Tempomathebel)) && (3.14)\\
& \wedge \neg(einstufiger\_Tempomathebel \wedge zweistufiger\_Tempomathebel) && (3.15)\\
& \wedge (Bremsassistent \rightarrow Nahbereichsradar) && (3.16)
\end{aligned}
$$

Zunächst werden die hierarchischen Beziehungen für die obligatorischen (Zeile 3.7 bis 3.8), optionalen (Zeile 3.9 bis 3.12) und alternativen (Zeile 3.13 bis 3.15) Merkmale formuliert, bevor in der letzten Zeile der Constraint die Formel komplettiert.

3.3. Eine Selektion

Der erste Schritt auf dem Weg zu einer Produktvariante ist die Auswahl von Merkmalen. Ist $\mathcal{M} = (M, H \cup C, r)$ ein Merkmalmodell, so beschreibt eine Belegung $f : M \rightarrow \{true, false\}$ eine Merkmalauswahl in dem Sinne, dass jedes Merkmal $m_i \in M$ mit $f(m_i) = true$ in der Belegung f ausgewählt ist und jedes Merkmal $m_j \in M$ mit $f(m_j) = false$ in der Belegung f nicht ausgewählt ist.

Die folgende Definition zeigt, dass sich eine Merkmalauswahl auch als aussagenlogische Formel ausdrücken lässt. Da das Merkmalmodell und im nächsten Abschnitt auch das Konfigurationsmodell mithilfe von aussagenlogischen Formeln formalisiert werden, konzentrieren wir uns im weiteren Verlauf der Arbeit auf diese Darstellung. Im Anhang findet sich ein Beweis, dass die Darstellungen einer Merkmalauswahl als Belegung bzw. als Formel äquivalent sind (s. dazu Abschnitt A.1.1).

Definition 3.10. *Sei $\mathcal{M} = (M, H \cup C, r)$ ein Merkmalmodell mit $M = \{m_1, \ldots, m_n\}$. Dann nennen wir $S \in F(M)$ eine **Selektion** von $\mathcal{M}$, falls gilt:*

$$S = s_1 \wedge \cdots \wedge s_n \text{ mit } s_i = m_i \text{ oder } s_i = \neg m_i \text{ für } 1 \leq i \leq n$$

*Wir bezeichnen außerdem mit $S(M) \subset F(M)$ die Menge aller Selektionen dieser Art. Eine Selektion $S \in F(M)$ heißt **gültig**, falls die Formel $S \wedge \phi(\mathcal{M}) \in F(M)$ erfüllbar ist.*

Die Menge aller gültigen Selektionen für ein gegebenes Merkmalmodell wird in der Literatur auch als die *Semantik des Merkmalmodells* bezeichnet [Bor09]. Wir werden im weiteren Verlauf der Arbeit Formulierungen benutzen, wie „die Selektion S enthält das Merkmal m_i“ bzw. „das Merkmal m_j ist nicht in der Selektion S enthalten“, um zum Ausdruck zu bringen, dass $S = s_1 \wedge \cdots \wedge s_n \in F(M)$ eine Selektion ist, mit $s_i = m_i$ bzw. $s_j = \neg m_j$.

Beispiel. *Die unten dargestellten Formeln S_1 und S_2 aus $F(M)$ stellen jeweils eine Selektion für das Merkmalmodell in Abb. 3.1 dar:*

$$\begin{aligned}
S_1 = {} & Fahrerassistenzsysteme \wedge Limiter \wedge \neg\, Tempomat \\
& \wedge \neg\, einstufiger_Tempomathebel \wedge \neg\, zweistufiger_Tempomathebel \\
& \wedge Bremsassistent \wedge Radarsysteme \\
& \wedge Nahbereichsradar \wedge \neg\, Fernbereichsradar \\
S_2 = {} & Fahrerassistenzsysteme \wedge Limiter \wedge \neg\, Tempomat \\
& \wedge \neg\, einstufiger_Tempomathebel \wedge \neg\, zweistufiger_Tempomathebel \\
& \wedge \neg\, Bremsassistent \wedge \neg\, Radarsysteme \\
& \wedge \neg\, Nahbereichsradar \wedge \neg\, Fernbereichsradar
\end{aligned}$$

Eine Darstellung der Selektion S_1 zeigt Abb. 3.2. Darin werden gewählte Merkmale durch einen schwarzen Haken und nicht gewählte Merkmale durch ein weißes „x“ in einem roten Kästchen gekennzeichnet. Während S_1 eine gültige Selektion für unser Merkmalmodell der Fahrerassistenzsysteme beschreibt, ist die Selektion S_2 ungültig, da das echt obligatorische Merkmal „Radarsysteme“ negiert ist.

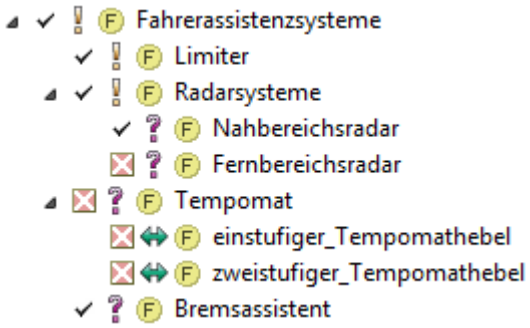


Abbildung 3.2.: Eine Darstellung der Selektion S_1

3.4. Das Konfigurationsmodell

Wie bereits in Abschnitt 2.3 erwähnt, werden im Konfigurationsmodell alle Variationspunkte des Softwaresystems modelliert. Deren Ausprägungen werden dabei als Variationen dokumentiert.

Definition 3.11. *Sei $\mathcal{M} = (M, H \cup C, r)$ ein Merkmalmodell. Ein zugehöriges **Konfigurationsmodell** ist eine endliche Menge von Variationspunkten $KM = \{VP_1, \ldots, VP_N\}$. Dabei besteht jeder **Variationspunkt** $VP = ((V_1, KF_1), \ldots, (V_p, KF_p))$ aus einer endlichen Menge von **Variationen** $V_1, \ldots, V_p$. Jeder Variation V_i ist eine **Konfigurationsformel** $KF_i \in F(M)$ zugeordnet.*

Damit sind Merkmalmodell und Konfigurationsmodell über die Konfigurationsformeln an den Variationspunkten miteinander verknüpft, da diese sich nur aus Merkmalen zusammensetzen können. Mithilfe der beiden Teilmodelle definieren wir nun das Variantenmodell.

Definition 3.12. *Das Tupel $VM = (\mathcal{M}, KM)$ aus einem Merkmalmodell und einem zugehörigen Konfigurationsmodell nennen wir ein **Variantenmodell**.*

Beispiel. *In Abb. 3.3 ist ein Konfigurationsmodell zu dem Merkmalmodell in Abb. 3.1 dargestellt. Es enthält zwei Variationspunkte:*

- *den Variationspunkt $VP_1 = VAR\_Bremsassistent$ mit den Variationen $V_1 = (Leermodul, \neg\ Bremsassistent)$ und $V_2 = (Vollmodul, Bremsassistent)$, und*
- *den Variationspunkt $VP_2 = VAR\_Tempomat$ mit den Variationen $V_1 = (Leermodul, \neg\ Tempomat)$,*

Variation Point	Assignment
VAR_Bremsassistent	
1 (Leermodul)	NOT(Bremsassistent)
2 (Vollmodul)	Bremsassistent
VAR_Tempomat	
1 (Leermodul)	NOT(Tempomat)
2 (einstufiger Tempomathebel)	einstufiger_Tempomathebel
3 (zweistufiger Tempomathebel)	zweistufiger_Tempomathebel

Abbildung 3.3.: Ein Konfigurationsmodell

$V_2 = (einstufiger\ Tempomathebel, einstufiger\_Tempomathebel)$ *und*
$V_3 = (zweistufiger\ Tempomathebel, zweistufiger\_Tempomathebel)$.

Die beiden Variationspunkte VP_1 und VP_2 haben damit vergleichsweise einfache Konfigurationsformeln, da sie jeweils nur aus einem Literal bestehen. Grundsätzlich sind jedoch komplexere Konfigurationsformeln möglich und auch realistisch. VP_1 könnte beispielsweise die Modellierung einer optionalen Komponente sein, welche, abhängig von einem Parameter, entweder die eingehenden Signale ohne Verarbeitung weitergibt (Leermodul) oder einen Bremsassistenten implementiert (Vollmodul). Wie dieser Parameter bestimmt wird, ist Inhalt des nächsten Abschnittes über den Konfigurationsprozess.

3.5. Der Konfigurationsprozess

Der Konfigurationsprozess sieht vor, dass durch die Auswahl der gewünschten Merkmale eine Systemvariante erstellt wird. Er beschreibt damit den Weg von einer gültigen Selektion $S \in F(M)$ zu einer korrekten Variante eines Funktionsmodells. Die eigentliche Konfigurationsaufgabe besteht darin, aus einer gegebenen Selektion genau die Menge an Variationen zu ermitteln, die dafür sorgen, dass das Funktionsmodell mit diesen Variationen die spezifizierte Funktionalität implementiert. Liegt eine Selektion $S \in F(M)$ vor, so lässt sich anhand der Konfigurationsformeln für jeden Variationspunkt die entsprechende Variation bestimmen.

Definition 3.13. *Sei dazu* $VP = ((V_1, KF_1), \ldots, (V_p, KF_p)) \in KM$ *ein Variationspunkt. Die* ***an dem Variationspunkt*** VP ***für eine gegebene Selektion*** S ***gültige***

Variation *$VP(S)$ kann mithilfe der Konfigurationsformeln ermittelt werden:*

$$VP(S) = \begin{cases} V_1, & \textit{falls } S \wedge KF_1 \textit{ erfüllbar ist} \\ \quad\vdots & \\ V_p, & \textit{falls } S \wedge KF_p \textit{ erfüllbar ist} \end{cases}$$

In diesem Fall wird im weiteren Verlauf der Arbeit auch die folgende Formulierung verwendet: Die Selektion S erfüllt die Konfigurationsformel KF_i.

Wären beispielsweise für eine Premium-Variante der Fahrerassistenzsysteme unter anderen die Merkmale *Tempomat* und *zweistufiger_Tempomathebel* ausgewählt, so würde der Konfigurationsprozess am Variationspunkt $VP_2 = VAR\_Tempomat$ (s. Abb. 3.3) die Variation V_3 als *gültig* bestimmen. In Abb. 3.4 ist der Konfigurationsprozess schematisch dargestellt. Der Ausgangspunkt ist eine Merkmalauswahl, welche, wie im vorherigen Abschnitt beschrieben, als Selektion S formalisiert wird. Mithilfe der Formel des Merkmalmodells $\phi(\mathcal{M})$ wird die Gültigkeit der Selektion festgestellt. Damit kann anschließend für jeden Variationspunkt VP_i anhand der Konfigurationsformeln $KF_{i1}, \ldots, KF_{ip}$ die für S gültige Variation $VP_i(S) = V_{ij}$ ermittelt werden. Auf diese Art und Weise erhält man für jeden Variationspunkt genau eine Variation, welche mithilfe des bereits erwähnten Parameters codiert wird. Diese Liste an Variationspunkt-Parameter-Paaren wird im letzten Schritt in das Funktionsmodell exportiert, um die noch offene Variabilität zu binden. In unserem Beispiel würde dabei in die Implementierung der Fahrerassistenzsysteme durch einen entsprechenden Mechanismus die Komponente des Tempomaten mit zweistufigem Tempomathebel eingefügt werden.

Dabei ist es für den Konfigurationsprozess essentiell, dass diese Auswertung eindeutig ist. Liegt eine gültige Merkmalselektion $S \in F(M)$ vor, so muss für jeden Variationspunkt im Konfigurationsmodell genau eine Variation gültig werden, was bedeutet, dass an jedem Variationspunkt genau eine Konfigurationsformel von S erfüllt wird. Ist an einem Variationspunkt diese Eindeutigkeit nicht gegeben, so kann dessen Parameter gar nicht oder ggf. nur inkorrekt ermittelt werden. Die Folge davon ist ein inkonsistentes bzw. falsch konfiguriertes Artefakt. In Kapitel 5 werden unter anderen Konsistenzbedingungen vorgestellt, welche Verletzungen dieser Eindeutigkeit im Modell formal beschreiben.

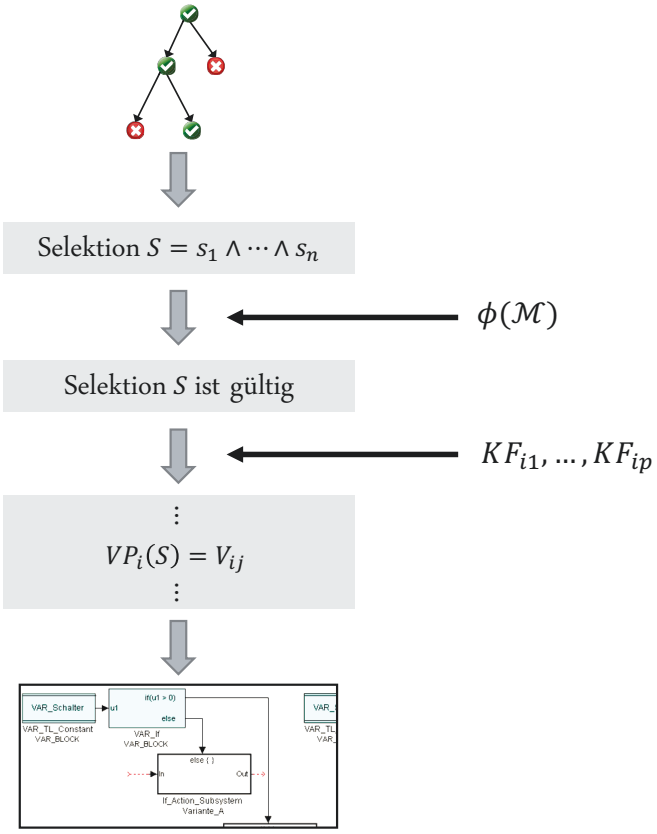


Abbildung 3.4.: Der Konfigurationsprozess

4. Konstruktiv - Evolution des Variantenmodells

Wie bereits in Abschnitt 1.1 beschrieben wurde, unterliegt im Automobilbereich die eingesetzte Software einer ständigen Evolution. So werden zum Funktionsmodell beispielsweise neue Komponenten hinzugefügt, bestehende fallen weg oder werden an neue Anforderungen angepasst. Der erste Abschnitt dieses Kapitels beschreibt daher die häufigsten solcher Evolutionsmuster, die für die Variabilität des Softwaresystems relevant sind. Für jedes Muster wird ein Änderungsprozess vorgeschlagen, der die nötigen elementaren Schritte zusammenfasst, um das Variantenmodell an die geänderte Situation anzupassen.

Für bestimmte Anpassungen des Variantenmodells hängt die Modellierung stark vom Kontext ab, wodurch keine allgemeingültige Vorgabe gemacht werden kann. Daher werden im zweiten Abschnitt Änderungen dieser Art näher betrachtet und Modellierungsalternativen diskutiert.

4.1. Die Änderungsprozesse

Die Änderungsprozesse verfolgen mehrere Ziele. Wenn zentral definiert ist, wie im Falle einer Evolution des Funktionsmodells das Variantenmodell angepasst wird, sinkt der Aufwand für die Synchronhaltung. Werden außerdem die wichtigsten Änderungen am Variantenmodell immer auf die gleiche Art und Weise durchgeführt, ist zu erwarten, dass die Modellstruktur lesbarer und verständlicher wird. Aus diesem Grund wird hier für jeden beschriebenen Evolutionsschritt des Funktionsmodells vorgestellt, wie anschließend das Variantenmodell manipuliert werden muss, damit es wieder die aktuelle Variabilitätsstruktur der Implementierung abbildet. Dafür werden die Variantenmodellmanipulationen als Hintereinanderausführung elementarer Änderungen dargestellt. Die folgenden Modifikationen stehen dabei für die jeweiligen Teilmodelle zur Verfügung:

- Änderungen im Merkmalmodell:
  - ein Merkmal zu M hinzufügen

  - ein Merkmal aus M entfernen
  - für $x, y \in M$ einen Constraint $(x, y) \in H$ hinzufügen und damit den Variationstyp von x festlegen
  - eine Vater-Kind-Beziehung (x, y) aus H entfernen
  - für $x, y \in M$ einen Constraint $(x, y) \in C$ hinzufügen und damit eine requires- bzw. conflicts-Beziehung definieren
  - einen Constraint (x, y) aus C entfernen

- Änderungen im Konfigurationsmodell
  - einen neuen Variationspunkt zu KM hinzufügen
  - ein Variationspunkt aus KM entfernen
  - an einem Variationspunkt eine Variation erstellen
  - an einem Variationspunkt eine Variation löschen
  - an einer Variation (V_i, KF_i) die Bezeichnung V_i ändern
  - an einer Variation (V_i, KF_i) die Konfigurationsformel KF_i ändern

Alle folgenden Änderungsprozesse werden anhand eines Fahrerassistenzsystems illustriert. Das Beispiel zeigt die evolutionäre Entwicklung dieses Softwaresystems und die dadurch nötigen Adaptionen des Variantenmodells. Der initiale Stand des Funktionsmodells ist in Abb. 4.1 dargestellt, welche ein vereinfachtes Simulink / TargetLink-Modell mit drei funktionalen Komponenten zeigt. Es enthält neben der Komponente, die den Tempomaten implementiert, noch zwei weitere optionale Komponenten: Der Bremsassistent kann im Ernstfall selbst eine Notbremsung durchführen, um einen Auffahrunfall zu vermeiden bzw. die Unfallfolgen zu verringern. Dagegen gibt die Komponente der Abstandswarnung dem Fahrer ein akustisches Signal, falls der Abstand zum vorausfahrenden Fahrzeug zu gering wird.

Alle drei Komponenten sind optional und wurden daher mithilfe eines *Enabled Subsystem*-Blockes modelliert (s. auch Abschnitt 2.2). Sie können somit durch die jeweiligen Kontrollblöcke *VAR_Tempomat*, *VAR_Bremsassistent* und *VAR_Abstandswarner*, die sich auf die gleichnamigen Variationspunkte im Konfigurationsmodell beziehen, konfiguriert werden. Wie auch in den folgenden Beispielen von Funktionsmodellen ist der

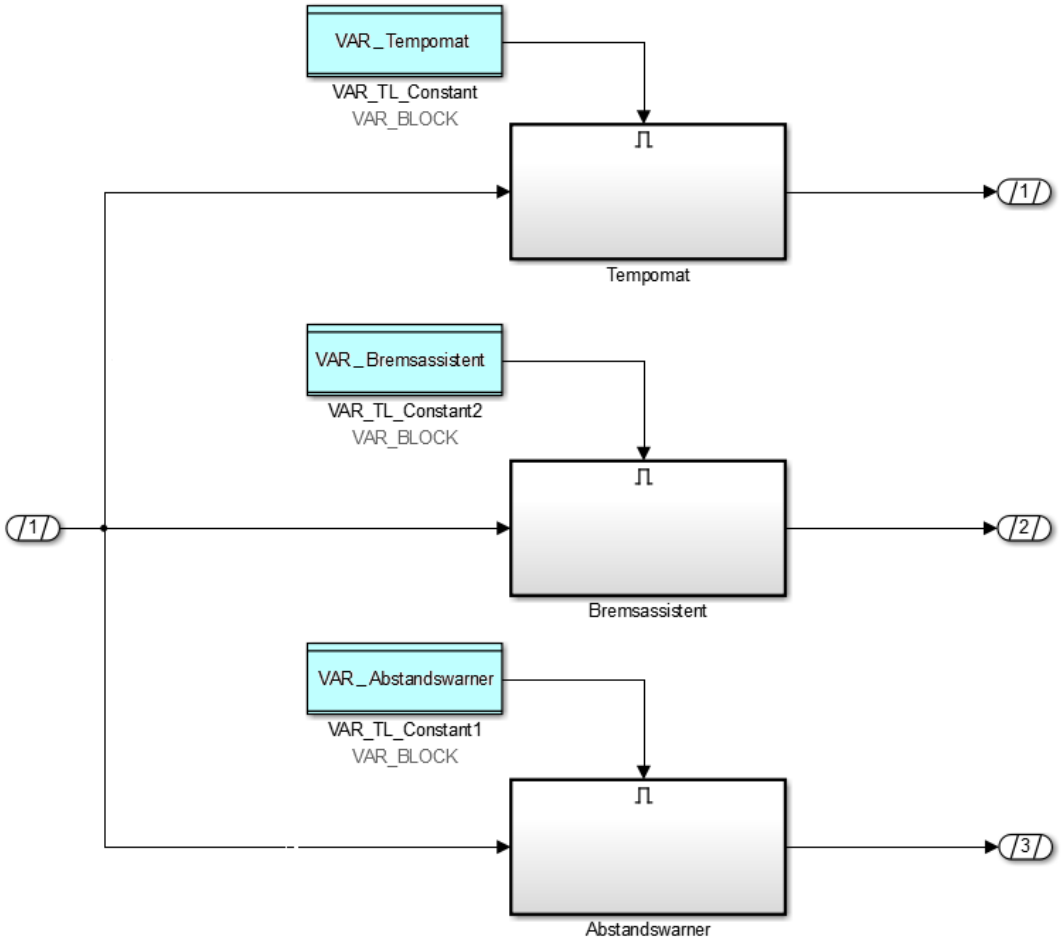
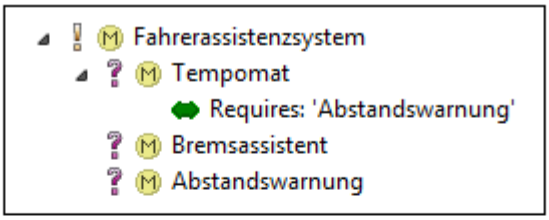


Abbildung 4.1.: Ein Funktionsmodell



Variation Point	Assignment
VAR_Abstandswarnung	
0 (Leermodul)	NOT(Abstandswarnung)
1 (Vollmodul)	Abstandswarnung
VAR_Bremsassistent	
0 (Leermodul)	NOT(Bremsassistent)
1 (Vollmodul)	Bremsassistent
VAR_Tempomat	
0 (Leermodul)	NOT(Tempomat)
1 (Vollmodul)	Tempomat

Abbildung 4.2.: Das Variantenmodell zu dem Funktionsmodell in Abb. 4.1

Variabilitätsmechanismus für die Variantenmodellierung des Systems zweitrangig. Wesentlich ist jedoch, ob die Komponenten optional, alternativ oder obligatorisch sind.

In Abb. 4.2 ist das zugehörige Variantenmodell abgebildet. Der obere Bildteil zeigt das Merkmalmodell mit je einem optionalen Merkmal für die drei oben genannten Komponenten. Wir gehen außerdem davon aus, dass aus Sicherheitsgründen der Tempomat nur in Verbindung mit der Abstandswarnung selektiert werden kann, was durch die *requires*-Beziehung ausgedrückt wird. In der unteren Bildhälfte ist das Konfigurationsmodell dargestellt, welches die drei Variationspunkte des Funktionsmodells zusammen mit deren Variationen und Konfigurationsformeln dokumentiert[1].

Dieses Variantenmodell ist zusammen mit dem zugehörigen Funktionsmodell der Ausgangspunkt des Beispielsystems, an welchem die in diesem Kapitel beschriebenen Änderungsprozesse veranschaulicht werden.

[1] Wir werden im gesamten Kapitel die Variantenmodelle auf diese Weise darstellen: die Baumstruktur des Merkmalmodells im oberen Bildteil und die Darstellung der Variationspunkte im Konfigurationsmodell in der unteren Bildhälfte.

4.1.1. Neue optionale Komponente

Beschreibung

In einem bestehenden System wird eine neu implementierte Komponente hinzugefügt. Es soll möglich sein, diese Komponente nur in bestimmte Software-Pakete einzubinden. Dafür wird ein Variabilitätsmechanismus implementiert und im System entsteht ein neuer Variationspunkt. Diese neue Variabilität muss im Variantenmodell durch neue Konfigurationsformeln für den entstandenen Variationspunkt und ein neues Merkmal abgebildet werden.

Beispiel

Der Abstandsregeltempomat stellt ein Beispiel für eine optionale Komponente dar. In Abb. 4.3 ist das weiterentwickelte Funktionsmodell dargestellt, welches nun die neue optionale Komponente für diese Funktionalität beinhaltet. Da der Abstandsregeltempomat eine Erweiterung des herkömmlichen Tempomaten darstellt, findet sich dessen Implementierung in der Komponente des Tempomaten wieder. Wie auch bei den optionalen Komponenten in der vorherigen Version des Funktionsmodells besteht die Möglichkeit, die Komponente des Abstandsregeltempomaten zu aktivieren bzw. zu deaktivieren, da diese Funktionalität nicht Bestandteil jedes Fahrerassistenzpaketes sein soll. Der Kontrollblock VAR_ART enthält den hierfür zuständigen Konfigurationsparameter.

In diesem Fall sieht die Anpassung des Merkmalmodells vor, dass ein neues Merkmal eingefügt wird. Wir modellieren es als Kind des bereits bestehenden Merkmals des Tempomaten, da der Abstandsregeltempomat die Funktionalität des herkömmlichen Tempomaten erweitert. Damit ist sichergestellt, dass mit der Auswahl des Abstandsregeltempomaten auch der herkömmliche Tempomat ausgewählt wird. Im Konfigurationsmodell muss ein neuer Variationspunkt angelegt werden, welcher die Auswahl der neuen Funktion abbildet (s. Abb. 4.4).

Modellierung

Sei $VM = (\mathcal{M}, KM)$ ein Variantenmodell mit den Merkmalen $M = \{m_1, \ldots, m_n\}$ und dem Konfigurationsmodell $KM = \{VP_1, \ldots, VP_N\}$. Ohne Beschränkung der Allgemeingültigkeit wollen wir das neue Merkmal m_{n+1} als Kind des bestehenden Merkmals $m_n \in M$ hinzufügen. Dann sind folgende Schritte nötig, um die Konsistenz zwischen Funktions- und Variantenmodell wiederherzustellen:

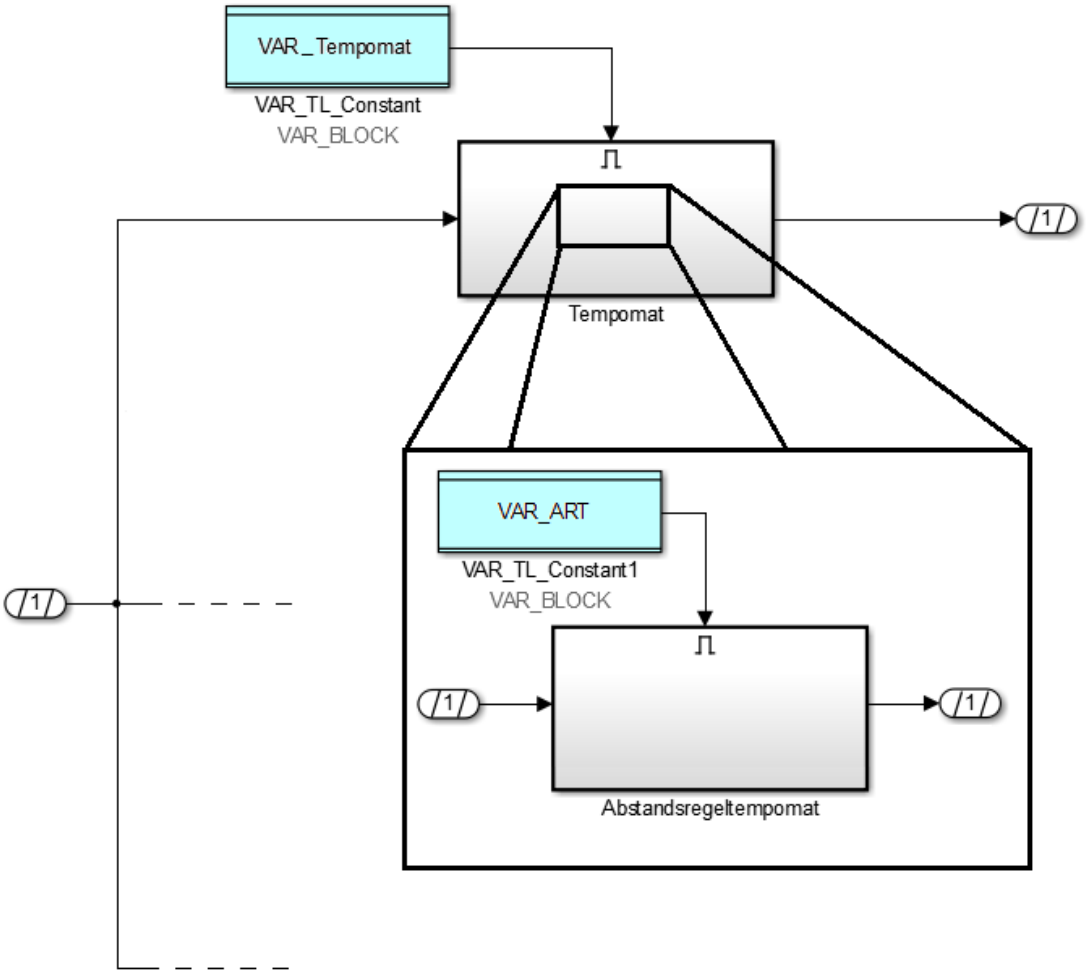
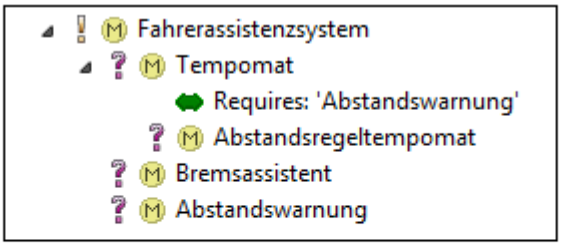


Abbildung 4.3.: Das Funktionsmodell nach Änderungsprozess 4.1.1



Variation Point	Assignment
VAR_Abstandsregeltempomat	
0 (Leermodul)	NOT(Abstandsregeltempomat)
1 (Vollmodul)	Abstandsregeltempomat
VAR_Abstandswarnung	
0 (Leermodul)	NOT(Abstandswarnung)
1 (Vollmodul)	Abstandswarnung
VAR_Bremsassistent	
0 (Leermodul)	NOT(Bremsassistent)
1 (Vollmodul)	Bremsassistent
VAR_Tempomat	
0 (Leermodul)	NOT(Tempomat)
1 (Vollmodul)	Tempomat

Abbildung 4.4.: Das Variantenmodell zu dem Funktionsmodell in Abb. 4.3

1. Das neue Merkmal m_{n+1} zu M hinzufügen, welches die neue optionale Komponente repräsentiert

2. Die Vater-Kind-Beziehung (m_n, m_{n+1}) zu $opt \subset H$ hinzufügen

3. Ggf. Constraints im Zusammenhang mit anderen Merkmalen definieren

4. Einen neuen Variationspunkt $VP_{N+1} = ((V_0, KF_0), (V_1, KF_1))$ zum Konfigurationsmodell KM hinzufügen

5. Setze $V_0 =$ *Leermodul* und $V_1 =$ *Vollmodul*

6. Setze $KF_0 = \text{NOT}(m_{n+1})$ und $KF_1 = m_{n+1}$

4.1.2. Optionale Komponente löschen

Beschreibung

Im System existiert eine optionale Komponente, welche einen Variationspunkt darstellt. Da die Funktionalität dieser Komponente ab sofort nicht mehr benötigt wird, entfernt man sie aus dem Funktionsmodell, wodurch auch der Variationspunkt entfällt. Im Variantenmodell müssen daher die betroffenen Variationen, deren Konfigurationsformeln und das entsprechende Merkmal gelöscht werden.

Beispiel

Das Fahrerassistenzsystem beinhaltet unter anderem die Funktionalität der Abstandswarnung, welche dem Fahrer eine Warnung gibt, sobald der Abstand zum vorausfahrenden Fahrzeug zu gering wird. Mittlerweile wurde die Komponente des Bremsassistenten weiterentwickelt, sodass sie auch die Funktionalität der Abstandswarnung übernimmt. Dementsprechend ist die Komponente für die Abstandswarnung obsolet geworden und wird gemeinsam mit ihrem Variabilitätsmechanismus gelöscht. Das Funktionsmodell nach dieser Änderung zeigt Abb. 4.5.

Bevor wir das zu dieser Funktion gehörende Merkmal aus dem Merkmalmodell löschen können, müssen wir den bestehenden Constraint *Tempomat* $\Rightarrow$ *Abstandswarnung* entweder umformulieren oder ebenfalls löschen. Wir haben in diesem Fall den Constraint umformuliert (s. Abb. 4.6), da die Abstandswarnfunktion in unserem Beispielmodell ab sofort Teil der Komponente des Bremsassistenten ist.

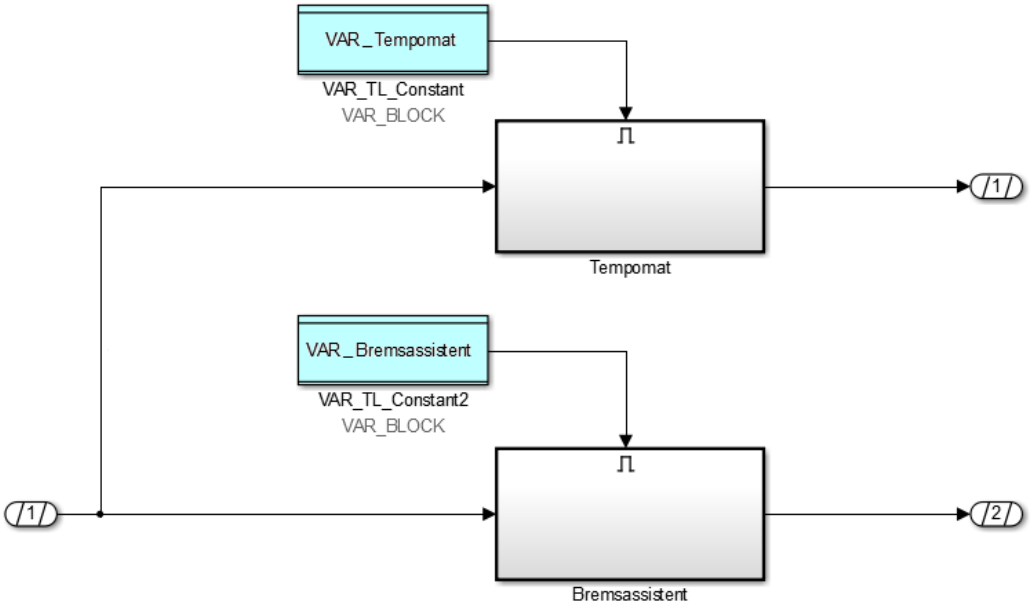
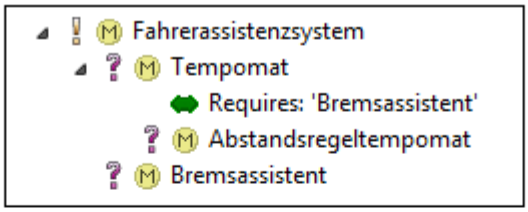


Abbildung 4.5.: Das Funktionsmodell nach Änderungsprozess 4.1.2



Variation Point	Assignment
VAR_Abstandsregeltempomat	
0 (Leermodul)	NOT(Abstandsregeltempomat)
1 (Vollmodul)	Abstandsregeltempomat
VAR_Bremsassistent	
0 (Leermodul)	NOT(Bremsassistent)
1 (Vollmodul)	Bremsassistent
VAR_Tempomat	
0 (Leermodul)	NOT(Tempomat)
1 (Vollmodul)	Tempomat

Abbildung 4.6.: Das Variantenmodell für das Funktionsmodell in Abb. 4.5

Modellierung

Sei $VM = (\mathcal{M}, KM)$ ein Variantenmodell mit den Merkmalen $M = \{m_1, \ldots, m_n\}$ und dem Konfigurationsmodell $KM = \{VP_1, \ldots, VP_N\}$. Ohne Beschränkung der Allgemeingültigkeit gehen wir davon aus, dass das bestehende Merkmal $m_n \in M$ die optionale Komponente repräsentiert, welche gelöscht wurde. Außerdem sei $m_{n-1} \in M$ dessen Vatermerkmal. Dann sind die folgenden Schritte nötig, um die Konsistenz zwischen Funktions- und Variantenmodell wiederherzustellen:

1. Im Konfigurationsmodell den Variationspunkt
   $VP = ((Leermodul, NOT(m_n)), (Vollmodul, m_n))$ löschen, falls vorhanden

2. Jeden Constraint $c_i \in C$ mit $m_n \in V(c_i)$ entsprechend anpassen (s. dazu Abschnitt 4.2.1.1)

3. Alle verbliebenen Variationspunkte nach Konfigurationsformeln KF_i mit $m_n \in V(KF_i)$ durchsuchen und diese anpassen (s. dazu Abschnitt 4.2.1.2)

4. Die Vater-Kind-Beziehung (m_{n-1}, m_n) aus $opt \subset H$ entfernen

5. Das Merkmal m_n aus der Merkmalmenge M entfernen

4.1.3. Optionale Komponente wird obligatorisch

Beschreibung

Im System befindet sich eine optionale Komponente, welche einen Variationspunkt darstellt. Ab sofort soll deren Funktionalität in jedes Software-Paket eingebunden werden. Die Komponente wird damit obligatorisch und der Variationspunkt entfällt. Dadurch müssen dessen Variationen und Konfigurationsformeln im Konfigurationsmodell entfernt und das Merkmalmodell angepasst werden. Evolutionsmuster dieser Art treten auch in anderen Domänen auf, z. B. im Linux Kernel [PCW12].

Beispiel

Die Komponente, welche die Funktionalität des Bremsassistenten implementiert, war bisher optional. Wir nehmen nun an, dass aus Sicherheitsgründen ab sofort jedes Fahrzeug serienmäßig mit dem Bremsassistenten ausgestattet wird. Dadurch ist diese Komponen-

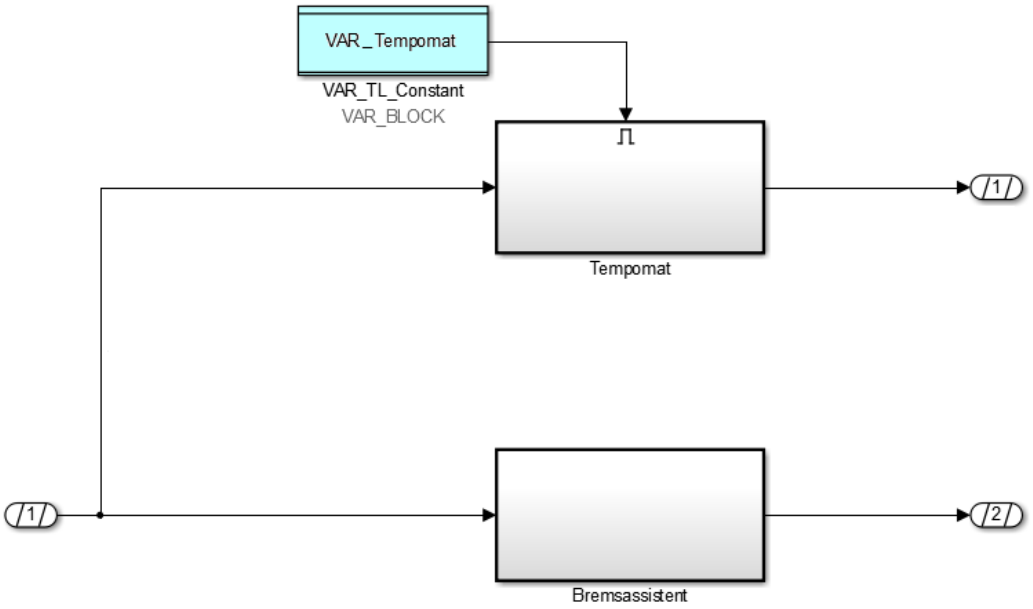


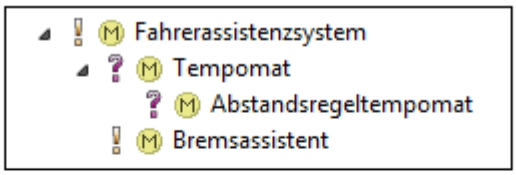
Abbildung 4.7.: Das Funktionsmodell nach Änderungsprozess 4.1.3

te obligatorisch und von jetzt an in jedem Fahrerassistenzsystem-Paket enthalten. Der Variabilitätsmechanismus wird aus dem Funktionsmodell entfernt (s. Abb. 4.7).

Das entsprechende Merkmal im Merkmalmodell wird somit echt obligatorisch, wodurch der bestehende Constraint *Tempomat* ⇒ *Bremsassistent* analysiert und angepasst werden muss. In diesem Fall ist der Constraint nach der Änderung redundant, da der Bremsassistent immer gewählt wird und die Bedingung damit stets erfüllt ist. Da es sich hier um keine technische Abhängigkeit zwischen Komponenten handelt, verbleibt der Constraint nicht aus Dokumentationszwecken im Merkmalmodell, sondern wird gelöscht (s. Abb. 4.8). Wäre das Merkmal des Bremsassistenten nach der Änderung nur bedingt obligatorisch, so würde der Constraint im Model verbleiben. Es wäre lediglich zu prüfen, ob es sinnvoller ist, ihn nicht auf den Bremsassistent, sondern auf dessen variablen Vater zu richten, da dieser für das Vorhandensein der Funktion in einer Selektion maßgebend ist (s. dazu auch Abschnitt 4.2.2.1).

Modellierung

Sei $VM = (\mathcal{M}, KM)$ ein Variantenmodell mit den Merkmalen $M = \{m_1, \ldots, m_n\}$ und dem Konfigurationsmodell $KM = \{VP_1, \ldots, VP_N\}$. Ohne Beschränkung der Allgemeingültigkeit gehen wir davon aus, dass das bestehende Merkmal $m_n \in M$ die optiona-



Variation Point	Assignment
VAR_Abstandsregeltempomat	
0 (Leermodul)	NOT(Abstandsregeltempomat)
1 (Vollmodul)	Abstandsregeltempomat
VAR_Tempomat	
0 (Leermodul)	NOT(Tempomat)
1 (Vollmodul)	Tempomat

Abbildung 4.8.: Das Variantenmodell für das Funktionsmodell in Abb. 4.7

le Komponente repräsentierte, welche obligatorisch wurde. Außerdem sei $m_{n-1} \in M$ dessen Vatermerkmal. Dann sind folgende Schritte nötig, um die Konsistenz zwischen Funktions- und Variantenmodell wiederherzustellen:

1. Im Konfigurationsmodell den Variationspunkt $VP = ((Leermodul, NOT(m_n)), (Vollmodul, m_n))$ löschen, falls vorhanden

2. Jeden Constraint $c_i \in C$ mit $m_n \in V(c_i)$ entsprechend anpassen (s. dazu Abschnitt 4.2.2.1)

3. Alle verbliebenen Variationspunkte nach Konfigurationsformeln KF_i mit $m_n \in V(KF_i)$ durchsuchen und diese anpassen (s. dazu Abschnitt 4.2.2.2)

4. Die Vater-Kind-Beziehung (m_{n-1}, m_n) aus $opt \subset H$ entfernen und entweder in $obl \subset H$ hinzufügen, falls das Merkmal erhalten bleibt, sonst das Merkmal m_n aus der Merkmalmenge M entfernen (s. dazu Abschnitt 4.2.2)

4.1.4. Neue alternative Komponente

Beschreibung

In diesem Fall besteht eine Komponente im System, zu der eine funktionale Alternative implementiert wird. Es lassen sich dabei drei Unterfälle unterscheiden, abhängig von der

Art der bereits bestehenden Komponente.

- Fall 1: War die bestehende Komponente obligatorisch, entsteht ein neuer alternativer Variationspunkt, um entweder die bestehende oder die neue Komponente in das System integrieren zu können.

- Fall 2: War die bestehende Komponente optional, so existiert bereits ein optionaler Variationspunkt. Dieser wird um einen alternativen Variationspunkt erweitert, sodass sich eine Verschachtelung ergibt. Der neue alternative Variationspunkt liegt im Funktionsmodell somit innerhalb des optionalen Variationspunktes. Wird der optionale Variationspunkt aktiviert, hat man die Wahl zwischen den beiden alternativen Komponenten. Bleibt er deaktiviert, so ist der innere alternative Variationspunkt nicht Teil des Funktionsmodells.

- Fall 3: Die bestehende Komponente war bereits Teil einer Gruppe alternativer Komponenten. In diesem Fall existieren wiederum zwei verschiedene Situationen. Entweder die neue Komponente stellt eine Alternative zu allen bereits bestehenden Komponenten dar, oder sie ist lediglich eine Alternative für eine der bestehenden Komponenten. Im ersten Fall erhält der existierende Variationspunkt eine weitere funktionale Alternative (Situation 1). Im zweiten Fall entsteht in einer der alternativen Komponenten ein neuer alternativer Variationspunkt (Situation 2).

In allen Fällen ist eine Anpassung des Merkmalmodells nötig. Im Konfigurationsmodell müssen die neu entstandenen Variationspunkte hinzugefügt bzw. die bereits bestehenden erweitert werden.

Beispiel

Bisher gab es mit dem Tempomathebel nur eine Möglichkeit, den Tempomat zu steuern. Mit der Steuerung über Lenkradtasten entsteht nun eine Alternative zum Tempomathebel, sodass innerhalb des Tempomaten eine neue Komponente implementiert wird, welche die Signale der Lenkradtasten verarbeiten kann. Zudem entsteht ein alternativer Variationspunkt, der die Wahlmöglichkeit zwischen Lenkradtasten und dem Tempomathebel abbildet. Der Konfigurationsparameter *VAR_Bedienkonzept* (s. Abb. 4.9) lässt die Auswahl zwischen den beiden alternativen Bedienkonzepten zu.

Im Merkmalmodell erstellen wir das neue strukturierende Merkmal *Bedieneinheit*, um die Bedienung von der restlichen Funktionalität des Tempomaten abzugrenzen[2]. Dieses neue Merkmal erhält außerdem zwei Kindmerkmale, welche die beiden alternativen

[2]In der Literatur werden diese Merkmale, die keinen direkten Einfluss auf die Konfiguration haben, auch als „abstract features“ bezeichnet [TKES11].

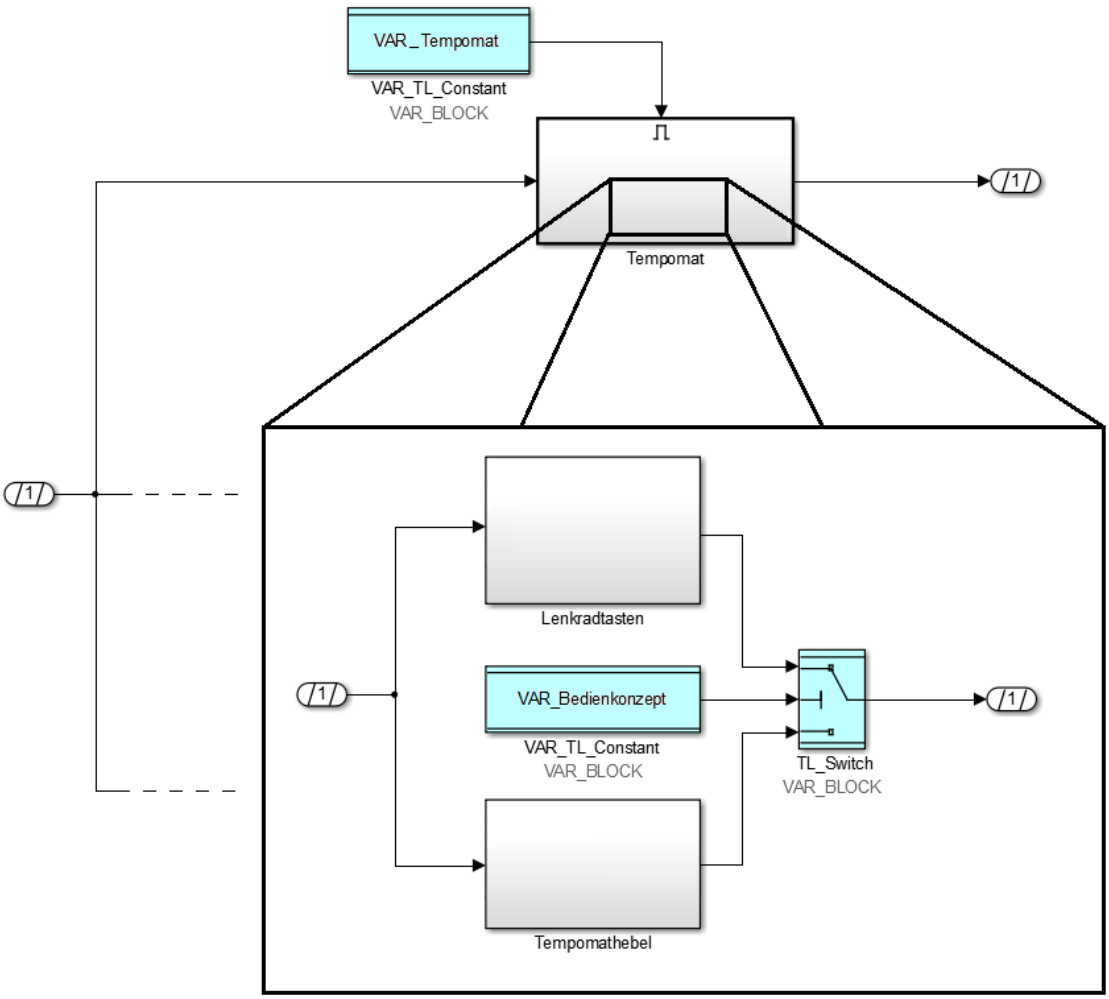
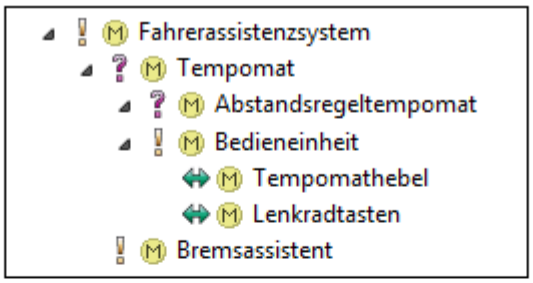


Abbildung 4.9.: Das Funktionsmodell nach Änderungsprozess 4.1.4, Fall 1



Variation Point	Assignment
VAR_Abstandsregeltempomat	
0 (Leermodul)	NOT(Abstandsregeltempomat)
1 (Vollmodul)	Abstandsregeltempomat
VAR_Bedienkonzept	
0 (not used)	NOT(Tempomat)
1 (Lenkradtasten)	Lenkradtasten
2 (Tempomathebel)	Tempomathebel
VAR_Tempomat	
0 (Leermodul)	NOT(Tempomat)
1 (Vollmodul)	Tempomat

Abbildung 4.10.: Das Variantenmodell zu dem Funktionsmodell in Abb. 4.9

Funktionalitäten der Bedienung repräsentieren. Im Konfigurationsmodell wird der neue Variationspunkt mit seinen Variationen und Konfigurationsformeln abgebildet (s. Abb. 4.10). Für die Modellierung ist dabei folgendes zu beachten: Der neue Variationspunkt *VAR_Bedienkonzept* erhält nicht nur die beiden Variationen für die neuen Alternativen Lenkradtasten und Tempomathebel, sondern noch eine dritte Variation. Diese steht für die Variante, in der kein Tempomat ausgewählt wurde und hat dementsprechend die Konfigurationsformel *NOT(Tempomat)*. Da in dieser Situation der neue Variationspunkt für die Entscheidung des Bedienkonzeptes nicht gebraucht wird, ist in der Praxis für diese Variation die Bezeichnung *not used* entstanden.

Im Laufe der Entwicklung wird der Tempomathebel dahingehend verbessert, dass der Fahrer die gewünschte Geschwindigkeit in kleinen (1 km/h) und großen (10 km/h) Schritten definieren kann. Der neue 2-stufige Tempomathebel ist die dritte Bedienungsalternative und damit eine neue Alternative zu einer bestehenden alternativen Komponente. Dadurch entsteht in der Komponente des Tempomathebels ein neuer Variationspunkt, wie es in Abb. 4.11 dargestellt ist. Mithilfe des neuen Konfigurationsparameters *VAR_Tempomathebel* kann nun zwischen den alternativen Komponenten 1-stufiger Tempomathebel und 2-stufiger Tempomathebel ausgewählt werden.

Im Merkmalmodell fügen wir unter dem Merkmal *Tempomathebel* zwei neue alternative Merkmale ein, um deutlich zu machen, dass von dieser Funktionalität ab sofort zwei Alternativen bestehen. Im Konfigurationsmodell bildet der neue Variationspunkt *VAR_Tempomathebel* die veränderte Situation ab. Auch hier benötigen wir wieder eine *not used*-Variation für die Varianten ohne Tempomathebel. Das Variantenmodell nach dieser Änderung ist in Abb. 4.12 dargestellt.

Modellierung

Im Merkmalmodell existieren grundsätzlich zwei verschiedene Möglichkeiten, neue Alternativen einer bestehenden Komponente abzubilden. Es werden in jedem Fall neue Merkmale für die neuen Alternativen definiert. Diese kann man jedoch im Merkmalbaum entweder unter dem Merkmal für die bestehende Komponente aufhängen oder dieses durch die neuen ersetzen. Im Allgemeinen ist vom Ersetzen abzuraten, da mit dieser Modellierung unter Umständen eine Gruppe alternativer Merkmale auf einer Hierarchieebene erstellt wird, auf der bereits eine solche Gruppe existiert. Es würden sich dadurch beide Alternativmerkmalgruppen vereinen, was nicht dem Ziel der Modellierung entspricht. Wie es auch im Beispiel der Bedieneinheit des Tempomaten modelliert wurde (s. Abb. 4.10), sollten die neuen alternativen Merkmale unter dem Merkmal der bestehenden Komponente eingefügt und ggf. durch ein strukturierendes Merkmal ergänzt werden. Lediglich wenn das Merkmal der bestehenden Komponente im Merkmalbaum ohnehin keine Geschwister hat, kann es sinnvoll sein, es durch die beiden neuen Merkmale zu ersetzen, um nicht unnötig viele Hierarchieebenen zu erzeugen. Wir gehen aber

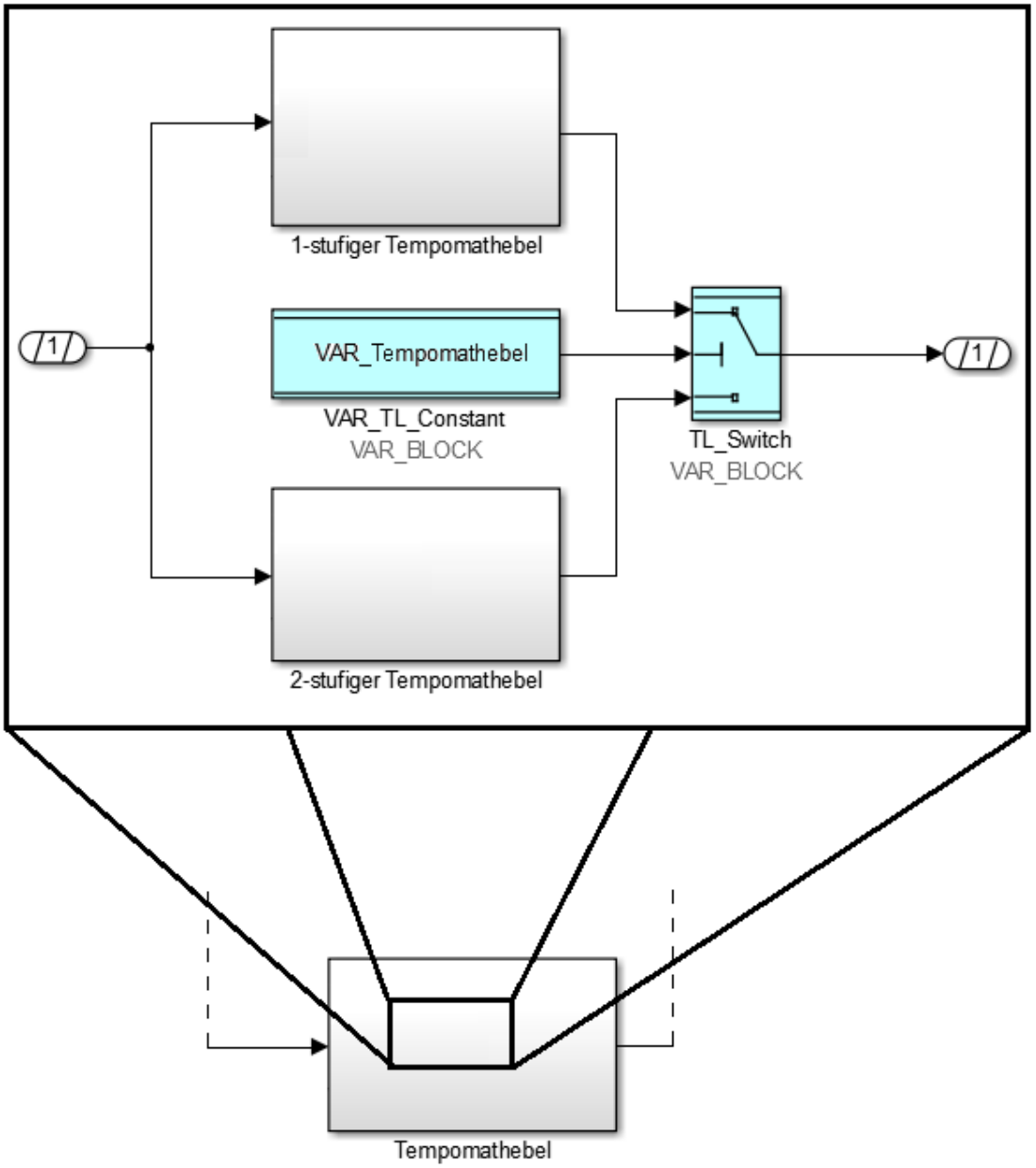
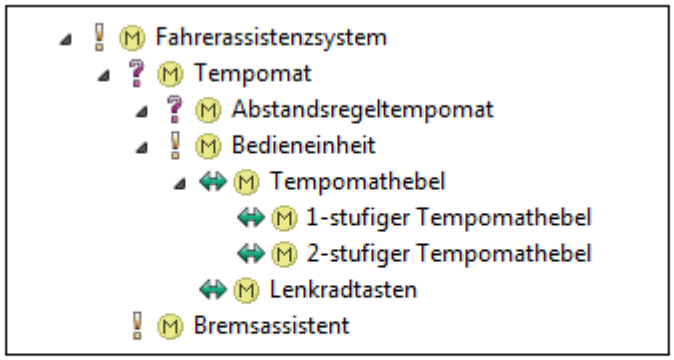


Abbildung 4.11.: Das Funktionsmodell nach Änderungsprozess 4.1.4, Fall 3



Variation Point	Assignment
VAR_Abstandsregeltempomat	
0 (Leermodul)	NOT(Abstandsregeltempomat)
1 (Vollmodul)	Abstandsregeltempomat
VAR_Bedienkonzept	
0 (not used)	NOT(Tempomat)
1 (Lenkradtasten)	Lenkradtasten
2 (Tempomathebel)	Tempomathebel
VAR_Tempomat	
0 (Leermodul)	NOT(Tempomat)
1 (Vollmodul)	Tempomat
VAR_Tempomathebel	
0 (not used)	NOT(Tempomathebel)
1 (1-stufiger Tempomathebel)	1-stufiger Tempomathebel
2 (2-stufiger Tempomathebel)	2-stufiger Tempomathebel

Abbildung 4.12.: Das Variantenmodell zu dem Funktionsmodell in Abb. 4.11

bei der folgenden Beschreibung der Modellierung von dem Fall aus, dass das bestehende Merkmal nicht ersetzt wird.

Gehen wir nun zur Modellierung der drei Fälle dieses Änderungsprozesses über, wie sie in der Beschreibung dargestellt wurden. Die Fälle unterschieden sich danach, ob die bereits bestehende Komponente obligatorisch, optional oder selbst alternativ war. Da dies auch Einfluss auf die Modellierung hat, wird in der folgenden Beschreibung auf jeden der drei Fälle einzeln eingegangen.

1. Fall: Die bestehende Komponente war obligatorisch:
   Sei $VM = (\mathcal{M}, KM)$ ein Variantenmodell mit den Merkmalen $M = \{m_1, \ldots, m_n\}$ und dem Konfigurationsmodell $KM = \{VP_1, \ldots, VP_N\}$. Wir gehen davon aus, dass die obligatorische Komponente, welche eine neue Alternative erhält, nicht durch ein Merkmal repräsentiert ist. Sie befindet sich jedoch innerhalb einer Komponente, die ohne Beschränkung der Allgemeingültigkeit durch das Merkmal $m_n \in M$ abgebildet wird. Dann sind folgende Schritte nötig, um die Konsistenz zwischen Funktions- und Variantenmodell wiederherzustellen:

   a) Das strukturierende obligatorische Merkmal m_{n+1} zu M hinzufügen[3]

   b) Die Vater-Kind-Beziehung (m_n, m_{n+1}) zu $obl \subset H$ hinzufügen

   c) Die beiden neuen Merkmale m_{n+2} und m_{n+3} zu M hinzufügen, welche die beiden Alternativen repräsentieren

   d) Die Vater-Kind-Beziehungen (m_{n+1}, m_{n+2}) und (m_{n+1}, m_{n+3}) zu $alt \subset H$ hinzufügen

   e) Ggf. Constraints im Zusammenhang mit anderen Merkmalen definieren

   f) Den neuen Variationspunkt $VP_{N+1} = ((V_0, KF_0), (V_1, KF_1), (V_2, KF_2))$ zum Konfigurationsmodell KM hinzufügen

   g) Setze $V_0 =$ *not used*, $V_1 =$ *Alternative 1* und $V_2 =$ *Alternative 2*

   h) Setze $KF_0 = \text{NOT}(m_n)$, $KF_1 = m_{n+2}$ und $KF_2 = m_{n+3}$

2. Fall: Die bestehende Komponente war optional:
   Sei $VM = (\mathcal{M}, KM)$ ein Variantenmodell mit den Merkmalen $M = \{m_1, \ldots, m_n\}$ und dem Konfigurationsmodell $KM = \{VP_1, \ldots, VP_N\}$. Wir gehen davon aus, dass die optionale Komponente, welche eine neue Alternative erhält, ohne Be-

[3] In dem obigen Beispiel der Fahrerassistenzsysteme ist dies das Merkmal *Bedieneinheit*.

schränkung der Allgemeingültigkeit durch das Merkmal $m_n \in M$ repräsentiert wird. Dann sind folgende Schritte nötig, um die Konsistenz zwischen Funktions- und Variantenmodell wiederherzustellen:

a) Die beiden neuen Merkmale m_{n+1} und m_{n+2} zu M hinzufügen, welche die beiden Alternativen repräsentieren

b) Die Vater-Kind-Beziehungen (m_n, m_{n+1}) und (m_n, m_{n+2}) zu $alt \subset H$ hinzufügen

c) Jeder Constraint $c_i \in C$ mit $m_n \in V(c_i)$ muss überprüft werden. Die Constraints, die inhaltlich für *beide* der neuen Alternativen relevant sind, können weiterhin auf das optionale Merkmal verweisen. Ist ein Constraint nur für eines der beiden neuen Alternativen relevant, so muss in dessen Formel m_n durch das entsprechende Merkmal ersetzt werden.

d) Ggf. Constraints im Zusammenhang mit anderen Merkmalen definieren

e) Den neuen Variationspunkt $VP_{N+1} = ((V_0, KF_0), (V_1, KF_1), (V_2, KF_2))$ zum Konfigurationsmodell KM hinzufügen

f) Setze $V_0 =$ *not used*, $V_1 =$ *Alternative 1* und $V_2 =$ *Alternative 2*

g) Setze $KF_0 = \text{NOT}(m_n)$, $KF_1 = m_{n+1}$ und $KF_2 = m_{n+2}$

3. Fall: Die bestehende Komponente war alternativ:
Bereits in der Beschreibung wurden hier zwei verschiedene Situationen unterschieden. Wir gehen zunächst auf die Schrittfolge ein, die nötig ist, falls für den bestehenden alternativen Variationspunkt eine neue alternative Komponente implementiert wird (Situation 1 in der Beschreibung). Sei $VM = (\mathcal{M}, KM)$ ein Variantenmodell mit den Merkmalen $M = \{m_1, \ldots, m_n\}$ und dem Konfigurationsmodell $KM = \{VP_1, \ldots, VP_N\}$. Wir gehen davon aus, dass sich die bereits bestehenden alternativen Komponenten innerhalb einer Komponente befinden, welche ohne Beschränkung der Allgemeingültigkeit durch das Merkmal $m_n \in M$ repräsentiert wird. Sei außerdem $VP_N = ((V_0, KF_0), \ldots, (V_p, KF_p)) \in KM$ der Variationspunkt, über den die alternativen Komponenten bisher konfiguriert wurden.

a) Das neue alternative Merkmal m_{n+1} zu M hinzufügen, welches die neue alternative Komponente repräsentiert

b) Die Vater-Kind-Beziehungen (m_n, m_{n+1}) zu $alt \subset H$ hinzufügen

c) Ggf. Constraints im Zusammenhang mit anderen Merkmalen einfügen

d) Der Variationspunkt VP_N muss um die Variation (V_{p+1}, KF_{p+1}) mit $V_{p+1} =$ *Alternative p+1* bzw. $KF_{p+1} = m_{n+1}$ erweitert werden.

Stellt die neue Komponente lediglich eine Alternative für eine der bestehenden Komponenten dar (Situation 2 in der Beschreibung), sieht die Modellierung anders aus. Sei dazu $VM = (\mathcal{M}, KM)$ ein Variantenmodell mit den Merkmalen $M = \{m_1, \ldots, m_n\}$ und dem Konfigurationsmodell $KM = \{VP_1, \ldots, VP_N\}$. Wir gehen davon aus, dass die alternative Komponente, welche eine neue Alternative erhält, ohne Beschränkung der Allgemeingültigkeit durch das Merkmal $m_n \in M$ repräsentiert wird.

a) Die beiden neuen Merkmale m_{n+1} und m_{n+2} zu M hinzufügen, welche die beiden Alternativen repräsentieren

b) Die Vater-Kind-Beziehungen (m_n, m_{n+1}) und (m_n, m_{n+2}) zu $alt \subset H$ hinzufügen

c) Jeder Constraint $c_i \in C$ mit $m_n \in V(c_i)$ muss überprüft werden. Die Constraints, die inhaltlich für *beide* der neuen Alternativen relevant sind, können weiterhin auf das bestehende alternative Merkmal verweisen. Ist ein Constraint nur für eines der beiden neuen Alternativen relevant, so muss in dessen Formel m_n durch das entsprechende Merkmal ersetzt werden.

d) Ggf. Constraints im Zusammenhang mit anderen Merkmalen definieren

e) Den neuen Variationspunkt $VP_{N+1} = ((V_0, KF_0), (V_1, KF_1), (V_2, KF_2))$ zum Konfigurationsmodell KM hinzufügen

f) Setze $V_0 =$ *not used*, $V_1 =$ *Alternative 1* und $V_2 =$ *Alternative 2*

g) Setze $KF_0 = \mathrm{NOT}(m_n)$, $KF_1 = m_{n+1}$ und $KF_2 = m_{n+2}$

4.2. Generelle Hinweise zu Änderungen im Variantenmodell

Dieser Abschnitt geht auf Herausforderungen der Modellierung ein, die in verschiedenen Änderungsprozessen auftreten können und für die es schwer ist, allgemeingültige Vorgaben zu machen. Daher ist es notwendig, diese Fälle weiter zu untergliedern, sodass genauere Handlungsanweisungen für die Modellierung möglich sind.

4.2.1. Merkmal löschen

Wird ein Merkmal gelöscht, so müssen alle Constraints im Merkmalmodell und alle Konfigurationsformeln im Konfigurationsmodell angepasst werden, welche das Merkmal enthalten haben.

4.2.1.1. Constraints anpassen

In diesem Abschnitt wird folgende Situation betrachtet: Es existiert ein Merkmal $m_i \in M$, welches Teil eines Constraints $(m_i, m_j) \in C$ ist. Ab sofort wird m_i nicht mehr benötigt und soll im Zuge einer Änderung gelöscht werden. Die vorgeschlagene Modellierung hängt dabei von der Art des Constraints ab:

- Fall 1: Es handelt sich um eine *requires*-Beziehung. Gilt dabei $(m_i, m_j) \in req \subset M \times M$, ist m_i also das Merkmal, welches ein anderes benötigt, so ist der Constraint nach dem Löschen des Merkmals überflüssig.

  Zeigt die *requires*-Beziehung jedoch in die entgegengesetzte Richtung, gilt also $(m_j, m_i) \in req \subset M \times M$, so ist eine weitere Analyse nötig. Dies kann beispielsweise geschehen, falls die entsprechende Komponente nicht mehr variabel ist. In diesem Fall kann auch der Constraint entfernt werden, da die Bedingung, welche durch diesen modelliert wird, ohnehin nicht mehr verletzt werden kann. Falls andererseits m_i gelöscht wird, um es durch ein neues bzw. bestehendes Merkmal zu ersetzen, sollte der Constraint beibehalten und auf dieses Merkmal gerichtet werden[4].

- Fall 2: Es handelt sich um eine *conflicts*-Beziehung. Es gilt also

  $$m_i, m_j \in M \text{ mit } (m_i, m_j) \in con \subset M \times M$$

  Ziel des Constraints ist es, dass Selektionen mit m_i *und* m_j ungültig sind und somit nicht konfiguriert werden. Da dies nach dem Löschen von m_i nicht mehr möglich ist, wird der Constraint in diesem Fall überflüssig und daher auch aus dem Merkmalmodell entfernt. Ein Sonderfall tritt auf, falls m_i nur gelöscht wird, um es durch ein anderes Merkmal zu ersetzen. Dann muss abhängig vom Kontext entschieden werden, ob der Constraint noch gültig bzw. eine Anpassung nötig ist.

Bei diesen Betrachtungen sind wir davon ausgegangen, dass m_i nur in einem Constraint vertreten ist. Sind dagegen mehrere Constraints beteiligt, muss die Modellierung

[4] Im Beispiel des Änderungsprozesses 4.1.2 wird das Merkmal *Abstandswarnung* gelöscht, da die Funktionalität in eine andere Komponente integriert wurde, welche bereits durch ein bestehendes Merkmal abgebildet ist.

situationsabhängig angepasst werden. Existiert beispielsweise eine Kette von *requires*-Constraints $(m_k, m_i), (m_i, m_j) \in req \subset C$ mit $m_i, m_j, m_k \in M$ und m_i soll gelöscht werden, kann es sinnvoll sein, die bestehenden Constraints durch die neue Beziehung $(m_k, m_j) \in req$ zu ersetzen.

4.2.1.2. Konfigurationsformeln anpassen

Dieser Abschnitt befasst sich mit folgender Situation: Ein Merkmal, welches Teil einer Konfigurationsformel ist, wird nicht mehr benötigt und soll im Zuge einer Änderung gelöscht werden. Nehmen wir also an, dass $m_i \in M$ gelöscht werden soll. Es existiert jedoch mindestens eine Konfigurationsformel KF_j im Konfigurationsmodell mit $m_i \in V(KF_j)$.

Da KF_j eine aussagenlogische Formel ist, kann man das Literal m_i nicht einfach entfernen (etwa durch einen Automatismus), da sonst die Gefahr einer Inkonsistenz besteht. Die Formel muss so angepasst werden, dass sie einerseits syntaktisch korrekt bleibt und außerdem nach dem Löschen des Merkmals das Konfigurationswissen wieder korrekt abbildet. Wird das Merkmal m_i gelöscht, um durch ein neues ersetzt zu werden, kann es eine Alternative sein, die Ersetzung auch in der Konfigurationsformel durchzuführen. Liegt der Grund für das Löschen des Merkmals darin, dass die zugehörige Komponente zukünftig nicht mehr variabel ist, so kann die Konfigurationsformel konsistent angepasst werden, in dem m_i durch das Literal *true* ersetzt wird. Mithilfe von Gesetzen der booleschen Algebra (z. B. $m \vee true \rightarrow true$ oder $m \wedge true \rightarrow m$) könnten die Formeln weiter vereinfacht werden. Dies stellt jedoch allenfalls eine Übergangslösung dar, die aus Gründen der Lesbarkeit und Verständlichkeit der Konfigurationsformeln nochmal überarbeitet werden sollte. Auch aus diesem Grund wurde in Abschnitt 5.1.1 als schwache Konsistenzbedingung formuliert, dass keine obligatorischen Merkmale in Konfigurationsformeln auftreten.

4.2.2. Ein variables Merkmal wird obligatorisch

Wird das variable Merkmal $m_i \in M$ im Zuge einer Änderung obligatorisch, ist es unter Umständen sinnvoll, es zu löschen. Dazu zwei Fälle:

- Das Merkmal m_i hat im Merkmalbaum Kinder: In dieser Situation sollte das Merkmal nicht gelöscht werden. Es steht im Modell für eine Gruppe von Funktionalitäten (z. B. verschiedene Bremsassistenz-Funktionen) und dient unter anderem dazu, den Merkmalbaum verständlich zu strukturieren. Es kann beispielsweise obligatorisch werden, falls die gesamte Funktionsgruppe in Zukunft in jedem Fahrzeug

enthalten sein soll, wobei jedoch bestimmte Unterfunktionen immer noch variabel sind.

- Falls das Merkmal m_i keine Kinder hat, es also ein Blatt im Merkmalbaum ist, so repräsentiert es nach der Änderung üblicherweise eine Funktion, die zukünftig in jedes Fahrzeug einfließen soll und somit nicht mehr konfiguriert werden muss. Daher kann es gelöscht werden, um das Merkmalmodell möglichst schlank und damit verständlich zu halten. Das Merkmal sollte dagegen im Modell verbleiben, falls entweder die Wahrscheinlichkeit besteht, dass es in naher Zukunft wieder variabel wird oder die vollständige Dokumentation der Domäne bei der Modellierung hohe Priorität hat.

4.2.2.1. Constraints anpassen

Wird ein Merkmal obligatorisch, müssen alle beteiligten Constraints angepasst werden. Wir betrachten zunächst den Fall, dass das Merkmal nach der Änderung echt obligatorisch ist, also in jeder gültigen Selektion vorhanden sein muss. Die vorgeschlagene Modellierung hängt dabei von der Art des Constraints ab:

- Handelt es sich um eine requires-Beziehung, ist der Constraint nach der Änderung streng genommen überflüssig: Denn wird in $(m_i, m_j) \in req \subset C$ der Variationstyp des Merkmals m_i auf obligatorisch gesetzt, muss anschließend auch m_j in jeder gültigen Selektion enthalten sein. Ist dieser Umstand gewollt, kann es sinnvoller sein, m_j selbst auch als obligatorisches Merkmal zu modellieren und den Constraint zu löschen. Ist das Merkmal m_j bereits obligatorisch, so war der Constraint ohnehin überflüssig.

  Wird dagegen in dem Constraint $(m_i, m_j) \in req \subset C$ das Merkmal m_j obligatorisch, ist der Constraint ebenfalls überflüssig und kann gelöscht werden. Denn nach Definition gilt stets $m_i \Rightarrow m_j$, was jedoch unabhängig vom Wahrheitswert von m_i immer erfüllt ist, sobald das Merkmal m_j zu *true* evaluiert.

  Betrachtet man also nur die Logik des Modells, kann eine requires-Beziehung gelöscht werden, falls ein beteiligtes Merkmal obligatorisch wird. Der Vorteil dieser Vorgehensweise ist ein schlankeres und somit verständlicheres Merkmalmodell. In manchen Situationen ist es jedoch sinnvoll, den Constraint beizubehalten, damit das Wissen um die Beziehung dokumentiert bleibt. Diese Modellierung ist zu empfehlen, wenn die Wahrscheinlichkeit besteht, dass die Änderung zurück genommen wird und das entsprechende Merkmal anschließend wieder variabel ist.

- Ist andererseits in dem Constraint $(m_i, m_j) \in con \subset C$ ohne Beschränkung der

Allgemeingültigkeit das Merkmal m_i obligatorisch, folgt daraus, dass m_j in keiner gültigen Selektion enthalten sein darf, wodurch es zu einem *toten* Merkmal wird[5] (s. dazu auch Abschnitt 5.1.3). Somit hat entweder der Constraint durch die Änderung seine Gültigkeit verloren, kann gelöscht werden und m_j verbleibt als variables Merkmal im Merkmalmodell, oder m_j sollte bei der Änderung ohnehin gelöscht werden, wodurch auch der Constraint überflüssig wird.

Da requires-Beziehungen mit echt obligatorischen Merkmalen zumindest logisch redundant sind und durch conflicts-Beziehungen dabei sogar Inkonsistenzen entstehen können, wird in Abschnitt 5.1.2 als schwache Konsistenzbedingung definiert, dass keine obligatorischen Merkmale in Constraints auftreten.

Ist das Merkmal m_i nach der Änderung dagegen nur bedingt obligatorisch, sollten die Constraints im Modell verbleiben, da m_i nicht in jeder gültigen Selektion vorhanden sein muss. Unter Umständen ist es in dem Fall sinnvoll, den Constraint auf den variablen Vater von m_i zu richten, da dieser über die Wahl des bedingt obligatorischen Merkmals entscheidet. Dadurch könnten an dem Constraint die Merkmale abgelesen werden, welche für dessen Erfüllung relevant sind. Jedoch kann mit dieser Modellierung auch der eigentliche Hintergrund der Abhängigkeit verloren gehen.

4.2.2.2. Konfigurationsformeln anpassen

Auch die betroffenen Konfigurationsformeln müssen angepasst werden, falls ein Merkmal obligatorisch wird. Nehmen wir an, dass $m_i \in M$ obligatorisch werden soll und mindestens eine Konfigurationsformel KF_j mit $m_i \in V(KF_j)$ im Konfigurationsmodell existiert.

Da nach der Änderung das Merkmal m_i obligatorisch ist, wird es in der Konfigurationsformel stets zu *true* evaluiert. Es nimmt auf die Konfiguration keinen tatsächlichen Einfluss mehr und ist dadurch in der Formel redundant. Wie bereits in Abschnitt 4.2.1.2 beschrieben, könnte man dann die Formeln vereinfachen. Dieses Vorgehen kann jedoch nicht in jedem Fall eine manuelle Überarbeitung ersetzen.

Ist das Merkmal dagegen nach der Änderung nur bedingt obligatorisch, kann die Konfigurationsformel unverändert bleiben. Da die Wahl des Merkmals jedoch gänzlich von dessen variablem Vater abhängt, kann es sinnvoll sein, m_i in der Formel durch diesen zu ersetzen. Dieses Vorgehen würde zur Verständlichkeit der Modellierung beitragen, kann jedoch den eigentlichen Hintergrund der Abhängigkeit verwischen.

[5] Ein totes Merkmal ist ein Merkmal, welches aufgrund von Constraints in keiner gültigen Selektion ausgewählt werden darf [TBC06].

5. Analytisch - Die Konsistenzbedingungen

In diesem Kapitel werden die Konsistenzbedingungen beschrieben, wobei wir zwischen schwachen und starken Konsistenzbedingungen unterscheiden. Erstere können als Hinweise auf potentielle Fehlmodellierungen interpretiert werden. Eine Verletzung dieser Bedingungen stellt in der Regel keine Gefahr für den Konfigurationsprozess dar und kann daher bei gewissen Projektkontexten bzw. in bestimmten Projektphasen geduldet werden bzw. sogar notwendig sein. Die schwachen Konsistenzbedingungen werden in Abschnitt 5.1 beschrieben. Die starken Konsistenzbedingungen sichern dagegen den Konfigurationsprozess ab und müssen unter allen Umständen eingehalten werden. Eine Verletzung dieser Bedingungen birgt ein signifikantes Risiko, dass die Konfiguration fehlschlägt bzw. ohne Fehlermeldung entweder eine syntaktisch fehlerhafte oder semantisch falsch konfigurierte Systemvariante entsteht. Die starken Konsistenzbedingungen werden in Abschnitt 5.2 beschrieben.

Ein wesentliches Ziel der Konsistenzbedingungen ist die Absicherung des Änderungsprozesses des Variantenmodells. Nach dem eine Änderung am Modell durchgeführt wurde, kann mithilfe dieser Bedingungen ermittelt werden, ob Fehlmodellierungen oder ein inkonsistenter Modellzustand aufgetreten sind. Im Fokus stehen dabei vor allem die Konfigurationsformeln, da diese in der Regel händisch eingegeben werden, wodurch es zu Fehlern kommen kann. Durch die Verwendung der formalen Repräsentation des Variantenmodells können alle theoretisch möglichen (also gültigen) Selektionen geprüft werden. Die Analysen sind somit unabhängig davon, welche Konfigurationen bereits bestehen bzw. geplant sind. Auf die prototypische Implementierung der hier beschriebenen Konsistenzbedingungen wird im nächsten Kapitel eingegangen.

Alle Konsistenzbedingungen sind nach dem gleichen Schema beschrieben. Es werden jeweils die folgenden Rubriken diskutiert:

- **Bedingung**: Hier wird die Bedingung in einem Satz kurz formuliert.
- **Beschreibung**: Diese Rubrik geht darauf ein, welchen Hintergrund die Bedingung hat und wohin eine Verletzung führen kann. Zudem werden Konsistenzbedingungen zueinander in Beziehung gesetzt.
- **Definition**: In Kapitel 3 wurde das Variantenmodell formal beschrieben. Die Kon-

sistenzbedingung wird in dieser Rubrik vor dem Hintergrund dieser Formalisierung definiert.

- **Prüfung**: Hier wird aus der Definition abgeleitet, wie man die entsprechende Konsistenzbedingung für ein gegebenes Variantenmodell überprüfen kann. Diese Betrachtungen bilden auch die Grundlage für die Implementierung des Prototypen.

- **Gegenbeispiel**: In dieser Rubrik wird ein beispielhaftes Variantenmodell angegeben, welches die Konsistenzbedingung verletzt.

- **Komplexität**: Hier wird kurz auf die algorithmische Komplexität der Prüfung eingegangen.

5.1. Schwache Konsistenzbedingungen

5.1.1. Obligatorische Merkmale in Konfigurationsformeln

Bedingung

Konfigurationsformeln enthalten keine obligatorischen Merkmale.

Beschreibung

Ein echt obligatorisches Merkmal ist in jeder Variante ausgewählt, wodurch im Konfigurationsprozess dessen Bezeichner in der Konfigurationsformel stets als *true* ausgewertet wird. Dadurch hat das Merkmal keinen Einfluss auf die Konfiguration und muss somit auch nicht Teil der Konfigurationsformel sein. Mit dieser Konsistenzbedingung soll die Lesbarkeit und Verständlichkeit der Konfigurationsformeln erhöht werden, da nur noch für die Konfiguration relevante Merkmale darin auftreten.

Es ist in bestimmten Fällen sinnvoll, eine Konfigurationsformel mit einem echt obligatorischen Merkmal zu formulieren, etwa wenn bereits bekannt ist, dass das betroffene Merkmal in naher Zukunft variabel wird. Daher ist die Verletzung dieser Konsistenzbedingung weniger als Fehler des Variantenmodells zu sehen, sondern eher als Modellierung, die es wert ist, den Entwickler darauf hinzuweisen. Dieser kann anschließend selbst entscheiden, wie er mit der Situation umgehen möchte.

Ein bedingt obligatorisches Merkmal in einer Konfigurationsformel ist dagegen weniger

kritisch, da es nicht in jeder gültigen Selektion enthalten sein muss. Wie in Abschnitt 4.2.2.2 beschrieben, ist es unter Umständen sinnvoll, das bedingt obligatorische Merkmal in der Konfigurationsformel durch seinen variablen Vater zu ersetzen. Daher werden dem Entwickler bei der Prüfung dieser Konsistenzbedingung *alle* obligatorischen Merkmale in Konfigurationsformeln angegeben.

Definition

Sei $VM = (\mathcal{M}, KM)$ ein Variantenmodell. Sei außerdem $VP \in KM$ ein Variationspunkt mit $VP = ((V_1, KF_1), \ldots, (V_p, KF_p))$. Dann erfüllt VP die oben beschriebene Konsistenzbedingung, falls gilt:

$$V(KF_i) \cap \{y \in M : (x, y) \in obl\}^1 = \emptyset \text{ für alle } i \in \{1, \ldots, p\}.$$

Das Variantenmodell selbst ist im Sinne der obigen Bedingung konsistent, falls jeder Variationspunkt aus KM die Konsistenzbedingung erfüllt.

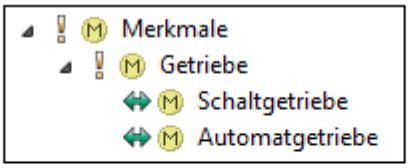
Prüfung

Um diese Konsistenzbedingung zu prüfen, müssen alle Konfigurationsformeln darauf hin untersucht werden, welchen Variationstyp die in den Formeln auftretenden Merkmale aufweisen.

Gegenbeispiel

Das Variantenmodell in Abb. 5.1 verletzt die beschriebene Konsistenzbedingung, da das obligatorische Merkmal *Getriebe* in beiden Konfigurationsformeln des Variationspunktes vertreten ist. Es könnte in diesem Fall auch weggelassen werden, ohne das Ergebnis der Konfiguration zu ändern.

[1] Die Menge $\{y \in M : (x, y) \in obl\}$ beschreibt in unserem Formalismus lediglich die Menge aller obligatorischen Merkmale.



Variation Point	Assignment
VAR_VP1	
0 (Schaltgetriebe)	Getriebe AND Schaltgetriebe
1 (Automatgetriebe)	Getriebe AND Automatgetriebe

Abbildung 5.1.: Verletzung der Konsistenzbedingung 5.1.1

Komplexität

Da lediglich die Konfigurationsformeln durchlaufen werden müssen, steigt der Aufwand linear mit deren Anzahl. Somit ist die Komplexität dieser Prüfung in Anbetracht der üblichen Größe unserer Variantenmodelle unbedenklich.

5.1.2. Obligatorische Merkmale in Constraints

Bedingung

Constraints enthalten keine obligatorischen Merkmale.

Beschreibung

Diese Konsistenzbedingung betrifft nur das Merkmalmodell. Sie soll den Entwickler dabei unterstützen, überflüssige Constraints zu finden. Ein echt obligatorisches Merkmal muss in jeder gültigen Selektion enthalten sein. Wenn die Constraints im Merkmalmodell ausgewertet werden, um etwa die Gültigkeit einer Selektion zu prüfen, wird der Bezeichner eines echt obligatorischen Merkmals stets zu *true* evaluiert. In Abschnitt 4.2.2.1 haben wir gesehen, dass echt obligatorische Merkmale in Constraints logisch redundant sind oder im Falle von *conflicts*-Beziehungen sogar zu Inkonsistenzen führen können. Aus diesem Grund wird dies hier als Inkonsistenz definiert.

In bestimmten Ausnahmefällen ist es jedoch sinnvoll, ein Constraint mit einem echt obligatorischen Merkmal zu formulieren. Beispielsweise wenn bereits bekannt ist, dass das betroffene Merkmal in naher Zukunft variabel wird. Daher ist die Verletzung dieser

Konsistenzbedingung weniger als Fehler des Variantenmodells zu sehen, sondern eher als Modellierung, auf die man den Entwickler hinweisen möchte. Dieser entscheidet anschließend selbst, wie er mit der Situation umgeht.

Die Modellprüfung soll den Entwickler zudem auch auf bedingt obligatorische Merkmale in Constraints hinweisen, da deren Existenz in einer Selektion und damit die Frage, ob der Constraint erfüllt ist, von einem anderen Merkmal abhängt. Daher kann eine Anpassung des Constraints die Verständlichkeit des Variantenmodells unterstützen (s. dazu auch Abschnitt 4.2.2.1).

Definition

Sei $VM = (\mathcal{M}, KM)$ ein Variantenmodell mit $\mathcal{M} = (M, H \cup C, r)$. Dann erfüllt VM die oben beschriebene Konsistenzbedingung, falls die Menge der obligatorischen Merkmale und die Menge aller Merkmale in Constraints keine gemeinsamen Elemente haben. Es muss also gelten:

$$\{y \in M : (x, y) \in obl\} \cap (\{x \in M : (x, y) \in C\} \cup \{y \in M : (x, y) \in C\}) = \emptyset$$

Prüfung

Um diese Konsistenzbedingung zu prüfen, müssen alle Elemente $(x, y) \in C$ darauf hin untersucht werden, welchen Variationstyp die in den Constraints auftretenden Merkmale aufweisen.

Gegenbeispiel

Das Variantenmodell in Abb. 5.2 verletzt die beschriebene Konsistenzbedingung, da von dem Merkmal *Automatgetriebe* eine *requires*-Beziehung hin zum Merkmal *Getriebe* definiert wurde. Der Constraint könnte in diesem Fall auch weggelassen werden, ohne dadurch die Semantik des Merkmalmodells zu ändern.

Komplexität

Da lediglich die Constraints des Merkmalmodells durchlaufen werden müssen, steigt der Aufwand linear mit deren Anzahl. Somit ist die Komplexität dieser Prüfung in Anbetracht der üblichen Größe unserer Variantenmodelle unbedenklich.

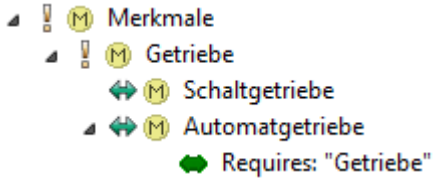


Abbildung 5.2.: Verletzung der Konsistenzbedingung 5.1.2

5.1.3. Tote Merkmale

Bedingung

Es gibt im Merkmalmodell keine Merkmale, die auf Grund von Constraints in keiner gültigen Selektion gewählt werden können (sogenannte *tote* Merkmale).

Beschreibung

Diese Konsistenzbedingung betrifft nur das Merkmalmodell. Existiert darin ein totes Merkmal, so kann dieses in keiner gültigen Selektion und damit auch in keiner Systemvariante enthalten sein. Dieser Umstand gefährdet keinesfalls den Konfigurationsprozess. Jedoch ist anzunehmen, dass es sich um eine Fehlmodellierung handelt, da wir grundsätzlich davon ausgehen, dass jedes Merkmal entweder in einer bestehenden Selektion ausgewählt ist oder theoretisch für eine geplante Selektion ausgewählt werden kann.

Definition

Sei $VM = (\mathcal{M}, KM)$ ein Variantenmodell mit $\mathcal{M} = (M, H \cup C, r)$. Dann erfüllt VM die Konsistenzbedingung, falls gilt: Es existiert kein $m_i \in M$, für welches gilt

$$\text{Sei } S \text{ eine Selektion mit } s_i = m_i \quad \Rightarrow \quad S \wedge \phi(\mathcal{M}) \text{ ist nicht erfüllbar}$$

Prüfung

Um ein gegebenes Variantenmodell bzgl. dieser Konsistenzbedingung zu analysieren, muss für jedes Merkmal $m_i \in M$ die folgende aussagenlogische Formel auf Erfüllbarkeit geprüft werden:

$$\phi(\mathcal{M}) \wedge m_i \tag{5.1}$$

Abbildung 5.3.: Verletzung der Konsistenzbedingung 5.1.3

Dann ist m_i ein totes Merkmal, falls die obige Formel *nicht* erfüllbar ist. Um das zu begründen, nehmen wir an, dass m_i kein totes Merkmal ist. Dann existiert eine gültige Selektion $S \in F(M)$, die m_i enthält. Aus der Gültigkeit folgt:

$$S \wedge \phi(\mathcal{M}) = s_1 \wedge \cdots \wedge s_n \wedge \phi(\mathcal{M}) \text{ ist erfüllbar.}$$

Da m_i in S enthalten ist, gilt $s_i = m_i$. Dadurch ist ebenfalls

$$s_1 \wedge \cdots \wedge s_n \wedge \phi(\mathcal{M}) \wedge m_i \text{ erfüllbar,}$$

also insbesondere auch die Formel 5.1.

Wir haben gezeigt: Ist m_i kein totes Merkmal, so ist $\phi(\mathcal{M}) \wedge m_i$ erfüllbar. Durch Kontraposition folgt daraus die Behauptung: Ist die Formel 5.1 nicht erfüllbar, so ist m_i ein totes Merkmal.

Eine Prüfung des gesamten Variantenmodells würde eine SAT-Analyse pro Merkmal erfordern. Um möglichst viele dieser für die Laufzeit kritischen Analysen einzusparen, lässt sich die Prüfung beschleunigen: bei dem Test des ersten nicht toten Merkmals erhält man eine gültige Belegung und damit ausgewählte Merkmale. Diese sind nach Definition nicht tot und können somit für die weitere Analyse ignoriert werden. Dieses Vorgehen wird solange wiederholt, bis alle Merkmale entweder geprüft oder ausgeschlossen werden konnten.

Gegenbeispiel

Das Merkmalmodell in Abb. 5.3 verletzt die beschriebene Konsistenzbedingung. Darin ist *US* ein totes Merkmal, da es in einer *conflicts*-Beziehung zu dem echt obligatorischen Merkmal *Repeater* steht. Es kann nicht ausgewählt werden und ist somit in keiner gültigen Selektion enthalten.

Komplexität

Bei dieser Konsistenzbedingung muss eine aussagenlogische Formel auf Erfüllbarkeit geprüft werden. Dieses Problem liegt in der Komplexitätsklasse *NP*, weshalb zunächst unklar ist, wie die Prüfung der starken Konsistenzbedingungen bei Variantenmodellen realistischer Größe skaliert. Verschiedene Versuche mit dem Prototypen haben jedoch gezeigt, dass die Performanz bei Erfüllbarkeitsanalysen auf Modellen aus der Praxis unbedenklich ist (s. dazu Abschnitt 6.3.2).

5.1.4. Merkmale ohne Einfluss auf die Konfiguration

Bedingung

Jedes variable Merkmal tritt in mindestens einer Konfigurationsformel auf.

Beschreibung

Um eine neue Variante zu erstellen, wird zuerst durch die Auswahl der gewünschten Merkmale eine Selektion definiert. Die Auswertung der Konfigurationsformeln an den Variationspunkten findet anschließend mit dieser Merkmalauswahl statt. Existiert ein variables (also nicht-obligatorisches) Merkmal, welches in keiner Konfigurationsformel enthalten ist, so hat es keinen Einfluss auf die Konfiguration. Es würde somit für die erstellte Systemvariante keinen Unterschied machen, ob das Merkmal ausgewählt ist oder nicht.

In Ausnahmefällen kann ein variables Merkmal ohne Verlinkung im Konfigurationsmodell jedoch vorkommen bzw. sinnvoll sein:

- Wenn bereits bekannt ist, dass in naher Zukunft der Funktionsumfang im System erweitert wird, kann das Merkmal dafür schon angelegt werden. Dieses wird verlinkt, wenn die Implementierung von Funktionalität und Variabilitätsmechanismus erfolgt und der Variationspunkt definiert ist. Bis dahin ist es in keiner Konfigurationsformel enthalten.

- Im Merkmalmodell werden sogenannte abstrakte Merkmale verwendet, um eine Struktur zu schaffen. Diese befinden sich in der Regel in den oberen Hierarchieebenen und fassen beispielsweise bestimmte Merkmalgruppen unter sich zusammen, die wiederum zur Konfiguration genutzt werden.

- Ein variables Merkmal kann indirekt auf die Konfiguration Einfluss nehmen. Auch wenn es in keiner Konfigurationsformel enthalten ist, kann es durch Constraints mit Merkmalen in Beziehung stehen, die im Konfigurationsmodell verlinkt sind. Damit ist für die Systemvariante unter Umständen entscheidend, ob dieses Merkmal gewählt ist oder nicht.

Wie bei allen schwachen Konsistenzbedingungen muss der Entwickler die gefundenen Auffälligkeiten selbst bewerten und ggf. die Modellierung anpassen.

Definition

Sei $VM = (\mathcal{M}, KM)$ ein Variantenmodell mit $\mathcal{M} = (M, H \cup C, r)$. Dann erfüllt VM die Konsistenzbedingung, falls gilt: Für alle $m \in M \setminus \{y \in M : (x, y) \in obl\}$ existiert eine Konfigurationsformel KF mit $m \in V(KF)$.

Prüfung

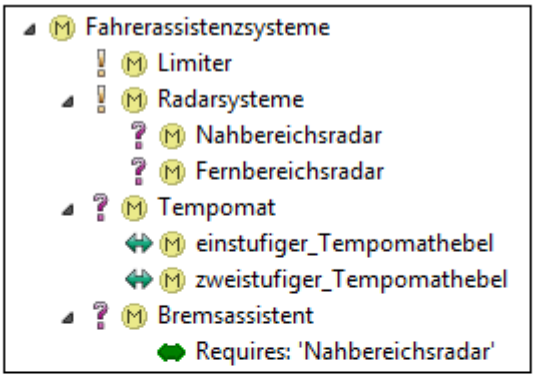
Um diese Konsistenzbedingung zu prüfen, muss für jedes optionale, alternative und disjunktive Merkmal $m \in M$ überprüft werden, ob es in einer Konfigurationsformel enthalten ist.

Gegenbeispiel

In Abb. 5.4 ist ein Variantenmodell dargestellt, welches die beschriebene Konsistenzbedingung verletzt. Das Merkmal *Fernbereichsradar* ist in keiner Konfigurationsformel enthalten. Da es zudem in keinem Constraint auftritt und auch keine Kinder im Merkmalbaum hat, ist davon auszugehen, dass entweder die Verlinkung im Konfigurationsmodell geplant, jedoch noch nicht durchgeführt wurde oder es sich tatsächlich um eine Fehlmodellierung handelt.

Komplexität

Bei dieser Analyse müssen für jedes Merkmal alle Konfigurationsformeln geprüft werden. Wie viel Zeit für eine einzelne Prüfung benötigt wird, hängt dabei von der Länge der Konfigurationsformel, also von der Anzahl der enthaltenen Merkmale ab. Der Aufwand



Variation Point	Assignment
VAR_Bremsassistent	
1 (Leermodul)	NOT(Bremsassistent)
2 (Vollmodul)	Bremsassistent
VAR_Tempomat	
1 (Leermodul)	NOT(Tempomat)
2 (einstufiger Tempomathebel)	einstufiger_Tempomathebel
3 (zweistufiger Tempomathebel)	zweistufiger_Tempomathebel

Abbildung 5.4.: Verletzung der Konsistenzbedingung 5.1.4

steigt daher im schlimmsten Fall polynomiell mit der Anzahl der Merkmale, der Konfigurationsformeln bzw. der Formellänge. Somit ist die Komplexität dieser Prüfung in Anbetracht der üblichen Größe unserer Variantenmodelle unbedenklich.

5.2. Starke Konsistenzbedingungen

In diesem Abschnitt werden die starken Konsistenzbedingungen beschrieben. Für jede dieser Bedingungen besteht die Prüfung aus Erfüllbarkeitsanalysen von aussagenlogischen Formeln. Das Erfüllbarkeitsproblem der Aussagenlogik gehört zur Komplexitätsklasse *NP*, weshalb zunächst unklar ist, wie die Prüfung der starken Konsistenzbedingungen bei Variantenmodellen realistischer Größe skaliert. Verschiedene Versuche mit dem Prototypen auf Modellen aus der Praxis haben jedoch gezeigt, dass die Performanz dabei unbedenklich ist (s. dazu Abschnitt 6.3.2). Da die Betrachtung der Komplexität für alle starken Konsistenzbedingungen identisch ist, wird diese Rubrik in den folgenden Beschreibungen nicht mehr erwähnt.

5.2.1. Kontradiktionen in Konfigurationsformeln

Bedingung

Konfigurationsformeln sind keine Kontradiktionen in Bezug auf die Merkmalmodellierung.

Beschreibung

Diese Bedingung behandelt die konsistente Formulierung des Konfigurationswissens. Es wird geprüft, ob Konfigurationsformeln existieren, die im Widerspruch zur Merkmalmodellierung stehen. Diese Fehlmodellierung würde dazu führen, dass keine gültige Selektion die betroffene Konfigurationsformel erfüllt, wodurch die korrespondierende Variation nie gewählt wird und damit überflüssig ist.

Definition

Sei $VM = (\mathcal{M}, KM)$ ein Variantenmodell. Sei außerdem KF eine Konfigurationsformel an einem Variationspunkt $VP \in KM$. Dann erfüllt KF die oben beschriebene Konsistenzbedingung, falls gilt:

Es existiert eine gültige Selektion $S \in F(M)$, sodass die Formel

$$S \wedge KF$$

erfüllbar ist.

Das Variantenmodell selbst ist im Sinne der obigen Bedingung konsistent, falls an jedem Variationspunkt aus KM jede Konfigurationsformel die Konsistenzbedingung erfüllt.

Prüfung

Bei dieser Konsistenzbedingung wird an jedem Variationspunkt jede Konfigurationsformel einzeln geprüft. Nach Definition ist die Konfigurationsformel KF konsistent, falls gilt: es existiert eine gültige Selektion $S \in F(M)$, sodass $S \wedge KF$ erfüllbar ist. In dem Fall ist für diese Selektion auch $S \wedge \phi(\mathcal{M})$ erfüllbar, da sie gültig ist. Nach Anwendung von Lemma 3 (Abschnitt A.1.3 im Anhang) gilt dann:

$$S \wedge \phi(\mathcal{M}) \wedge KF \text{ ist erfüllbar}$$

und damit insbesondere auch $\phi(\mathcal{M}) \wedge KF$.

Im Umkehrschluss dieser Überlegungen ergibt sich: Ist $\phi(\mathcal{M}) \wedge KF$ *nicht* erfüllbar, so liegt bei der Konfigurationsformel KF Inkonsistenz vor, da es sich um eine Kontradiktion bzgl. des Merkmalmodells handelt. Es muss somit für jede Konfigurationsformel KF der folgende Ausdruck auf Erfüllbarkeit getestet werden:

$$\phi(\mathcal{M}) \wedge KF$$

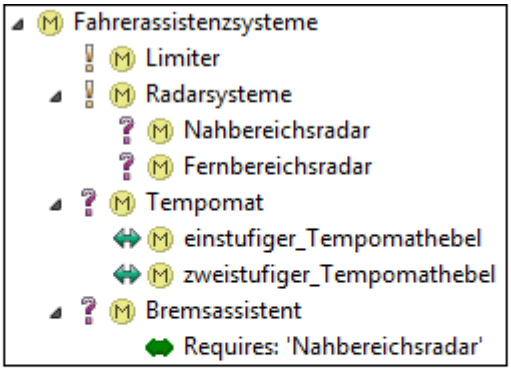
Gegenbeispiel

Das Variantenmodell in Abb. 5.5 verletzt die beschriebene Konsistenzbedingung. Die Konfigurationsformel von Variation 2 kann nie erfüllt sein, da die Merkmale *einstufiger_Tempomathebel* und *zweistufiger_Tempomathebel* im Merkmalmodell als Alternativen modelliert sind und somit in einer gültigen Selektion nicht beide Merkmale ausgewählt werden dürfen.

5.2.2. Tautologien in Konfigurationsformeln

Bedingung

Konfigurationsformeln sind keine Tautologien in Bezug auf die Merkmalmodellierung.



Variation Point	Assignment
VAR_VP_1	
1 (Variation 1)	NOT(Tempomat)
2 (Variation 2)	einstufiger_Tempomathebel AND zweistufiger_Tempomathebel

Abbildung 5.5.: Verletzung der Konsistenzbedingung 5.2.1

Beschreibung

Diese Konsistenzbedingung zielt auf ein aussagekräftiges Konfigurationswissen ab. Es soll verhindert werden, dass Konfigurationsformeln Tautologien in Bezug auf die Merkmalmodellierung sind, also von jeder gültigen Selektion $S \in F(M)$ erfüllt werden. Eine Inkonsistenz dieser Art kann zwei verschiedene Folgen haben, die wiederum als Inkonsistenzen aufgefasst werden können:

1. Die restlichen Konfigurationsformeln an dem betroffenen Variationspunkt wurden auch fehlerhaft modelliert in dem Sinne, dass sie widersprüchlich zur Merkmalmodellierung sind und somit von keiner gültigen Selektion erfüllt werden. Diese Inkonsistenz ist näher in Abschnitt 5.2.1 beschrieben. Dadurch würde an diesem Variationspunkt stets dieselbe Variation ausgewählt werden und alle weiteren Variationen sowie deren Konfigurationsformeln wären überflüssig. Dies kann nicht Ziel der Variantenmodellierung sein, da die Konfiguration eines Variationspunktes immer von der Selektion abhängen sollte.

2. Alle weiteren Konfigurationsformeln an diesem Variationspunkt sind korrekt formuliert. Dann wird es bei mindestens einer Selektion dazu kommen, dass mehr als eine Konfigurationsformel erfüllt wird und damit die Konfiguration fehlschlägt. Diese Inkonsistenz ist in Abschnitt 5.2.4 beschrieben.

Dies zeigt, dass die Folgen der angesprochenen Fehlmodellierung ebenfalls mithilfe anderer Konsistenzbedingungen gefunden werden können. Um jedoch die Ursache dafür zu

ermitteln, ist es sinnvoll, das Variantenmodell auch dieser Prüfung zu unterziehen.

Definition

Sei $VM = (\mathcal{M}, KM)$ ein Variantenmodell. Sei außerdem KF eine Konfigurationsformel an einem Variationspunkt $VP \in KM$. Dann erfüllt KF die oben beschriebene Konsistenzbedingung, falls gilt:

Es existiert eine gültige Selektion $S \in F(M)$, sodass die Formel

$$S \wedge KF$$

nicht erfüllbar ist.

Das Variantenmodell selbst ist im Sinne der obigen Bedingung konsistent, falls an jedem Variationspunkt aus KM jede Konfigurationsformel die Konsistenzbedingung erfüllt.

Prüfung

Bei dieser Konsistenzbedingung wird an jedem Variationspunkt jede Konfigurationsformel einzeln geprüft. Nach Definition ist die Konfigurationsformel KF konsistent, falls gilt: es existiert eine gültige Selektion $S \in F(M)$, sodass $S \wedge KF$ *nicht* erfüllbar ist. In dem Fall ist für diese Selektion auch $S \wedge \phi(\mathcal{M})$ erfüllbar (da sie gültig ist). Außerdem lässt sich aus Lemma 4 (Abschnitt A.1.4 im Anhang) folgern:

$$S \wedge KF \text{ nicht erfüllbar} \Rightarrow S \wedge (\neg KF) \text{ erfüllbar.}$$

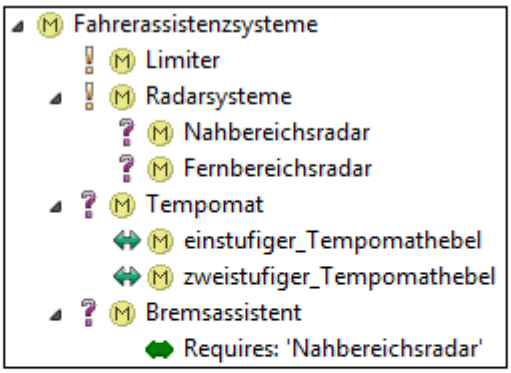
Damit gilt nach Anwendung von Lemma 3 (Abschnitt A.1.3 im Anhang):

$$S \wedge \phi(\mathcal{M}) \wedge (\neg KF) \text{ ist erfüllbar}$$

und damit insbesondere auch $\phi(\mathcal{M}) \wedge (\neg KF)$.

Im Umkehrschluss dieser Überlegungen ergibt sich: Ist $\phi(\mathcal{M}) \wedge (\neg KF)$ *nicht* erfüllbar, so liegt bei der Konfigurationsformel KF Inkonsistenz vor, da es sich um eine Tautologie bzgl. des Merkmalmodells handelt. Es muss somit für jede Konfigurationsformel KF der folgende Ausdruck auf Erfüllbarkeit getestet werden:

$$\phi(\mathcal{M}) \wedge (\neg KF)$$



Variation Point	Assignment
VAR_VP_2	
1 (Variation 1)	Limiter
2 (Variation 2)	Tempomat

Abbildung 5.6.: Verletzung der Konsistenzbedingung 5.2.2

Gegenbeispiel

Das Variantenmodell in Abb. 5.6 verletzt die beschriebene Konsistenzbedingung. Die Konfigurationsformel von Variation 1 wird von jeder gültigen Selektion erfüllt, da das Merkmal *Limiter* im Merkmalmodell echt obligatorisch ist und somit stets ausgewählt werden muss.

5.2.3. Erfüllbare Konfigurationsformeln

Bedingung

An jedem Variationspunkt muss für jede Selektion mindestens eine Konfigurationsformel erfüllt sein.

Beschreibung

In Abschnitt 3.5 über den Konfigurationsprozess wurde darauf hingewiesen, wie wichtig die eindeutige Konfiguration ist. Dafür muss sichergestellt werden, dass an jedem Variationspunkt für jede gültige Selektion mindestens eine Konfigurationsformel erfüllt ist. Sollte eine Selektion existieren, die an einem Variationspunkt von keiner Konfigu-

rationsformel erfüllt wird, kann in diesem Fall keine Variation ermittelt werden. Dies kann zur Folge haben, dass das Zielartefakt nicht vollständig konfiguriert werden kann, sodass unter Umständen eine inkonsistente bzw. syntaktisch fehlerhafte Artefaktvariante entsteht.

Definition

Sei $VM = (\mathcal{M}, KM)$ ein Variantenmodell. Sei außerdem $VP \in KM$ ein Variationspunkt mit $VP = ((V_1, KF_1), \ldots, (V_p, KF_p))$. Dann erfüllt VP die oben beschriebene Konsistenzbedingung, falls gilt:

Für jede beliebige gültige Selektion $S \in F(M)$ existiert mindestens ein $i \in \{1, \ldots, p\}$, sodass die Formel

$$S \wedge KF_i$$

erfüllbar ist.

Das Variantenmodell selbst ist im Sinne der obigen Bedingung konsistent, falls jeder Variationspunkt aus KM die Konsistenzbedingung erfüllt.

Prüfung

Jeder Variationspunkt wird bei dieser Bedingung einzeln geprüft. Die Bedingung ist für den Variationspunkt $VP = ((V_1, KF_1), \ldots, (V_p, KF_p)) \in KM$ verletzt, falls die folgende Formel aus $F(M)$ erfüllbar ist:

$$\phi(\mathcal{M}) \wedge \bigwedge_{i=1}^{p} (\neg KF_i)$$

Begründung: Aus der Erfüllbarkeit der obigen Formel gilt nach Lemma 2 (Abschnitt A.1.2 im Anhang), dass eine Selektion $S \in F(M)$ existiert, sodass

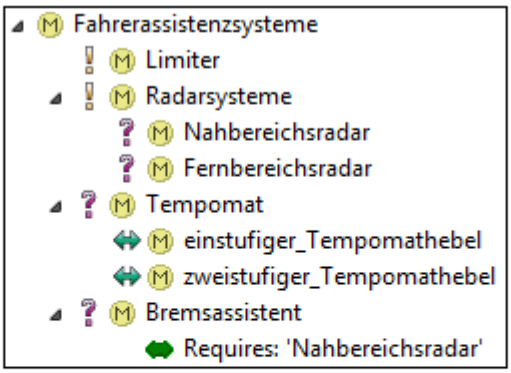
$$S \wedge \phi(\mathcal{M}) \wedge \bigwedge_{i=1}^{p} (\neg KF_i)$$

erfüllbar ist. Damit ist insbesondere auch

$$S \wedge \phi(\mathcal{M})$$

erfüllbar, womit die Gültigkeit der Selektion S gezeigt ist.
Außerdem ist

$$S \wedge \bigwedge_{i=1}^{p} (\neg KF_i)$$



Variation Point	Assignment
VAR_VP_3	
1 (Variation 1)	Tempomat AND einstufiger_Tempomathebel
2 (Variation 2)	Tempomat AND zweistufiger_Tempomathebel

Abbildung 5.7.: Verletzung der Konsistenzbedingung 5.2.3

erfüllbar und damit auch

$$S \wedge \neg KF_i \text{ für alle } 1 \leq i \leq p.$$

In Lemma 4 (Abschnitt A.1.4 im Anhang) haben wir gesehen, dass damit

$$S \wedge KF_i$$

für kein $i \in \{1, \ldots, p\}$ erfüllbar ist. Folglich erfüllt die gültige Selektion $S \in F(M)$ nach Definition keine Konfigurationsformel an dem untersuchten Variationspunkt, wodurch dieser die Konsistenzbedingung verletzt.

Gegenbeispiel

Das Variantenmodell in Abb. 5.7 verletzt die beschriebene Konsistenzbedingung. Das Merkmal *Tempomat* ist im Merkmalmodell optional, kann damit an- und abgewählt werden. Beide Konfigurationsformeln werden jedoch nur von Selektionen erfüllt, in denen dieses Merkmal ausgewählt wurde. Dadurch kann für eine Fahrzeugvariante ohne Tempomat keine Variation an diesem Variationspunkt ermittelt werden, wodurch die Konfiguration fehlschlägt.

5.2.4. Eindeutige Konfigurationsformeln

Bedingung

An jedem Variationspunkt darf für jede gültige Selektion höchstens eine Konfigurationsformel erfüllt sein.

Beschreibung

In Abschnitt 3.5 über den Konfigurationsprozess wurde darauf hingewiesen, wie wichtig die eindeutige Konfiguration ist. Dafür muss unter anderem sichergestellt werden, dass an jedem Variationspunkt für jede gültige Selektion höchstens eine Konfigurationsformel erfüllt ist. Sollte eine gültige Selektion existieren, die an einem Variationspunkt mehr als eine Konfigurationsformel erfüllt, so kann in dieser Situation keine Variation ermittelt werden. Das hat zur Folge, dass das Zielartefakt nicht vollständig konfiguriert werden kann, sodass unter Umständen eine inkonsistente bzw. syntaktisch fehlerhafte Artefaktvariante entsteht.

Definition

Sei $VM = (\mathcal{M}, KM)$ ein Variantenmodell. Sei außerdem $VP \in KM$ ein Variationspunkt mit $VP = ((V_1, KF_1), \ldots, (V_p, KF_p))$. Dann erfüllt VP die oben beschriebene Konsistenzbedingung, falls gilt:

Für keine gültige Selektion $S \in F(M)$ existieren Indizes $i \neq j \in \{1, \ldots, p\}$, sodass die Formeln

$$S \wedge KF_i \text{ } und$$
$$S \wedge KF_j$$

erfüllbar sind.

Das Variantenmodell selbst ist im Sinne der obigen Bedingung konsistent, falls jeder Variationspunkt aus KM die Konsistenzbedingung erfüllt.

Prüfung

Jeder Variationspunkt wird bei dieser Bedingung einzeln geprüft. Die Konsistenzbedingung ist für den Variationspunkt $VP = ((V_1, KF_1), \ldots, (V_p, KF_p)) \in KM$ verletzt, falls die folgende Formel erfüllbar ist:

$$\phi(\mathcal{M}) \wedge (\bigvee_{i<j} (KF_i \wedge KF_j))$$

Denn ist der obige Ausdruck erfüllbar, so folgt aus Lemma 2 (Abschnitt A.1.2 im Anhang), dass eine Selektion $S \in F(M)$ existiert, sodass die Formel

$$S \wedge \phi(\mathcal{M}) \wedge (\bigvee_{i<j} (KF_i \wedge KF_j))$$

erfüllbar ist. Damit ist insbesondere auch

$$S \wedge \phi(\mathcal{M})$$

erfüllbar, woraus folgt, dass S eine gültige Selektion ist. Darüber hinaus ist außerdem

$$S \wedge (\bigvee_{i<j} (KF_i \wedge KF_j))$$

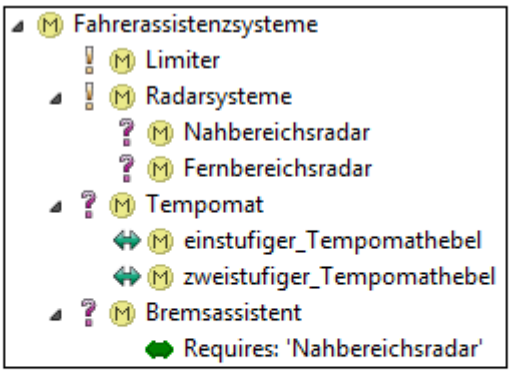
erfüllbar. Es muss also mindestens ein Paar von Indizes $1 \leq i, j \leq p$ existieren, sodass

$$S \wedge KF_i \wedge KF_j$$

erfüllbar ist. Daraus folgt unmittelbar die obige Definition für die Verletzung dieser Konsistenzbedingung, da die gültige Selektion S die beiden Konfigurationsformeln KF_i *und* KF_j erfüllt.

Dieses Vorgehen bei der Prüfung der Konsistenzbedingung hat einen vermeintlichen Nachteil. Aus der Erfüllbarkeit der oben genannten Formel lässt sich nur folgern, dass der untersuchte Variationspunkt die Konsistenzbedingung verletzt. Liefert die maschinelle Prüfung auch die erfüllende Belegung, erhält man zusätzlich die für die Inkonsistenz verantwortliche Selektion. Es bleibt jedoch zunächst unklar, *welche* beiden Konfigurationsformeln von dieser Selektion erfüllt werden.

Aus diesem Grund muss anschließend an dem inkonsistenten Variationspunkt noch jede Konfigurationsformel einzeln geprüft werden, ob sie von der gefundenen Selektion erfüllt wird. Erfahrungsgemäß tritt diese Inkonsistenz nicht bei der Mehrheit der Variationspunkte auf. Daher lohnt sich eine initiale Analyse, die zeigen kann, welche Variationspunkte überhaupt betroffen sind.



Variation Point	Assignment
VAR_VP_4	
1 (Variation 1)	Tempomat
2 (Variation 2)	einstufiger_Tempomathebel

Abbildung 5.8.: Verletzung der Konsistenzbedingung 5.2.4

Gegenbeispiel

Das Variantenmodell in Abb. 5.8 verletzt die beschriebene Konsistenzbedingung. In einer Selektion, welche das Merkmal *einstufiger_Tempomathebel* enthält, muss auch dessen Vatermerkmal *Tempomat* ausgewählt werden. Damit wären die Konfigurationsformeln beider Variationen erfüllt. Innerhalb des Konfigurationsprozesses kann dann nicht entschieden werden, welche Variation an diesem Variationspunkt gebunden werden soll.

6. Der Prototyp

Dieses Kapitel beschäftigt sich mit der prototypischen Implementierung der Änderungsprozesse und der Konsistenzbedingungen für das Variantenmodell, wie sie in den Kapiteln 4 und 5 dargestellt wurden. Zunächst beschäftigt sich Abschnitt 6.1 mit dem Aufbau und den wesentlichen Merkmalen des Prototypen. Anschließend wird die Implementierung der Änderungsprozesse und der Konsistenzbedingungen beschrieben und dabei auf Aspekte wie Umsetzung, Ausführung und Performanz eingegangen.

6.1. Beschreibung des Prototypen

Der Prototyp setzt auf das Eclipse Plug-In pure::variants auf. Dieses wird bei der Daimler AG eingesetzt, um merkmalbasierte Variantenmodelle von Softwaresystemen zu erstellen. Mithilfe verschiedener Konnektoren zu Entwicklungswerkzeugen, wie z. B. Matlab Simulink bzw. TargetLink oder IBM Rational DOORS können anhand der Variantenmodelle die entsprechenden Entwicklungsartefakte automatisch konfiguriert werden.

In Abb. 6.1 ist die Architektur des Prototypen schematisch dargestellt. Das Werkzeug pure::variants stellt eine Schnittstelle zur Verfügung, die mit der Programmiersprache JavaScript angesprochen werden kann. Dementsprechend wurden die Änderungsprozesse bzw. Konsistenzbedingungen in Skripten dieser Sprache realisiert. Zwischen der werkzeugseitigen Schnittstelle und den einzelnen Skripten wurde außerdem eine Zwischenschicht erstellt. Dabei ist für das Merkmalmodell (*Feature Model*), das Konfigurationsmodell (*Family Model*) und für eine Selektion (*Variant Model*) jeweils ein Skript entstanden, welches alle Funktionen enthält, die aus einer Instanz dieses Modelltyps entweder Informationen abfragen oder Manipulationen an dieser vornehmen. Das *Base*-Skript enthält alle Funktionen, die vom Modelltyp unabhängig sind.

Dieser Aufbau bietet eine Übersicht der zur Verfügung stehenden Funktionen für die verschiedenen Modelltypen. Darüber hinaus sind die Skripte der Änderungsprozesse bzw. Konsistenzbedingungen damit weniger abhängig von der pure::variants-Schnittstelle. Ändert sich diese, muss lediglich an einer Stelle in der Zwischenschicht überprüft werden, welche Auswirkungen sich daraus ergeben.

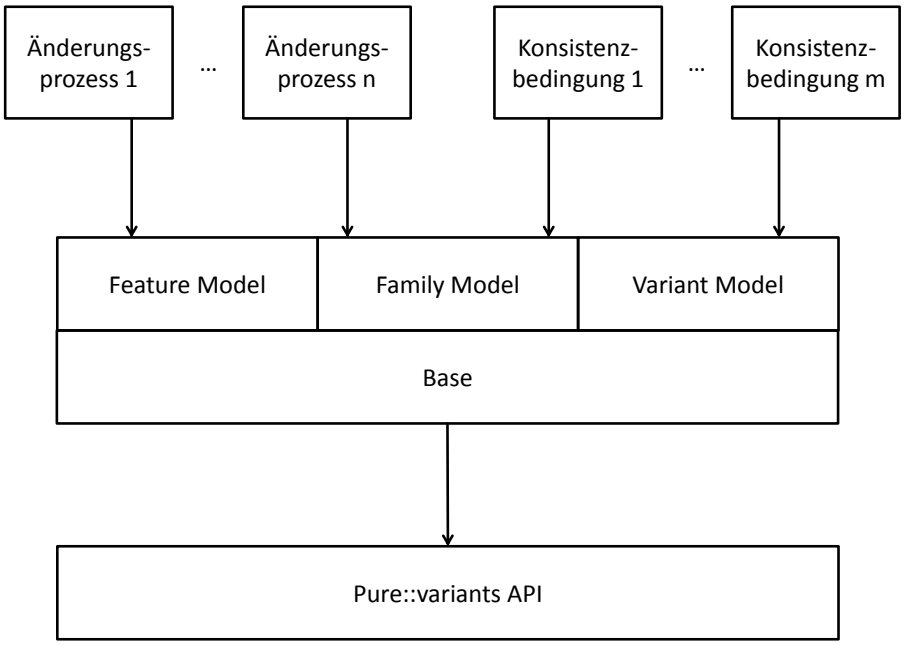


Abbildung 6.1.: Architektur des Prototypen

6.2. Implementierung der Änderungsprozesse

Für die in Abschnitt 4.1 beschriebenen Änderungsprozesse sind Skripte entstanden, welche die entsprechenden Änderungen im Variantenmodell durchführen. Dieser Abschnitt geht auf die Umsetzung und die Ausführung der Skripte ein und beschreibt anschließend einen Beispielprozess.

6.2.1. Umsetzung

Der Großteil dieser Implementierungen enthält Benutzerinteraktionen, um z. B. die Namen neuer Merkmale abzufragen oder eine Entscheidung zwischen verschiedenen Modellierungsalternativen zu treffen. Dabei sind die Skripte so aufgebaut, dass zunächst alle Benutzereingaben gesammelt werden, bevor eine tatsächliche Transformation des Modells stattfindet. Dadurch kann zu Beginn die Konsistenz der Eingabeparameter geprüft werden. Darüber hinaus wird verhindert, dass ein inkonsistentes Modell zurückbleibt, falls beispielsweise ein Änderungsprozess zwischen Eingabe- und Modelltransformationsschleifen vom Benutzer abgebrochen wird.

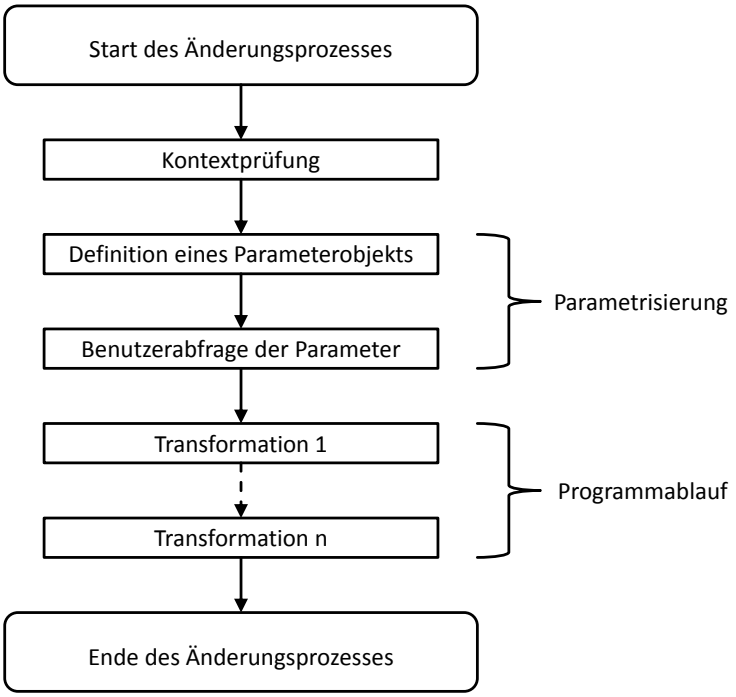


Abbildung 6.2.: Ablauf der Änderungsprozesse

Diese Parametrisierung hat die Implementierungen der Änderungsprozesse außerdem einem automatisierten Testkonzept zugänglich gemacht, da alle Eingaben in einem Parameterobjekt gesammelt werden und die eigentliche Änderung des Modells im Anschluss ohne Benutzerinteraktion ablaufen kann[1]. In Abb. 6.2 ist die generische Vorgehensweise der Änderungsprozesse in einem Ablaufdiagramm dargestellt. Durch diesen Aufbau und die dadurch mögliche Qualitätssicherung der Änderungsprozesse trägt auch die Implementierung dazu bei, das Risiko für Inkonsistenzen in den Variantenmodellen zu minimieren.

6.2.2. Ausführung

Jedes Skript eines Änderungsprozesses kann direkt aus pure::variants aufgerufen werden. Über das Kontextmenü eines Merkmals öffnet sich ein Dialogfenster, in dem der Benutzer auswählen kann, welches Skript ausgeführt werden soll. Dabei ist entscheidend, über welches Merkmal das Dialogfenster geöffnet wird. Für die Änderungsprozesse

[1]Das Testkonzept zur Qualitätssicherung der Änderungsprozesse ist im Rahmen einer Abschlussarbeit entstanden [Win13].

- *optionale Komponente löschen* und
- *optionale Komponente wird obligatorisch*

wird die Transformation an dem Merkmal durchgeführt, welches ausgewählt wurde. Dagegen muss man für die Änderungsprozesse

- *neue optionale Komponente* und
- *neue alternative Komponente*[2]

das Skript über das Kontextmenü jenes Merkmals selektieren, welches künftig der Vater der neuen Merkmale im Merkmalbaum sein wird.

Wählt der Benutzer auf diese Weise ein Skript aus, so führt es zunächst eine Prüfung durch, ob der gewählte Kontext durch den Änderungsprozess bearbeitet werden kann. Beispielsweise ändert das Skript *optionale Komponente wird obligatorisch* den Variationstyp des gewählten Merkmals von optional auf obligatorisch, prüft alle relevanten Constraints und Konfigurationsformeln und passt sie ggf. an. Würde man dieses Skript auf ein Merkmal anwenden, welches nicht optional ist, gibt es eine Fehlermeldung aus und bricht ab.

Nachdem der Kontext geprüft und die Benutzereingaben entgegen genommen sind, werden die Modelltransformationen durchgeführt. Das Skript gibt im Anschluss alle realisierten Änderungen aus, sodass diese ggf. vom Entwickler nochmal geprüft werden können.

6.2.3. Beispielprozess

In Abb. 6.3 ist ein Aktivitätsdiagramm dargestellt, welches die Implementierung des Änderungsprozesses *optionale Komponente löschen* beschreibt. Wie bereits im vorherigen Abschnitt erwähnt, wird der Prozess über das Kontextmenü des entsprechenden Merkmals gestartet und zunächst geprüft, ob es sich tatsächlich um ein optionales Merkmal handelt bzw. dieses auch keine Kinder im Merkmalbaum hat. Ist beides der Fall, wird auf die Abhängigkeiten des betroffenen Merkmals eingegangen.

Dazu werden alle Constraints bzw. Konfigurationsformeln, in denen das Merkmal auf-

[2]Für diesen Änderungsprozess wurde bereits in der Beschreibung danach unterschieden, welchen Variationstyp die bestehende Komponente hat (optional, obligatorisch oder selbst alternativ, vgl. Abschnitt 4.1.4). Dementsprechend sind auch drei verschiedene Skripte entstanden. Da der Aufruf jedoch für alle drei gleich ist, wurden sie hier zusammengefasst.

tritt, ermittelt und angepasst. Für diesen Änderungsprozess bedeutet das ein Löschen der Constraints und einen Hinweis an den Benutzer, welche Konfigurationsformeln er noch händisch überarbeiten muss. Hängen an einem Variationspunkt alle Konfigurationsformeln *nur* von dem zu löschenden Merkmal ab, so kann dieser gelöscht werden. Der Variationspunkt $VP = ((Leermodul, \neg\, m_i), (Vollmodul, m_i))$ ist ein klassisches Beispiel dafür, da er lediglich von dem Merkmal m_i abhängt. Damit sind alle Abhängigkeiten des Merkmals bearbeitet und es kann schlussendlich selbst gelöscht werden.

Für jeden Änderungsprozess ist vor der Implementierung ein Aktivitätsdiagramm entstanden, welches die Schrittfolge des Skriptes aufzeigt und alle nötigen Benutzerinteraktionen beinhaltet. In Abschnitt A.2 im Anhang sind die Aktivitätsdiagramme der restlichen Änderungsprozesse dargestellt.

6.3. Implementierung der Konsistenzbedingungen

Dieser Abschnitt geht zunächst auf die Umsetzung der Konsistenzbedingungen ein. Anschließend wird die Performanz des Prototypen im Praxiseinsatzes auf realistischen Modellen analysiert. Das ist besonders relevant, da bestimmte Modellprüfungen auf SAT-Analysen des Variantenmodells basieren, was auf Grund der algorithmischen Komplexität theoretisch zu hohen Laufzeiten führen kann.

6.3.1. Umsetzung

Für die in Kapitel 5 beschriebenen Konsistenzbedingungen ist jeweils ein Skript entstanden, welches ein beliebiges Variantenmodell der entsprechenden Prüfung unterziehen kann und im Fehlerfall das betroffene Element (Merkmal, Variationspunkt, Konfigurationsformel, etc.) in der Konsole ausgibt. Der für das Variantenmodell verantwortliche Entwickler entscheidet selbst, welche Konsistenzbedingung er an seinem Modell prüfen möchte, weil die Verletzung bestimmter schwacher Konsistenzbedingungen in gewissen Kontexten legitim sein kann.

Die in Abschnitt 5.2 beschriebenen starken Konsistenzbedingungen basieren im Kern auf Erfüllbarkeitsanalysen von aussagenlogischen Formeln, welche aus dem Variantenmodell extrahiert werden. Daher rufen die Skripte, welche diese Bedingungen implementieren, einen externen SAT-Solver auf[3], der die Erfüllbarkeitsanalyse durchführt. Der Solver liefert anschließend das Ergebnis und ggf. eine erfüllende Belegung an das Skript zurück,

[3]Es wurde SAT4J verwendet (s. z. B. [LBPRS]).

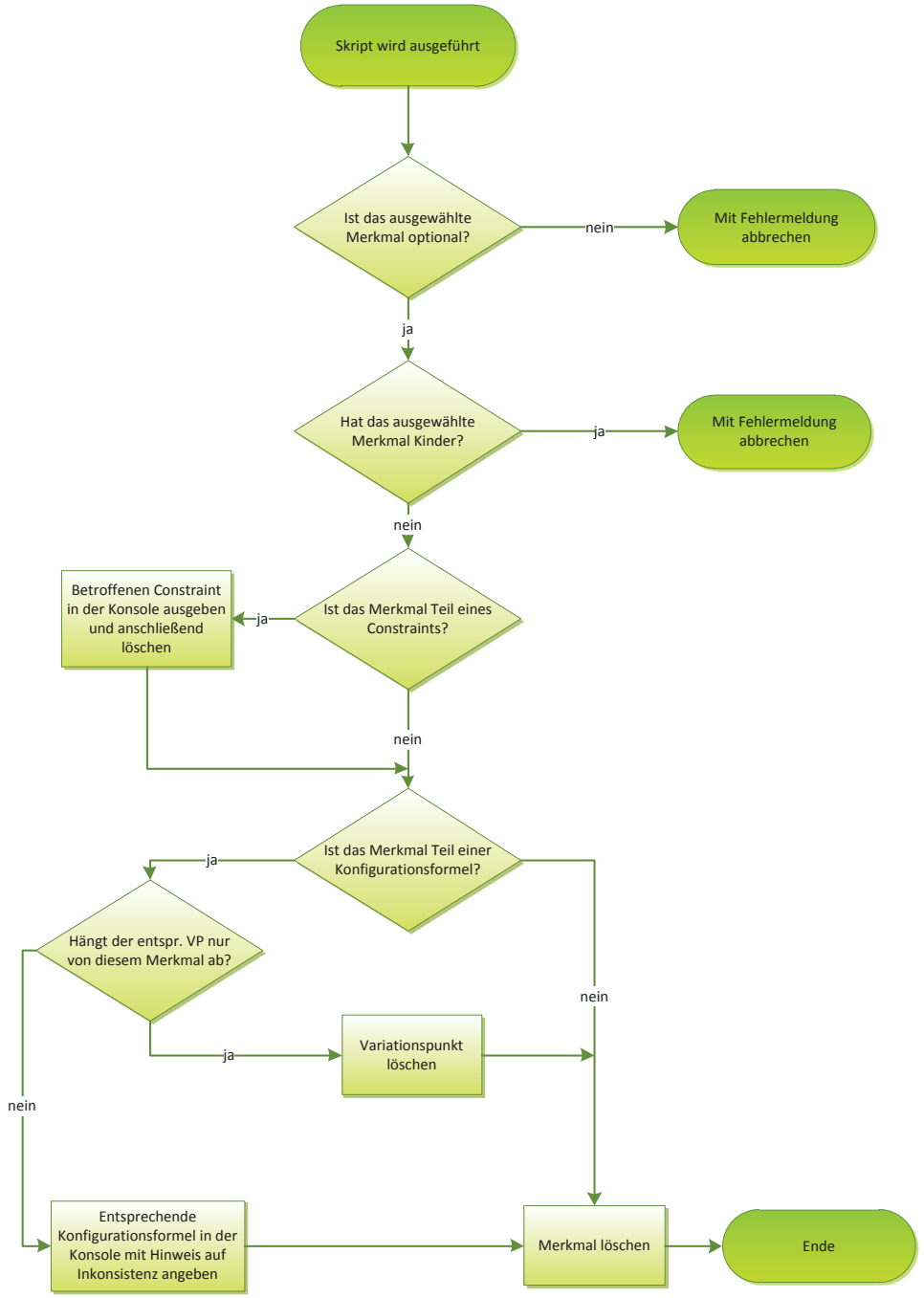


Abbildung 6.3.: Der Änderungsprozess *Optionale Komponente löschen*

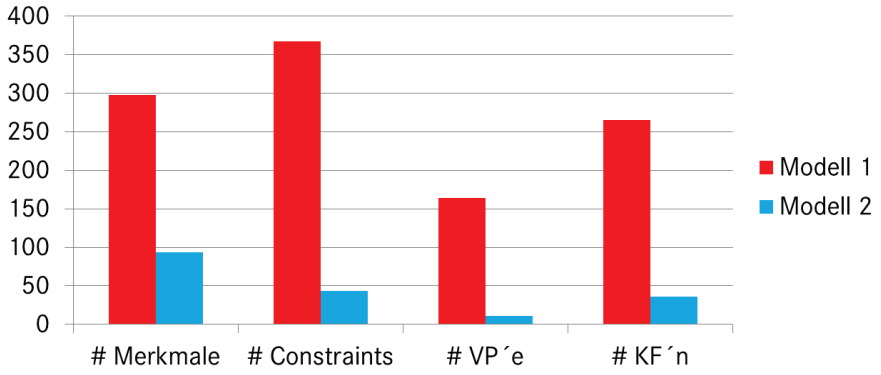


Abbildung 6.4.: Kennzahlen der Variantenmodelle[6]

welches diese Daten auswertet und für den Benutzer interpretiert. Auf die Performanz der Konsistenzprüfungen wird im nächsten Abschnitt eingegangen.

6.3.2. Performanz

Um die Performanz des Prototypen in einer realistischen Umgebung zu testen, wurde die automatische Prüfung der Konsistenzbedingungen an zwei Variantenmodellen durchgeführt, welche in der industriellen Praxis entstanden sind. Das Merkmalmodell von Variantenmodell 1 besteht aus 298 Merkmalen und 367 Constraints. Nach [MWCC08] beträgt die *extra constraint representativeness*[4] (ECR) 74,8 %. Das Konfigurationsmodell hat 164 Variationspunkte mit insgesamt 265 Konfigurationsformeln[5]. Die Größe des Modells liegt damit in einem Bereich, der für Domänen wie Motorsteuerung, Fahrerassistenz oder e-Drive üblich ist.

Das zweite Variantenmodell befindet sich noch in der Entwicklung und ist daher etwas kleiner: 94 Merkmale, 43 Constraints (ECR = 34 %) und 11 Variationspunkte mit insgesamt 36 Konfigurationsformeln. Die Grafik in Abb. 6.4 vergleicht in einer Übersicht die Kennzahlen der beiden Variantenmodelle.

[4]Die *extra constraint representativeness* (ECR) ist das Verhältnis zwischen Merkmalen, die in Constraints beteiligt sind (wobei Mehrfachbeteiligungen einfach gezählt werden) und der Anzahl der Merkmale insgesamt. Sie soll einen Aufschluss über die Dichte dieser Constraints im Merkmalmodell geben und kann so als Komplexitätsmaß gelten.

[5]Das Konfigurationsmodell beinhaltet einige Variationspunkte mit zwei Variationen, von denen eine als default-Variation definiert ist. Diese default-Variationen benötigen keine Konfigurationsformeln, wodurch der entsprechende Variationspunkt nur eine Konfigurationsformel enthält.

[6]VP´e = Variationspunkte, KF´n = Konfigurationsformeln

Tabelle 6.1.: Laufzeiten des Prototyps

	KB 5.2.1	KB 5.2.2	KB 5.2.3	KB 5.2.4	alle 4
VM 1					
einzelne Analyse	0,143 Sek.	0,143 Sek.	0,133 Sek.	0,145 Sek.	-
# Analysen	265	265	69	69	-
gesamtes Modell	134,8 Sek.	135,4 Sek.	33,3 Sek.	37,3 Sek.	320 Sek.
VM 2					
einzelne Analyse	0,147 Sek.	0,146 Sek.	0,143 Sek.	0,155 Sek.	-
# Analysen	36	36	11	11	-
gesamtes Modell	6,34 Sek.	6,38 Sek.	2,16 Sek.	2,28 Sek.	16,6 Sek.

Auf diesen beiden Modellen wurde die automatische Prüfung aller starken Konsistenzbedingungen durchgeführt. Eine Übersicht der Ergebnisse zeigt Tabelle 6.1[7]. Die ersten drei Zeilen enthalten für Variantenmodell 1 (*VM* 1) folgende Daten:

- Die durchschnittliche Dauer[8] *einer* SAT-Analyse für die vier verschiedenen Konsistenzbedingungen (Zeile 3).
- Die Anzahl der Analysen, welche bei der jeweiligen Konsistenzbedingungen nötig sind, um das gesamte Modell zu prüfen (Zeile 4).
- Die im Durchschnitt benötigte Zeit, um das gesamte Variantenmodell hinsichtlich der entsprechenden Konsistenzbedingung zu prüfen (Zeile 5).

Die Zeilen 7-9 enthalten die entsprechenden Informationen für Variantenmodell 2 (*VM* 2).

Da bei den Konsistenzbedingungen 5.2.1 bzw. 5.2.2 jede Konfigurationsformel einzeln analysiert wird, müssen bei dem ersten Variantenmodell 265 Analysen durchgeführt werden, um das gesamte Modell zu prüfen. Die Prüfung von *VM* 2 benötigt entsprechend 36 Analysen. Bei den Konsistenzbedingungen 5.2.3 bzw. 5.2.4 wird dagegen nur für jeden Variationspunkt eine SAT-Analyse durchgeführt. Hinzu kommt, dass diese beiden Konsistenzbedingungen nur für Variationspunkte sinnvoll sind, an denen mindestens zwei Konfigurationsformeln existieren. Wie bereits oben erwähnt, ist das nicht bei jedem Variationspunkt aus Variantenmodell 1 der Fall. Daher benötigen die entsprechenden Skripte nur 69 Analysen, um das gesamte Variantenmodell hinsichtlich dieser Konsistenzbedingungen zu prüfen. Bei *VM* 2 werden alle 11 Variationspunkte analysiert.

[7] Ein Teil dieser Laufzeiten wurden bereits in einer früheren Arbeit veröffentlicht [Hol14].

[8] Die Laufzeiten stellen jeweils den Durchschnittswert von mehreren Versuchsreihen dar.

Auffällig ist, dass die Dauer einer einzelnen SAT-Analyse nicht von der Modellgröße abzuhängen scheint. Obwohl die beiden Merkmalmodelle einen signifikanten Größenunterschied aufweisen und dadurch die für die Konsistenzbedingungen relevanten Formeln ebenfalls unterschiedlich komplex sind, nehmen die einzelnen SAT-Analysen von Variantenmodell 1 bzw. 2 im Durchschnitt in etwa dieselbe Zeit in Anspruch. Welcher Teil der Implementierung dagegen anscheinend von der Modellgröße beeinflusst wird, ist die Erstellung des Inputs für den SAT-Solver. Dabei wird das Merkmalmodell in eine aussagenlogische Formel überführt, die Konfigurationsformeln geparst, ein Log-File erstellt und später beschrieben. Während bei Variantenmodell 1 die SAT-Analysen selbst nur den kleineren Teil der Gesamtlaufzeit ausmachen, stellen diese den Großteil der Laufzeit bei Variantenmodell 2 dar[9].

Mit durchschnittlich ca. 0.14 Sekunden sind die Laufzeiten des SAT-Solvers für uns zufriedenstellend, zumal auch nach zahlreichen Durchläufen keine Analyse länger als eine halbe Sekunde dauerte. Dadurch kann ein Variantenmodell mit einer für die Industrie üblichen Größe in wenigen Minuten bzgl. mehrerer Inkonsistenzen geprüft werden. Alle Laufzeiten wurden dabei auf einem Windows 7-PC mit 2,53 GHz und 4 GB RAM gemessen.

[9]Zum Beispiel Konsistenzbedingung 5.2.2: 265 * 0,143 Sek. = 37,895 Sek. reine Analysezeit bei *VM* 1 entspricht 28% der Gesamtlaufzeit; 36 * 0,146 Sek. = 5,256 Sek. reine Analysezeit bei *VM* 2 entspricht 82% der Gesamtlaufzeit

7. Einordnung in die Literatur

In diesem Kapitel werden die hier vorgestellten Lösungsansätze in die Landschaft jener Arbeiten eingeordnet, welche sich mit ähnlichen Fragestellungen beschäftigen. Der erste Abschnitt befasst sich dabei mit der Evolution des Variantenmodells bzw. seiner Teilmodelle. Anschließend behandelt Abschnitt 7.2 verschiedene Ansätze, die Prüfungen des Variantenmodells auf Inkonsistenzen und Fehlmodellierungen beschreiben sowie deren Abgrenzung gegenüber den in Kapitel 5 dargestellten Konsistenzbedingungen.

7.1. Evolution des Variantenmodells

Viele Autoren beschäftigen sich mit Evolution in Software-Produktlinien und der Modifikationen der zugehörigen Variantenmodelle. Man stößt in der Literatur zu diesen Themen häufig auf den Begriff *Refactoring*. In unserem Verständnis ist das eine Änderung ohne Einfluss auf die Semantik des Variantenmodells. Wir sehen daher die in Kapitel 4 beschriebenen Änderungsprozesse nicht als Refactorings an, da sie sehr wohl die Semantik des Modells ändern. Trotzdem sind Arbeiten über Refactorings für uns interessant, da andere Autoren diesen Begriff unterschiedlich auslegen. Beispielsweise beschreibt ein verwandter Ansatz Merkmalmodell-Refactorings im Kontext der Software-Produktlinien als Änderungen, welche die Konfigurabilität des Modells nicht beschneiden [AGM+06]. Eine Modifikation des Merkmalmodells ist danach ein Refactoring, falls das geänderte Modell alle Variantenkonfigurationen des ursprünglichen Modells zulässt oder noch weitere dazu kommen. Dadurch kann ein Refactoring als eine Art Weiterentwicklung gesehen werden, wodurch der beschriebene Katalog von Refactorings zum Teil relevant für unsere Arbeit ist. Jedoch betrachten die Autoren im Gegensatz zu unserer Arbeit nur das Merkmalmodell. Weitere Arbeiten [Bor09, BTG10] behalten diese Definition bei, gehen bei ihrer formalen Darstellung aber über das Merkmalmodell hinaus und beschreiben auch weitere Artefakte, wie z. B. das Konfigurationswissen. Im Gegensatz zu deren allgemeineren Beschreibung von sprachen-unabhängigen „transformation templates“ liegt bei der vorliegenden Arbeit der Fokus auf dem Zusammenhang zwischen Evolution in einem komponentenbasierten Funktionsmodell und der nötigen Anpassung des zugehörigen Variantenmodells.

Bei einem weiteren Ansatz werden Änderungen am Merkmalmodell in vier Klassen einge-

teilt [TBK09]. Auch hier ist entscheidend, wie sich die Menge der Variantenkonfigurationen nach der Modifikation des Modells geändert hat. Nach diesem Kriterium können die Autoren alle Merkmalmodelländerungen in die Klassen Refactoring, Spezialisierung, Generalisierung oder beliebige Änderung einordnen und stellen einen Algorithmus vor, der für eine gegebene Änderung die Klasse ermittelt. Wie bereits oben erwähnt, sind für uns alle vier Klassen relevant, jedoch zielt unsere Arbeit eher darauf ab, genaue schrittweise Variantenmodellanpassungen zu beschreiben, um eine geänderte Variabilität in der Implementierung abzubilden. Schulze et al. beschreiben variantenerhaltende Refactorings in merkmalorientierten Software-Produktlinien [STKS12]. Angelehnt an Code-Refactorings aus der objektorientierten Programmierung [Fow99] stellen sie dabei vier Refactorings dar, die jeweils den zugehörigen Code von mehr als einem Merkmal betreffen und keine vorher gültigen Varianten beeinträchtigen. Die Autoren beziehen das Merkmalmodell in ihre Überlegungen ein, beschreiben aber im Gegensatz zur vorliegenden Arbeit nicht die Auswirkungen ihrer Refactorings auf dieses Modell.

Die Analyse des Linux Kernel von Lotufo et al. zeigt, dass die hier beschriebenen Evolutionsschritte auch in anderen Systemen auftreten [LSB+10]. Die Autoren beobachteten die Evolution des Variantenmodells des Linux Kernel über einen längeren Zeitraum und sammelten dabei Informationen über die inhaltlich und strukturell unterschiedlichen Arten von Änderungen und Refactorings am Merkmalmodell. Auch in dieser Arbeit wird es als große Herausforderung beschrieben, die Konsistenz von Implementierung und Variantenmodell zu bewahren, was uns in unserer Problembeschreibung bestätigt.

7.2. Konsistenzprüfungen des Variantenmodells

Es existieren viele Ansätze, die Analysen des Merkmalmodells beschreiben, um generelle Eigenschaften oder auch Inkonsistenzen zu ermitteln. Benavides et al. haben verschiedene Arbeiten auf diesem Gebiet zusammengefasst [BSC10]. Es finden sich dort beispielsweise Analysen, die ein Merkmalmodell auf die Existenz bzw. die Anzahl gültiger Selektionen prüfen. Eine weitere Arbeit beschreibt explizit Fehlmodellierungen in Merkmalmodellen [ML04]. Diese werden in drei verschiedene Klassen eingeteilt. Als *Redundanzen* werden mehrfach formulierte Variabilitätsinformationen bezeichnet, welche z. B. bei der Kombination von hierarchischen Beziehungen und Constraints entstehen können. Die Klasse der *Anomalien* fasst alle Fälle zusammen, in denen die Domäne widersprüchlich modelliert wurde. Muss etwa ein optionales Merkmal durch einen Constraint stets gewählt werden, so bildet das Modell nicht die eigentliche Variabilität der Funktionalität ab, die an dem betroffenen Merkmal hängt. Außerdem wird auf *Inkonsistenzen* eingegangen. Damit sind Fehlmodellierungen gemeint, wie z. B. ein Constraint, der den Widerspruch zwischen zwei obligatorischen Merkmalen ausdrückt. Solche Inkonsistenzen führen in der Regel dazu, dass sich für das Merkmalmodell keine gültige Selektion definieren lässt und dadurch keine Variante konfiguriert werden kann.

Durch den Fokus auf das Merkmalmodell sind die präsentierten Analysen für uns nur teilweise relevant. Viele der Konsistenzbedingungen aus Kapitel 5 beziehen sich auf den Zusammenhang zwischen Merkmalen und Konfigurationswissen, worauf in den oben genannten Arbeiten nicht eingegangen wird. Thaker et al. beschreiben dagegen einen Mechanismus für die „safe composition of product lines“ [TBKC07]. Auch dort werden SAT-Analysen vorgestellt, um zu prüfen, ob gültige Merkmalselektionen zu inkonsistenten Systemvarianten führen können. Während wir auf Inkonsistenzen innerhalb des Variantenmodells eingehen, werden dort Constraints beschrieben, die sich aus der Zusammensetzung von „feature modules“ und somit aus der Produktlinie selbst ergeben.

Auch andere Arbeiten beschäftigen sich mit dem Mapping zwischen Merkmalen und Komponenten. Reiser et al. präsentieren einen Ansatz für das Variantenmanagement hierarchischer komponentenbasierter Systeme [RKW09]. Dabei beschreiben Merkmalmodelle die Variabilität der jeweiligen Hierarchieebene, wobei mit sogenannten *configuration links* Konfigurationsinformationen zwischen den Modellen ausgetauscht werden können. Obwohl sich auch unsere Modellierung auf variable Komponenten bezieht, liegt der Unterschied darin, dass das gesamte System in *einem* Variantenmodell (und damit einem Merkmalmodell) abgebildet wird, auf das sich die vorgestellten Konsistenzbedingungen beziehen.

In [MRM+12] wird ein mengentheoretischer Formalismus vorgestellt, um ein Tracing zwischen Merkmalen (Spezifikation) und Komponenten (Implementierung) zu beschreiben. Darauf basierend werden Analysen präsentiert, welche mithilfe von QSAT-Solvern geprüft werden. Die Autoren arbeiten dabei mit einer anderen Beschreibung von Variabilität und der Verknüpfung zwischen Merkmalen und Komponenten. Während in deren Mapping ein Merkmal stets von einer oder mehreren Teilmengen von Komponenten implementiert werden kann, wird bei unserer Modellierung der Zusammenhang mithilfe einer booleschen Formel ausgedrückt. Diese kann beliebig komplex sein, sodass eine Komponente (in unserem Fall eine Variation) von mehreren Merkmalen abhängen kann oder in das System integriert wird, sobald ein Merkmal bzw. eine Kombination von Merkmalen *nicht* ausgewählt wird.

Einen ähnlichen Ansatz verfolgen die Autoren in [MHP+07]. Hier wird zwischen Produktlinien- und Softwarevariabilität unterschieden und eine separate Modellierung mit OVM- bzw. Merkmalmodellen vorgeschlagen. Auf einer formalisierten Beschreibung dieser Modelle und deren Beziehungen können anschließend Analysen automatisch ausgeführt werden. Teile dieser Analysen (beispielsweise nicht realisierbare Merkmalselektionen) können auf unsere Art der Modellierung übertragen werden, andere sind dagegen für uns nicht relevant (z. B. verschiedene Implementierungen für eine Merkmalselektion). Insgesamt zielen die vorgestellten Analysen eher auf die Frage ab, ob die Architektur der Produktlinie flexibel genug ist, um die geplanten Produkte zu konfigurieren. Der hier vorgestellte Ansatz konzentriert sich eher auf potentielle Fehler in den Modellen bzw. mögliche Ursachen dafür, damit der Konfigurationsprozess abgesichert und eine konsis-

tente Darstellung der Variabilität gewährleistet werden kann.

Zudem existieren Arbeiten, die sich direkt mit SAT-basierten Analysen auf Merkmalmodellen und der damit verbundenen Komplexität beschäftigen [TBK09]. Mit generierten Modellen wurden dort bzgl. der Performanz gute Ergebnisse erzielt und daher die Prognose formuliert, dass SAT-Analysen auf Merkmalmodellen tendenziell unproblematisch sind [MWC09]. Wir können das mit unseren Beobachtungen bestätigen, die auf Modellen entstanden sind, welche tatsächlich in der Praxis eingesetzt werden.

In einer weiteren Arbeit [KS00] werden ebenfalls SAT-Solver verwendet, um Inkonsistenzen in der Produktdokumentation, also auf Hardware-Ebene zu finden. Die vorgestellten Konzepte wurden ebenfalls in Zusammenarbeit mit der Daimler AG entwickelt und evaluiert. Die Autoren konnten dabei auf eine bestehende Datenbasis zurückgreifen, welche bereits in Aussagenlogik vorlag. Darauf aufbauend wurden Konsistenzbedingungen definiert und anschließend in Erfüllbarkeitsanalysen übersetzt, welche unseren Prüfungen auf der Software-Seite ähnlich sind (z. B. nicht baubare Konfigurationen). Durch die Implementierung dieser Analysen im Baubarkeits-Informations-System (BIS) konnten Inkonsistenzen effizient lokalisiert und die Qualität der Produktdokumentation verbessert werden. Eine ausführliche Beschreibung des Systems findet sich in [SKK03].

8. Evaluation

In diesem Kapitel wird das Lösungskonzept evaluiert, welches in den vorangegangenen Kapiteln vorgestellt wurde. Als wesentlicher Bestandteil der Evaluation wurde eine Expertenbefragung durchgeführt, auf welche im ersten Abschnitt eingegangen wird. Im Anschluss wird betrachtet, welchen Einfluss der Ansatz auf die Kriterien für Wartbarkeit nimmt, wie sie in Abschnitt 1.2 beschrieben werden. Dementsprechend beschäftigt sich der zweite Abschnitt mit Betrachtungen zu Modifizierbarkeit, Stabilität und Testbarkeit. Im Sinne der Praxistauglichkeit wird in Abschnitt 8.3, diskutiert, wie sich die erstellte Methodik in den Entwicklungsprozess der Daimler AG integrieren lässt. Abschließend fassen wir die Ergebnisse der Evaluation zusammen.

8.1. Expertenbefragung

Im Rahmen der Evaluation dieser Arbeit wurde eine Expertenbefragung durchgeführt, die sich mit den Themen Evolution in Entwicklungsartefakten, Änderungen an Variantenmodellen und ggf. dabei auftretenden Inkonsistenzen befasst. Befragt wurden dabei zehn Experten, die im Durchschnitt über mehr als sieben Jahre Erfahrung in der Variantenmodellierung verfügen, welche sie in zahlreichen Projekten erworben haben (s. dazu Frage 1 und 2 in Abschnitt A.3.1 im Anhang). Abgesehen von einem Teilnehmer aus dem akademischen Umfeld sind alle restlichen Experten Vertreter der Industrie, von denen die Mehrheit bei der Daimler AG beschäftigt ist.

Die Fragen mit den zugehörigen Antworten der Teilnehmer sind in Abschnitt A.3 im Anhang zusammengefasst. In der nun folgenden Betrachtung der Kriterien für Wartbarkeit werden an den geeigneten Stellen die entsprechenden Ergebnisse der Befragung interpretiert, um die Argumentation zu unterstützen.

8.2. Kriterien für Wartbarkeit

Dieser Abschnitt erklärt, welchen Einfluss die dargestellten Lösungsansätze auf die Kriterien für Wartbarkeit haben. In Abb. 8.1 wird nochmal daran erinnert, auf welche

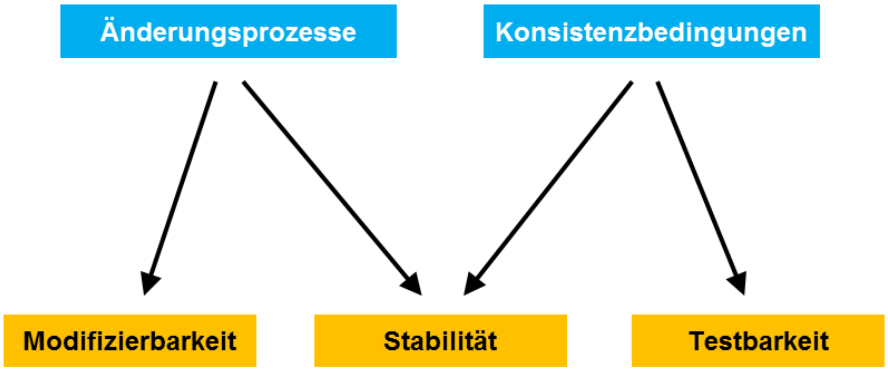


Abbildung 8.1.: Einflussnahme der Lösungsansätze auf die Kriterien

Kriterien sich die Änderungsprozesse und Konsistenzbedingungen auswirken.

8.2.1. Modifizierbarkeit

Die Modifizierbarkeit beschreibt im Kontext dieser Arbeit, wie aufwändig eine Anpassung des Variantenmodells an eine geänderte Umgebung ist. Dabei ist relevant, welche Arten von Anpassungen häufig sind und wie diese modelliert werden können. Die nächsten drei Unterabschnitte sollen erläutern, welchen Beitrag die in Kapitel 4 beschriebenen Änderungsprozesse für die Modifizierbarkeit des Variantenmodells leisten.

8.2.1.1. Einheitliche Modellierung

In Kapitel 4 wurden Änderungen des Funktionsmodells in Anpassungen des Variantenmodells übersetzt. Es ist damit eine grundsätzliche Modellierung vorgegeben, an der sich die Entwickler orientieren können. Diese muss lediglich in Ausnahmefällen an den konkreten Kontext angepasst werden. Wird dieser vorgeschlagenen Modellierung bei jeder Anpassung gefolgt, so versprechen wir uns davon eine einheitliche Struktur im Variantenmodell, die wiederum zur Verständlichkeit beiträgt. Das Modell kann dadurch auch leichter weitergegeben werden, was die Abhängigkeit von einzelnen Personen mindert.

8.2.1.2. Abdeckung der häufigsten Änderungen

Um Anpassungen am Variantenmodell unterstützen zu können, ist besonders relevant, welche Änderungen häufig auftreten. Daher wurden bei der Daimler AG verschiedene

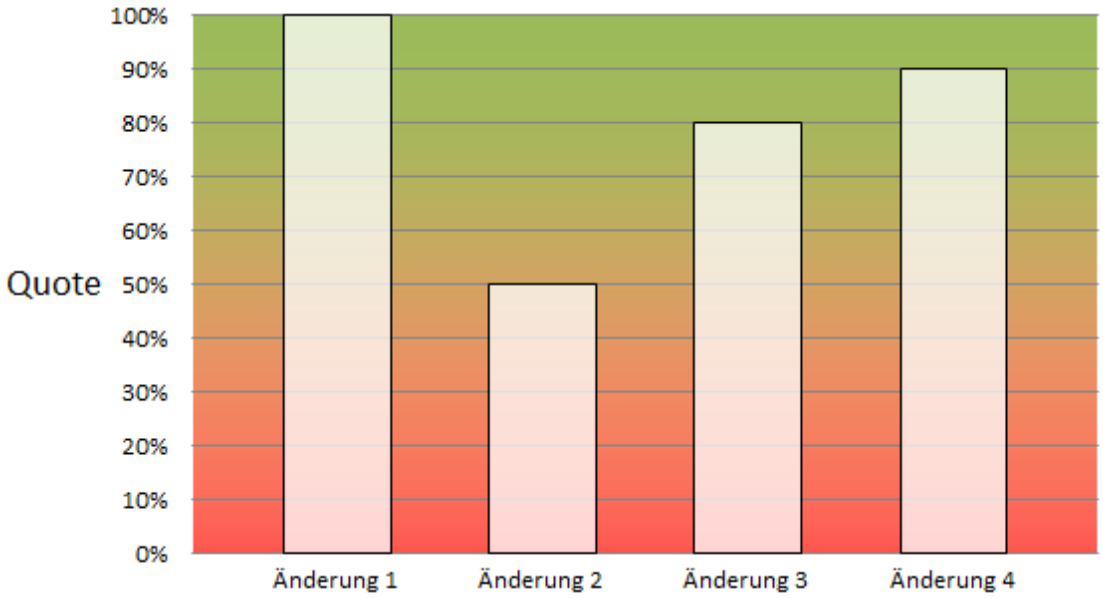


Abbildung 8.2.: Auftrittshäufigkeit der beschriebenen Änderungen

Evolutionsmuster in Funktionsmodellen untersucht, von denen die folgenden im Kapitel 4 in Anpassungen des Variantenmodells übersetzt wurden:

- Änderung 1: Neue optionale Komponente
- Änderung 2: Optionale Komponente löschen
- Änderung 3: Optionale Komponente wird obligatorisch
- Änderung 4: Neue alternative Komponente

Mithilfe von Frage 4 der Expertenbefragung (s. Abschnitt A.3.2.2) sollte bewertet werden, wie häufig diese Änderungen in der Praxis sind. Die Befragten mussten dabei für die obigen vier Änderungen angeben, ob diese in ihren Bereichen auftreten. Das Diagramm in Abb. 8.2 zeigt, wie viel Prozent der Experten die jeweilige Änderung aus ihrem Bereich kennen. Während die Änderung *Optionale Komponente löschen* zumindest in jedem zweiten Bereich auftritt, sind die restlichen in nahezu allen Bereiche bekannt.

Die Teilnehmer konnten außerdem in einem Freifeld weitere häufige Änderungen angeben, die nicht zur Wahl standen. Dabei wurden die folgenden drei Änderungen jeweils einmal angegeben. Diese lassen sich mit den bestehenden Änderungsprozessen abbilden bzw. können auf elementare Modelländerungen zurückgeführt werden, sodass eine Automatisierung nicht sinnvoll ist:

Tabelle 8.1.: Häufigkeit von variabilitätsrelevanten Änderungen

Häufigkeit	mind. täglich	mind. wöchentlich	mind. monatlich	seltener
# Antworten	2	2	4	2

- *Eine obligatorische Komponente wird optional:* Falls die obligatorische Komponente noch nicht im Variantenmodell abgebildet ist, lässt sich diese Änderung auch mit dem bestehenden Änderungsprozess *Neue optionale Komponente* durchführen, da das Merkmal, die Abhängigkeiten und der Variationspunkt ohnehin neu erstellt werden müssen. Wenn die obligatorische Komponente dagegen bereits modelliert und Abhängigkeiten in Form von Constraints definiert sind, um vorzubereiten, dass sie optional wird, so muss lediglich der Variationstyp geändert werden. Für diese elementare Operation lohnt sich eine Automatisierung nicht.

- *Es wird eine Abhängigkeit zwischen Merkmalen definiert:* Ein neuer Constraint zwischen Merkmalen ist eine elementare Operation, welche Bestandteil vieler Änderungsprozesse ist. Diesen Schritt isoliert zu automatisieren, würde im Vergleich zum Aufwand wenig Nutzen bringen und eine parallele Implementierung zu dem genutzten Variantenmodellierungswerkzeug pure::variants darstellen.

- *Eine optionale Komponente wird in mehrere aufgeteilt:* Hier existieren zwei Möglichkeiten, wie sich diese Änderung auf die Variabilität des Modells auswirkt. Wird die optionale Komponente in verschiedene Alternativen aufgeteilt, so kann der Änderungsprozess *Neue alternative Komponente (zu einer optionalen Komponente)* verwendet werden. Soll dagegen eine optionale Komponente in mehrere optionale Komponenten aufgeteilt werden, eignet sich der Änderungsprozess *Neue optionale Komponente*, um die geänderte Variabilität zu modellieren.

Es können somit alle zusätzlich angegebenen Änderungen mit den existierenden Änderungsprozessen abgedeckt werden. Die restlichen sieben der insgesamt zehn Teilnehmer der Befragung gaben keine weiteren Änderungen an, was die Annahme bekräftigt, dass in Kapitel 4 die Modellierung von Änderungen beschrieben wurde, die für die Praxis relevant sind.

Dass diese zudem sehr häufig auftreten können, zeigen die Ergebnisse von Frage 3 der Expertenbefragung (s. Abschnitt A.3.2.1). Dabei sollten die Teilnehmer angeben, wie häufig variabilitätsrelevante Änderungen in ihren Bereichen vorkommen. In Tabelle 8.1 sind die Ergebnisse zusammengefasst. Für die Mehrheit der Befragten sind demnach monatliche Änderungen die Regel. Jedoch haben nahezu alle Teilnehmer bei dieser Frage im Freifeld als Bemerkung angegeben, dass diese Häufigkeit stark davon abhängt, in welcher Projektphase man sich befindet. In frühen Phasen sind tägliche Änderungen demnach keine Seltenheit. Wird dagegen die gesamte Projektlaufzeit betrachtet, ergeben sich im Durchschnitt wöchentliche oder monatliche Änderungszyklen. Daher ist für die

Entwickler vor allem in bestimmten Projektphasen eine Unterstützung wichtig, die zum Teil tägliche Änderungen am Variantenmodell erleichtert.

8.2.1.3. Automatisierung der Änderungsprozesse

Die ersten beiden Unterabschnitte haben gezeigt, dass in dieser Arbeit für die häufigsten Änderungen am Variantenmodell eine einheitliche Vorgehensweise dargestellt wurde. Ein weiterer Beitrag zur Modifizierbarkeit leistet die Implementierung der Änderungsprozesse. Dadurch kann der Aufwand von umfangreicheren Anpassungen des Variantenmodells reduziert werden, da der Prototyp die meisten Teilschritte automatisch durchführt. Dem Benutzer bleiben lediglich Aufgaben, wie z. B. die Definition von Namen neuer Elemente oder die Entscheidung für oder gegen einen bestimmten Modellierungsstil.

Das soll zunächst beispielhaft an dem Änderungsprozess *optionale Komponente wird obligatorisch* verdeutlicht werden. In Abb. 8.3 ist dessen Aktivitätsdiagramm abgebildet. Alle automatisierten Schritte des Änderungsprozesses sind dabei grün eingefärbt, wohingegen jene Aktivitäten rot hervorgehoben sind, die eine Benutzerinteraktion erfordern. Wird vom Benutzer das entsprechende Skript ausgeführt, muss er lediglich entscheiden, wie die Constraints angepasst werden, welche das betroffene Merkmal enthalten. Falls außerdem Variationspunkte existieren, die neben dem entsprechenden Merkmal noch von weiteren abhängen, werden diese von der Implementierung nicht bearbeitet. Das Skript weist den Benutzer jedoch auf alle Variationspunkte hin, die er ggf. noch manuell ändern muss. Alle weiteren Aktivitäten des Änderungsprozesses werden von der Implementierung übernommen.

Der Automatismus übernimmt demnach den Großteil der elementaren Anpassungen, wie z. B. Modellelemente hinzufügen oder entfernen. Auch die Modellanalyse wird dadurch unterstützt, wenn beispielsweise ein Merkmal gelöscht werden soll und alle relevanten Abhängigkeiten (Constraints, Konfigurationsformeln) geprüft werden müssen. Würde der Entwickler die oben genannte Änderung dagegen manuell durchführen, wären die folgenden Arbeitsschritte nötig:

1. Im Merkmalmodell prüfen, ob das betroffene Merkmal Teil eines Constraints ist und ggf. den/die Constraint(s) anpassen bzw. löschen

2. Im Konfigurationsmodell prüfen, ob das Merkmal in einer Konfigurationsformel auftritt

3. Ggf. den entsprechenden Variationspunkt anpassen bzw. löschen

4. Den Variationstyp des Merkmals auf obligatorisch setzen

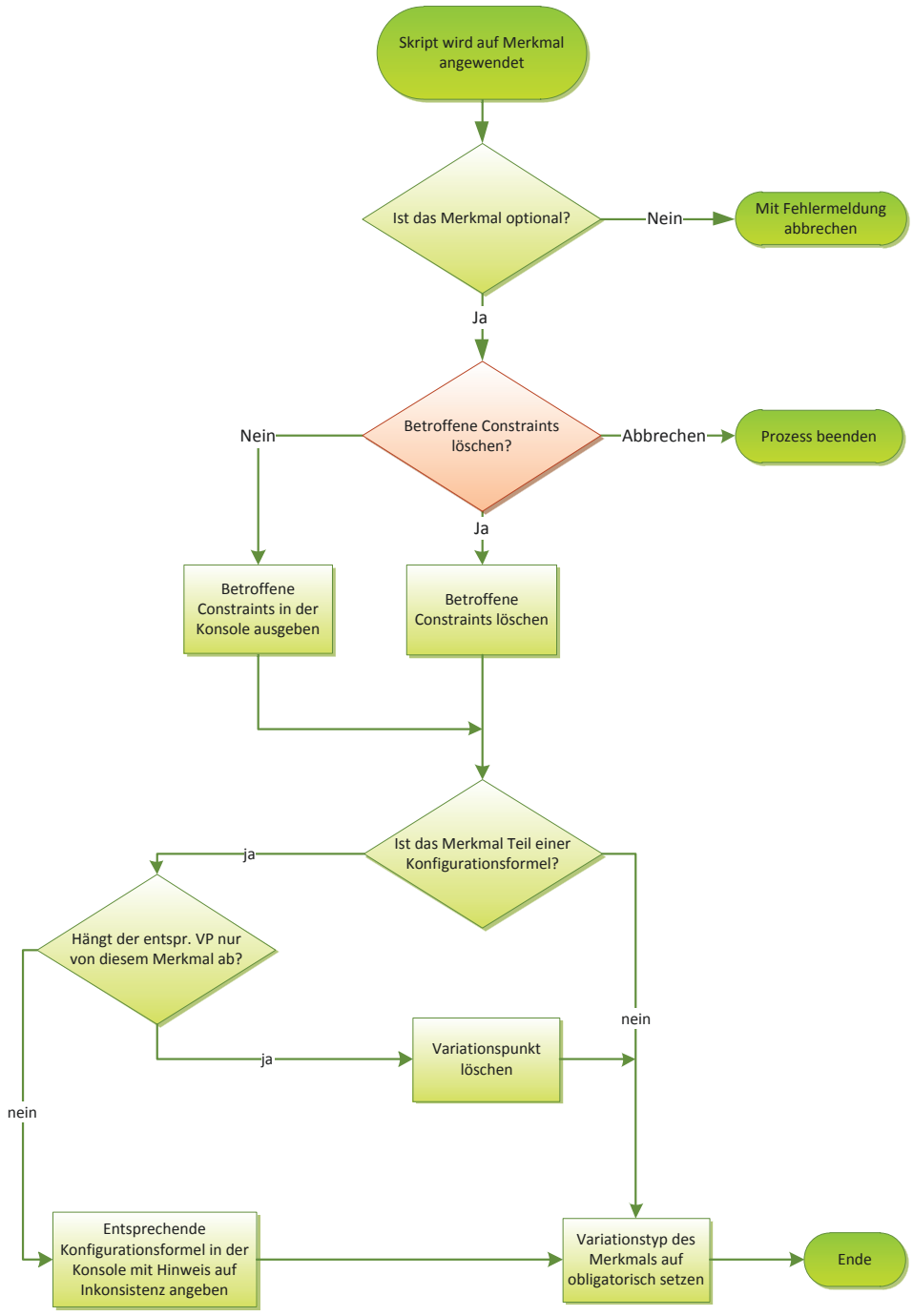


Abbildung 8.3.: Aktivitätsdiagramm des Änderungsprozesses *Optionale Komponente wird obligatorisch*

Ein weiteres Beispiel für die Automatisierung ist der Änderungsprozess *optionale Komponente löschen*[1]. Hier führt das Skript die komplette Änderung automatisch durch, nachdem der Benutzer es an dem gewünschten Merkmal ausgeführt hat. Bei einer manuellen Durchführung müssten, wie bei dem Beispiel oben, alle Constraints und Konfigurationsformeln mit dem entsprechenden Merkmal händisch analysiert, angepasst und anschließend das Merkmal gelöscht werden.

Auch wenn der Automatisierungsgrad dieser beiden Beispiele nicht bei allen Änderungsprozessen erreicht wurde[2], erhöht deren Implementierung trotzdem die Modifizierbarkeit des Variantenmodells. Die Entwickler können sich auf Aufgaben wie Namensgebung oder Designentscheidungen konzentrieren, wodurch der Aufwand von Änderungen reduziert wird.

8.2.2. Stabilität

Dieser Abschnitt stellt dar, inwiefern die in den vorangegangenen Kapiteln dargestellten Änderungsprozesse und Konsistenzbedingungen einen Beitrag für die Stabilität des Variantenmodells leisten. Die Stabilität als Kriterium für die Wartbarkeit beschreibt, wie hoch das Risiko für eine Inkonsistenz nach einer Änderung des Variantenmodells ist. Für die Risikoanalyse in der Softwareentwicklung sind die beiden Parameter Eintrittswahrscheinlichkeit und Ausmaß relevant [BE08], weshalb die Teilnehmer der Expertenbefragung gebeten wurden, diese für gegebene Inkonsistenzen zu bewerten.

Zunächst sollte die Wahrscheinlichkeit eingeschätzt werden, dass die folgenden Modellierungsfehler auftreten (Frage 7 in Abschnitt A.3.3.3):

- Fehler 1: Neue(s) Merkmal(e) anlegen und dessen/deren Verlinkung im Konfigurationsmodell vergessen (in der Konfigurationsformel eines Variationspunktes)
- Fehler 2: Geplanter Constraint zwischen Merkmalen vergessen
- Fehler 3: Semantischer Fehler (z. B. Merkmalname verwechseln) bei manueller Eingabe von Konfigurationsformeln
- Fehler 4: Die Konfigurationsformeln an einem Variationspunkt sind so formuliert, dass die eindeutige Konfiguration gefährdet ist[3]

[1] Dessen Aktivitätsdiagramm ist in Kapitel 6 abgebildet (s. Abb. 6.3).

[2] Vgl. dazu die Aktivitätsdiagramme aller Änderungsprozesse in Abschnitt A.2.

[3] Das ist der Fall, wenn für bestimmte Merkmalselektionen entweder gar keine oder mehrere Konfigurationsformeln erfüllt sind.

Im Anschluss sollte die Schwere der Auswirkung folgender Inkonsistenzen eingeschätzt werden (Frage 8 in Abschnitt A.3.3.3):

- Inkonsistenz 1: Ein für die Konfiguration nicht benötigtes Merkmal befindet sich im Merkmalmodell.
- Inkonsistenz 2: Merkmalselektionen, die man ausschließen möchte, sind gültig bzw. Selektionen, die man konfigurieren möchte, sind ungültig.
- Inkonsistenz 3: Der Konfigurationsprozess läuft ohne Fehlermeldung ab, wobei jedoch inhaltliche Fehler in der Variationspunktbelegung auftreten, sodass das Entwicklungsartefakt ggf. falsch konfiguriert wird.
- Inkonsistenz 4: Die Konfiguration schlägt fehl bzw. kann nicht vollständig durchgeführt werden. Die Ursache dafür ist jedoch unklar.

Um nun von der Eintrittswahrscheinlichkeit der Modellierungsfehler (aus Frage 7) auf die der Inkonsistenzen (aus Frage 8) zu schließen, benötigt man lediglich die Information, welcher Fehler zu welchen Inkonsistenzen führen kann:

- Fehler 1 $\rightarrow$ Inkonsistenz 1: Wird ein Merkmal angelegt, ohne es im Konfigurationsmodell zu verlinken, hat es vorerst keinen Einfluss auf die Konfiguration.
- Fehler 2 $\rightarrow$ Inkonsistenz 2: Constraints schränken die Auswahl von Merkmalen ein und haben daher großen Einfluss darauf, welche Selektionen gültig sind.
- Fehler 3 bzw. 4 $\rightarrow$ Inkonsistenzen 3 und 4: Ist eine Konfigurationsformel entweder semantisch (z. B. falsches Merkmal referenziert) oder syntaktisch fehlerhaft (z. B. eine Tautologie), kann das zu einer falsch konfigurierten Artefaktvariante oder aber auch zu einem Fehler im Konfigurationsprozess führen.

Damit existiert für jede der vier obigen Inkonsistenzen eine Einschätzung von Eintrittswahrscheinlichkeit und Ausmaß. Daraus ergibt sich das Risiko der Inkonsistenzen, welches in einer Risikomatrix illustriert werden kann. Da die Inkonsistenzen 3 und 4 jeweils durch die Fehler 3 und 4 verursacht werden können, haben wir sie für die grafische Darstellung der Risiken in die Inkonsistenzen 3.1 (verursacht durch Fehler 3) und 3.2 (verursacht durch Fehler 4) bzw. analog in 4.1 und 4.2 unterteilt.

In Abb. 8.4 ist die Risikomatrix der Inkonsistenzen dargestellt[4]. Dabei sind an X- bzw. Y-Achse diejenigen Skalen aufgetragen, welche die Teilnehmer der Befragung für die

[4]Die numerische Auswertung der dafür relevanten Fragen, welche die Grundlage der Matrix darstellt, findet sich im Anhang in Abschnitt A.3.3.3.

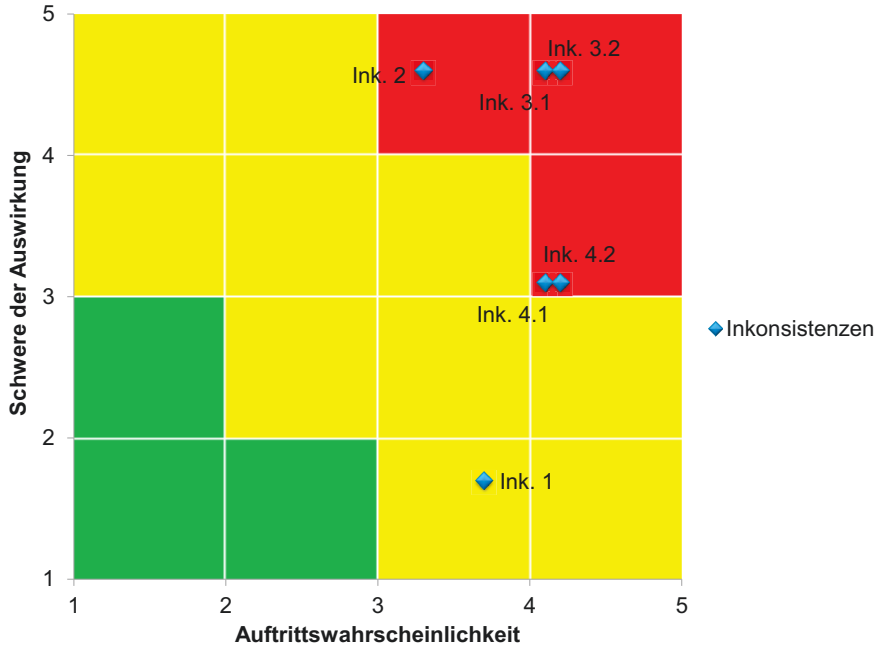


Abbildung 8.4.: Die Risikomatrix der Inkonsistenzen

Bewertung der entsprechenden Eigenschaften zur Verfügung hatten. Sie reichen für die Einschätzung der Eintrittswahrscheinlichkeit (X-Achse) von *1 = sehr unwahrscheinlich* bis *5 = sehr wahrscheinlich* bzw. für die Einschätzung der Schwere der Auswirkung (Y-Achse) von *1 = keine Auswirkung* bis *5 = sehr starke Auswirkung*.

Grundsätzlich ist ein Risikodiagramm in drei Risikozonen unterteilt. In der akzeptablen Risikozone, welche in der Matrix grün dargestellt ist, sind Maßnahmen nicht zwingend erforderlich. In der sogenannten ALARP-Zone (*As Low As Reasonably Practicable*) wird versucht, das Risiko mit angemessenem Aufwand zu reduzieren. Sie ist in der Matrix gelb dargestellt. Dahingegen sind in der rot gekennzeichneten inakzeptablen Zone Maßnahmen zur Risikominderung unbedingt erforderlich.

Es wird deutlich, dass das von Inkonsistenz 1 ausgehende Risiko noch vertretbar ist. Im Gegensatz dazu stellen alle weiteren Folgen von Fehlmodellierungen große Risiken für das Variantenmodell dar, sodass dringender Handlungsbedarf besteht. Damit geht von allen betrachteten Inkonsistenzen eine mittlere bis große Gefahr für die Stabilität des Variantenmodells aus. Diese Gefahr kann jedoch mithilfe der in dieser Arbeit vorgestellten Lösungskonzepte deutlich verringert werden. Die implementierten Änderungsprozesse tragen in folgendem Sinne dazu bei:

1. In Kapitel 4 wird beschrieben, welche Änderungsschritte nötig sind, um bestimmte

Anpassungen des Variantenmodells durchzuführen. Die Skripte, welche die Änderungsprozesse implementieren, führen diese Schritte durch, wobei der Benutzer an bestimmten Punkten eingebunden wird. Der Automatismus hat den Vorteil, dass dabei kein Zwischenschritt vernachlässigt wird. Die Fehler 1 und 2 (Verlinkung von neuem Merkmal im Konfigurationsmodell bzw. einen geplanten Constraint vergessen), deren Eintrittswahrscheinlichkeiten in Frage 7 bewertet wurden (s. oben), lassen sich somit vermeiden, da das Konfigurationsmodell stets automatisch angepasst bzw. der Benutzer explizit gefragt wird, ob er Constraints einfügen möchte.

2. Um die Abhängigkeit zwischen Merkmalen und Variationspunkten zu definieren, mussten bisher alle Konfigurationsformeln manuell eingegeben werden. Die obige Risikobewertung hat jedoch gezeigt, dass gerade Inkonsistenzen an den Konfigurationsformeln (3 und 4) wahrscheinlich sind und schwere Auswirkungen haben können. Dass die Bearbeitung dieser Modellelemente fehleranfällig ist, lassen auch die Antworten auf die Frage vermuten, wie bestimmte Modelleigenschaften die Verständlichkeit des Variantenmodells beeinflussen (Frage 6 der Befragung, s. Abschnitt A.3.3.2). Danach wird die Aussagenlogik in den Konfigurationsformeln als komplex wahrgenommen und hat im Vergleich zu anderen Modelleigenschaften eher einen negativen Einfluss. Aus diesen Gründen werden neue Konfigurationsformeln durch die Implementierung automatisch definiert, um potentielle Fehler dabei zu verhindern.

3. Generell besteht ein weiteres Risiko darin, dass der Entwickler eine größere Änderung am Variantenmodell beginnt, diese aber aufgrund äußerer Umstände nicht abschließen kann. Es bleibt somit unter Umständen ein inkonsistenter Modellzustand zurück. Wird dagegen die automatische Durchführung eines Änderungsprozesses zwischendurch abgebrochen, bleibt kein inkonsistentes Modell zurück, da die tatsächlichen Manipulationen erst zum Ende des Skriptes ausgeführt werden (s. dazu auch Abschnitt 6.2.1).

Somit unterstützt die Automatisierung der Änderungsprozesse die Entwickler dabei, Fehler bei einer Änderung zu vermeiden. Jedoch kann nicht jede beliebige Änderung automatisiert werden. Es kann durchaus dazu kommen, dass manuelle Anpassungen des Modells nötig sind. Um diese abzusichern, wurden die Konsistenzbedingungen definiert und deren Prüfung implementiert. Dadurch können beispielsweise die Fehler 1 und 4 von oben im Variantenmodell identifiziert und anschließend behoben werden, welche sonst unter Umständen zu schwerwiegenden Inkonsistenzen führen.

Die risikoreichsten Inkonsistenzen können demnach verhindert werden, indem die Benutzer Änderungen am Variantenmodell mithilfe der Änderungsprozesse durchführen und anschließend durch die Konsistenzbedingungen absichern. In diesem Sinne leisten die erarbeiteten Konzepte einen Beitrag zur Stabilität des Variantenmodells.

8.2.3. Testbarkeit

Wie effizient die Konsistenz des Variantenmodells geprüft werden kann, beschreibt das Kriterium Testbarkeit. Dieser Abschnitt erläutert, welchen Beitrag die Definition und Implementierung der Konsistenzbedingungen aus Kapitel 5 für die Testbarkeit leisten können. Dafür wurde zunächst im Rahmen der Expertenbefragung die automatische Analyse eines Beispielmodells mit der manuellen Prüfung verglichen. Im Anschluss wird die Performanz der Implementierung auf realistischen Variantenmodellen aus der industriellen Praxis betrachtet.

8.2.3.1. Analyse eines Beispielmodells

Zunächst vergleichen wir die manuelle Analyse mit der Konsistenzprüfung unseres Prototypen. Dafür haben wir im Rahmen unserer Expertenbefragung die Teilnehmer gebeten, ein beispielhaftes Variantenmodell zu untersuchen. Das Modell und die Ergebnisse finden sich im Anhang (s. Abschnitt A.3.3.4). Für unsere Betrachtungen ist vor allem relevant, dass die Befragten durchschnittlich 33,21 Sekunden benötigt haben, um einen Variationspunkt bzgl. der folgenden Inkonsistenzen zu prüfen:

- Kontradiktionen in Konfigurationsformeln (s. Abschnitt 5.2.1)
- Tautologien in Konfigurationsformeln (s. Abschnitt 5.2.2)
- Erfüllbare Konfigurationsformeln (s. Abschnitt 5.2.3)
- Eindeutige Konfigurationsformeln (s. Abschnitt 5.2.4)

Das angesprochene Beispielmodell (s. Abb. A.9 in Abschnitt A.3.3.4) hat 9 Merkmale im Merkmalmodell und 4 Variationspunkte im Konfigurationsmodell. Die Experten haben für die Prüfung des gesamten Modells somit im Durchschnitt 132,84 Sekunden benötigt, also etwas mehr als zwei Minuten.

Der in Kapitel 6 beschriebene Prototyp hat das gesamte Beispielmodell dagegen in ca. 3,4 Sekunden[5] analysiert. Die Implementierung ist auf diesem Modell demnach fast 40 mal so schnell wie die manuelle Prüfung und somit wesentlich effizienter. Hinzu kommt, dass die Befragten bei 10% ihrer Analysen falsch lagen, teilweise sogar fehlerhafte Variationspunkte für konsistent erklärten. Die automatische Analyse konnte dagegen jede Inkonsistenz finden und im Modell lokalisieren. Insgesamt kommt der Prototyp somit in

[5]Diese Laufzeit hat sich nach mehreren Durchläufen als Durchschnitt ergeben.

Tabelle 8.2.: Vergleich von manueller und automatischer Modellprüfung

	manuelle Modellprüfung	automatische Modellprüfung
benötigte Zeit	132,84 Sek.	3,4 Sek.
Fehlerquote	10%	0%

wesentlich kürzerer Zeit zu einem besseren Ergebnis. In Tabelle 8.2 wird dieser Vergleich nochmal zusammengefasst.

8.2.3.2. Analyse eines realistischen Variantenmodells

Das Beispielmodell, dessen Analyse im vorherigen Abschnitt ausgewertet wird, ist vergleichsweise klein. Wie die Antworten auf die Frage nach den durchschnittlichen Modellgrößen zeigt (Frage 5 der Expertenbefragung, s. Abschnitt A.3.3.1), sind typische Variantenmodelle in der industriellen Praxis deutlich größer. Es ist zu erwarten, dass mit der Modellgröße auch die benötigte Zeit einer manuellen Prüfung steigt. Das ist vor allem für die oben genannten Inkonsistenzen wahrscheinlich, da hier das Wissen um alle gültigen Selektionen und damit das Verständnis des gesamten Merkmalmodells erforderlich ist. Daher würde selbst die Analyse eines einzelnen Variationspunktes mehr Zeit erfordern, wenn das Merkmalmodell etwa 10 bis 11 mal so viele Merkmale hat[6]. Da mit der Größe in der Regel auch die Komplexität steigt, ist ein Rückgang der Fehlerquote bei größeren Modellen ebenfalls nicht zu erwarten.

Wie dagegen der Prototyp mit steigender Modellgröße skaliert, deutet bereits die Beschreibung in Abschnitt 6.3.2 an. Dort wurde die Performanz der implementierten Skripte auf zwei realistischen Variantenmodellen untersucht. Das größere der beiden Modelle besteht aus einem Merkmalmodell mit 298 Merkmalen und einem Konfigurationsmodell mit 69 Variationspunkten. Der Prototyp lief insgesamt ca. 320 Sekunden, um das gesamte Variantenmodell hinsichtlich der vier oben genannten Inkonsistenzen zu prüfen.

Wie bereits im vorherigen Abschnitt erwähnt, benötigt ein erfahrener Variantenmodellierer etwas mehr als eine halbe Minute für die Analyse eines einzelnen Variationspunktes aus unserem Beispielmodell. Geht man von der optimistischen Annahme aus, dass die befragten Experten ebenso viel Zeit für die Prüfung eines Variationspunktes in dem (viel größeren) realistischen Modell benötigen, so würde eine manuelle Prüfung des gesamten Modells immer noch ca. 34,5 Minuten in Anspruch nehmen. Vielmehr ist jedoch zu erwarten, dass die für eine Analyse benötigte Zeit überproportional ansteigt, wenn man

[6] Unser Beispielmodell hat 9 Merkmale, wohingegen nach den Angaben der befragten Experten in der industriellen Praxis Modelle mit ca. 100 Merkmalen der Regelfall sind (Frage 5 der Befragung, s. Abschnitt A.3.3.1).

statt eines simplen Beispielmodells mit 9 Merkmalen ein Variantenmodell aus der industriellen Praxis mit knapp 300 Merkmalen untersucht. Da dieses mit über 300 Constraints auch eine entsprechende Komplexität aufweist, ist auch ein Ansteigen der Fehlerquote wahrscheinlich.

8.3. Praxistauglichkeit der Methode

Dieser Abschnitt beschäftigt sich mit der Frage, wie die in dieser Arbeit beschriebenen Konzepte in den bestehenden Entwicklungsprozess bei der Daimler AG eingebunden werden können. Dabei geht es um die Änderungsprozesse aus Kapitel 4 und die Konsistenzbedingungen aus Kapitel 5, die merkmalbasierte Variantenmodelle anpassen bzw. Inkonsistenzen darin identifizieren. Wie bereits erwähnt, werden diese Modelle bei der Daimler AG in dem Eclipse-Plug-In pure::variants der Firma pure systems erstellt und verwaltet. Daher wurde die in Kapitel 6 beschriebene Implementierung der genannten Konzepte in dieses Werkzeug integriert. Den Entwicklern müssen somit nur die Skripte zur Verfügung gestellt werden. Nach einer kurzen Einführung in deren Verwendung können die Modellverantwortlichen mit den Änderungsprozessen ihre Modelle anpassen. Die Benutzung der Konsistenzbedingungen kann in den Arbeitsablauf integriert werden, indem beispielsweise die Entwickler dazu angehalten werden, für die Variantenmodelle vor jeder Datensicherung die Konsistenz per Analyse sicherzustellen.

Bei einem Einsatz für serienrelevante Variantenmodelle sind den Entwicklungsbereichen neben der Funktionalität jedoch noch weitere Aspekte der Implementierung wichtig. Davon abgesehen, dass die im Rahmen dieser Arbeit entstandene Implementierung prototypischen Charakter hat, sind dann beispielsweise auch die Wartung der Skripte oder die Haftung für eventuelle Fehler relevant. Dafür steht die Möglichkeit offen, die Funktionalitäten von der Firma pure systems implementieren zu lassen. Dabei würde auf Grundlage der hier beschriebenen Konzepte ein weiteres Modul innerhalb der bereits bestehenden Daimler-spezifischen Erweiterungen des Werkzeugs entstehen.

8.4. Zusammenfassung

Die Evaluation sollte zeigen, wie die Lösungskonzepte dieser Arbeit die Wartbarkeit des Variantenmodells beeinflussen können. Die Expertenbefragung und die Möglichkeit, den Prototypen auf realistische Modelle der industriellen Praxis anzuwenden, haben eine zentrale Rolle gespielt, um die eingangs erwähnten Kriterien genauer zu betrachten:

- Zunächst wurde die **Modifizierbarkeit** betrachtet. Wir haben gesehen, dass durch die Änderungsprozesse eine einheitliche Modellierung der häufigsten Änderungen

des Variantenmodells beschrieben wurde. Auch wenn bei der Benutzung der Änderungsprozesse in manchen Fällen noch eine manuelle Fertigstellung nötig ist, kann vor allem durch den Einsatz des Prototypen und der damit verbundenen Automatisierung elementarer Anpassungen der Änderungsaufwand reduziert werden.

- Um die **Stabilität** des Variantenmodells zu steigern, wurden zuerst mithilfe der befragten Experten die risikoreichsten Inkonsistenzen identifiziert. Die implementierten Änderungsprozesse unterstützen die Reduktion dieser Risiken, indem beispielsweise kein Einzelschritt einer Änderung vernachlässigt bzw. der Benutzer an potentiell mögliche Anpassungen erinnert wird. Auch die automatische Erstellung neuer Konfigurationsformeln hilft, von den Teilnehmern der Befragung als wahrscheinlich und schwerwiegend eingeschätzte Fehler zu vermeiden. Zwar kann die Implementierung bestehende Konfigurationsformeln nicht anpassen. Dafür hat der Benutzer jedoch die Möglichkeit, das Modell nach der manuellen Bearbeitung mithilfe der Konsistenzbedingungen zu analysieren, um kritische Inkonsistenzen zu lokalisieren.

- Diese Konsistenzbedingungen tragen auch zur **Testbarkeit** bei. Es hat sich gezeigt, dass erfahrene Entwickler fast 40 mal so viel Zeit wie der Prototyp benötigen, um in einem verhältnismäßig kleinen Beispiel-Variantenmodell Inkonsistenzen zu finden, deren Prüfung auf SAT-Analysen basieren. Auch die Fehlerquote spricht für eine erhöhte Testbarkeit: während die Implementierung fehlerlos blieb, kam jede zehnte Analyse der Entwickler zu einem falschen Ergebnis. Hinzu kommt, dass nicht nur Beispielmodelle, sondern auch deutlich größere Variantenmodelle der industriellen Praxis effizient geprüft werden können.

9. Ausblick

In diesem Kapitel werden weiterführende Inhalte diskutiert, die auf den in dieser Arbeit vorgestellten Konzepten für die Wartbarkeit von Variantenmodellen aufbauen. Wir sind bei unseren Überlegungen stets von Variabilität in einem Funktionsmodell ausgegangen. Zunächst wird im ersten Abschnitt betrachtet, ob und wie sich die erarbeiteten Ergebnisse auf die weiteren Entwicklungsphasen übertragen lassen. Der zweite Abschnitt beschäftigt sich mit dem Thema des durchgängigen Variantenmanagements. Dieses geht von der Hypothese aus, dass sich bei der Variantenmodellierung verschiedener Entwicklungsartefakte einer Domäne (z. B. Anforderungen, Funktionsmodellierung, Testspezifikationen, Parametrierung der Fahrerassistenzsysteme) inhaltliche Überschneidungen ergeben und schlägt daher vor, die gemeinsamen Aspekte artefaktübergreifend, also durchgängig zu modellieren. Der letzte Abschnitt thematisiert die Frage, wie man den Automatisierungsgrad der Änderungsprozesse weiter steigern kann, indem zusätzliche Informationen von der zugrunde liegenden Änderung im Funktionsmodell herangezogen werden.

9.1. Variantenmodellierung der weiteren Entwicklungsphasen

In dieser Arbeit werden Konzepte vorgestellt, um die Wartbarkeit eines Variantenmodells zu gewährleisten, welches die funktionalen Eigenschaften und die Konfigurationslogik eines modellbasierten Implementierungsartefaktes abbildet. Variabilität tritt jedoch im Entwicklungsprozess auch früher in den Anforderungen oder im Nachhinein bei der Softwareintegration, dem Testen oder der Applikation auf. Für diese Artefakte werden bei der Daimler AG in manchen Bereichen ebenfalls merkmalbasierte Variantenmodelle verwendet. Daher stellt sich die Frage, ob sich die Lösungsansätze dieser Arbeit auf die weiteren Entwicklungsphasen übertragen lassen.

Bei den Änderungsprozessen hängt das davon ab, ob sich die in Kapitel 4 dargestellten Evolutionsmuster in Funktionsmodellen auch z. B. in Anforderungen, Testdokumenten oder Parametersätzen wiederfinden. Dafür müsste für diese Artefakte ein analoger Begriff zu dem der Komponente definiert werden, wie er in dieser Arbeit benutzt wurde, um auf dessen Basis die variabilitätsrelevanten Änderungen im jeweiligen Artefakt zu beschreiben. Die Konsistenzbedingungen können grundsätzlich auf jedes merkmalbasierte Variantenmodell angewendet werden, in welchem die Variationspunkte per Aussagenlo-

gik konfiguriert werden. Es ist jedoch möglich, dass für andere Artefakte neben den hier vorgestellten noch weitere Modellprüfungen relevant sind.

9.2. Durchgängiges Variantenmanagement

Das Konzept des durchgängigen Variantenmanagements zielt darauf ab, die Methodik, den Prozess und die Werkzeuge zur Handhabung von Softwarevarianten über alle Artefakte des Entwicklungsprozesses zu vereinheitlichen. Darüber hinaus sollen gemeinsame Variabilitätsaspekte von z. B. Anforderungen, Funktionsmodellierung, Testdokumentation auch in einem übergreifenden Merkmalmodell festgehalten werden. Da die Variationspunkte artefaktspezifisch sind, wird dieses gemeinsame Merkmalmodell über die Konfigurationsformeln mit je einem Konfigurationsmodell pro Entwicklungsartefakt verlinkt[1].

Dieses durchgängige Variantenmodell muss nun die Variabilität verschiedener Artefakte abbilden und somit auch deren Evolution unterstützen. Das führt zu höheren Anforderungen bzgl. der Wartbarkeit des Modells. Für diesen Anwendungsfall ist eine Erweiterung der Änderungsprozesse nötig. Wird beispielsweise eine neue Funktion in den Anforderungen beschrieben, so hat dies Einfluss auf die Implementierung und die Testfälle. Dadurch wird nicht nur eine Anpassung des Merkmalmodells, sondern auch mehrerer Konfigurationsmodelle notwendig. Als Basis dafür müssen zudem ggf. isoliert erstellte Merkmalmodelle harmonisiert und Entwicklungsprozessadaptionen durchgeführt werden, um die bereichsübergreifende Dokumentation eines zentralen Merkmalmodells zu ermöglichen [MR13].

9.3. Automatisierter Update-Prozess

Ein weiterer Ausbau der hier vorgestellten Konzepte beschäftigt sich mit der Frage, ob man den Prozess der Anpassung des Variantenmodells weiter automatisieren kann. Dafür wären zusätzliche Informationen nötig, wie beispielsweise die Namen neuer Komponenten bzw. Merkmale, die in unseren Änderungsprozessen manuell eingegeben werden. In [TD13] sind Refactorings auf Simulink-Modellen mit dem Ziel beschrieben, diese zu automatisieren. Darunter finden sich ebenfalls Änderungen, die die Variabilität des Modells betreffen. Diese müssen auf die in Kapitel 4 dargestellten Änderungsprozesse abgebildet werden, wodurch die Anpassung des Variantenmodells von dem Funktionsmodell-

[1] Dieser Ansatz wird im BMBF geförderten Entwicklungsprojekt SPES_XT von der Daimler AG und weiteren industriellen und akademischen Partnern erarbeitet und evaluiert. In [GRMM+13] stellen die Autoren dar, welche Anforderungen aus der industriellen Praxis dabei zu Grunde gelegt wurden.

Refactoring ausgelöst werden könnte. Ein Entwickler würde anschließend lediglich die Änderung im Variantenmodell kontrollieren, wobei ihn wiederum die Konsistenzbedingungen unterstützen können.

A. Anhang

A.1. Aussagenlogische Beweise

Dieser Abschnitt führt die Beweise von aussagenlogischen Lemmata aus, die in der Arbeit vor allem für die Beschreibung der Konsistenzbedingungen in Kapitel 5 verwendet werden.

A.1.1. Äquivalenzbeweis der Darstellungen einer Merkmalauswahl

In Abschnitt 3.3 wurden zwei verschiedene Möglichkeiten vorgestellt, eine Auswahl von Merkmalen zu formalisieren. Einerseits kann eine Belegung verwendet werden, also eine Abbildung der Merkmale in M auf *true* oder *false*. In Abschnitt 3.1 haben wir gesehen, dass

$$B(M) = \{f \mid f : M \to \{true, false\}\}$$

die Menge dieser Belegungen zusammenfasst.

In der vorliegenden Arbeit wurde jedoch die Darstellung einer Merkmalauswahl als aussagenlogische Formel bevorzugt. Eine Selektion der Merkmalmenge $M = \{m_1, \ldots, m_n\}$ ist in Abschnitt 3.3 definiert als

$$S = s_1 \wedge \cdots \wedge s_n \text{ mit } s_i = m_i \text{ oder } s_i = \neg m_i \text{ für } 1 \leq i \leq n,$$

wobei mit $S(M)$ die Menge aller Selektionen von M bezeichnet wurde. Wir zeigen nun, dass diese beiden Darstellungen äquivalent sind.

Lemma 1. *Die Mengen $B(M)$ und $S(M)$ lassen sich bijektiv (also eineindeutig) aufeinander abbilden.*

Beweis. Wir zeigen die Aussage, in dem wir zunächst eine Abbildung $\tau : B(M) \to S(M)$ definieren. Sei dazu $f \in B(M)$ eine beliebige Belegung der Menge M. Dann ist $\tau(f) \in S(M)$ mit

$$\tau(f) = s_1 \wedge \cdots \wedge s_n \text{ mit } s_i := \begin{cases} m_i, & \text{falls } f(m_i) = \mathit{true} \\ \neg m_i, & \text{falls } f(m_i) = \mathit{false} \end{cases}$$

Jetzt zeigen wir, dass τ eine injektive Abbildung ist. Das ist der Fall, wenn die Bilder zweier ungleicher Elemente aus $B(M)$ ebenfalls nicht gleich sind:

$$\tau \text{ ist injektiv, falls gilt: } f, f' \in B(M) \text{ mit } f \neq f' \Rightarrow \tau(f) \neq \tau(f')$$

Seien also f und f' zwei beliebige Elemente aus $B(M)$ mit $f \neq f'$. Dann existiert ein $i \in \{1, \ldots, n\}$ mit $f(m_i) \neq f'(m_i)$. Damit ist

$$\tau(f) = s_1 \wedge \cdots \wedge s_n \text{ bzw. } \tau(f') = s'_1 \wedge \cdots \wedge s'_n \text{ mit } s_i \neq s'_i,$$

woraus $\tau(f) \neq \tau(f')$ folgt. Somit ist τ injektiv. Nun ist eine injektive Abbildung zwischen endlichen Mengen bijektiv, falls diese die gleiche Mächtigkeit haben. Wie oben erwähnt, enthält $B(M)$ alle Abbildungen von der n-elementigen Menge M auf die Menge $\{\mathit{true}, \mathit{false}\}$. Dafür gibt es insgesamt 2^n verschiedene Möglichkeiten. $S(M)$ ist die Menge aller Selektionen, also aller aussagenlogischen Formeln mit n Literalen, die negiert oder nicht negiert sein können. Daher ergeben sich hier 2^n unterschiedliche Selektionen. Die beiden Mengen sind damit gleichmächtig, woraus folgt, dass τ bijektiv ist. Es existiert also eine bijektive Abbildung von $B(M)$ nach $S(M)$, womit das Lemma gezeigt ist. □

Bemerkung. *Die Umkehrabbildung $\tau^{-1} : S(M) \to B(M)$ gibt an, auf welche Belegung eine beliebige Selektion abgebildet wird:*

Sei $S \in S(M)$ eine Selektion, so ist $\tau^{-1}(S) \in B(M)$ mit

$$\tau^{-1}(S)(m_i) = \begin{cases} \mathit{true}, & \mathit{falls}\ s_i = m_i \\ \mathit{false}, & \mathit{falls}\ s_i = \neg m_i \end{cases}$$

Dabei ist zu bemerken, dass für eine beliebige Selektion $S \in S(M)$ das Urbild $f_S = \tau^{-1}(S)$ jene Belegung ist, für die gilt: $f_S(S) = \mathit{true}$. Darüber hinaus ist f_S die einzige Belegung mit dieser Eigenschaft. Denn existiert eine weitere Belegung g_S mit $g_S(S) = \mathit{true}$, so folgt

$$\begin{aligned} f_S(S) = f_S(s_1 \wedge \cdots \wedge s_n) = \mathit{true} \ &\Rightarrow\ f_S(s_1) = \cdots = f_S(s_n) = \mathit{true} \\ \mathit{und}\ \ g_S(S) = g_S(s_1 \wedge \cdots \wedge s_n) = \mathit{true} \ &\Rightarrow\ g_S(s_1) = \cdots = g_S(s_n) = \mathit{true} \end{aligned}$$

Damit folgt

$$f_S(s_i) = g_S(s_i) \text{ für alle } 1 \leq i \leq n.$$

Da jedoch entweder $s_i = m_i$ oder $s_i = \neg m_i$ gilt, haben f_S und g_S die gleichen Werte auf allen Elementen von M und sind somit identisch. Wir werden diese Eindeutigkeit später verwenden.

A.1.2. Existenz einer erfüllenden Selektion

Dieser Abschnitt liefert den Beweis für das folgende Lemma:

Lemma 2. *Sei $\phi \in F(M)$ eine beliebige aussagenlogische Formel mit Literalen aus M. Dann ist ϕ genau dann erfüllbar, wenn eine Selektion $S \in F(M)$ existiert, sodass $S \wedge \phi$ erfüllbar ist.*

Beweis. Wir beginnen mit der Hinrichtung: Sei also $\phi \in F(M)$ eine erfüllbare Formel. Nach Definition (s. Abschnitt 3.1) existiert dann eine Belegung $f_\phi : M \to \{true, false\}$ mit $f_\phi(\phi) = true$. Im vorherigen Lemma haben wir gesehen, dass zwischen Belegungen und Selektionen von M eine Bijektion besteht. Verwendet man dessen Abbildungsvorschrift, ergibt sich die in diesem Lemma gesuchte Selektion aus f_ϕ:

$$S_\phi := \tau(f_\phi)$$

Jetzt bleibt noch zu zeigen, dass $S_\phi \wedge \phi$ erfüllbar ist, dass also eine Belegung f mit $f(S_\phi \wedge \phi) = true$ existiert. Dafür eignet sich wiederum f_ϕ. Zunächst gilt $f_\phi(S_\phi \wedge \phi) = f_\phi(S_\phi) \wedge f_\phi(\phi)$. Dabei ist $f_\phi(S_\phi) = f_\phi(s_1 \wedge \cdots \wedge s_n) = f_\phi(s_1) \wedge \cdots \wedge f_\phi(s_n)$, wobei jedoch gilt:

$$f_\phi(s_i) = \begin{cases} f_\phi(m_i), & \text{falls } f_\phi(m_i) = true \\ f_\phi(\neg m_i), & \text{falls } f_\phi(m_i) = false \end{cases}$$

Somit ist $f_\phi(s_i) = true$ für alle $i \in \{1, \ldots, n\}$ und es gilt $f_\phi(S_\phi) = true$. Dass $f_\phi(\phi) = true$ ist, haben wir bereits zu Beginn gesehen. Mithin ist $f_\phi(S_\phi \wedge \phi) = f_\phi(S_\phi) \wedge f_\phi(\phi) = true$ und somit $S_\phi \wedge \phi$ erfüllbar.

Damit ist die Hinrichtung gezeigt. Die Rückrichtung ist weniger aufwändig: Aus der Erfüllbarkeit von $S \wedge \phi$ folgt auch die von ϕ. □

Bemerkung. *Aus der Rückrichtung des gerade bewiesenen Lemmas ergibt sich durch Kontraposition die folgende Aussage, welche in der Arbeit an mehreren Stellen verwendet wird: Ist eine beliebige Formel $\phi \in F(M)$ nicht erfüllbar, so existiert keine Selektion $S \in F(M)$, sodass $\phi \wedge S$ erfüllbar ist.*

A.1.3. Erfüllbarkeit der Konjunktion

In diesem Abschnitt wird das folgende Lemma bewiesen:

Lemma 3. *Seien $\phi_A, \phi_B \in F(M)$ zwei beliebige aussagenlogische Formeln und $S \in F(M)$ eine Selektion. Sind dann die Formeln $S \wedge \phi_A$ und $S \wedge \phi_B$ erfüllbar, so ist es auch die Formel $S \wedge \phi_A \wedge \phi_B$.*

Beweis. Sind $S \wedge \phi_A$ und $S \wedge \phi_B$ erfüllbar, so existieren zwei Belegungen f_A und f_B mit

$$f_A(S \wedge \phi_A) = f_B(S \wedge \phi_B) = true$$

Diese zwei Belegungen erfüllen damit insbesondere auch die Selektion S. In der Bemerkung nach dem Lemma in Abschnitt A.1.1 haben wir gesehen, dass die erfüllende Belegung einer Selektion eindeutig ist. Damit folgt $f_A = f_B$ und

$$f_A(S \wedge \phi_A \wedge \phi_B) = true,$$

womit $S \wedge \phi_A \wedge \phi_B$ erfüllbar ist, was zu zeigen war. □

A.1.4. Erfüllbarkeit der Negation

Hier wird das folgende Lemma bewiesen:

Lemma 4. *Sei $\phi \in F(M)$ eine beliebige aussagenlogische Formel und $S \in F(M)$ eine Selektion. Dann ist $S \wedge \phi$ genau dann erfüllbar, wenn $S \wedge \neg\phi$ nicht erfüllbar ist.*

Beweis. Wir zeigen zunächst die Hinrichtung: Ist $S \wedge \phi$ erfüllbar, so existiert nach Definition eine Belegung $f : M \to \{true, false\}$ mit $f(S \wedge \phi) = f(S) \wedge f(\phi) = true$, also $f(S) = true$ und $f(\phi) = true$.

Wir nehmen nun an, dass dann auch $S \wedge \neg\phi$ erfüllbar ist und führen dies zum Widerspruch. Nach dieser Annahme existiert ein f' mit $f'(S \wedge \neg\phi) = f'(S) \wedge f'(\neg\phi) = true$, also $f'(S) = true$ und $f'(\neg\phi) = true$ für die gleiche Selektion S wie oben. Aus $f(S) = f'(S) = true$ folgt damit $f = f'$, wie wir in der Bemerkung nach dem ersten Lemma (s. Abschnitt A.1.1) gesehen haben. Damit muss aber

$$f(\phi) = f'(\neg\phi) = f(\neg\phi) = \neg f(\phi)$$

gelten, was den gewünschten Widerspruch liefert.

Für die Rückrichtung gehen wir davon aus, dass $S \wedge \neg\phi$ nicht erfüllbar ist. In der Bemerkung nach dem Lemma in Abschnitt A.1.1 hat sich gezeigt, dass sich mithilfe der Umkehrfunktion der bijektiven Abbildung τ für jede Selektion die eindeutig bestimmte erfüllende Belegung ermitteln lässt. Sei also $f_S = \tau^{-1}(S)$ diese Belegung. Aus $f_S(S) = true$ und $f_S(S \wedge \neg\phi) = false$ (da nicht erfüllbar) folgt $f_S(\neg\phi) = false$. Damit ist

$$f_S(\phi) = \neg f_S(\neg\phi) = true$$

und es gilt $f_S(S \wedge \phi) = f_S(S) \wedge f_S(\phi) = true$. Mithin ist $S \wedge \phi$ erfüllbar und das Lemma ist gezeigt. □

A.2. Aktivitätsdiagramme der Änderungsprozesse

Dieser Abschnitt beinhaltet die Aktivitätsdiagramme der Änderungsprozesse, welche die Grundlage für deren Implementierung darstellen. Der Änderungsprozess *optionale Komponente löschen* wurde bereits in Abschnitt 6.2 als Beispiel beschrieben. Daher finden sich hier die Aktivitätsdiagramme für die restlichen Änderungsprozesse

- *Optionale Komponente wird obligatorisch* und
- *Neue Alternative (zu einer bestehenden Komponente)*,

welcher wie in Kapitel 4 noch danach unterschieden wird, ob die bestehende Komponente obligatorisch, optional oder alternativ ist. Außerdem wurde für die Darstellung in den Diagrammen der Unterprozess *Constraints einfügen* aus bestimmten Änderungsprozessen extrahiert und isoliert beschrieben.

Optionale Komponente wird obligatorisch

In Abb. A.1 ist das Aktivitätsdiagramm für den Änderungsprozess *Optionale Komponente wird obligatorisch* dargestellt. Nachdem man das entsprechende Skript auf ein Merkmal anwendet, wird zunächst der Kontext geprüft. Anschließend entscheidet der Benutzer über die Behandlung der Constraints, bevor im Konfigurationsmodell jene Variationspunkte gelöscht werden, die lediglich von dem ausgewählten Merkmal abhängen. Danach wird schließlich der Variationstyp des Merkmals geändert.

Neue alternative Komponente

Wie bereits in Abschnitt 6.2 erwähnt wurde, sind für diesen Änderungsprozess drei verschiedene Skripte entstanden, die sich darin unterscheiden, welchen Variationstyp die bestehende Komponente hat. Alle diese Skripte werden an dem Merkmal ausgeführt, welches die bestehende Komponente repräsentiert und neue Alternativen erhalten soll. Zu Beginn prüft das Skript, ob es im richtigen Kontext ausgeführt wurde, nimmt die Namen der neuen Merkmale vom Benutzer entgegen und stellt die Möglichkeit bereit, neue Constraints einzufügen. Anschließend muss der Benutzer entscheiden, ob die neuen Alternativen das bestehende (für den Prozess ausgewählte) Merkmal ersetzen sollen oder nicht. Abhängig davon werden Constraints bzw. Konfigurationsformeln behandelt, in welchen dieses Merkmal auftritt. Während der erste Teil in allen drei Skripten nahezu gleich abläuft, hängen die Anpassungen der Abhängigkeiten im letzten Teil davon ab, ob das bestehende Merkmal obligatorisch, optional oder alternativ ist.

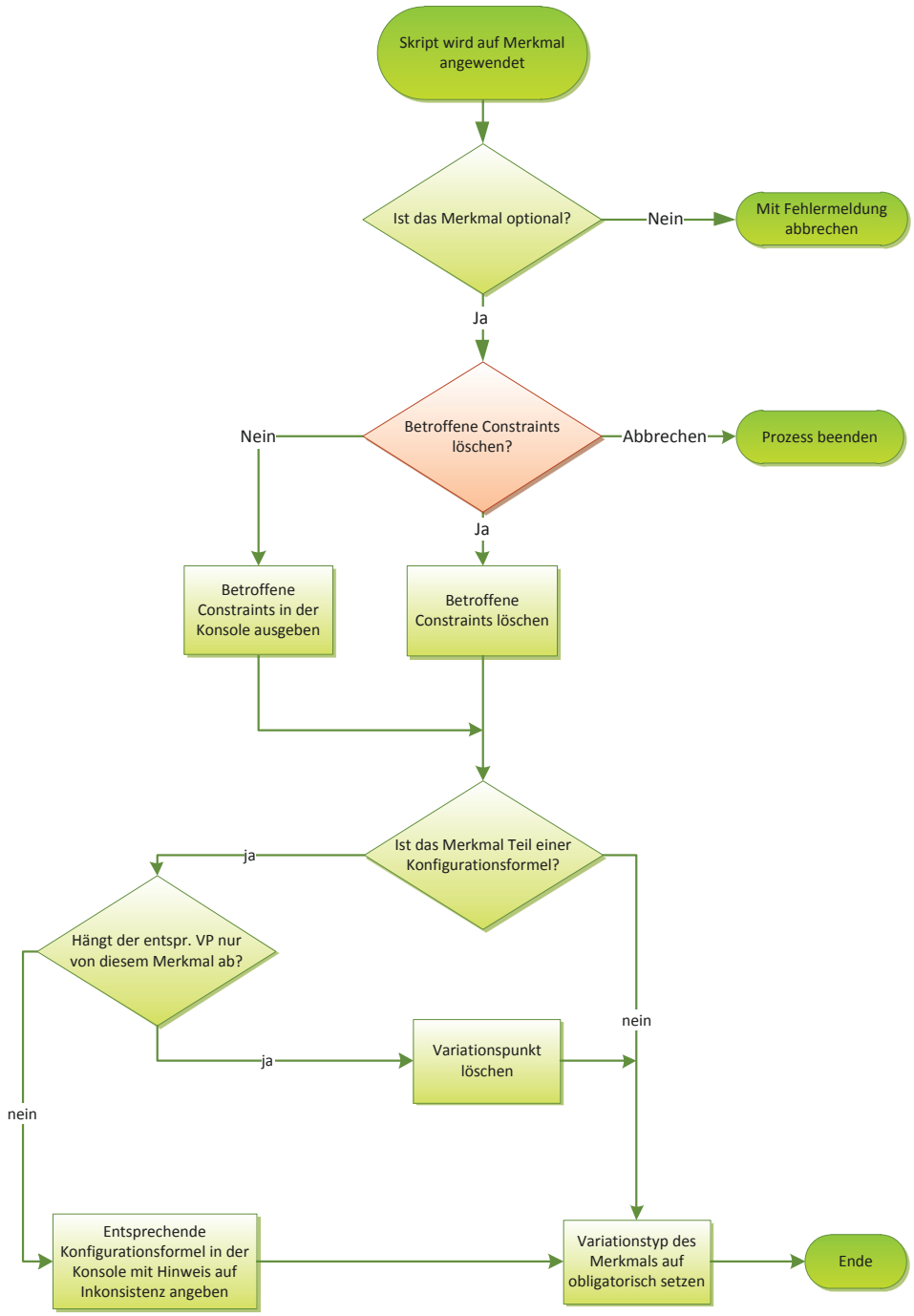


Abbildung A.1.: Der Änderungsprozess *Optionale Komponente wird obligatorisch*

Die nachfolgenden Abbildungen zeigen die Aktivitätsdiagramme der Änderungsprozesse *Neue Alternative zu einer*

- *obligatorischen Komponente*: Abb. A.2 und A.3
- *optionalen Komponente*: Abb. A.4 und A.5
- *alternativen Komponente*: Abb. A.6 und A.7

Unterprozess: Constraint einfügen

In Abb. A.8 ist das Aktivitätsdiagramm für den Unterprozess *Constraint einfügen* dargestellt.

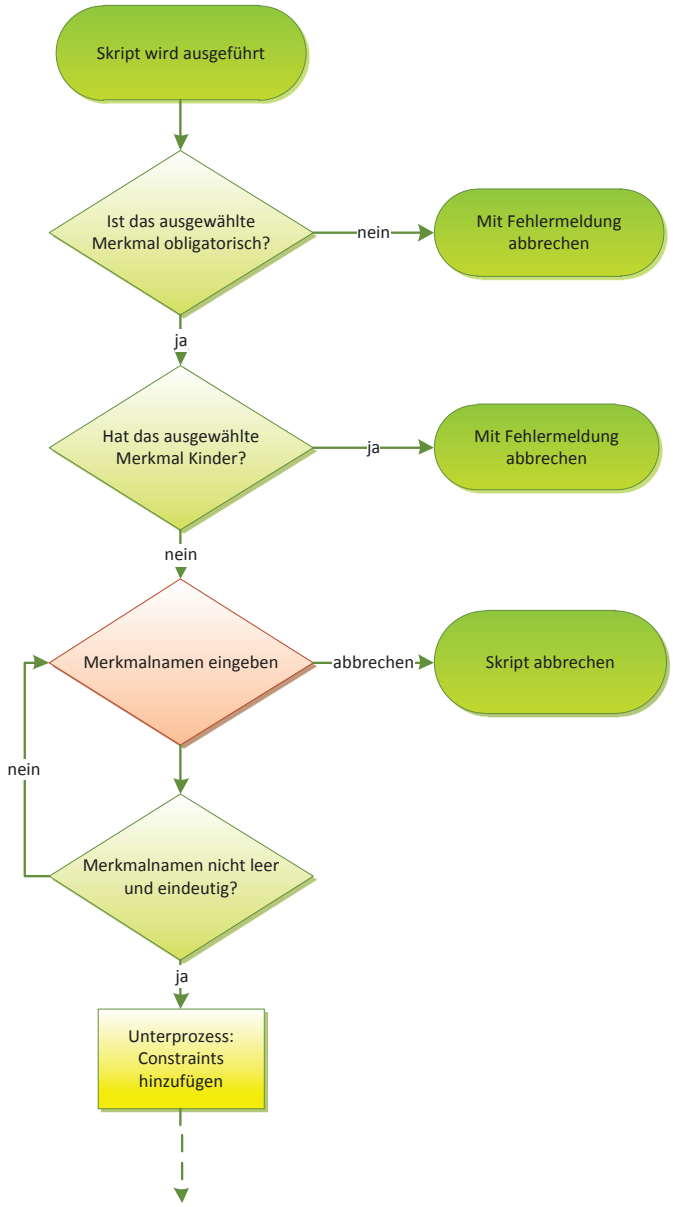


Abbildung A.2.: Der Änderungsprozess *Neue Alternative zu einer obligatorischen Komponente* - Teil 1

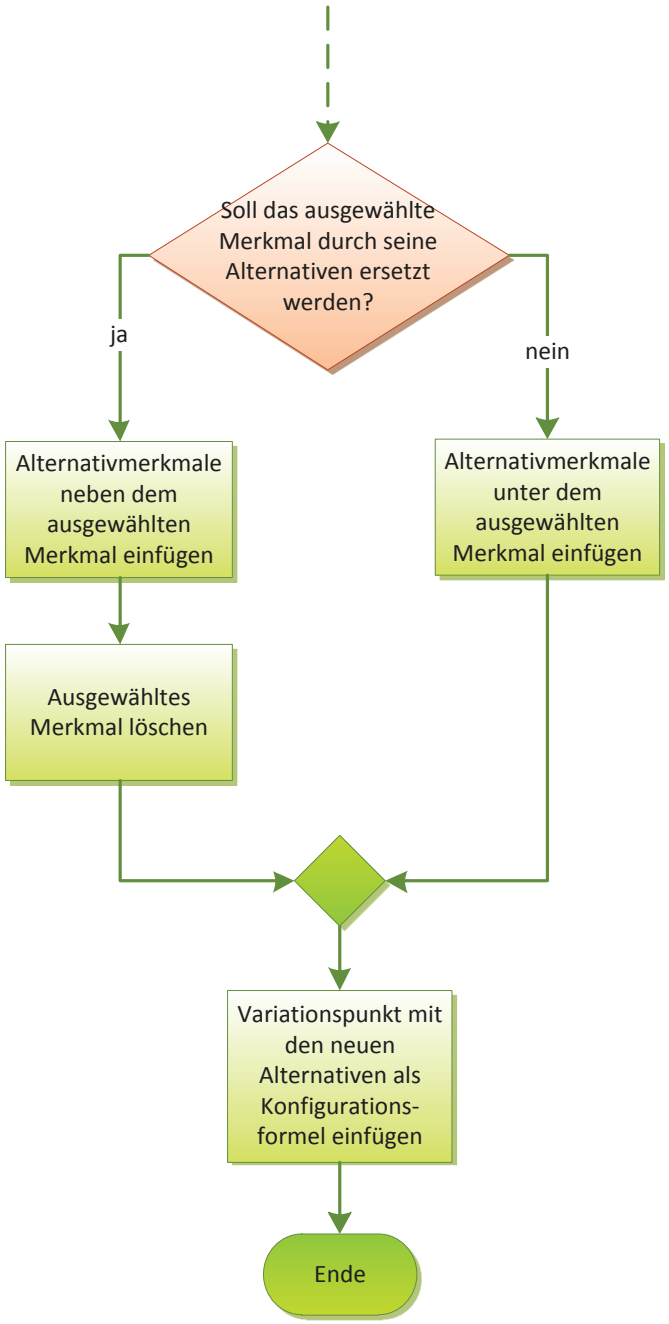


Abbildung A.3.: Der Änderungsprozess *Neue Alternative zu einer obligatorischen Komponente* - Teil 2

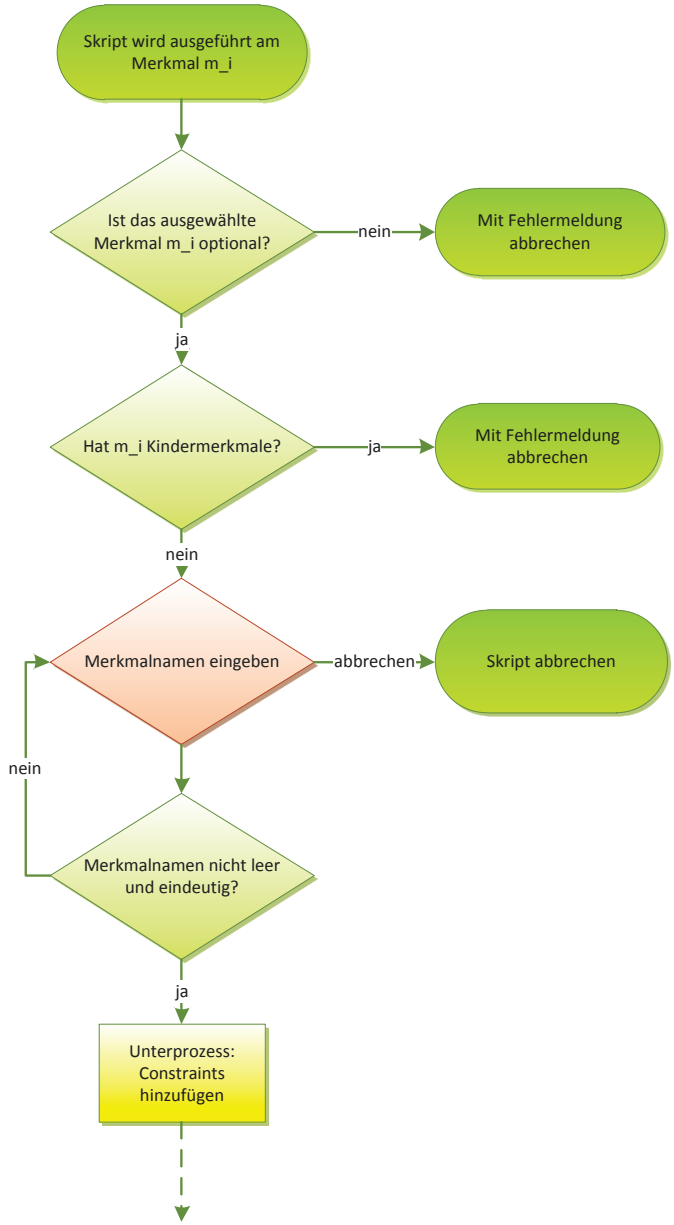


Abbildung A.4.: Der Änderungsprozess *Neue Alternative zu einer optionalen Komponente* - Teil 1

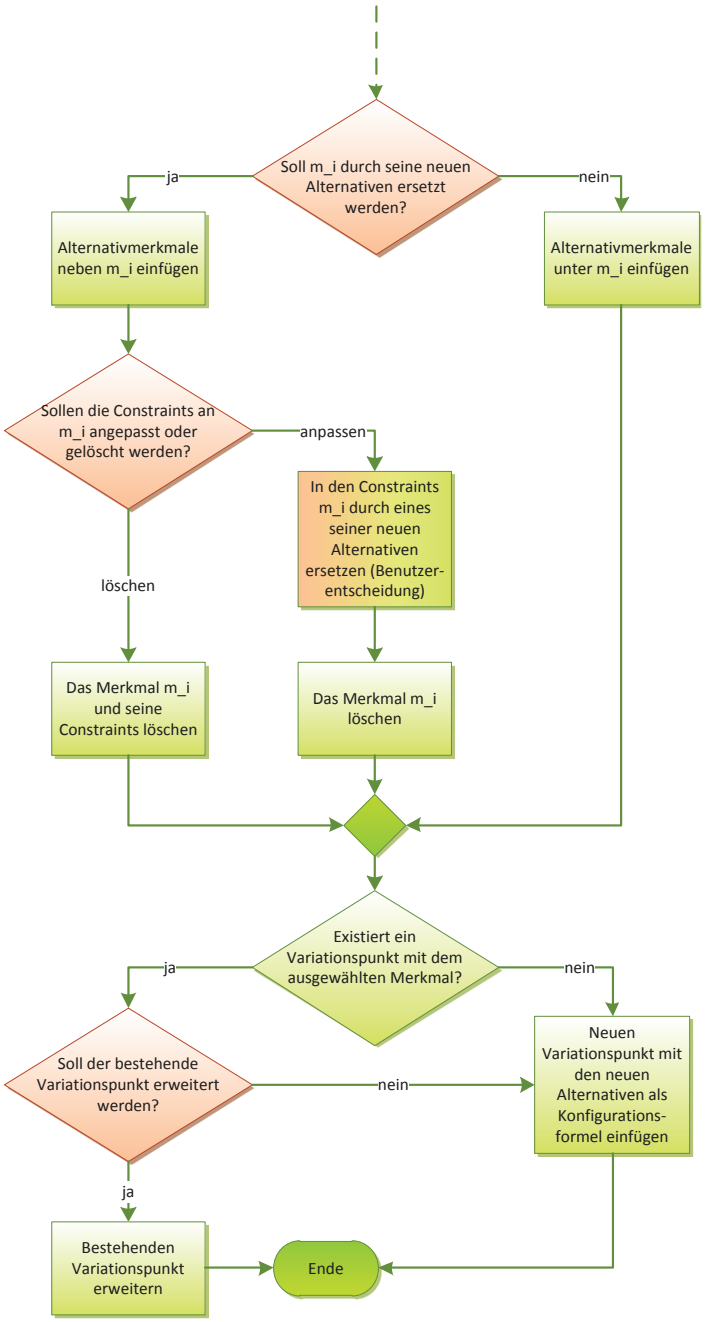


Abbildung A.5.: Der Änderungsprozess *Neue Alternative zu einer optionalen Komponente* - Teil 2

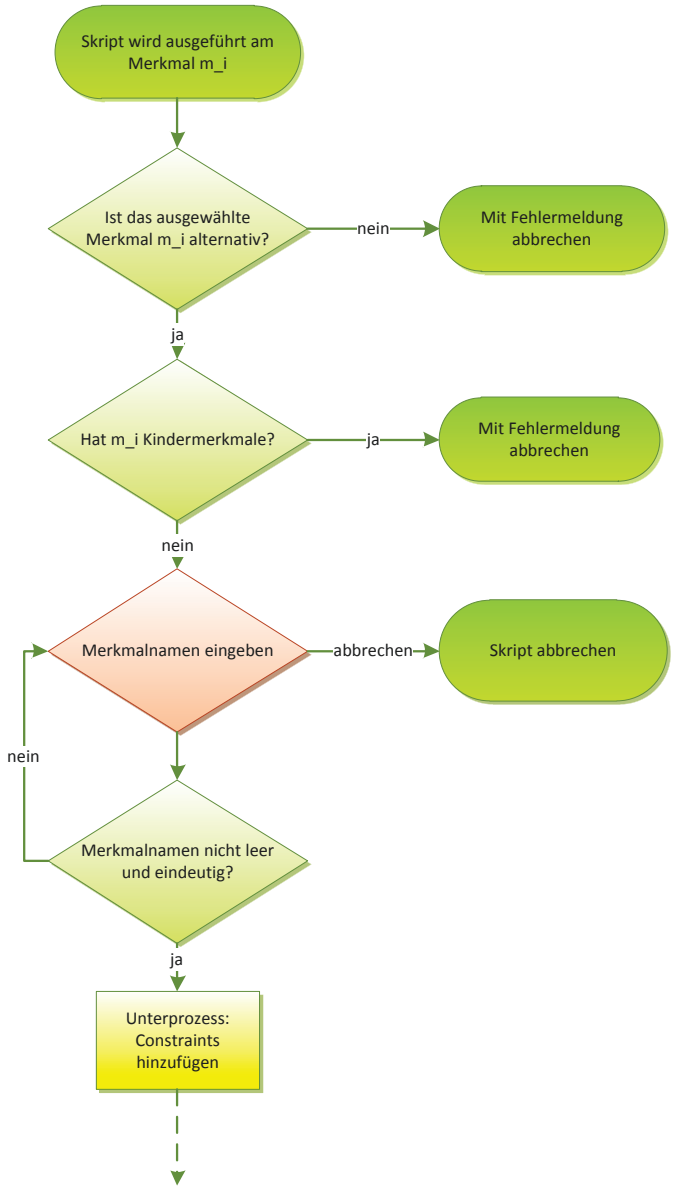


Abbildung A.6.: Der Änderungsprozess *Neue Alternative zu einer alternativen Komponente* - Teil 1

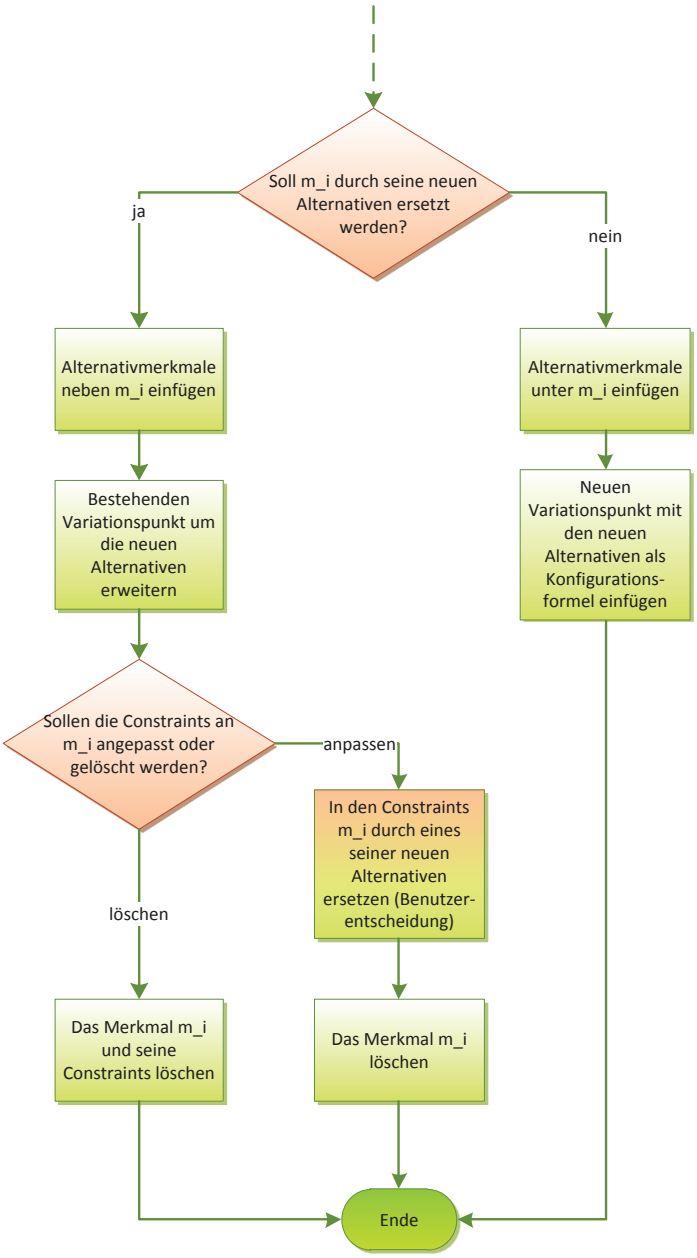


Abbildung A.7.: Der Änderungsprozess *Neue Alternative zu einer alternativen Komponente* - Teil 2

Abbildung A.8.: Der Unterprozess *Constraint einfügen*

Tabelle A.1.: Erfahrung der befragten Entwickler

Teilnehmer	1	2	3	4	5	6	7	8	9	10	**Ø**
Jahre Erfahrung	3	6	16	9	4	1	5	14	3	15	**7,6**

Tabelle A.2.: Anzahl der Projekte der befragten Entwickler

Teilnehmer	1	2	3	4	5	6	7	8	9	10	**Ø**
Anzahl Projekte	4	15	5	7,5	3	3	15	25	2	10	**8,95**

A.3. Ergebnisse der Expertenbefragung

In diesem Abschnitt wird auf den Inhalt und die Ergebnisse der Expertenbefragung eingegangen. Die Antworten der Teilnehmer werden hier lediglich dargelegt, während in Kapitel 8 deren Interpretation erfolgt.

A.3.1. Einleitung

Zu Beginn sollte der Erfahrungsstand der Teilnehmer ermittelt werden. Die ersten beiden offenen Fragen lauteten daher:

Frage 1. *Seit wie vielen Jahren beschäftigen Sie sich mit Variantenmodellierung?*

Frage 2. *An wie viel verschiedenen Variantenmodellierungsprojekten haben Sie bisher gearbeitet?*

Die Tabellen A.1 und A.2 zeigen die Antworten auf die ersten beiden Fragen[1]. Die Teilnehmer verfügen demnach im Durchschnitt über mehr als 7 Jahre Erfahrung in der Variantenmodellierung, welche sie durchschnittlich in ca. 9 Projekten erworben haben.

[1] In diesem Abschnitt zeigt Tabelle A.x stets das Ergebnis von Frage x.

Tabelle A.3.: Häufigkeit von variabilitätsrelevanten Änderungen

Häufigkeit	mind. täglich	mind. wöchentlich	mind. monatlich	seltener
# Antworten	2	2	4	2

A.3.2. Evolution des Entwicklungsartefaktes

Der nächste Abschnitt der Befragung beschäftigte sich mit der Evolution in Entwicklungsartefakten, z. B. in einem Funktionsmodell.

A.3.2.1. Häufigkeit von Änderungen

Zunächst ging es um die Häufigkeit, mit der Änderungen auftreten, die anschließend im Variantenmodell abgebildet werden müssen:

Frage 3. *Wie regelmäßig treten im Durchschnitt in Ihrem Bereich variabilitätsrelevante Änderungen auf?*

Dabei standen neben einem Freifeld für Bemerkungen die folgenden Häufigkeiten zur Auswahl: *mindestens täglich*, *mindestens wöchentlich*, *mindestens monatlich* und *seltener*. In Tabelle A.3 sieht man die Ergebnisse von Frage 3. Im Freifeld wurde zudem von vielen Teilnehmern als Bemerkung angemerkt, dass diese Häufigkeit stark davon abhängt, in welcher Projektphase man sich gerade befindet. In frühen Phasen sind tägliche Änderungen den Angaben nach keine Seltenheit, wohingegen wöchentliche oder sogar monatliche Änderungen gegen Projektende eher die Regel sind.

A.3.2.2. Arten von Änderungen

Die nächste Frage in diesem Abschnitt geht auf die Arten der Änderungen im Entwicklungsartefakt ein:

Frage 4. *Welche Arten von Änderungen treten in Ihrem Bereich auf?*

Tabelle A.4.: Auftrittsquote bestimmter Änderungen

Änderung	tritt auf	tritt nicht auf	Quote
neue optionale Komponente	10	0	100 %
optionale Komponente löschen	5	5	50 %
optionale Komponente wird obligatorisch	8	2	80 %
neue alternative Komponente	9	1	90 %

Dabei mussten die Teilnehmer für die folgenden vier Änderungen angeben, ob diese in ihrem Bereich auftreten.

- Änderung 1: Eine optionale Komponente wird neu erstellt
- Änderung 2: Eine optionale Komponente wird nicht mehr verwendet und kann gelöscht werden
- Änderung 3: Eine bisher optionale Komponente wird künftig immer eingesetzt (Sonderausstattung geht in Serienausstattung über)
- Änderung 4: Es wird eine neue Komponente erstellt, die zukünftig eine Alternative zu einer bereits bestehenden Komponente darstellt

Tabelle A.4 zeigt die Ergebnisse von Frage 4. Während die ersten drei Änderungen für nahezu alle Bereiche relevant sind, tritt Änderung 4 zumindest in jedem zweiten Bereich auf. Die Teilnehmer konnten außerdem in einem Freifeld weitere häufige Änderungen angeben, die nicht zur Wahl standen. Dabei wurden von je einem Teilnehmer die folgenden Änderungen angegeben:

- Eine obligatorische Komponente wird optional
- Es wird eine Abhängigkeit zwischen Merkmalen definiert
- Eine optionale Komponente wird in mehrere aufgeteilt

Die restlichen sieben der insgesamt zehn Teilnehmer gaben keine weitere Änderung an.

A.3.3. Merkmalbasierte Variantenmodelle

Der letzte Abschnitt der Befragung thematisierte allgemeine Eigenschaften und potentielle Inkonsistenzen von merkmalbasierten Variantenmodellen.

Tabelle A.5.: Durchschnitt der angegebenen Modellgrößen

Element	Merkmale	Variationspunkte	Variationen pro VP
durchschnittliche Anzahl	98,9	238,2	2,7

A.3.3.1. Größe des Variantenmodells

Um einen Eindruck zu gewinnen, welche Dimensionen merkmalbasierte Variantenmodelle üblicherweise aufweisen, wurden die Teilnehmer befragt, wie groß die Modelle durchschnittlich in ihrem Bereich sind:

Frage 5. *Wie viele Merkmale / Variationspunkte / Variationen pro Variationspunkt haben Variantenmodelle typischerweise in Ihrem Bereich?*

Das Ergebnis zeigt Tabelle A.5.

A.3.3.2. Komplexität des Variantenmodells

Die nächste Frage sollte ermitteln, was in den Augen der Entwickler die Komplexität eines Variantenmodells ausmacht.

Frage 6. *Welchen Einfluss haben Ihrer Meinung nach die folgenden Faktoren auf die Verständlichkeit eines Variantenmodells?*

Die Teilnehmer wurden gebeten, die drei folgenden Eigenschaften zu bewerten:

- Eigenschaft 1: Baumdarstellung mit Variationstypen (obligatorisch, optional,...)
- Eigenschaft 2: Einsatz von Constraints zwischen Merkmalen (requires, conflicts,...)
- Eigenschaft 3: Aussagenlogik der Konfigurationsformeln

Dabei war die folgende Skala gegeben: 1 (negativen Einfluss), 2 (leicht negativen Einfluss), 3 (neutral), 4 (leicht positiven Einfluss), 5 (positiven Einfluss). Außerdem stand auch ein Freifeld zur Verfügung, in dem eine selbst gewählte Eigenschaft hinsichtlich

Tabelle A.6.: Einfluss auf die Verständlichkeit des Variantenmodells

Teilnehmer	1	2	3	4	5	6	7	8	9	10	**Ø**
Baumdarstellung m. Variationstypen	5	5	5	5	5	4	5	5	5	5	**4,9**
Einsatz von Constraints	3	3	5	4	5	5	4	3	4	3	**3,9**
Aussagenlogik d. Konfigurationsformeln	2	1	2	2	3	5	2	3	2	3	**2,5**

Tabelle A.7.: Wahrscheinlichkeit für Fehlmodellierungen

Teilnehmer	1	2	3	4	5	6	7	8	9	10	**Ø**
neues Merkmal ohne Verlinkung	5	5	4	2	5	4	4	2	4	2	**3,7**
Constraint vergessen	4	4	5	1	5	2	4	3	3	2	**3,3**
semantischer Fehler bei KF´n	2	5	4	4	3	5	5	4	4	5	**4,1**
nicht eindeutige Konfigurationsformeln	5	4	5	5	5	4	3	3	4	4	**4,2**

ihres Beitrages zur Verständlichkeit bewerten werden konnte. In Tabelle A.6 sind die Ergebnisse dieser Frage dargestellt. Der Einfluss der Aussagenlogik an den Konfigurationsformeln wurde dabei von den meisten Teilnehmern negativ bewertet. Zudem haben zwei Teilnehmer im Freifeld explizit angegeben, dass komplexe Konfigurationsformeln die Verständlichkeit negativ beeinflussen.

A.3.3.3. Risiko von Inkonsistenzen

In den nächsten beiden Fragen sollte von den Experten das Risiko von Inkonsistenzen des Variantenmodells eingeschätzt werden. Dementsprechend wurde einerseits nach der Wahrscheinlichkeit für Fehler gefragt, die bei Änderungen auftreten können. Andererseits gaben die Teilnehmer auch Auskunft über die Schwere der Auswirkung bestimmter Inkonsistenzen. Das Risiko kann dann, wie im Risikomanagement üblich, als Produkt von Eintrittswahrscheinlichkeit und Auswirkung ermittelt werden. Die erste der beiden Fragen lautete demnach:

Frage 7. *Im Folgenden sind potentielle Fehler beschrieben, die bei einer Änderung des Variantenmodells auftreten können. Wie bewerten Sie die Wahrscheinlichkeit, dass diese einem Entwickler unterlaufen?*

Die Teilnehmer mussten dabei die folgenden Fehler bewerten:

- Fehler 1: Neue(s) Merkmal(e) anlegen und dessen/deren Verlinkung im Konfigurationsmodell vergessen (in der Konfigurationsformel eines Variationspunktes)

Tabelle A.8.: Schwere der Auswirkungen von Inkonsistenzen

Teilnehmer	1	2	3	4	5	6	7	8	9	10	**Ø**
unnötiges Merkmal	1	2	2	1	2	2	2	1	3	1	**1,7**
überflüssige bzw. fehlende Selektionen	5	5	5	5	5	5	4	4	4	4	**4,6**
fehlerhafte Konfiguration	5	5	5	5	5	5	3	4	4	5	**4,6**
unvollständige Konfiguration	2	2	3	3	4	3	4	2	5	3	**3,1**

- Fehler 2: Geplanter Constraint zwischen Merkmalen vergessen
- Fehler 3: Semantischer Fehler (z. B. Merkmalname verwechseln) bei manueller Eingabe von Konfigurationsformeln
- Fehler 4: Die Konfigurationsformeln an einem Variationspunkt sind so formuliert, dass die eindeutige Konfiguration gefährdet ist[2]

Dabei war die folgende Skala gegeben: 1 (sehr unwahrscheinlich), 2 (unwahrscheinlich), 3 (mittelmäßig wahrscheinlich), 4 (wahrscheinlich), 5 (sehr wahrscheinlich). Tabelle A.7 zeigt die Ergebnisse von Frage 7. Wie bereits erwähnt, ging es in der nächsten Frage um die Schwere der Auswirkung bestimmter Inkonsistenzen. Sie lautete:

Frage 8. *Der zweite Teil der Frage beschreibt potentielle Folgen eines Fehlers im Variantenmodell. Wie bewerten Sie die folgenden Inkonsistenzen hinsichtlich der Schwere ihrer Auswirkungen?*

Die Teilnehmer mussten dabei die folgenden Fälle bewerten:

- Inkonsistenz 1: Ein für die Konfiguration nicht benötigtes Merkmal befindet sich im Merkmalmodell.
- Inkonsistenz 2: Merkmalselektionen, die man ausschließen möchte, sind gültig bzw. Selektionen, die man konfigurieren möchte, sind ungültig.
- Inkonsistenz 3: Der Konfigurationsprozess läuft ohne Fehlermeldung ab, wobei jedoch inhaltliche Fehler in der Variationspunktbelegung auftreten, sodass das Entwicklungsartefakt ggf. falsch konfiguriert wird.

[2]Das ist der Fall, wenn für bestimmte Merkmalselektionen entweder gar keine oder mehrere Konfigurationsformeln erfüllt sind.

- Inkonsistenz 4: Die Konfiguration schlägt fehl bzw. kann nicht vollständig durchgeführt werden. Die Ursache dafür ist jedoch unklar.

Dabei war die folgende Skala gegeben: 1 (keine Auswirkung), 2 (geringe Auswirkung), 3 (mittelmäßige Auswirkung), 4 (starke Auswirkung), 5 (sehr starke Auswirkung). In Tabelle A.8 sind die Ergebnisse von Frage 8 dargestellt. In Abschnitt 8.2.2 werden die Ergebnisse von den Fragen 7 und 8 kombiniert, um mithilfe von Eintrittswahrscheinlichkeit und Schwere der Auswirkung eine Risikoeinschätzung der Inkonsistenzen zu erhalten und diese grafisch darstellen zu können.

A.3.3.4. Konsistenzprüfung eines Beispielmodells

Zum Abschluss der Befragung mussten die Teilnehmer eine Aufgabe erfüllen. Gegeben war ein beispielhaftes Variantenmodell, dessen Konfigurationsmodell aus vier Variationspunkten besteht (s. Abb. A.9). Die Befragten sollten jeden Variationspunkt einzeln analysieren und entscheiden, ob eine der folgenden Inkonsistenzen vorliegt, welche ebenfalls gegeben waren:

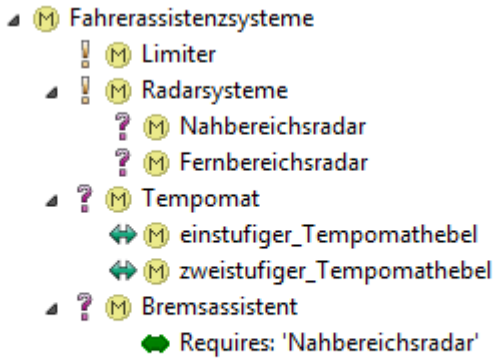
An dem betroffenen Variationspunkt

- existiert eine kontradiktorische Konfigurationsformel
- existiert eine tautologische Konfigurationsformel
- wird für mindestens eine gültige Merkmalselektion keine Konfigurationsformel erfüllt
- werden für mindestens eine gültige Merkmalselektion alle Konfigurationsformeln erfüllt[3]

Jeder der Variationspunkte in Abb. A.9 ist inkonsistent, manche von ihnen verletzen sogar mehrere Konsistenzbedingungen. Die Aufgabe galt als erfüllt, falls mindestens eine Inkonsistenz gefunden werden konnte. Wie lange die Experten im Durchschnitt benötigt haben, um die Variationspunkte zu analysieren, zeigt Tabelle A.9. Demnach hat sich bei den 40 Analysen (10 Befragte * 4 Variationspunkte) eine durchschnittliche Zeit von 33,21 Sekunden ergeben, die ein erfahrener Variantenmodellierer benötigt, um einen Variationspunkt hinsichtlich der oben genannten Inkonsistenzen zu prüfen.

Interessant ist dabei auch die Fehlerquote. Bei zwei Analysen wurden Inkonsistenzen identifiziert, die nicht vorlagen. Der noch brisantere Fehlerfall ist bei zwei weiteren Ana-

[3]Diese Konsistenzbedingungen sind in Abschnitt 5.2 beschrieben.



Variation Point	Assignment
VAR_VP_1	
1 (Variation 1)	NOT(Tempomat)
2 (Variation 2)	einstufiger_Tempomathebel AND zweistufiger_Tempomathebel
VAR_VP_2	
1 (Variation 1)	Limiter
2 (Variation 2)	Tempomat
VAR_VP_3	
1 (Variation 1)	Tempomat AND einstufiger_Tempomathebel
2 (Variation 2)	Tempomat AND zweistufiger_Tempomathebel
VAR_VP_4	
1 (Variation 1)	Tempomat
2 (Variation 2)	einstufiger_Tempomathebel

Abbildung A.9.: Das zu prüfende Beispielmodell

Tabelle A.9.: Benötigte Zeiten für die Konsistenzprüfung

Variationspunkt	1	2	3	4
durchschnittlich benötigte Zeit	36,18 Sek.	26,18 Sek.	41,41 Sek.	29,07 Sek.

lysen aufgetreten: Dort wurden die Variationspunkte jeweils für konsistent erklärt. Damit sind insgesamt 10% der Analysen zu einem falschen Ergebnis gekommen.

Literaturverzeichnis

[AGM+06] Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba und Carlos Lucena: *Refactoring Product Lines.* In: *Proceedings of the 5th international conference on Generative programming and component engineering*, GPCE '06, Seiten 201–210, New York, NY, USA, 2006. ACM.

[Akc11] Özgür Akcasoy: *Solution Patterns for Variability in Model-Based Product Lines.* Masterarbeit, RWTH Aachen, 2011.

[Ass83] Günter Asser: *Einführung in die mathematische Logik: Aussagenkalkül*, Band 1 der Reihe *Einführung in die mathematische Logik.* H. Deutsch, 1983.

[Bat05] Don S. Batory: *Feature Models, Grammars, and Propositional Formulas.* In: J. Henk Obbink und Klaus Pohl (Herausgeber): *SPLC*, Band 3714 der Reihe *Lecture Notes in Computer Science*, Seiten 7–20. Springer, 2005.

[BCTS06] David Benavides, Antonio Ruiz Cortés, Pablo Trinidad und Sergio Segura: *A Survey on the Automated Analyses of Feature Models.* In: *JISBD*, Seiten 367–376, 2006.

[BE08] H. Balzert und C. Ebert: *Lehrbuch der Softwaretechnik: Softwaremanagement.* Lehrbuch der Software-Technik. Spektrum Akademischer Verlag, 2008, ISBN 9783827411617.

[Bee07] Michael Beeck: *Development of logical and technical architectures for automotive systems.* Software & Systems Modeling, 6(2):205–219, 2007.

[BKPS04] Günter Böckle, Peter Knauber, Klaus Pohl und Klaus Schmid (Herausgeber): *Software-Produktlinien: Methoden, Einführung und Praxis.* Dpunkt Verlag, 2004.

[Bor09] Paulo Borba: *An Introduction to Software Product Line Refactoring.* In: João M. Fernandes, Ralf Lämmel, Joost Visser und João Saraiva (Herausgeber): *GTTSE*, Band 6491 der Reihe *Lecture Notes in Computer Science*, Seiten 1–26. Springer, 2009.

[BPK09] Goetz Botterweck, Andreas Polzer und Stefan Kowalewski: *Using Higher-Order Transformations to Derive Variability Mechanism for Embedded Systems.* In: Sudipto Ghosh (Herausgeber): *MoDELS Workshops*, Band 6002 der Reihe *Lecture Notes in Computer Science*, Seiten 68–82. Springer, 2009.

[BRN+13] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker 0002, Krzysztof Czarnecki und Andrzej Wasowski: *A Survey of Variability Modeling in Industrial Practice.* In: Stefania Gnesi, Philippe Collet und Klaus Schmid (Herausgeber): *The Seventh International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '13, Pisa, Italy, January 23 - 25, 2013.* ACM, 2013.

[BSC10] David Benavides, Sergio Segura und Antonio Ruiz Cortés: *Automated Analysis of Feature Models 20 Years Later: A Literature Review.* Information Systems, 35(6):615–636, 2010.

[BTG10] Paulo Borba, Leopoldo Teixeira und Rohit Gheyi: *A Theory of Software Product Line Refinement.* In: Ana Cavalcanti, David Déharbe, Marie Claude Gaudel und Jim Woodcock (Herausgeber): *ICTAC*, Band 6255 der Reihe *Lecture Notes in Computer Science*, Seiten 15–43. Springer, 2010.

[CE00] Krzysztof Czarnecki und Ulrich Eisenecker: *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, Reading, MA, USA, 2000.

[CN01] Paul C. Clements und Linda Northrop: *Software Product Lines: Practices and Patterns.* SEI Series in Software Engineering. Addison-Wesley, August 2001.

[CW07] Krzysztof Czarnecki und Andrzej Wasowski: *Feature Diagrams and Logics: There and Back Again.* In: *Software Product Line Conference, 2007. SPLC 2007. 11th International*, Seiten 23–34. IEEE, 2007.

[Cza98] Krzysztof Czarnecki: *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models.* Dissertation, Fakultät für Informatik, Technische Universität Illmenau, 1998.

[Fow99] Martin Fowler: *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA, 1999.

[GA01] Cristina Gacek und Michalis Anastasopoules: *Implementing Product Line Variabilities.* In: *Proceedings of the 2001 symposium on Software reusabi-*

lity: putting software reuse in context, SSR '01, Seiten 109–117, New York, NY, USA, 2001. ACM.

[GRMM+13] Martin Große-Rhode, Peter Manhart, Ralf Mauersberger, Sebastian Schröck, Michael Schulze und Thorsten Weyer: *Anforderungen von Leitbranchen der deutschen Industrie an Variantenmanagement und Wiederverwendung und daraus resultierende Forschungsfragestellungen.* In: *Tagungsband Software Engineering 2013, Workshopband*, Seiten 251–260, 2013.

[GW11] Benjamin Gutekunst und Jens Weiland: *Handhabung von Varianz in Simulink aus funktionsorientierter Sicht.* In: *Informatik 2011 - 9. Workshop Automotive Software Engineering*, Berlin, 2011.

[HC13] Hannes Holdschick und Walter Commerell: *Variantenmanagement in der modellbasierten Produktlinienentwicklung von Fahrzeugsystemen.* In: *Tagungsband ASIM-Treffen STS/GMMS*, ASIM '13, Seiten 111–118. Arbeitsgemeinschaft Simulation in der GI, 2013.

[Hol12] Hannes Holdschick: *Challenges in the Evolution of Model-Based Software Product Lines in the Automotive Domain.* In: *Proceedings of the 4th International Workshop on Feature-Oriented Software Development*, FOSD '12, New York, NY, USA, 2012. ACM.

[Hol14] Hannes Holdschick: *Konzepte zur Absicherung der Variantenkonfiguration von eingebetteter Fahrzeugsoftware.* In: *Tagungsband des vierten Workshops zur Zukunft der Entwicklung softwareintensiver, eingebetteter Systeme (ENVISION 2020)*, Seiten 87–98, 2014.

[ISO11] ISO/IEC: *ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models.* Technischer Bericht, 2011.

[KCH+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak und A. S. Peterson: *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technischer Bericht, Carnegie-Mellon University Software Engineering Institute, 1990.

[KS00] Wolfgang Küchlin und Carsten Sinz: *Proving Consistency Assertions for Automotive Product Data Management.* In: I. Gent, H. van Maaren und T. Walsh (Herausgeber): *SAT2000 - Highlights of Satisfiability Research in the Year 2000*, Band 63 der Reihe *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2000.

[LBPRS] Daniel Le Berre, Anne Parrain, Olivier Roussel und Lakhdar Sais: *SAT4J: A satisfiability library for Java, 2005.*

[LSB+10] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki und Andrzej Wąsowski: *Evolution of the Linux Kernel Variability Model.* In: *Proceedings of the 14th international conference on Software product lines: going beyond*, SPLC'10, Seiten 136–150, Berlin, Heidelberg, 2010. Springer-Verlag.

[Man02] Mike Mannion: *Using First-Order Logic for Product Line Model Validation.* In: *Proceedings of the 2nd international conference on Software product lines (SPLC) 2002*, Seiten 176–187. Springer, 2002.

[MHP+07] Andreas Metzger, Patrick Heymans, Klaus Pohl, Pierre Yves Schobbens und Germain Saval: *Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis.* In: *RE*, Seiten 243–253. IEEE, 2007.

[ML04] Thomas von der Maßen und Horst Lichter: *Deficiencies in Feature Models.* In: *Proceedings of SPLC'04 - Workshop on Software Variability Management for Product Derivation - Towards Tool Support*, 2004.

[MR13] Christian Manz und Manfred Reichert: *Herausforderungen an ein durchgängiges Variantenmanagement in Software-Produktlinien und die daraus resultierende Entwicklungsprozessadaption.* In: *Tagungsband Software Engineering 2013, Workshopband.* Koellen-Verlag, 2013.

[MRM+12] Swarup Mohalik, S Ramesh, Jean Vivien Millo, Shankara Narayanan Krishna und Ganesh Khandu Narwane: *Tracing SPLs Precisely and Efficiently.* In: *Proceedings of the 16th International Software Product Line Conference-Volume 1*, Seiten 186–195. ACM, 2012.

[MWC09] Marcilio Mendonca, Andrzej Wąsowski und Krzysztof Czarnecki: *SAT-based Analysis of Feature Models is Easy.* In: *Proceedings of the 13th International Software Product Line Conference*, Seiten 231–240. Carnegie Mellon University, 2009.

[MWCC08] Marcilio Mendonca, Andrzej Wasowski, Krzysztof Czarnecki und Donald Cowan: *Efficient Compilation Techniques for Large Scale Feature Models.* In: *Proceedings of the 7th international conference on Generative programming and component engineering*, Seiten 13–22. ACM, 2008.

[PCA+13] Leonardo Passos, Krzysztof Czarnecki, Sven Apel, Andrzej Wąsowski, Christian Kästner und Jianmei Guo: *Feature-Oriented Software Evolution.* In: *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, Seite 17. ACM, 2013.

[PCW12] Leonardo Passos, Krzysztof Czarnecki und Andrzej Wąsowski: *Towards a Catalog of Variability Evolution Patterns: The Linux Kernel Case.* In: *Proceedings of the 4th International Workshop on Feature-Oriented Software Development*, Seiten 62–69. ACM, 2012.

[RKW09] M O Reiser, Ramin Tavakoli Kolagari und Matthias Weber: *Compositional Variability - Concepts and Patterns.* In: *System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on*, Seiten 1–10. IEEE, 2009.

[SBDT10] Ina Schaefer, Lorenzo Bettini, Ferruccio Damiani und Nico Tanzarella: *Delta-oriented Programming of Software Product Lines.* In: *Proceedings of the 14th international conference on Software product lines: going beyond*, SPLC'10, Seiten 77–91, Berlin, Heidelberg, 2010. Springer-Verlag.

[SHA12] Christoph Seidl, Florian Heidenreich und Uwe Aßmann: *Co-Evolution of Models and Feature Mapping in Software Product Lines.* In: *Proceedings of the 16th International Software Product Line Conference-Volume 1*, Seiten 76–85. ACM, 2012.

[SKK03] Carsten Sinz, Andreas Kaiser und Wolfgang Küchlin: *Formal Methods for the Validation of Automotive Product Configuration Data.* Artificial Intelligence for Engineering Design, Analysis and Manufacturing, 17(1):75–97, Januar 2003. Special issue on configuration.

[SLB+11] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski und Krzysztof Czarnecki: *Reverse Engineering Feature Models.* In: Richard N. Taylor, Harald Gall und Nenad Medvidovic (Herausgeber): *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21-28, 2011*, Seiten 461–470. ACM, 2011.

[SRP05] Periklis Sochos, Matthias Riebisch und Ilka Philippow: *Featuregesteuerte Architekturgestaltung zwecks Wartbarkeit und Evolution von Produktlinien.* In: Peter Liggesmeyer, Klaus Pohl und Michael Goedicke (Herausgeber): *Software Engineering*, Band 64 der Reihe *LNI*, Seiten 29–42. GI, 2005.

[STKS12] Sandro Schulze, Thomas Thüm, Martin Kuhlemann und Gunter Saake: *Variant-Preserving Refactoring in Feature-Oriented Software Product Li-*

nes. In: Ulrich W. Eisenecker, Sven Apel und Stefania Gnesi (Herausgeber): *VaMoS*, Seiten 73–81. ACM, 2012.

[Sys14] Pure Systems: *pure::variants User's Guide.* 2014.

[TBC06] Pablo Trinidad, David Benavides und Antonio Ruiz Cortés: *Isolated Features Detection in Feature Models.* In: *CAiSE Forum*, 2006.

[TBK09] Thomas Thüm, Don S. Batory und Christian Kästner: *Reasoning about Edits to Feature Models.* In: *Proceedings of the International Conference on Software Engineering, ICSE 2009*, Seiten 254–264. IEEE, 2009.

[TBKC07] Sahil Thaker, Don Batory, David Kitchin und William Cook: *Safe Composition of Product Lines.* In: *Proceedings of the 6th international conference on Generative programming and component engineering*, Seiten 95–104. ACM, 2007.

[TD13] Quang Minh Tran und Christian Dziobek: *Ansatz zur Erstellung und Wartung von Simulink-Modellen durch den Einsatz von Transformationen/Refactorings und Generierungsoperationen.* In: Holger Giese, Michaela Huhn, Jan Phillips und Bernhard Schätz (Herausgeber): *MBEES*, Seiten 1–12. fortiss GmbH, München, 2013.

[TKES11] Thomas Thüm, Christian Kästner, Sebastian Erdweg und Norbert Siegmund: *Abstract Features in Feature Modeling.* In: Eduardo Santana de Almeida, Tomoji Kishi, Christa Schwanninger, Isabel John und Klaus Schmid (Herausgeber): *Software Product Line Conference, 2011*, Seiten 191–200. IEEE, 2011.

[VG07] Markus Völter und Iris Groher: *Product Line Implementation using Aspect-Oriented and Model-Driven Software Development.* In: *SPLC*, Seiten 233–242. IEEE Computer Society, 2007.

[Wei08] Jens Weiland: *Variantenkonfiguration eingebetteter Automotive Software mit Simulink.* Dissertation, Universität Leipzig, 2008.

[Win13] Marco Winkelbauer: *Konzeption eines Testrahmens für die automatisierte Prüfung von Variantenmodelltransformationen.* Bachelorarbeit, Hochschule Karlsruhe - Fakultät für Informatik und Wirtschaftsinformatik, 2013.