Otto von Guericke Universität Magdeburg
Fakultät für Informatik

Masterarbeit

Automatisierte Analysen der Modellierung von Multiproduktlinien in VELVET

Verfasser:

Reimar Schröter

11. April 2012

Betreuer:

Prof. Dr. rer. nat. habil. Gunter Saake
Dipl.-Inform. Thomas Thüm
Institut für Technische und Betriebliche Informationssysteme (ITI)

Schröter, Reimar:
Automatisierte Analysen der Modellierung von Multiproduktlinien in VELVET
Masterarbeit, Otto von Guericke Universität Magdeburg, 2012.

Danksagung

Besonderer Dank geht an Herrn Prof. Dr. rer. nat. habil. Gunter Saake, der es mir ermöglichte meine Masterarbeit am Institut für Technische und Betriebliche Informationssysteme (ITI) zu verfassen. Des Weiteren möchte ich mich auch bei Dipl.-Inform. Thomas Thüm für die ausgezeichnete Betreuung, Verbesserungsvorschläge und die hilfreichen Diskussionen bedanken. Zusätzlich möchte ich mich auch besonders bei Herrn Prof. Dr. Frank Ortmeier für die Bereitschaft bedanken, als Zweitgutachter bei meiner Arbeit zu fungieren.
Außerdem bedanke ich mich bei Susann Johannes, Alexander Grebhahn und Norbert Siegmund für Korrekturlesen und Anregungen zur Verbesserung dieser Arbeit. Weiterer Dank geht auch an meine Freunde und meine Familie, die mich während der Masterarbeit moralisch unterstützt haben.

Inhaltsverzeichnis

Abbildungsverzeichnis viii

Tabellenverzeichnis ix

Quellcodeverzeichnis xii

1 Einleitung 1

2 Grundlagen 5
2.1 Software-Produktlinien . . . 5
2.1.1 Feature-Modelle . . . 6
2.1.2 Aussagenlogische Repräsentation von Feature-Modellen . . . 8
2.1.3 Beziehungen zwischen Produktlinien . . . 9
2.1.4 Schrittweiser Konfigurationsprozess . . . 10
2.2 Multiproduktlinien . . . 11
2.2.1 Motivation . . . 11
2.2.2 Darstellung von Multiproduktlinien . . . 12

3 Modellierung in VELVET 17
3.1 Produktlinienmodellierung in VELVET . . . 18
3.2 Multiproduktlinienmodellierung in VELVET . . . 21
3.2.1 Vererbung in VELVET . . . 22
3.2.2 Aggregation in VELVET . . . 25
3.3 Zusammenfassung . . . 29

4 Konsistenzprüfungen für Multiproduktlinien in VELVET 31
4.1 Automatisierung von Konsistenzprüfungen . . . 32
4.2 Syntaktische Konsistenzprüfungen . . . 33
4.3 Konsistenzprüfungen für Produktlinien in Multiproduktlinien . . . 35
4.3.1 Dead-Features und False-Optional-Features . . . 36
4.3.2 Anzahl und Ausgabe von Varianten . . . 42
4.3.3 Core-Features . . . 47
4.3.4 Atomare Mengen . . . 50
4.4 Zusammenfassung . . . 53

5 Implementierung von Konsistenzprüfungen für VELVET 55

5.1 Bisherige Werkzeugunterstützung für VELVET 55
5.2 Parser und syntaktische Überprüfungen für VELVET-Modelle 57
5.3 Test auf Erfüllbarkeit für VELVET-Modelle 58
5.4 Konsistenzprüfungen für VELVET-Modelle 60
5.5 Zusammenfassung 61

6 Evaluierung **63**
6.1 Testgetriebene Entwicklung 63
6.2 Evaluierung bestehender Multiproduktlinien 65
6.3 Zusammenfassung 68

7 Verwandte Arbeiten **69**

8 Zusammenfassung **73**

9 Zukünftige Arbeiten **75**

A VELVET-Grammatik **77**

B Testgetriebene Entwicklung **81**
B.1 Modelle für syntaktische Analysen 81
B.2 Modelle für semantische Analysen 83

Literaturverzeichnis **85**

Abbildungsverzeichnis

1.1 Kosten der Produktlinienentwicklung. 1

2.1 Feature-Modell einer Auto-Produktlinie. 7
2.2 Schrittweiser Konfigurationsprozess. 10
2.3 Die Produktlinie `Mail_Client` und deren Abhängigkeiten. 12
2.4 Abhängige Produktlinien im Feature-Diagramm. 13
2.5 Klassendiagramme zur Modellierung von Kompositionen. 15

3.1 Fehlende Eindeutigkeit bei der Erweiterung von Gruppen. 23
3.2 Feature-Modelle nach Spezialisierung durch Vererbung. 24
3.3 Feature-Modell nach Generalisierung durch Vererbung. 25
3.4 Überführung von abhängigen Produktlinien in ein Gesamtmodell bei der Verwendung von Aggregation. 28

4.1 Werkzeuganforderungen zur Unterstützung von Konsistenzprüfungen. . 32
4.2 Dead-Features und False-Optional-Features in Produktlinien. 37
4.3 Dead-Features in Multiproduktlinien. 40
4.4 Anzahl der Varianten einer Instanz in Multiproduktlinien. 44
4.5 Core-Features in Multiproduktlinien. 49
4.6 Core-Instanzen in Multiproduktlinien. 50
4.7 Atomare Mengen in Produktlinien. 51
4.8 Atomare Mengen in Multiproduktlinien. 52

5.1 Existierende Werkzeugunterstützung für VELVET. 57
5.2 Probleme bei der Erweiterung von Features. 59

6.1 Ablauf der testgetriebenen Entwicklung. 64
6.2 Kompositionsmodell für die Multiproduktlinie Sensor-Netzwerk. 65
6.3 Analyse von Core-Features über die Vererbungshierarchie. 67
6.4 Analyse von atomaren Mengen über die Vererbungshierarchie. 67

Tabellenverzeichnis

2.1 Klassifikation von Änderungen an Feature-Modellen. 9
2.2 Vergleich objektorientierter Konzepte mit Produktlinien. 14

5.1 Syntaktische Konsistenzprüfungen in FmConfig und VeAnalyzer. 58
5.2 Unterstütze Mechanismen zur Bildung eines aussagenlogischen Ausdrucks. 60
5.3 Integrierte Konsistenzprüfungen in VeAnalyzer. 61

6.1 Ergebnisse der Analysen des Sensor-Netzwerks. 66

Quellcodeverzeichnis

3.1 VELVET-Grammatik Teil I. 19
3.2 Verbesserung der VELVET-Grammatik Teil I. 19
3.3 VELVET-Grammatik Teil II. 20
3.4 Auto-Produktlinie in VELVET. 21
3.5 VELVET-Grammatik Teil III. 22
3.6 Verbesserung der VELVET-Grammatik Teil II. 23
3.7 Spezialisierung der Produktlinie `Liste` durch Vererbung. 24
3.8 Generalisierung der Produktlinie `Liste` durch Vererbung. 25
3.9 VELVET-Grammatik Teil IV. 26
3.10 Verbesserung der VELVET-Grammatik Teil III. 27
3.11 Multiproduktlinie `Mail_Client` in VELVET. 27
4.1 Syntaktischer Test auf doppelte Namensvergabe. 34
4.2 Syntaktischer Test auf Korrektheit der Referenzen. 34
4.3 Syntaktischer Test auf doppelte Vererbung. 34
4.4 Syntaktischer Test bei fehlender Vererbung. 34
4.5 Syntaktischer Test bei `mandatory` in Gruppen. 35
4.6 Syntaktischer Test bei Rekursion in der Modellierung. 35
4.7 Pseudocode 1: Berechnung von Dead-Features. 37
4.8 Pseudocode 2: Berechnung der False-Optional-Features. 38
4.9 Dead-Features in Multiproduktlinien bei Vererbung in VELVET. 40
4.10 Pseudocode 3: Berechnung von Dead-Features und False-Optional-Features in Multiproduktlinien. 41
4.11 Pseudocode 4: Berechnung der Anzahl der Varianten. 42
4.12 Pseudocode 5: Berechnung von Core-Features. 47
4.13 Core-Features in Multiproduktlinien bei Vererbung in VELVET. 49
4.14 Pseudocode 6: Berechnung von atomaren Mengen. 50
5.1 Probleme bei der Erweiterung des Features `Sortierung`. 59
A.1 Vollständige, aktuelle VELVET-Grammatik. 77
A.2 Vollständige, angepasste VELVET-Grammatik. 78
B.1 Mandatory in Gruppen I. 81
B.2 Mandatory in Gruppen II. 81
B.3 Doppelte Vererbung. 82
B.4 Doppelte Namensvergabe I. 82
B.5 Doppelte Namensvergabe II. 82
B.6 Doppelte Namensvergabe III. 82

B.7 Doppelte Namensvergabe IV. 82
B.8 Doppelte Namensvergabe V. 83
B.9 Dead-Features I. 83
B.10 Dead-Features II. 83
B.11 Dead-Features III. 84
B.12 Dead-Features IV. 84
B.13 Dead-Features V. 84

1. Einleitung

Der Traum der massiven Wiederverwendung von Software ist so alt wie die Softwareentwicklung selbst [vdLSR07]. Die Wiederverwendung von Software hat den Vorteil, dass weniger Software geschrieben werden muss und dass mehr Aufwand in andere Aspekte wie Korrektheit und Robustheit von Software investiert werden kann [Mey00]. Durch die Entwicklung von verschiedenen neuartigen Konzepten wurde in den letzten Jahrzehnten versucht die Wiederverwendung von Software zu verbessern. Diesbezüglich wurden neue Möglichkeiten durch die Einführung von objektorientierter Programmierung geschaffen. Die beiden häufigsten Methoden zur Wiederverwendung von bestehender Funktionalität in objektorientierten Systemen sind Vererbung und die Komposition von Objekten [GHJV95].

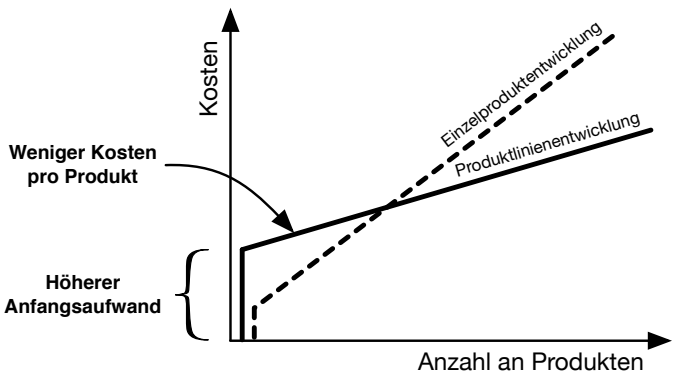


Abbildung 1.1: Kosten der Produktlinienentwicklung [JKB08].

Eine weitere erhebliche Verbesserung der Softwareentwicklung im Vergleich zu der objektorientierten Programmierung erfolgte durch die Einführung von Software-Produktlinien. Im Bereich der Wiederverwendung konnte in diesem Zusammenhang eine neue Ebene erreicht werden [vdLSR07]. Software-Produktlinien ermöglichen eine Entwicklung von ähnlichen Programmen auf einer gemeinsamen Basis, weshalb auf eine Neuentwicklung für einzelne Programme verzichtet und somit die Wiederverwendung der

einzelnen Quelltextfragmente besser unterstützt werden kann. In diesem Zusammenhang können ähnliche Programme mit geringeren Kosten, in kürzerer Zeit und mit höherer Qualität entwickelt werden [PBvdL05]. Bei der Entwicklung einer Software-Produktlinie muss jedoch beachtet werden, dass die grundlegende Neuentwicklung von einer Software-Produktlinie einen höheren Anfangsaufwand und höhere Kosten erfordert, als die jeweilige Neuentwicklung eines einzelnen Programms (siehe Abbildung 1.1 auf der vorherigen Seite). Der Vorteil der Entwicklung einer Software-Produktlinie im Gegensatz zu der vollständigen Neuentwicklung von Programmen, stellt sich jedoch schon bei etwa drei verschiedenen Programmen ein [PBvdL05].
In den letzten Jahren erhielt die Entwicklung von Software-Produktlinien zunehmende Aufmerksamkeit von Unternehmen aller Größenordnungen und von verschiedenen Märkten [vdLSR07]. Dementsprechend stellt die Entwicklung ähnlicher Programme mittels Software-Produktlinien einen vielversprechenden Mechanismus dar. Jedoch kann die Komplexität von Software-Produktlinien sehr unterschiedliche Ausmaße annehmen. Bei der Neuentwicklung einer Software-Produktlinie mit großer Komplexität kann es vorkommen, dass ein Teil der gewünschten Basisfunktionalität schon in einer bestehenden Software-Produktlinie realisiert wurde. Demnach ist die Wiederverwendung dieser Funktionalität in einer neu entwickelten Software-Produktlinie ein erwünschter Aspekt, der nach Krüger in Produktlinien aus Produktlinien beziehungsweise verschachtelten Produktlinien [Kru06] resultiert. Im Gegensatz dazu prägten Rosenmüller und Siegmund diesbezüglich den Begriff Multi-Software-Produktlinien oder kurz Multiproduktlinien für jegliche Art der Komposition von Produktlinien [RS10].
Die Modellierung von Software-Produktlinien ist zum Teil gut erforscht, dokumentiert und wird von Werkzeugen (zum Beispiel `FeatureIDE`[1]) gut unterstützt [Bat05, KCH+90, CE00, BSRC10]. Beispielsweise können verschiedene Analysen auf bestehenden Modellen von Software-Produktlinien angewandt werden, die Rückschlüsse auf deren Korrektheit ermöglichen [BSRC10]. Diese frühe Fehlererkennung bietet dem Modellierer und Entwickler eine gute Grundlage für die jeweilige Entwicklung einer Software-Produktlinie. Diese Sicherheit und Kontrolle bei der Modellierung und Entwicklung ist ebenfalls für Multiproduktlinien wünschenswert, um den Einsatz von Multiproduktlinien etablieren zu können. Durch verschiedene Möglichkeiten der Zusammenstellung von Software-Produktlinien ergeben sich zusätzliche Herausforderungen, die bei der Sicherstellung der Konsistenz bei der Modellierung einer Multiproduktlinie bewältigt werden müssen.

Zielstellung der Arbeit

Das Ziel dieser Arbeit ist es, bestehende Analysen zu untersuchen und neue Analysen zu entwickeln, die zur Sicherstellung der Korrektheit und Konsistenz bei der Modellierung von Multiproduktlinien verwendet werden können. Im Speziellen sollen Analysen für die Modellierungssprache VELVET[2] realisiert werden, die eine sichere Zusammenstellung der verschiedenen, einzelnen Modelle der betroffenen Software-Produktlinien ermöglichen.

[1] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/
[2] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/MultiPLe/modeling.htm

VELVET ist eine textuelle Modellierungssprache [RSTS11], die Sprachmittel der objektorientierten Programmierung zur Zusammenstellung von verschiedenen Software-Produktlinien anbietet. Dementsprechend kann unter Anderem die Verwendung von Vererbung und Aggregation zur Erstellung von komplexen Modellen für Multiproduktlinien verwendet werden.
Durch die verschiedenen Möglichkeiten der Zusammenstellung von Software-Produktlinien zu Multiproduktlinien ergeben sich verschiedene Abhängigkeiten. Demnach muss sichergestellt werden, dass die Zusammenstellung der Multiproduktlinie korrekt und gültig ist. In diesem Zusammenhang soll im Rahmen der Arbeit überprüft und diskutiert werden, ob die verschiedenen existierenden Analysen bezogen auf die Konsistenz von Software-Produktlinien auch auf Multiproduktlinien angewandt werden können. Zur Automatisierung von Analysen bezüglich der Konsistenz soll prototypisch ein Werkzeug entwickelt werden, das die entwickelten und diskutierten Analysen unterstützt. Anschließend sollen verschiedene Evaluierungen durchgeführt werden, um die Korrektheit der entwickelten Konzepte im Prototyp zu untermauern.

Gliederung der Arbeit

In Kapitel 2 werden die Grundlagen zum Verständnis der Arbeit dargelegt. Diesbezüglich erfolgt eine detaillierte Einführung in Software-Produktlinien, Multiproduktlinien und deren jeweilige Modellierung. Anschließend wird in Kapitel 3 die Modellierungssprache VELVET genauer vorgestellt. In diesem Zusammenhang werden auch notwendige Verbesserungen dieser Modellierungssprache eingeführt und diskutiert. Aufbauend auf diesen Verbesserungen werden in Kapitel 4 verschiedene Analysen, bezogen auf die Konsistenz, vorgestellt und deren Verwendung in Multiproduktlinien von VELVET diskutiert. Des Weiteren werden verschiedene Konzepte aufgestellt, die die weitere Analyse von Multiproduktlinien in VELVET ermöglichen. Im folgenden Kapitel 5 wird die Implementierung der realisierten Analysen vorgestellt. Die Evaluierung, die die Korrektheit der realisierten Konzepte untermauert, erfolgt in Kapitel 6. Verwandte Arbeiten werden in Kapitel 7 beschrieben, bevor abschließend eine Zusammenfassung im Kapitel 8 und ein Ausblick über weitere, mögliche Forschung in Kapitel 9 gegeben wird.

2. Grundlagen

In dieser Arbeit wird die Konsistenz und Gültigkeit der Kombination von verschiedenen Software-Produktlinien zu Multiproduktlinien betrachtet. Daher werden in diesem Kapitel grundlegende Kenntnisse zum Verständnis der Arbeit vermittelt. Dementsprechend werden zunächst Grundlagen bezogen auf Software-Produktlinien und deren mögliche Darstellung in *Feature-Modellen* und als *aussagenlogischer Ausdruck* gegeben. Anschließend erfolgt eine Übersicht über die Kategorisierung von möglichen Änderungen an Software-Produktlinien und eine Erläuterung des schrittweisen Konfigurationsprozesses von Software-Produktlinien. Des Weiteren wird eine Einführung zu Multiproduktlinien gegeben.

2.1 Software-Produktlinien

Clements et al. beschreibt eine Produktlinie als ein Gruppe von Produkten mit einer gemeinsam verwalteten Menge an Features, welche bestimmten Ansprüchen von einem gewählten Markt oder Auftragsgebiet entsprechen [CN01]. Der bekannteste Markt in dem Produktlinien eingesetzt werden, ist die Automobilindustrie. Kein Auto wird heutzutage von Grund auf neu designt, es werden lediglich neue Features zu einem bestehenden Framework hinzugefügt [KCH+90].
Die Produktlinientechnologie kann direkt auf die Softwareentwicklung übertragen werden, weshalb auch die Definition einer Software-Produktlinie von Clements et al. wie folgt dargelegt wird [CN01]:

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

Demnach handelt es sich bei einer Software-Produktlinie um eine Menge von Softwareprodukten mit gleichen Features, die die Anforderungen an ein bestimmtes Marktsegment oder einem bestimmten Auftrag erfüllen, indem eine gemeinsame Menge von

entwickelten Kernkonzepten in vorgeschriebener Weise verwendet wird.
Durch die Nutzung einer Software-Produktlinie, können vom gleichen Quelltext verschiedene Produkte abgeleitet werden, die auf die Anforderungen des jeweiligen Kunden zugeschnitten sind. Diese verschiedenen Endprodukte werden im Folgenden *Varianten* genannt. Die Gemeinsamkeiten und Unterschiede der verschiedenen Varianten werden durch *Features* ausgedrückt, die folgendermaßen von Böckle et al. definiert werden [BKPS04]:

> *„Als Feature bezeichnet man wesentliche, meist für Benutzer sichtbare funktionale oder nichtfunktionale Leistungsmerkmale (Charakteristika) eines Produkts.“*

Die Entwicklung mittels Produktlinientechnologie anstelle von einer grundlegenden Neuentwicklung eines Programms verspricht einige Vorteile. Ein großer Vorteil von Produktlinien ist beispielsweise die Verringerung von Entwicklungskosten eines Programms, die sich schon bei etwa drei verschiedenen Varianten einstellt [PBvdL05]. Demnach ist die Wiederverwendung von Softwareartefakten eines der größten Vorteile einer Produktlinie, die einen Großteil an Entwicklungskosten einspart. Weitere Vorteile stellen die verkürzte Entwicklungszeit der Produkte einer bestehenden Produktlinie oder auch die höhere Qualität der Produkte, welche durch den gemeinsamen Quelltext öfter getestet werden als bei einer Neuentwicklung, dar. Kostenersparnis, Zeitersparnis und Qualitätssteigerung sind lediglich eine kleine Auswahl an vielversprechenden Vorteilen, die aus der Verwendung einer Produktlinie resultieren. Weitere Vorteile werden unter Anderem von Pohl et al. in [PBvdL05] aufgelistet und können dort nachgeschlagen werden.
Anschließend werden weitere Erläuterungen zur Darstellung einer Software-Produktlinie vorgestellt. Da sich diese Arbeit mit dem Bereich der Softwaretechnik befasst, wird im Folgenden anstelle des Begriffs Software-Produktlinie der Begriff Produktlinie als Synonym verwendet.

2.1.1 Feature-Modelle

Feature-Modelle wurden von Kang et al. eingeführt [KCH+90] und beispielsweise von Batory als eine hierarchisch geordnete Menge von Features beschrieben [Bat05]. Im Speziellen können durch Feature-Modelle die verschiedenen gültigen Kombinationen einer Produktlinie beschrieben und sich gegenseitig ausschließende Features verhindert werden. Beispielsweise kann durch die Nutzung von Feature-Modellen auf der Ebene der Modellierung sichergestellt werden, dass in einem Auto nicht zugleich ein automatisches als auch ein manuelles Getriebe eingebaut wird.
Die grafische Darstellung eines Feature-Modells wird als *Feature-Diagramm* bezeichnet. Die genaue Darstellung der Feature-Diagramme variiert in verschiedenen Quellen [KCH+90, CE00, CHE05, CW07], weshalb zunächst die hier verwendete Syntax an einem einfachen Beispiel dargestellt wird.
In Abbildung 2.1 auf der nächsten Seite ist ein Feature-Diagramm einer Produktlinie der Automobilindustrie abgebildet [CW07]. Anhand dieses Beispiels erfolgt zunächst die Beschreibung der einzelnen Komponenten eines Feature-Diagramms. Das oberste

Element des hierarchisch geordneten Diagramms ist das sogenannte *Wurzel*-Feature, das den Namen `Auto` besitzt und zugleich die gesamte Auto-Produktlinie repräsentiert. Bei der Einführung von Feature-Modellen durch Kang et al. wurde die hierarchische Ordnung durch einen *und*/*oder*-Baum beschrieben [KCH+90]. Darin wurden alle Features, die hierarchisch untergeordnet sind, als *Kind*-Features bezeichnet. Wohingegen ein übergeordnetes Feature als *Eltern*-Feature bezeichnet wurde. Demnach handelt es sich bei dem Feature `Karosserie` um ein Kind-Feature des Wurzel-Features `Auto`, wobei dies wiederum das Eltern-Feature der `Karosserie` ist. Im und/oder-Baum werden *optionale* Features, wie die Features `Schluesselloses_Zugangssystem` und `Zentralverriegelung`, mit einem leeren Kreis dargestellt. Die Darstellung von *alternativen* Features, bei der jeweils genau ein Feature ausgewählt werden muss, erfolgt durch einen Kreisbogen. In Abbildung 2.1 werden beispielsweise die Features `Automatik` und `Manuell` als verschiedene alternative Getriebearten eines Autos abgebildet. Des Weiteren wird das notwendige Vorhandensein der Eltern-Features bei der Wahl der Kind-Features durch die verschiedenen Linien repräsentiert. Demnach muss bei der Wahl von `Automatik` beziehungsweise `Manuell` ein Auto das Feature `Getriebe` besitzen. Features, die weder alternative noch optionale Features sind, müssen immer gewählt werden, wenn das zugehörige Eltern-Feature gewählt wurde. Diese Features wurden von Batory als *obligatorische* (Englisch: *mandatory*) Features bezeichnet [Bat05], die in der zugehörigen Abbildung durch einen gefüllten Kreis dargestellt werden. Des Weiteren bietet die Darstellung von Batory die Möglichkeit *oder*-Beziehungen zu formulieren, bei der mindestens ein Feature ausgewählt werden muss. Beispielsweise handelt es sich bei den Features `Elektrisch` und `Benzin` des Features `Motor` um eine oder-Beziehung, dessen Repräsentation durch einen gefüllten Kreis erfolgt.

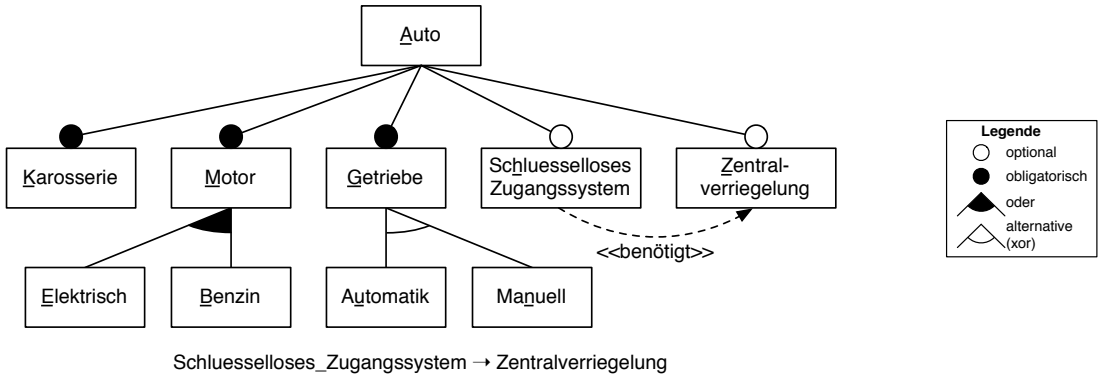


Abbildung 2.1: Feature-Modell einer Auto-Produktlinie [CW07].

Jedoch können nicht alle Beziehungen zwischen den einzelnen Features immer direkt in einer Baumstruktur abgebildet werden. Daher werden aussagenlogische Ausdrücke verwendet, um diese Beziehungen zwischen den verschiedenen Features darstellen zu können. Diese Beziehungen werden in zusätzliche Bedingungen formuliert, die meist direkt unter dem Feature-Diagramm angegeben werden. In Abbildung 2.1 wird für die Produktlinie des Autos beispielsweise zusätzlich gefordert, dass bei der Wahl des Features `Schluesselloses_Zugangssystem` auch die `Zentralverriegelung` vorhanden sein muss. Des Weiteren gibt es auch die Möglichkeit diese Beziehung grafisch darzustellen,

wie es durch den Pfeil <<*benötigt*>> angedeutet wird. Dies ist äquivalent zu der aussagenlogischen Darstellung der Beispielbedingung. Im Allgemeinen ist die Repräsentation mittels Pfeil nicht so flexibel wie der aussagenlogische Ausdruck, da in diesem auch komplexere Beziehungen dargestellt werden können.
Anhand des Feature-Modells kann ein Kunde nun sein Auto selbst zusammenstellen. In diesem Zusammenhang wählt der Kunde die jeweiligen Features unter Beachtung der dargestellten Bedingungen des Feature-Modells aus. Diese Featureauswahl wird als *Konfiguration* bezeichnet. Beispielsweise kann durch die Auswahl der Features `{Auto, Karosserie, Motor, Benzin, Getriebe, Automatik}` ein dementsprechend konfiguriertes Auto aus der Auto-Produktlinie (siehe Abbildung 2.1 auf der vorherigen Seite) erzeugt werden.
Die Darstellung einer Produktlinie als Feature-Modell ermöglicht eine gute Übersicht über die verschiedenen möglichen Varianten, die aus der Produktlinie generiert werden können. Im Folgenden wird eine weitere Darstellung eines Feature-Modells vorgestellt, die weitere Analysen bezüglich dieser Modelle ermöglicht.

2.1.2 Aussagenlogische Repräsentation von Feature-Modellen

Feature-Diagramme ermöglichen eine einfache Vorstellung über die Produktvielfalt einer Produktlinie und deren möglichen Konfigurationen. Im Gegensatz dazu bieten *aussagenlogische Ausdrücke* eine logische Repräsentation der Feature-Modelle [Bat05]. Unter Verwendung dieser Darstellung kann die Gültigkeit einer gegebenen Konfiguration bezogen auf ein bestimmtes Feature-Modell getestet werden.
Für die Darstellung eines Feature-Modells als aussagenlogischen Ausdruck wird jedem Feature eine boolesche Variable zugewiesen und die Beziehungen zwischen den Features durch die Verwendung von den logischen Operatoren $\wedge, \vee, \neg, \rightarrow, \leftrightarrow$ dargestellt.
Das Feature-Modell der Auto-Produktlinie (siehe Abbildung 2.1 auf der vorherigen Seite) kann wie folgt in einen aussagenlogischen Ausdruck dargestellt werden [CW07]. Aus Gründen der Übersichtlichkeit, werden die einzelnen Features durch Abkürzungen repräsentiert, die im Feature-Modell durch die unterstrichenen Buchstaben gekennzeichnet wurden:

$$
\begin{aligned}
Root - Feature: &\quad a \\
Kind - Eltern: &\wedge \quad (k \rightarrow a) \\
&\wedge \quad (m \rightarrow a) \wedge (e \rightarrow m) \wedge (b \rightarrow m) \\
&\wedge \quad (g \rightarrow a) \wedge (u \rightarrow g) \wedge (n \rightarrow g) \\
&\wedge \quad (h \rightarrow a) \\
&\wedge \quad (z \rightarrow a) \\
Obligatorisch: &\wedge \quad (a \rightarrow k) \wedge (a \rightarrow m) \wedge (a \rightarrow g) \\
Oder - Gruppe: &\wedge \quad (m \rightarrow e \vee b) \\
Alternativ - Gruppe: &\wedge \quad (g \rightarrow (\neg n \wedge u) \vee (n \wedge \neg u)) \\
Zusatzbedingungen: &\wedge \quad (h \rightarrow z)
\end{aligned}
$$

Des Weiteren kann durch die Verwendung der aussagenlogischen Repräsentation die Korrektheit des Feature-Modells kontrolliert werden. Die Modellierung eines Feature-

Modells ist korrekt, wenn aus dem Feature-Modell mindestens eine Variante abgeleitet werden kann. Im Fall der logischen Repräsentation als aussagenlogischer Ausdruck ist dies gleichbedeutend mit dem Erfüllbarkeitsproblem der Aussagenlogik. Demnach wird für den gegebenen logischen Ausdruck der Auto-Produktlinie eine Wertzuweisung der Variablen gesucht, bei der der Ausdruck erfüllt (wahr) ist. In gleicher Weise kann auch eine Konfiguration eines Feature-Modells auf Gültigkeit überprüft werden. Zu diesem Zweck wird jeder Variable eines ausgewählten Features der Wert „wahr“ sowie alle anderen Variablen der Wert „falsch“ zugewiesen und der aussagenlogische Ausdruck auf Erfüllbarkeit getestet.
Das Problem der Erfüllbarkeit von aussagenlogischen Ausdrücken ist NP-vollständig, demnach gibt es theoretisch keinen bekannten Algorithmus der für alle aussagenlogischen Ausdrücke in polynomialer Zeit eine Lösung bestimmen kann. Zur Bestimmung einer Lösung für aussagenlogische Ausdrücke werden meist SAT-Solver (Englisch: Satisfiability-Solver) verwendet. SAT-Solver sind eine Klasse von Algorithmen, die zumindest für einen Großteil der aussagenlogischen Ausdrücke eine Lösung bestimmen können. In der Praxis finden sie daher für die meisten Feature-Modelle innerhalb weniger Sekunden eine Lösung [MWC09].

2.1.3 Beziehungen zwischen Produktlinien

Bei der Bearbeitung von Produktlinien wird ein Feature-Modell als Eingabe verwendet, um ein neues verändertes Feature-Modell zu erzeugen. Thüm et al. kategorisierten die verschiedenen Beziehungen des Eingabemodells und des Ausgabemodells anhand der zugehörigen Varianten [TBK09]. Demzufolge können bei der Bearbeitung von einem Feature-Modell sowohl Produkte entstehen als auch entfernt werden. Die folgende Tabelle (siehe Tabelle 2.1) wurde von Thüm et al. übernommen und ermöglicht einen Überblick über die Klassifikation der Bearbeitung von Feature-Modellen.

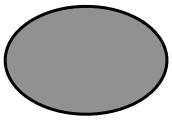
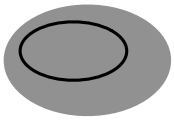
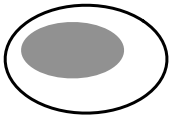
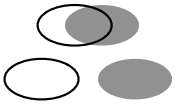
	Keine Variante hinzugefügt	Variante hinzugefügt
Keine Variante entfernt	 Refaktorisierung	 Generalisierung
Variante entfernt	 Spezialisierung	 willkürliche Änderung

Tabelle 2.1: Klassifikation von Änderungen an Feature-Modellen [TBK09].

Dementsprechend werden bei einer *Spezialisierung* einer Produktlinie existierende Varianten entfernt aber keine neuen Varianten hinzugefügt. Genau entgegengesetzt verhält es sich bei der *Generalisierung*, bei der neue Varianten hinzugefügt und keine existierenden Varianten entfernt werden. Die Bearbeitung eines Feature-Modells, bei der sowohl neue Varianten entstehen als auch existierende Varianten entfernt werden, bezeichnet

Thüm et al. als willkürliche Änderungen. Zur Vollständigkeit der möglichen Bearbeitungen eines Feature-Modells wurde in dieser Übersicht ebenfalls die Refaktorisierung angegeben, die aber für diese Arbeit nicht von Relevanz ist. Hierbei handelt es sich um eine Änderung des Feature-Modells, bei der weder Varianten hinzugefügt noch entfernt werden. Dies kann beispielsweise aus der Umbenennung eines Features resultieren.

2.1.4 Schrittweiser Konfigurationsprozess

Czarnecki et al. führten zudem den Begriff des *schrittweisen Konfigurationsprozesses* (Englisch: Staged Configuration) ein [CHE05]. Hierbei handelt es sich um eine Konfiguration die in mehreren Schritten vollzogen wird und nicht in einem Schritt alle benötigen Features zur Beschreibung einer Variante definieren muss. In jedem Schritt wird demzufolge eine Auswahl von verschiedenen Features getroffen, die wiederum die Auswahl weiterer Features in einem folgenden Schritt einschränkt. Bei jedem dieser Schritte wird ein Feature-Modell als Eingabe verwendet, eine Auswahl von Features getroffen und ein entsprechendes spezialisiertes Feature-Modell ausgegeben.

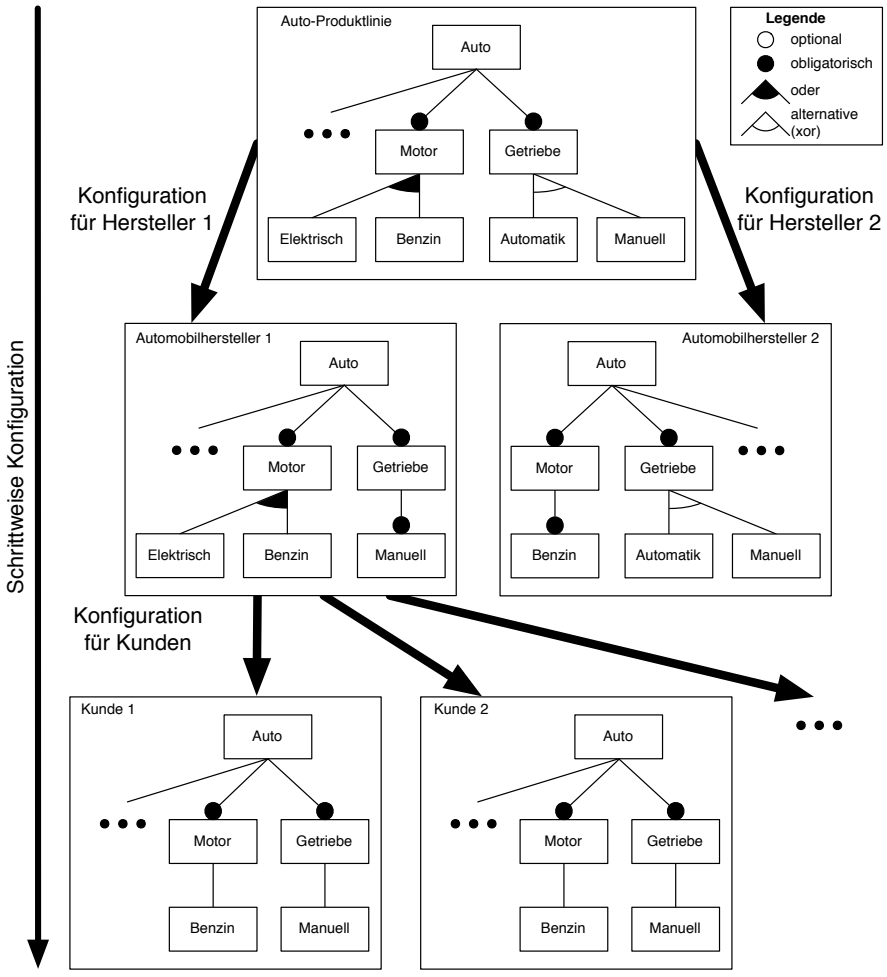


Abbildung 2.2: Schrittweiser Konfigurationsprozess am Beispiel der Auto-Produktlinie.

Zur Veranschaulichung und zur Motivation des schrittweisen Konfigurationsprozesses wird im Folgenden die Auto-Produktlinie verwendet (siehe Abbildung 2.2 auf der vorherigen Seite). Hierbei handelt es sich um eine Produktlinie, die für Fertigungsanlagen verschiedener Firmen verwendet werden könnte. In diesem Zusammenhang kann die Notwendigkeit bestehen, die resultierenden Produkte der Produktlinie weiter zu beschränken. Dieser Vorgang kann durch den schrittweisen Konfigurationsprozess vollzogen werden. Beispielsweise unterstützt ein Automobilhersteller keine Fertigung von elektrischen Motoren und ein Anderer verbaut kein Automatik-Getriebe. In einem ersten Konfigurationsschritt können daher zwei unterschiedliche Feature-Modelle aus dem ursprünglichen Feature-Modell generiert werden. Zur Bestimmung einer Variante durch den Endkunden wurde somit die Anzahl der notwendigen restlichen Konfigurationen vermindert. Bei dem ersten Automobilhersteller kann der Endkunde zwar zwischen den verschiedenen Motoren wählen, bei dem verwendeten Getriebe muss er jedoch das einzig verbliebene manuelle Getriebe auswählen. Ähnlich verhält sich die Situation beim zweiten Automobilhersteller. In diesem Fall hat der Endkunde die Wahl zwischen dem Getriebe, aber keine Wahl bezüglich des Motors.
Das hier verwendete Beispiel verbildlicht lediglich die Notwendigkeit einer schrittweisen Konfiguration. Jedoch sind jeweilige Konfigurationsketten in Wirklichkeit deutlich länger. Komplexere Beispiele können den zugehörigen Ausführungen von Czarnecki et al. entnommen werden [CHE05].

2.2 Multiproduktlinien

Der größte Vorteil von Produktlinien ist die Wiederverwendung von Softwareartefakten und die damit verbundene Kostenersparnis. Die Wiederverwendung von bestehender Funktionalität ist daher von großer Bedeutung, weshalb weitere Möglichkeiten der Wiederverwendung sehr vielversprechend sind. Aus diesem Grund ist auch bei der Entwicklung von komplexen Produktlinien die Wiederverwendung von Basisfunktionalität erwünscht, die schon in bestehenden Produktlinien realisiert wurde. Dies resultiert nach Krüger in Produktlinien aus Produktlinien beziehungsweise verschachtelten Produktlinien [Kru06]. Im Gegensatz dazu prägten Rosenmüller und Siegmund den Begriff *Multi-Softwareproduktlinien* oder kurz *Multiproduktlinien* für jegliche Art der Komposition von Produktlinien [RS10]. Angelehnt an die Definition von Rosenmüller und Siegmund wird auch in dieser Arbeit der Begriff Multiproduktlinien verwendet.
Zunächst wird im folgenden Abschnitt anhand eines einfachen Beispiels die Notwendigkeit von Multiproduktlinien motiviert. Anschließend wird eine weitere Möglichkeit zur Darstellung von Multiproduktlinien vorgestellt, die von Rosenmüller und Siegmund eingeführt wurde. Wenn nicht anders angegeben, wurden die folgenden Ausführungen von Rosenmüller et al. übernommen [RS10, RSuRK08].

2.2.1 Motivation

In Abbildung 2.3 auf der nächsten Seite erfolgt die Darstellung einer Produktlinie `Mail_Client`. Diese Produktlinie verwendet eine Produktlinie `Mail_Framework` und verschiedene Varianten einer zweiten Produktlinie für Listen, `Sortierte_Liste` und

`Synchronisierte_Liste`. Die Varianten einer Produktlinie in einer anderen Produktlinie werden von Rosenmüller et al. als *Instanzen* bezeichnet [RSuRK08].

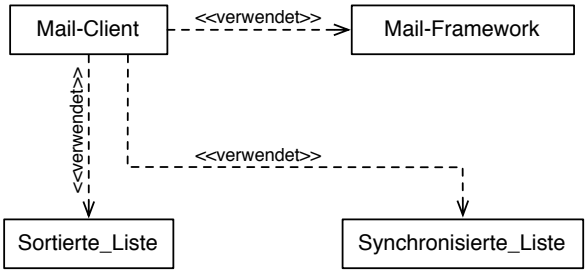


Abbildung 2.3: Die Produktlinie `Mail_Client` verwendet Produktlinie `Mail_Framework` und Varianten einer Produktlinie Liste [RSuRK08].

Durch die Wiederverwendung der bestehenden Produktlinie `Mail_Framework` und die mehrfache Verwendung der jeweiligen Instanzen der Produktlinie `Liste` kann ein enormer Implementierungsaufwand für die Produktlinie `Mail_Client` eingespart werden. Des Weiteren wird durch die Verwendung von Instanzen die mehrfache Modellierung und Implementierung einer Liste verhindert, dessen Erstellung für die verschiedenen Konfigurationen notwendig gewesen wäre.
Durch verschiedene mögliche Abhängigkeiten zwischen Produktlinien unterscheiden Rosenmüller und Siegmund zwischen (1) *hierarchische Multiproduktlinien* und (2) *flache Multiproduktlinien* [RS10]. Bei hierarchischen Multiproduktlinien bestehen Abhängigkeiten zwischen den Produktlinien einer Multiproduktlinie, die eine Hierarchie aus Produktlinien bilden. Demnach handelt es sich bei dem Beispiel der Produktlinie `Mail_Client` um eine hierarchische Multiproduktlinie, da unter Anderem die Produktlinie `Mail_Client` von der Produktlinie `Sortierte_Liste` abhängig ist. Im Gegensatz dazu, ist die `Sortierte_Liste` unabhängig von der Produktlinie `Mail_Client` und kann isoliert entwickelt werden.
Als flache Multiproduktlinien werden diejenigen Multiproduktlinien bezeichnet, die keine Hierarchie bilden [RS10]. In diesem Zusammenhang kann man wiederum das Beispiel des `Mail_Client` betrachten. Wenn bei der Entwicklung der Produktlinie `Mail_Client` zugleich auch eine Produktlinie `Mail_Server` entwickelt wurde, besteht zwischen diesen beiden Produktlinien eine flache Multiproduktlinien-Beziehung. Diese beiden Produktlinien müssen aufeinander abgestimmt werden, um beispielsweise das gleiche Protokoll zu unterstützen. Die Produktlinien `Mail_Client` und `Mail_Server` sind gegenseitig voneinander abhängig, da die Auswahl eines Features der einen Produktlinie die Auswahl eines Features der anderen Produktlinie bedingt.

2.2.2 Darstellung von Multiproduktlinien

Bei der Einführung der Produktlinien wurde das Feature-Diagramm als grafische Darstellung vorgestellt (siehe Abschnitt 2.1.1 auf Seite 6). Nach Czarnecki et al. können in einem Feature-Diagramm auch mehrere Wurzel-Features und somit auch mehrere Produktlinien in einem Feature-Diagramm abgebildet werden [CHE05]. Aus diesem

Grund können auch teilweise Abhängigkeiten zwischen den verschiedenen Produktlinien einer Multiproduktlinie in einem Feature-Diagramm dargestellt werden (siehe Abbildung 2.4).

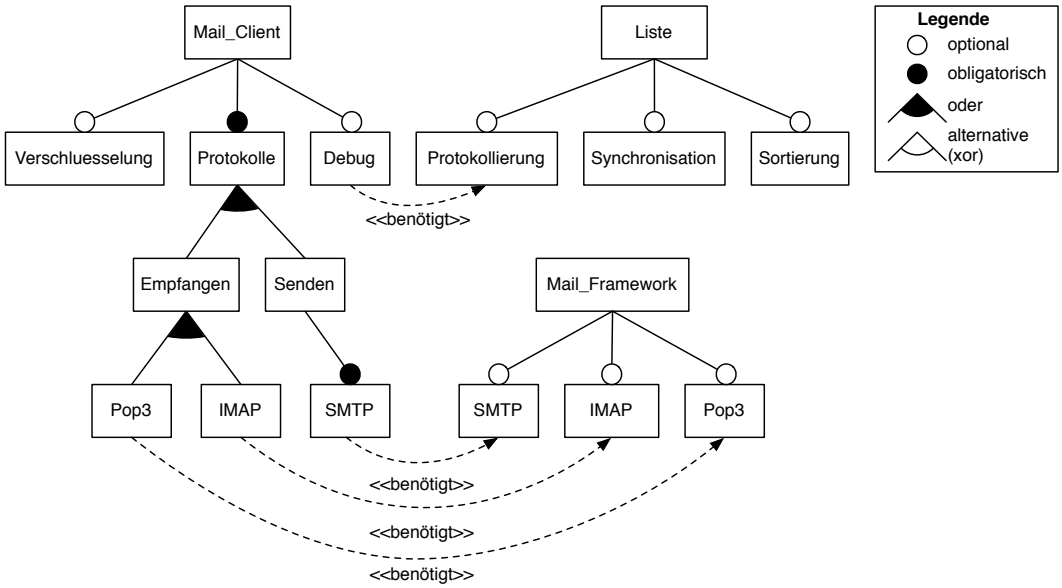


Abbildung 2.4: Darstellungen von abhängigen Produktlinien im Feature-Diagramm [RSuRK08].

Das Feature-Diagramm der Multiproduktlinie `Mail_Client` besitzt die drei Wurzel-Features `Mail_Client`, `Mail_Framework` und `Liste`. Wie bei Feature-Modellen, die einzelne Produktlinien darstellen, können zusätzliche Bedingungen zwischen Features angegeben werden. Diese Bedingungen können auch zwischen Features verschiedener Produktlinien definiert werden [CHE05]. Beispielsweise bedingt die Wahl eines bestimmten Protokolls (`POP3`, `IMAP`, `SMTP`) der Produktlinie `Mail_Client` auch das entsprechende Protokoll der kooperierenden Produktlinie `Mail_Framework`.
Im Feature-Diagramm der Produktlinie `Mail_Client` (siehe Abbildung 2.4) werden die Abhängigkeiten zwischen den einzelnen Produktlinien dargestellt. In dieser Darstellung können jedoch keine Abhängigkeiten zwischen den einzelnen Instanzen der Produktlinie `Liste` und der Produktlinie `Mail_Client` dargestellt werden, da diese Instanzen unterschiedliche Konfigurationen benötigen. Beispielsweise verwendet die Instanz `Synchronisierte_Liste` der Produktlinie `Liste` das Feature `Synchronisation` und die Instanz `Sortierte_Liste` das Feature `Sortieren`. Lediglich die Bedingung bezüglich des Features `Protokollierung` kann im Feature-Diagramm dargestellt werden, da es bei Auswahl des Feature `Debug` in beiden Instanzen integriert werden muss. Des Weiteren handelt es sich bei diesem Beispiel um kleinere Produktlinien. Bei größeren Produktlinien ist die Darstellung dementsprechend unübersichtlicher und kaum noch handhabbar. Zur Darstellung von verschiedenen Instanzen einer Produktlinie und deren Abhängigkeiten untereinander sowie zu einer anderen Produktlinie, wurde von Rosenmüller et al. eine erweiterte Modellierung von Produktlinien vorgeschlagen. Diese ermöglicht die Darstellung von komplexen Multiproduktlinien und deren Instanzen [RSuRK08]. Die Erweiterungen basieren auf den Ausführungen von Czarnecki et al., der Klassen von

objektorientierter Programmierung (OOP) mit einer Produktlinie und Varianten einer Produktlinie mit der Instantiierung eines Objekts vergleicht [CHE05]. Zusätzlich wurden von Rosenmüller et al. weitere Beziehungen zwischen objektorientierter Programmierung und Produktlinien zusammengetragen (siehe Tabelle 2.2). Unter Verwendung der Aggregation kann demnach eine Produktlinie verschiedene Variablen vom Typ einer Produktlinie besitzen und eine konkrete Instanz zugewiesen werden. Zusätzlich wird durch die Vergabe von eindeutigen Namen der verschieden konfigurierten Instanzen eine eindeutige Identifizierung und Definition von Bedingungen bezüglich dieser Instanzen ermöglicht. Des Weiteren wird der schrittweise Konfigurationsprozess (siehe Abschnitt 2.1.4 auf Seite 10) von Feature-Modellen in Beziehung mit der Spezialisierung von objektorientierten Klassen gesetzt.

Objektorientiertes Konzept	Produktlinien-Darstellung
Klasse	Produktlinie
Objekt	Produktlinien-Instanz
Spezialisierung	Schrittweiser Konfigurationsprozess
Aggregation	Verwendet-Beziehung zwischen Produktlinien (siehe Abbildung 2.3 auf Seite 12)
Typ einer Variable	Typ der Produktlinien Instanz
Name einer Variable	Name einer Produktlinien Instanz

Tabelle 2.2: Objektorientierte Konzepte und zugehörige Repräsentation für Produktlinien [RSuRK08].

Durch den Vergleich von Produktlinien mit objektorientierter Programmierung ist es möglich, ebenfalls Klassendiagramme von UML (Unified Modeling Language) zur Modellierung von Produktlinien-Kompositionen zu verwenden. In Abbildung 2.5 auf der nächsten Seite wird unter Nutzung von UML-Klassendiagrammen veranschaulicht, wie die verschiedenen Instanzen in der Multiproduktlinie `Mail_Client` modelliert werden können. Ergänzende Bedingungen, die weitere Beziehungen zwischen den Produktlinien angeben, werden später noch genauer erläutert.
Im UML-Klassendiagramm sind die verschiedenen Produktlinien `Mail_Client`, `Liste`, `Sortierte_Liste` und die `Synchronisierte_Liste` abgebildet. Die Konfigurationen der Produktlinie `Liste` zu den Spezialisierungen `Synchronisierte_Liste` und `Sortierte_Liste` werden hier durch Vererbung repräsentiert. Die damit in Verbindung stehende Verringerung der Anzahl der Konfiguration wird durch die Zuweisung der jeweiligen Features `Synchronisation` beziehungsweise `Sortierung` dargestellt. Diese Zuweisung wird in der Abbildung 2.5 auf der nächsten Seite in Form von Attributen `Synchronisation : bool = true` und `Sortierung : bool = true` angegeben. Des Weiteren wird durch die Aggregation von UML-Klassendiagrammen, die Verwendung der verschiedenen Instanzen in einer Produktlinie dargestellt. In Abbildung 2.5 auf der nächsten Seite wird dies durch die Instanzen mit dem Namen `mailListe` und `fehlerListe` veranschaulicht. Dieses UML-Klassendiagramm beschreibt nun eindeutig die Zusammenhänge zwischen den verschiedenen Produktlinien der Multiproduktlinie `Mail_Client`.

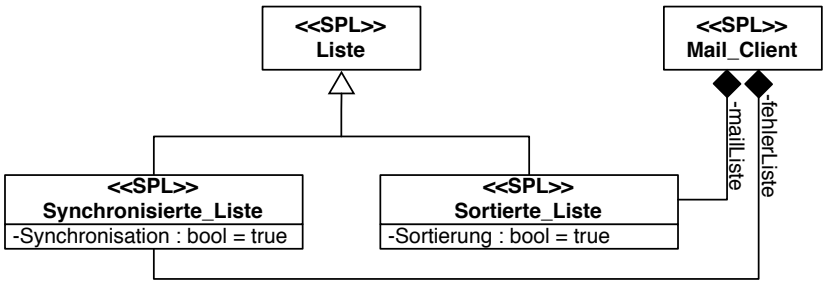


Abbildung 2.5: Verwendung von Klassendiagrammen zur Modellierung von Kompositionen in Multiproduktlinien [RSuRK08].

Zur Sicherstellung einer gültigen Konfiguration der Multiproduktlinie `Mail_Client` können ähnlich wie bei einem Feature-Modell weitere Bedingungen angegeben werden. In diesem Zusammenhang unterscheiden Rosenmüller et al. verschiedene Arten von Bedingungen, den Domänen-Bedingungen (Englisch: domain constraints) und den Instanz-Bedingungen (Englisch: instance constraints) [RSuRK08].

Domänen-Bedingungen:

(1) Mail-Client.Debug → Liste.Protokollierung

Instanz-Bedingungen:

(2) Mail-Client.Protokolle.Empfang.Pop3 → Mail-Client.mail-framework.Pop3

(3) Mail-Client.Protokolle.Empfang.IMAP → Mail-Client.mail_framework.IMAP

(4) Mail-Client.Protokolle.Empfang.SMTP → Mail-Client.mail_framework.SMTP

Als Domänen-Bedingungen gelten zwei verschiedene Arten von Bedingungen. Einerseits handelt es sich um Domänen-Bedingungen wenn Abhängigkeiten zwischen verschiedenen Features einer oder mehrerer Produktlinien dargestellt werden (alle Bedingungen in Abbildung 2.4 auf Seite 13). Andererseits handelt es sich auch um eine Domänen-Bedingung wenn diese für alle Instanzen einer bestimmten Produktlinie gilt. Beispielsweise handelt es sich bei der Bedingung (1) um eine solche Bedingung, da das Feature `Protokollierung` für alle Instanzen der Produktlinie `Liste` gefordert wird.
Im Gegensatz dazu beschreibt eine Instanz-Bedingung die Abhängigkeit zwischen Produktlinien und konkreten Instanzen einer anderen Produktlinie. Demnach handelt es sich bei jeder Bedingung, in der eine Instanz aufgeführt wird, um eine Instanz-Bedingung. Unter Verwendung dieser Bedingung kann beispielsweise die Übereinstimmung der verwendeten Protokolle (`Pop3`, `IMAP` und `SMTP`) vom `Mail-Client` mit der Instanz der Produktlinie `Mail-Framework` beschrieben werden (Bedingung (2)-(4)).

3. Modellierung in VELVET

Im vorherigen Kapitel wurden verschiedene Beispiele für Produktlinien und Multiproduktlinien aufgeführt. Im Gegensatz zu diesen Beispielen können Feature-Modelle von Produktlinien in der Realität sehr komplexe Ausmaße annehmen und so die Übersichtlichkeit und die Handhabbarkeit der Feature-Modelle stark beeinträchtigen. Aus diesem Grund ist es oftmals sinnvoll, verschiedene kleinere Produktlinien zu modellieren, um weiterhin die Handhabbarkeit zu gewährleisten [RSTS11]. Des Weiteren können diese kleineren Produktlinien leichter wiederverwendet und in komplexere Multiproduktlinien integriert werden. Zur einfachen Beschreibung von Produktlinien und zur Wiederverwendung von Produktlinien in komplexeren Multiproduktlinien entwickelten Rosenmüller et al. die Modellierungssprache VELVET [RSTS11]. VELVET vereint die Modellierung von Produktlinien und Multiproduktlinien sowie die Beschreibung eines Konfigurationsprozesses.
Die Modellierung unter Verwendung von VELVET verspricht verschiedene Vorteile: (1) Wiederverwendung der Produktlinien, (2) Wiederverwendung von Konfigurationsentscheidungen und (3) Unterstützung des schrittweisen Konfigurationsprozesses.

(1) Die Wiederverwendung von Produktlinien in VELVET wird auf unterschiedlichste Weise unterstützt. In diesem Zusammenhang stellt VELVET sogenannte Kompositionsmechanismen bereit, die zur Kombination der verschiedenen Produktlinien verwendet werden können. Bei den Kompositionsmechanismen handelt es sich um *Vererbung*, *Aggregation* und *Superimposition*.

(2) Da VELVET neben der Modellierung von Feature-Modellen auch die Konfiguration von Feature-Modellen unterstützt, können getroffene Konfigurationsentscheidungen wiederverwendet und somit Zeit und Geld gespart werden.

(3) Ein weiterer Vorteil von VELVET ist die Unterstützung des schrittweisen Konfigurationsprozesses (siehe Abschnitt 2.1.4 auf Seite 10). Demnach ermöglicht VELVET ein bestehendes Feature-Modell als Eingabe zu verwenden und ein neues spezialisiertes Feature-Modell zu beschreiben.

Das Ziel dieser Arbeit ist die Sicherstellung der Konsistenz von Multiproduktlinien. Da VELVET eine Modellierungssprache für Multiproduktlinien mit großen Vorteilen ist, wird in diesem Zusammenhang speziell die Konsistenz von VELVET-Modellen analysiert. Im Detail werden dabei die verschiedenen Kompositionsmechanismen der Vererbung und der Aggregation genauer betrachtet. Aus diesem Grund wird im Folgenden die VELVET-Grammatik schrittweise vorgestellt. Dabei werden auch notwendige Änderungen der Grammatik vorgestellt, die zur Sicherstellung der Konsistenz in dieser Arbeit durchgeführt wurden. Des Weiteren muss die gesamte Multiproduktlinie in einem aussagenlogischen Ausdruck überführt werden. Dies ermöglicht weitere Tests, die beispielsweise überprüfen, ob eine gültige Variante aus dem Modell abgeleitet werden kann. Aus diesem Grund wird die jeweils vorgestellte Komposition aus verschiedenen VELVET-Modellen in einem Gesamtmodell dargestellt. Dadurch ist die einfache Überführung in einen aussagenlogischen Ausdruck gewährleistet.
Die ursprüngliche Beschreibung von VELVET basiert auf den Ausführungen von Rosenmüller et al. [RSTS11]. In diesen Ausführungen wurden Ausschnitte der VELVET-Grammatik dargestellt. Die Grammatik wurde in der Zwischenzeit jedoch von den Autoren verfeinert. Daher diente als Grundlage dieser Arbeit die aktuelle Version der VELVET-Grammatik, die von der zugehörigen Webseite entnommen werden kann[1] (siehe Kapitel A auf Seite 77). Des Weiteren werden alle Änderungen, die an VELVET in dieser Arbeit vorgenommen wurden, in Kapitel A auf Seite 77 noch einmal vollständig dargestellt.

3.1 Produktlinienmodellierung in VELVET

VELVET ist eine Modellierungs- und Konfigurationssprache, die es erlaubt verschiedene Produktlinien zu kombinieren und zu erweitern. Grundlage für diese Kombination ist die Beschreibung der einzelnen Produktlinien, deren Definition unter Verwendung von VELVET im Folgenden beschrieben wird.

Grammatik

Zunächst wird im Quelltext 3.1 auf der nächsten Seite ein Auszug der VELVET-Grammatik gegeben, der zur Definition einer einfachen Produktlinie verwendet wird. Bestandteile, die für eine erweiterte Modellierung benötigt werden, werden in diesem Quelltext nicht angegeben oder *gelb* hinterlegt.
Das Wurzel-Feature einer Produktlinie wird in VELVET mit der Definition eines Konzepts unter Verwendung des Schlüsselwortes `concept` und dem zugehörigen Produktlinien-Namen (`ConceptName`) abgebildet (Zeile 1). Diese Definition gleicht einer Klassendefinition in objektorientierter Programmierung, der in diesem Fall die Konzept-Definition (`ConceptBody`) mit den verschiedenen Features, Bedingungen und Gruppen folgt (Zeile 4, 5). Ein Feature wird durch das Schlüsselwort `feature` definiert (Zeile 8). Es ist standardmäßig optional, kann aber durch das Schlüsselwort `mandatory` als obligatorisch definiert werden (Zeile 8). Im Block eines Features können abermals Features,

[1] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/MultiPLe/modeling.htm

Gruppen sowie Bedingungen angegeben werden (Zeile 10, 5). Des Weiteren können auch die verschiedenen Gruppen durch die Schlüsselwörter `someof` (Zeile 14) für eine Oder-Gruppe und `oneof` (Zeile 15) für eine Alternative-Gruppe definiert werden.

```
1  Concept: ["refines"] "concept" ConceptName [ConceptBases]
2                                                  ConceptBody ;
3  ConceptName: ID ;
4  ConceptBody: "{" Definition* "}" ;
5  Definition: (Feature | FeatureGroup | Constraint | Instance
6                                                  | Attribute) ;
7
8  Feature: ["mandatory"] "feature" FeatureName (FeatureDef | ";") ;
9  FeatureName: ID ("," ID)* ;
10 FeatureDef: "{" Definition* "}" ;
11
12 FeatureGroup: GroupType GroupChilds ;
13 GroupType: (SomeOfGroup | OneOfGroup) ;
14 SomeOfGroup: "someof" ;
15 OneOfGroup: "oneof" ;
16 GroupChilds: "{" Feature+ "}" ;
```

Quelltext 3.1: VELVET-Grammatik Teil I.

Da die Regeln `ConceptBody` und `FeatureDef` übereinstimmen (siehe Quelltext 3.1), wurde im Rahmen dieser Arbeit eine Vereinfachung der Grammatik vorgenommen. Hierbei wurden diese beiden Elemente unter einer neuen Regel `Body` zusammengefasst. Diese grammatikalische Änderung wird noch einmal in Quelltext 3.2 dargestellt. Angepasste Bereiche werden dabei *grün* hinterlegt.

```
1  Concept: ["refines"] "concept" ConceptName [ConceptBases] Body ;
2  ConceptName: ID ;
3  Body: "{" Definition* "}" ;
4  Definition: (Feature | FeatureGroup | Constraint | Instance
5                                                  | Attribute) ;
6
7  Feature: ["mandatory"] "feature" FeatureName ( Body | ";") ;
```

Quelltext 3.2: Verbesserung der VELVET-Grammatik Teil I.

Der bisher vorgestellte Ausschnitt von VELVET ermöglicht die vollständige Definition eines Feature-Modells ohne zusätzliche Bedingungen. Zur Beschreibung von Bedingungen zwischen den Features besitzt VELVET weitere Regeln, aus denen komplexe Bedingungen formuliert werden können. Diese Regeln werden in Quelltext 3.3 auf der nächsten Seite abgebildet. Demnach kann eine Bedingung unter Verwendung des Schlüsselwortes `constraint` und einer folgenden Definition (`ConstraintDefinition`) beschrieben werden (Zeile 1, 5). Die Definition von Bedingungen kann aus einem Element (`UnaryConstraint`) oder auch aus mehreren Elementen zusammengesetzt werden.

Eine einelementige Bedingung kann sowohl ein Feature als auch eine negierte einelementige Bedingung darstellen. Des Weiteren werden auch Definitionen, die in Klammern angegeben werden, als einelementige Bedingung angesehen, in der weitere Bedingungen definiert werden können. Eine zweielementige Bedingung (`BinaryConstraint`, Zeile 7) besteht wiederum aus einer einelementigen Bedingung, einem binären Operator und einer weiteren Definition. Daher können nicht nur zweielementige sondern auch n-elementige Bedingungen definiert werden. In diesem Zusammenhang können die binären, aussagenlogischen Operatoren „Und“ (`&&`), „Oder“ (`||`), „Exklusiv-Oder“ (`xor`), „Implikation“ (`->`) sowie „Äquivalenz“ (`<->`) zur Darstellung komplexer Bedingungen verwendet werden.

```
1  Constraint: "constraint"  [ConstraintName "="] ConstraintDefinition
2                                                              ";" ;
3  ConstraintName: ID ;
4
5  ConstraintDefinition: (BinaryConstraint | UnaryConstraint
6                                          | AttribConstraint) ;
7  BinaryConstraint: UnaryConstraint BinaryOp ConstraintDefinition ;
8  UnaryConstraint: (UnaryOp UnaryConstraint | ParensConstraint
9                                              | FeatureName) ;
10 ParensConstraint: "(" ConstraintDefinition ")" ;
11
12 UnaryOp: "!" ;
13 BinaryOp: ("&&" | "||" | "xor" | "->" | "<->") ;
```

Quelltext 3.3: VELVET-Grammatik Teil II.

Verwendung von VELVET am Beispiel

In Quelltext 3.4 auf der nächsten Seite wird die bekannte Auto-Produktlinie (siehe Abbildung 2.1 auf Seite 7) mittels VELVET dargestellt. Da das Wurzel-Feature als Konzept in VELVET beschrieben wird, erfolgt zunächst die Definition des Konzepts `Auto`. Innerhalb des Konzepts werden anschließend die einzelnen Features der Produktlinie `Auto` dargestellt. In diesem Zusammenhang werden die optionalen Features `Schluesselloses_Zugangssystem` und `Zentralverriegelung` (Zeile 20, 21) beschrieben. Die obligatorischen Features `Karosserie` (Zeile 3), `Motor` (Zeile 5) und `Getriebe` (Zeile 12) werden zusätzlich mit dem Schlüsselwort `mandatory` versehen. Des Weiteren wird das Schlüsselwort `someof` für die Darstellung der verschiedenen Motor-Varianten und das Schlüsselwort `oneof` für die jeweiligen Getriebe-Varianten verwendet (zum Vergleich siehe Abbildung 2.1 auf Seite 7). Im Feature-Modell der Auto-Produktlinie wurde zusätzlich die Abhängigkeit des Features `Schluesselloses_Zugangssystem` zu der `Zentralverriegelung` dargestellt, dessen Beschreibung in VELVET durch die Verwendung einer Implikation erfolgt (Zeile 24).
Dieses Beispiel zeigt die einfache Modellierung von Produktlinien mittels VELVET. Im nächsten Abschnitt werden die verschiedenen Kompositionsmechanismen beschrieben, die von VELVET unterstützt werden. Zur Verwendung dieser Mechanismen werden in den jeweiligen Abschnitten weitere Schlüsselwörter der Grammatik eingeführt.

```
concept Auto{
    //obligatorisches Feature
    mandatory feature Karosserie;

    mandatory feature Motor{
        someof{ //Oder Gruppe
            feature Elektrisch;
            feature Benzin;
        }
    }

    mandatory feature Getriebe{
        oneof{ //Alternative Gruppe
            feature Automatik;
            feature Manuell;
        }
    }

    //optionale Features
    feature Schluesselloses_Zugangssystem;
    feature Zentralverriegelung;

    //zusätzliche Bedingung
    constraint Schluesselloses_Zugangsystem -> Zentralverriegelung;
}
```

Quelltext 3.4: Auto-Produktlinie in VELVET.

3.2 Multiproduktlinienmodellierung in VELVET

VELVET ist eine Modellierungssprache die unter Anderem zur Kombination verschiedener Produktlinien verwendet werden kann. Zu diesem Zweck werden von VELVET verschiedene, sogenannte Kompositionsmechanismen bereitgestellt. VELVET unterstützt die Verwendung von: *Vererbung*, *Superimposition* und *Aggregation*. Diese Mechanismen ermöglichen verschiedene Modifikationen eines existierenden Feature-Modells, die einer Refaktorisierung, Spezialisierung, Generalisierung oder einer willkürlichen Änderung entsprechen (siehe Abschnitt 2.1.3 auf Seite 9).
Diese Arbeit konzentriert sich vor allem auf die sicherere Komposition von Produktlinien in Verbindung mit den Kompositionsmechanismen Vererbung und Aggregation. Aus diesem Grund werden im Folgenden lediglich diese Kompositionsmechanismen genauer betrachtet. Weitere Informationen zu den jeweiligen Kompositionsmechanismen können den Ausführungen von Rosenmüller et al. entnommen werden [RSTS11].
Im Folgenden werden jeweils weitere grammatikalische Elemente eingeführt und notwendige Verbesserungen der Grammatik zur Sicherstellung der Konsistenz dargestellt. In diesem Zusammenhang folgt jeweils ein Beispiel über die Verwendung des Kompositionsmechanismus in VELVET und eine mögliche Darstellung der Komposition in einem einzelnen Feature-Modell. Dies bildet die Grundlage für weitere Analysen.

3.2.1 Vererbung in VELVET

Rosenmüller et al. integrierten in die Modellierungssprache VELVET den Kompositionsmechanismus der Vererbung, um ähnlich wie bei objektorientierter Programmierung bestehende Konzepte erweitern oder zusammenstellen zu können [RSTS11].

Grammatik

Zur Modellierung der Vererbung verwendet VELVET weitere grammatikalische Elemente. In Quelltext 3.5 wird ein zusätzlicher Ausschnitt der VELVET-Grammatik dargestellt, der zur Beschreibung dieser Elemente dient. Dieser Ausschnitt konzentriert sich auf die Definition eines Konzeptes. Im vorherigen Abschnitt wurde gezeigt, wie ein einzelnes Konzept durch die Angabe des Schlüsselwortes `concept` definiert werden kann. Dazu werden nach dem Schlüsselwort `concept` der Konzept-Name und eine weitere Definition der Produktlinie durch die Angabe von Bedingungen, Gruppen und Features angegeben. Zwischen der Angabe des Konzept-Namens und der eigentlichen Konzept-Definition können optional sogenannte Basiskonzepte (`ConceptBases`, Zeile 1) angegeben werden. Diese Basiskonzepte werden durch das Zeichen „:“ eingeleitet und durch die Angaben weiterer Konzept-Namen (`ID`, Zeile 3) beschrieben. Da VELVET eine Auflistung mehrerer Basiskonzepte erlaubt, unterstützt VELVET Mehrfachvererbung.

```
1 Concept: ["refines"] "concept" ConceptName [ConceptBases]
2                                                    ConceptBody ;
3 ConceptName: ID ;
4 ConceptBases: ":" ID ("," ID)* ;
```

Quelltext 3.5: VELVET-Grammatik Teil III.

Durch die Verwendung verschiedener Basiskonzepte ermöglicht VELVET die Erweiterung bestehender Konzepte. Einerseits können durch diese Erweiterung Features hinzugefügt und somit eine Generalisierung eines bestehenden Feature-Modells erreicht werden (siehe Abschnitt 2.1.3 auf Seite 9). Andererseits können durch das Hinzufügen von Bedingungen ableitbare Varianten eines bestehenden Feature-Modells eingeschränkt werden, was wiederum einer Spezialisierung entspricht. Des Weiteren können aber auch Features und Bedingungen in gemischter Form durch Vererbung zu einem bestehenden Konzept hinzugefügt werden. Diese zusätzlichen Bedingungen und Features entsprechen einer willkürlichen Änderung des ursprünglichen Konzeptes.
Durch die mögliche Erweiterung bestehender Features, kann die Verwendung von Gruppen zu einem Problem bezüglich der Eindeutigkeit führen. In Abbildung 3.1 auf der nächsten Seite wird eine erweiterte Form der Produktlinie `Liste` dargestellt, in der das Feature `Sortierung` eine Alternative-Gruppe mit den Features `QuickSort` und `MergeSort` besitzt. Bei einer Erweiterung des Features `Sortierung` um einen weiteren Sortieralgorithmus, ist nicht automatisch entscheidbar, ob dieses Feature zur bestehenden Gruppe oder parallel dazu eingefügt werden soll. Aus diesem Grund ist eine Anpassung der Grammatik sinnvoll, die eine eindeutige Modellierung ermöglicht. In diesem Zusammenhang ergeben sich zwei Möglichkeiten: (1) jede Gruppe benötigt einen speziellen Namen, oder (2) jedes Feature darf nur genau eine Gruppe enthalten. In dieser

Arbeit wurde die zweite Möglichkeit in die VELVET-Grammatik integriert, da diese eine gewöhnliche Darstellung in einem Feature-Diagramm ermöglicht. Dementsprechend wird in Quelltext 3.6 ein angepasster Ausschnitt von VELVET angegeben, in dem jeweils nur eine Gruppe in einem Feature beziehungsweise im Konzept definiert werden kann.

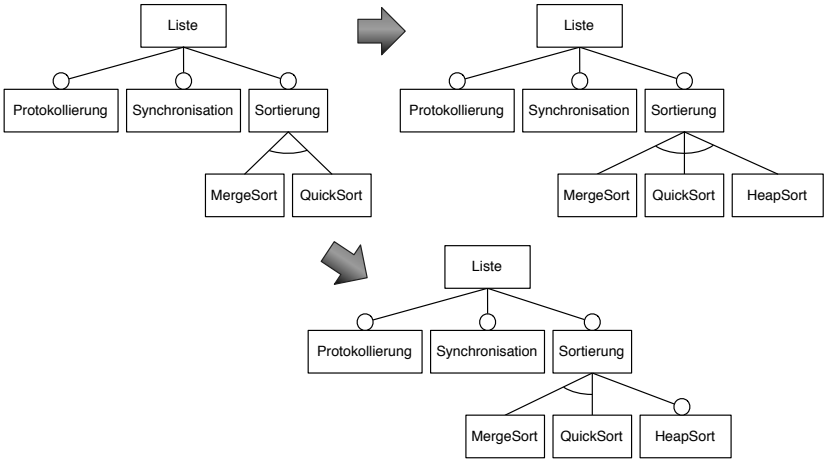


Abbildung 3.1: Fehlende Eindeutigkeit bei der Erweiterung von Gruppen.

Zur Einschränkung der Verwendung von einer Gruppe pro Feature wurde in Quelltext 3.6 die Definition der Regel `Body` beziehungsweise `Definition` abgeändert (Zeile 2-4). Weiterhin sollte bei der neuen Definition die Möglichkeit bestehen parallel zu einer Gruppe zusätzlich Bedingungen davor oder dahinter anzugeben. Aus diesem Grund können zunächst Bedingungen definiert werden. Wenn eine Gruppe definiert wurde, können nur noch weitere Bedingungen folgen. Erfolgte jedoch die Definition eines Features, kann keine Gruppe mehr definiert werden.

```
1 Concept:  ["refines"] "concept" ConceptName [ConceptBases] Body ;
2 Body: "{"  [Definition] "}" ;
3 Definition:  (ConstInstAttr* [((FeatureGroup ConstInstAttr*) |
4                                (ConstInstAttr | Feature)*)]) ;
5
6 ConstInstAttr: (Constraint  | Instance | Attribute) ;
7
8 Feature: ["mandatory"] "feature" FeatureName (Body | ";") ;
```

Quelltext 3.6: Verbesserung der VELVET-Grammatik Teil II.

Verwendung der Vererbung am Beispiel

Zur Darstellung von Vererbung in VELVET wird nun das Beispiel der Produktlinie `Mail_Client` betrachtet. In der Produktlinie `Mail_Client` wurde die Beziehung zu der Produktlinie `Liste` und dessen verwendeter Instanzen in einem UML-Klassendiagramm dargestellt (siehe Abbildung 2.5 auf Seite 15). Hierbei wurde die Produktlinie `Liste` durch Vererbung wiederverwendet und die neuen Produktlinien `Sortierte_Liste` und `Synchronisierte_Liste` erzeugt. Die entsprechende Darstellung dieser Produktlinien in VELVET erfolgt in Quelltext 3.7 auf der nächsten Seite.

Dargestellt wird zunächst die Produktlinie `Liste` (Zeilen 1-5) mit den optionalen Features `Protokollierung`, `Sortierung` und `Synchronisation`. Des Weiteren wird die Produktlinie `Sortierte_Liste` (Zeile 7-9) dargestellt, die durch die Verwendung von Vererbung alle Features und Bedingungen der Produktlinie `Liste` übernimmt. Daher können die Features der Produktlinie `Liste` in der Produktlinie `Sortierte_Liste` verwendet und beispielsweise neue Bedingungen, bezogen auf diese Features, definiert werden. Zur Konfiguration der Produktlinie `Liste` in der Produktlinie `Sortierte_Liste` wird daher zusätzlich die Bedingung `constraint Sortierung` (Zeile 8) angegeben. In gleicher Weise wird auch die Produktlinie `Synchronisierte_Liste` (Zeile 11-13) erzeugt, indem alle Features der Produktlinie `Liste` übernommen werden und anschließend die Bedingung `constraint Synchronisation` (Zeile 12) hinzugefügt wird.

```
1  concept Liste{
2      feature Protokollierung;
3      feature Sortierung;
4      feature Synchronisation;
5  }
6
7  concept Sortierte_Liste : Liste {
8      constraint Sortierung;
9  }
10
11 concept Synchronisierte_Liste : Liste {
12     constraint Synchronisation;
13 }
```

Quelltext 3.7: Spezialisierung der Produktlinie `Liste` durch Vererbung.

Zur späteren Analyse ist es notwendig die Kompositionen der verschiedenen Produktlinien in einen gemeinsamen aussagenlogischen Ausdruck zu überführen. Als Grundlage dient ein Feature-Modell, das die gesamte Multiproduktlinie repräsentiert. Die zugehörigen Feature-Modelle für die `Synchronisierte_Liste` und die `Sortierte_Liste` werden in Abbildung 3.2 dargestellt.

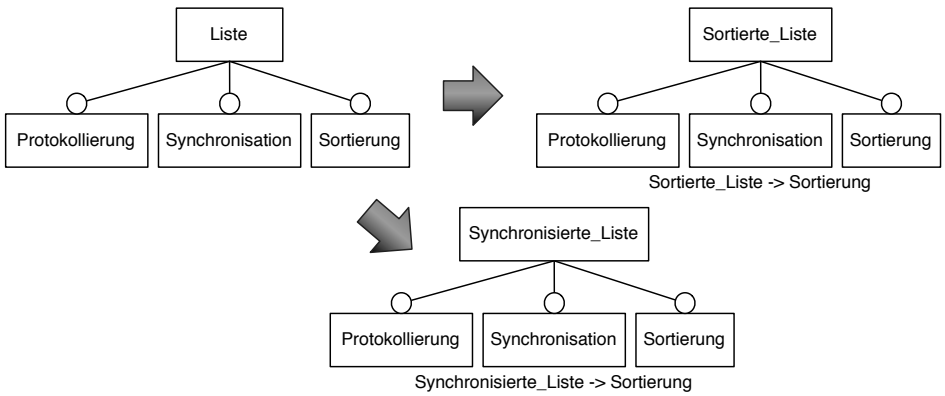


Abbildung 3.2: Feature-Modelle nach Spezialisierung durch Vererbung.

Im Quelltext 3.7 auf der vorherigen Seite wurde die Möglichkeit der Spezialisierung mittels Vererbung dargestellt. Im Gegensatz dazu, repräsentiert der Quelltext 3.8 eine Generalisierung. Zur Veranschaulichung wurde dazu das Feature `Protokollierung` der ursprünglichen Produktlinie `Liste` auskommentiert. Dieses Feature wird erst anschließend in der Erweiterung `Protokollierte_Liste` hinzugefügt. Daher können aus der Produktlinie `Protokollierte_Liste` mehr Produkte als aus der ursprünglichen Produktlinie `Liste` generiert werden.

```
1 concept Liste{
2     //feature Protokollierung; //In dieser SPL Version nicht vorhanden
3     feature Sortierung;
4     feature Synchronisation;
5 }
6
7 concept Protokollierte_Liste : Liste {
8     feature Protokollierung;
9 }
```

Quelltext 3.8: Generalisierung der Produktlinie `Liste` durch Vererbung.

Wie bei der Spezialisierung der Produktlinie `Liste` wird zur weiteren Analyse ein gesamtes Feature-Modell benötigt, das die vollständige Multiproduktlinie repräsentiert. Die Darstellung der Produktlinie `Protokollierte_Liste` erfolgt in Abbildung 3.3.

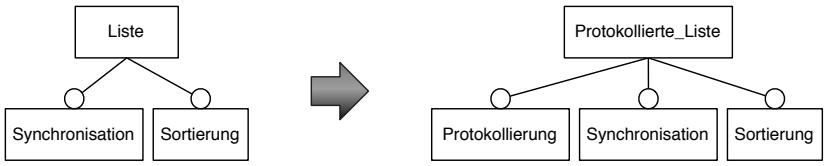


Abbildung 3.3: Feature-Modell nach Generalisierung durch Vererbung.

Wie zuvor erwähnt, können auch willkürliche Änderungen durch die gleichzeitige Definition von zusätzlichen Features und Bedingungen erreicht werden. Zudem ermöglicht die Unterstützung von Mehrfachvererbung die Generalisierung, Spezialisierung oder auch die willkürlichen Änderungen mehrerer Modelle zugleich.

3.2.2 Aggregation in VELVET

Die Aggregation ist ein weiterer wichtiger Mechanismus, um verschiedene Produktlinien zu kombinieren. Wie die Vererbung ist auch die Aggregation ein bekannter Mechanismus der objektorientierten Programmierung, der ebenfalls auf die Entwicklung von Produktlinien übertragen werden kann. Im Fall von Produktlinien ermöglicht die Aggregation verschiedene Instanzen einer oder mehrerer Produktlinien in eine andere Produktlinie zu integrieren.

Grammatik

Der zugehörige Ausschnitt der VELVET-Grammatik zur Darstellung von Instanzen wird in Quelltext 3.9 auf der nächsten Seite abgebildet (Zeile 5, 15, 16). Bei der Einführung von VELVET wurde erläutert, wie eine Produktlinie modelliert werden kann. In

diesem Zusammenhang konnte ein Konzept lediglich Bedingungen, Gruppen und Features enthalten. Die Möglichkeit der Definition von Instanzen wurde daher zunächst vernachlässigt, um diesen Aspekt nun im Speziellen betrachten zu können. Dementsprechend wird in der Grammatik von VELVET, bei der Definition eines Konzepts beziehungsweise eines Features, die Definition einer Instanz (`Instance`, Zeile 5) ermöglicht. Diese Instanz besteht aus einem speziellen Konzept-Namen (`ConceptName`) und aus dem eigentlichen Instanz-Namen (`InstanceName`, Zeile 15, 16).

```
1  Concept:  ["refines"] "concept" ConceptName [ConceptBases] Body ;
2  Body: "{" Definition* "}" ;
3  Definition: (ConstInstAttr* [((FeatureGroup ConstInstAttr*) | (
4                                        ConstInstAttr | Feature)*)]) ;
5  ConstInstAttr: (Constraint | Instance  | Attribute) ;
6
7  Feature: ["mandatory"] "feature" FeatureName (Body | ";") ;
8  FeatureName: ID ("." ID)* ;
9
10 //Definition von Bedingungen
11 //...
12 UnaryConstraint: (UnaryOp UnaryConstraint | ParensConstraint
13                                                  | FeatureName) ;
14
15 Instance: ConceptName InstanceName ";" ;
16 InstanceName: ID ;
```

Quelltext 3.9: VELVET-Grammatik Teil IV.

Bei der Einführung von Multiproduktlinien wurden unter Anderem verschiedene Arten von Bedingungen, bezogen auf Instanzen, vorgestellt (zum Beispiel Instanz-Bedingungen, siehe Abschnitt 2.2.2 auf Seite 12). Diesbezüglich muss die VELVET-Grammatik auch die Definition von verschiedenen Bedingungen, im Zusammenhang mit diesen Instanzen, unterstützten. Bei der Einführung von VELVET durch Rosenmüller et al. wurde die Möglichkeit zur Beschreibung von Bedingungen für Instanzen auch an einem Beispiel beschrieben. Betrachtet man jedoch die Definition der verschiedenen Bedingungen, die grundsätzlich jeweils aus einelementigen Bestandteilen (`UnaryConstraint`) zusammengesetzt sind, können diese bisher keine Instanzen enthalten (siehe Quelltext 3.9, Zeile 12, 13). In dem Zusammenhang ist nach der Grammatik bisher nur möglich einen speziellen Feature-Namen (`FeatureName`) zu verwenden (Zeile 13). Zur Behebung dieses Problems musste die Definition von Instanzen in der VELVET-Grammatik angepasst werden.
Da die Regeln `FeatureName` und `InstanceName` jeweils aus `ID`-Elementen bestehen, können diese Regeln zur neuen Regel `Name` zusammengefasst und somit die Grammatik vereinfacht werden (siehe Quelltext 3.10 auf der nächsten Seite). Die Anpassung ermöglicht zugleich die Unterstützung von Instanz-Namen bei der Definition von Bedingungen, da diese nicht mehr auf Feature-Namen beschränkt werden. Notwendige Änderungen wurden zusätzlich farblich gekennzeichnet.

```
1 Name: ID ("." ID)* ;
2 Feature: ["mandatory"] "feature"  Name (Body | ";") ;
3
4 UnaryConstraint: (UnaryOp UnaryConstraint | ParensConstraint | Name) ;
5
6 Instance: ConceptName  Name ";" ;
```

Quelltext 3.10: Verbesserung der VELVET-Grammatik Teil III.

Verwendung der Aggregation am Beispiel

Die erweiterte Grammatik von VELVET kann nun genutzt werden, um verschiedene Instanzen einer Produktlinie in einer anderen Produktlinie zu beschreiben. Zur Veranschaulichung wird in Quelltext 3.11 die Produktlinie `Mail_Client` in VELVET dargestellt (zum Vergleich siehe Abbildung 2.5 auf Seite 15), die mehrere Instanzen der Produktlinie `Liste` verwendet. Die Verwendung der verschiedenen Varianten der Produktlinie `Liste` im `Mail_Client` wurde im vorherigen Abschnitt durch Vererbung repräsentiert. Im Gegensatz zu dieser Darstellung können in VELVET (siehe Quelltext 3.11) die verschiedenen Instanzen direkt beschrieben und konfiguriert werden ohne weitere Produktlinien durch Vererbung erzeugen zu müssen.

```
1  concept Mail_Client{
2      //Instanz-Definitionen
3      Liste mailListe;
4      Liste fehlerListe;
5
6      //Instanz-Bedingungen
7      constraint mailListe.Sortierung;
8      constraint fehlerListe.Synchronisation;
9
10     mandatory feature Protokolle{...}
11
12     feature Verschluesselung;
13
14     feature Debug{
15         //Domänen-Bedingung
16         constraint Liste.Protokollierung;
17     }
18 }
```

Quelltext 3.11: Multiproduktlinie `Mail_Client` in VELVET.

Zunächst benötigt die Produktlinie `Mail_Client` neben den eigenen Features eine Instanz der Produktlinie `Liste` zur sortierten Speicherung von verschiedenen E-Mails der Produktlinie `Mail_Client`. Demnach wird eine Instanz der Produktlinie unter Angabe der Produktlinie und des eindeutigen Namens `mailListe` (Zeile 3) erstellt. Zusätzlich wird die Bedingung `constraint mailListe.Sortierung` (Zeile 7) hinzugefügt, die eine Auswahl des Features `Sortierung` für diese spezielle Instanz `mailListe` vornimmt. Bei dieser Bedingung handelt es sich daher um eine Instanz-Bedingung (siehe Abschnitt 2.2.2 auf Seite 12). Des Weiteren benötigt die Produktlinie `Mail_Client`

eine weitere Instanz der Produktlinie `Liste` mit dem Namen `fehlerListe` (Zeile 4), bei der zusätzlich das Feature `Synchronisation` erforderlich ist. Aus diesem Grund erhält auch diese Instanz eine zusätzliche Bedingung zur Auswahl des Features `Synchronisation` (Zeile 8).

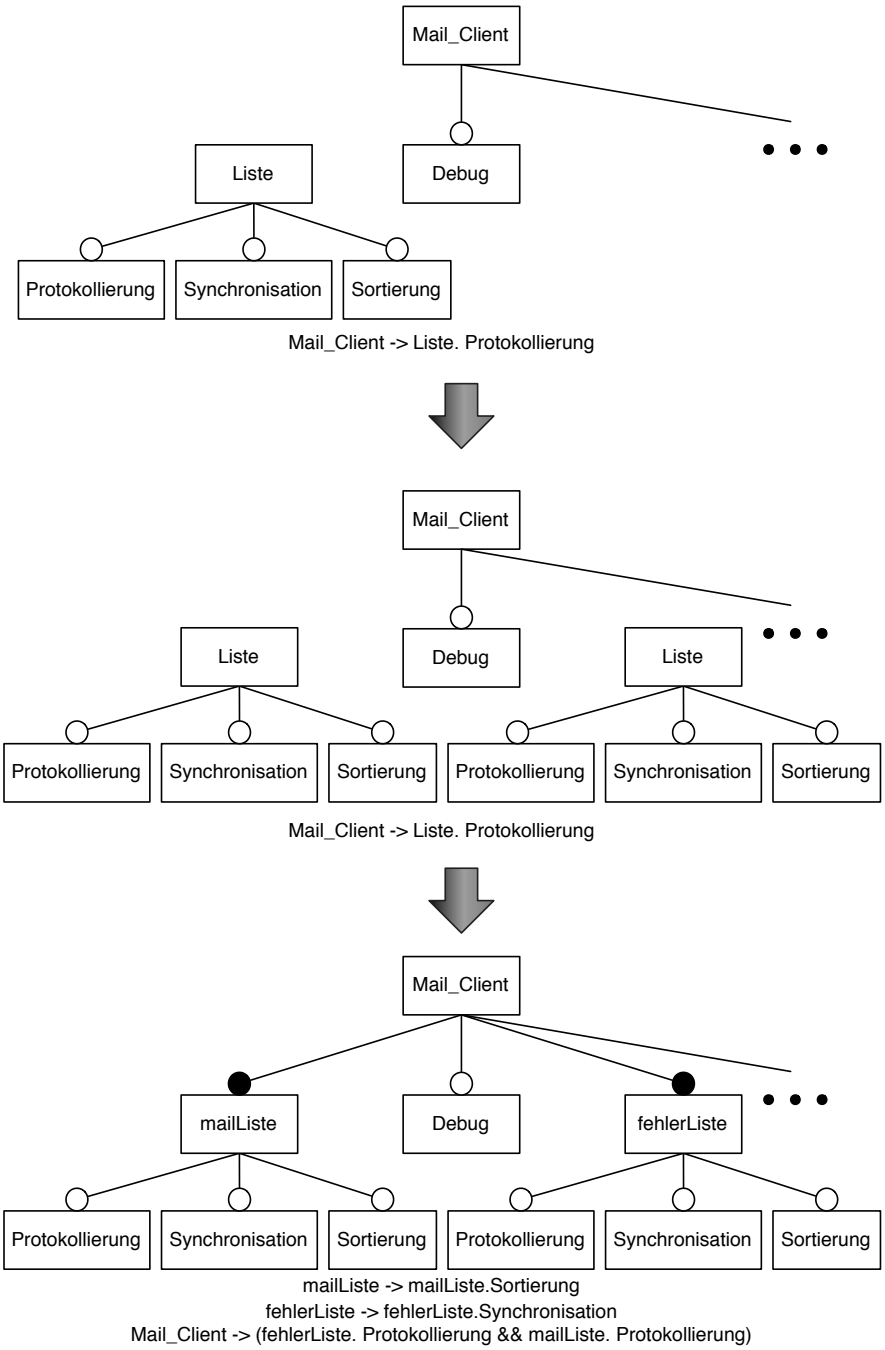


Abbildung 3.4: Überführung von abhängigen Produktlinien in ein Gesamtmodell bei der Verwendung von Aggregation.

Bei dem Feature-Modell zur Beschreibung der Multiproduktlinie `Mail_Client` (siehe Abbildung 2.4 auf Seite 13) wurde unter Anderem bei der Auswahl des Features `Debug` die Aktivierung des Features `Protokollierung` der Produktlinie `Liste` gefordert. Hierbei handelt es sich um eine Domänen-Bedingung (siehe Abschnitt 2.2.2 auf Seite 12), die für alle Instanzen der Produktlinie `Liste` benötigt wird. Diese Bedingung kann ebenfalls in VELVET dargestellt werden. In diesem Zusammenhang wird der Produktlinienname `Liste` bei der Definition der Bedingung angegeben (Zeile 16).
Zur weiteren Analyse der Multiproduktlinie ist es wiederum erforderlich ein Feature-Modell zu erzeugen, das die gesamte Multiproduktlinie repräsentiert. In diesem Zusammenhang müssen die Produktlinie `Mail_Client` und die Produktlinie `Liste` kombiniert werden. Da die Produktlinie `Mail_Client` zwei verschiedene Instanzen mit verschiedenen Konfigurationen der Produktlinie `Liste` verwendet, muss zunächst die Produktlinie `Liste` verdoppelt werden. Anschließend werden die Wurzel-Features der zwei Produktlinien `Liste` in die jeweiligen Instanz-Namen umbenannt. Des Weiteren können die Wurzel-Features der Produktlinie `Liste` (`fehlerListe`, `mailListe`) als obligatorische Features in die Multiproduktlinie `Mail_Client` eingefügt werden. In diesem Fall wird als neues Eltern-Feature der Instanzen das Wurzel-Feature der gesamten Multiproduktlinie verwendet. Im Allgemeinen wird als neues Eltern-Feature das Feature gewählt, in dem eine Instanz definiert wird. Dies muss nicht zwingend direkt im Konzept geschehen, sondern kann auch in einem beliebigen Kind-Feature sein. Anschließend können die Instanz-Bedingungen eindeutig in das neue Feature-Modell eingefügt werden. In diesem Zusammenhang werden die Bedingungen `mailListe -> mailListe.Sortierung` und `fehlerListe -> fehlerListe.Synchronisation` für die jeweiligen Instanzen definiert. Des Weiteren wird die Domänen-Bedingung `Mail_Client -> Liste.Protokollierung` für die neue Darstellung im Gesamtmodell angepasst.

3.3 Zusammenfassung

Als weitere Grundlage dieser Arbeit wurde die von Rosenmüller et al. eingeführte, textuelle Modellierungssprache VELVET vorgestellt [RSTS11]. VELVET unterstützt verschiedene Kompositionsmechanismen, die zur Komposition der verschiedenen Produktlinien einer Multiproduktlinie verwendet werden können. In diesem Kapitel wurden verschiedene kleinere Verbesserungen bezogen auf die Grammatik von VELVET vorgeschlagen und integriert. Dementsprechend wird beispielsweise durch die Beschränkung der Verwendung von einer Gruppe pro Feature sichergestellt, dass Erweiterungen einer vorhandenen Produktlinie durch Vererbung eindeutig modelliert werden können. Des Weiteren wurde in diesem Kapitel die Überführung der einzelnen Feature-Modelle in ein Gesamtmodell im Detail beschrieben, was eine Grundlage für weitere Test bezogen auf die gesamte Multiproduktlinie darstellt.

4. Konsistenzprüfungen für Multiproduktlinien in VELVET

Mit wachsender Größe kann selbst die Modellierung einer einzelnen Produktlinie sehr unübersichtlich werden. Durch die wachsende Komplexität der Feature-Modelle können daher Inkonsistenzen, also Widersprüche in der Modellierung, leicht übersehen werden. Dies kann dazu führen, dass gewünschte Konfigurationen nicht durch das Modell unterstützt werden. Aus diesem Grund muss zunächst überprüft werden, ob eine gegebene Konfiguration gültig ist. Im schlimmsten Fall kann es sogar vorkommen, dass vorhandene Inkonsistenzen zu einem sogenannten leeren Feature-Modell, einem Feature-Modell aus dem keine Varianten generiert werden können, führen. Daher ist es von großer Bedeutung solche Inkonsistenzen in der Modellierung zu lokalisieren und verschiedene Fehler aufzudecken. Aufgrund dessen wurden verschiedene Analysen entwickelt, die die Konsistenz eines Feature-Modells überprüfen. Diese Analysen unterstützen den Modellierer bei der Erstellung eines korrekten Feature-Modells, das keine Inkonsistenzen oder andere Fehler enthält.
Die aufgeführten Probleme bei der Modellierung von Produktlinien verstärken sich zum Teil bei der Modellierung von Multiproduktlinien. Da es sich bei Multiproduktlinien um die Komposition von verschiedenen Produktlinien handelt, entstehen bei der Modellierung oft sehr komplexe Modelle. Wie auch bei einem großen Feature-Modell einer einzelnen Produktlinie verstärken sich daher die Probleme bezüglich der Übersicht und der Handhabbarkeit. Dies fördert Fehler in der Modellierung, weshalb Konsistenzprüfungen in Bezug auf Multiproduktlinien von großer Bedeutung sind. Deshalb ist die Entwicklung eines Werkzeugs notwendig, das verschiedenen Prüfungen bezüglich der Konsistenz von Multiproduktlinien unterstützt und dem Modellierer verschiedene Fehler anzeigt. In diesem Zusammenhang ist es ebenfalls notwendig zu überprüfen, wie die bestehenden Analysen bezüglich der Konsistenz von Produktlinien für Multiproduktlinien verwendet werden können.

Im folgenden Abschnitt wird zunächst untersucht, welche Anforderungen an ein Werkzeug zur Realisierung von Konsistenzprüfungen bestehen. Anschließend werden verschiedene syntaktische Konsistenzprüfungen vorgestellt und die Verwendung von bestehenden Konsistenzprüfungen von Produktlinien in Multiproduktlinien diskutiert und mögliche Konzepte vorgestellt.

4.1 Automatisierung von Konsistenzprüfungen

Zur Überprüfung der Konsistenz von Multiproduktlinien, die mit VELVET modelliert wurden, ist es notwendig ein Werkzeug zu entwickeln, das auch verschiedene Analysen für Multiproduktlinien bereitstellt. Ein solches Werkzeug muss verschiedene Aufgaben erfüllen: (1) Parsen von Multiproduktlinien in VELVET, (2) syntaktische Überprüfungen der Modelle, (3) Test auf Erfüllbarkeit der Multiproduktlinien und (4) Realisierung von verschiedenen Konsistenzprüfungen. Diese Aufgaben bauen aufeinander auf und sind voneinander abhängig (siehe Abbildung 4.1). Demnach erfolgt beispielsweise eine syntaktische Überprüfung einer gegebenen Multiproduktlinie nur, wenn zuvor der Parser die Modelle ordnungsgemäß einlesen konnte. Die verschiedenen Aufgaben zur Überprüfung der Konsistenz werden im Folgenden noch einmal genauer betrachtet:

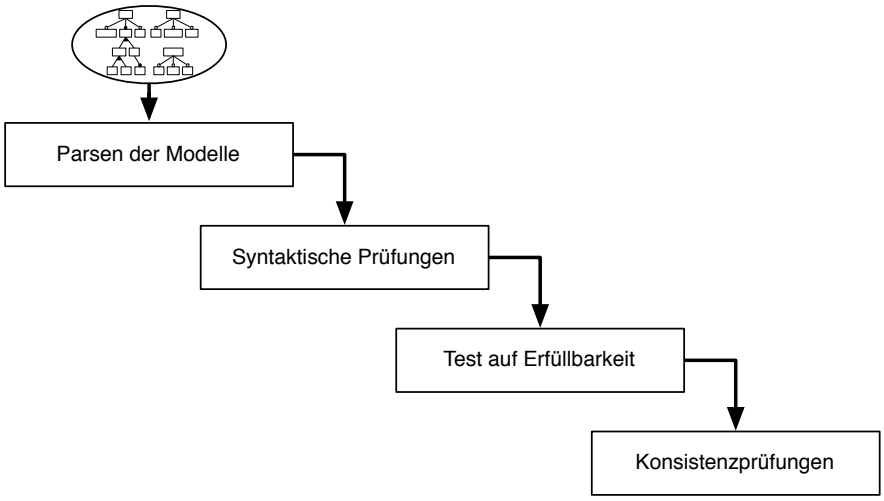


Abbildung 4.1: Werkzeuganforderungen zur Unterstützung von Konsistenzprüfungen.

(1) Modelle auf Grundlage der vorgestellten VELVET-Grammatik müssen anhand eines Parsers eingelesen und auf syntaktische Korrektheit überprüft werden. Hierfür werden alle Einzelmodelle der Multiproduktlinie eingelesen und überprüft. Wenn dieser Vorgang erfolgreich abgeschlossen wurde, können weitere Test bezüglich der Konsistenz durchgeführt werden. Falls das Einlesen fehlschlägt, muss zunächst das Modell korrigiert werden, bevor weitere Aufgaben durchgeführt werden können.

(2) Da nicht alle syntaktischen Beziehungen in einer Grammatik ausgedrückt und vom Parser überprüft werden können, müssen weitere syntaktische Prüfungen vorgenommen werden, um die Korrektheit der Modellierung zu gewährleisten. Beispielsweise kann durch einen Parser nicht garantiert werden, dass ein verwendeter Feature-Name noch nicht genutzt wurde.

(3) Nach den erfolgreichen syntaktischen Überprüfungen durch den Parser und weiteren syntaktischen Analysen folgt nun die Überprüfung der Erfüllbarkeit der Multiproduktlinien. Zur Überprüfung einer Multiproduktlinie auf Erfüllbarkeit, werden alle Einzelmodelle der Multiproduktlinie in ein Gesamtmodell überführt. Die semantischen Interpretationen der Kompositionsmechanismen Aggregation und Vererbung wurden im letzten Kapitel genauer betrachtet (siehe Kapitel 3 auf Seite 17). Diesbezüglich wurde jeweils ein Feature-Modell dargestellt, das die gesamte Multiproduktlinie repräsentiert. Die Erfüllbarkeit des Gesamtmodells kann beispielsweise durch einen SAT-Solver oder anderen bestehenden Analysewerkzeugen für Feature-Modelle überprüft werden.

(4) Der Test auf Erfüllbarkeit der Multiproduktlinie ermöglicht anschließend weitere Konsistenzprüfungen, die dem Modellierer zusätzliche Informationen über die Korrektheit der modellierten Kompositionsmechanismen des Gesamtmodells liefern. Um weitere Analysen durchzuführen, können bestehende Konsistenzprüfungen von Produktlinien verwendet werden, die auf das Gesamtmodell oder auf Teile des Modells angewandt werden können. Auf der einen Seite können zum Teil bestehende Analysen von Produktlinien übernommen werden. Auf der anderen Seite müssen aber auch bestehende Analysen im Kontext von Multiproduktlinien neu definiert und deren Auswertung angepasst werden. Die einzelnen Konsistenzprüfungen und deren zum Teil notwendige Erweiterungen werden in den folgenden Abschnitten noch einmal genauer betrachtet.

4.2 Syntaktische Konsistenzprüfungen

Durch die Verwendung eines Parsers kann die syntaktische Korrektheit eines VELVET-Modells überprüft werden. Dies ist gleichbedeutend mit einer ersten Konsistenzprüfung eines gegebenen Modells. In diesem Zusammenhang sind jedoch weitere Test notwendig, da nicht jede syntaktische Beziehung in einer Grammatik formuliert werden kann. Aus diesem Grund müssen nach dem Parsen des Modells weitere syntaktische Analysen durchgeführt werden, um die Konsistenz eines gegebenen Modells sicherstellen zu können. Im Folgenden werden verschiedene syntaktische Tests vorgestellt, die zur Sicherstellung der Konsistenz durchgeführt werden müssen.
Im Quelltext 4.1 auf der nächsten Seite erfolgt die erneute Darstellung der Produktlinie `Liste`. Bei der Erstellung des Konzepts wurde jedoch das Feature `Sortierung` anstelle des Features `Synchronisation` (Zeile 4) zweimal definiert. Hierbei handelt es sich um ein typisches Beispiel für eine Beziehung, die nicht durch ein Parser erkannt werden kann. Dementsprechend muss zusätzlich getestet werden, ob ein Element (Feature oder Instanz) mit dem gleichen Namen bereits existiert.

```
1 concept Liste{
2       feature Protokollierung;
3       feature Sortierung;
4       feature Sortierung;
5 }
```

Quelltext 4.1: Syntaktischer Test auf doppelte Namensvergabe.

Des Weiteren muss bei der Verwendung von Vererbung und Aggregation analysiert werden, ob referenzierte Features beziehungsweise Instanzen existieren (siehe Quelltext 4.2, `Sortierung`). Existiert ein referenziertes Feature nicht im umgebenen Konzept, muss die Vererbungshierarchie nach dem Feature durchsucht werden.

```
1 concept Sortierte_Liste : Liste {
2     constraint Sortierung; //Feature existiert in Vererbungshierarchie?
3 }
```

Quelltext 4.2: Syntaktischer Test auf Korrektheit der Referenzen.

Die Korrektheitsüberprüfung bezogen auf Vererbung ist ein weiterer wichtiger syntaktischer Test, der nicht durch den Parser vollzogen werden kann. Im Quelltext 4.3 ist das Konzept `Sortierte_Liste` abgebildet, welches von der Produktlinie `Liste` erbt. In diesem Beispiel wird das Konzept der `Liste` doppelt als Basiskonzept angegeben. Diese Angabe lässt jedoch keine sinnvolle Interpretation zu, weshalb derartige Angaben als Fehler ausgegeben werden.

```
1 concept Sortierte_Liste : Liste  , Liste {
2     constraint Sortierung;
3 }
```

Quelltext 4.3: Syntaktischer Test auf doppelte Vererbung.

Im Gegensatz zum vorherigen Problem, bei dem das gleiche Konzept doppelt zur Vererbung angegeben wurde, resultiert nun ein Problem durch das Fehlen einer expliziten Vererbungs-Definition. Im Quelltext 4.4 wird eine zusätzliche Bedingung für das Feature `Sortierung` angegeben, das aber nicht im Konzept `Sortierte_Liste` definiert wurde. Des Weiteren ist kein Basiskonzept angegeben, indem das Feature eventuell existieren könnte. Aufgrund dessen kann dieses Problem auch als fehlende Referenz interpretiert werden. Jedoch mangelt es in diesem speziellen Fall lediglich an der Angabe des Basiskonzeptes `Liste`, um eine eindeutige und fehlerfreie Modellierung zu garantieren.

```
1 concept Sortierte_Liste  {
2     constraint Sortierung;
3 }
```

Quelltext 4.4: Syntaktischer Test bei fehlender Vererbung.

Ein weiterer Aspekt, der auf syntaktischer Ebene untersucht werden muss, ist ein obligatorisches Feature in einer Gruppe (siehe Quelltext 4.5). Beispielsweise darf in eine alternative Gruppe lediglich ein Feature ausgewählt werden. Daher ist die obligatorische Definition von Features unerwünscht. Dieser Aspekt kann jedoch nur sehr umständlich in der Grammatik ausgedrückt werden. Deshalb ist auch hier ein zusätzlicher syntaktischer Test notwendig.

```
1 concept Liste{
2   constraint Sortierung{
3     oneof{
4       feature MergeSort;
5       mandatory feature QuickSort;
6     }
7   }
8 }
```

Quelltext 4.5: Syntaktischer Test bei `mandatory` in Gruppen.

Ein weiterer bedeutender Test bezieht sich auf Rekursion in der Konzept-Definition. In diesem Zusammenhang darf beispielsweise in einem Konzept keine Instanz vom Typ des eigenen Konzeptes angelegt werden. Ein entsprechendes Beispiel wird in Quelltext 4.6 abgebildet. Die Definition einer Instanz vom Typ `Liste` im Konzept `Liste` resultiert in einem Gesamtmodell, das unendlich viele Ebenen mit je einer Instanz `liste` besitzt. Die sinnvolle Verwendung des Modells ist daher nicht möglich. Aus diesem Grund ist die Erkennung von Rekursionen in der Modellierung ein wichtiger Aspekt der überprüft werden muss.

```
1 concept Liste{
2     Liste liste;
3 }
```

Quelltext 4.6: Syntaktischer Test bei Rekursion in der Modellierung.

Die zuvor betrachteten Prüfungen ermöglichen eine automatisierte Erkennung syntaktischer Fehler. Wenn das Modell fehlerfrei geparst wurde und alle zusätzlichen syntaktischen Tests erfolgreich verliefen, können weitere Prüfungen durchgeführt werden. Auf Grundlage des syntaktisch korrekten Modells können anschließend semantische Prüfungen durchgeführt werden, die die Korrektheit der Komposition beziehungsweise die Erfüllbarkeit des Gesamtmodells analysieren.

4.3 Konsistenzprüfungen für Produktlinien in Multiproduktlinien

Im vorherigen Abschnitt wurden syntaktische Prüfungen betrachtet, die die Grundlage für automatisierte, semantische Prüfungen bilden. Eine Möglichkeit die semantische Korrektheit der jeweiligen Multiproduktlinien zu überprüfen, ist die Überführung der Gesamt-Komposition der einzelnen Feature-Modelle in einen aussagenlogischen Ausdruck und ein anschließender Test auf Erfüllbarkeit durch einen SAT-Solver. Wie bei

der Verwendung von Vererbung und Aggregation ein Gesamtmodell erzeugt werden kann, wurde im vorherigen Kapitel genauer betrachtet (siehe Kapitel 3 auf Seite 17). Diese Gesamtmodelle können wie gewöhnliche Produktlinien in einen aussagenlogischen Ausdruck überführt und auf Erfüllbarkeit überprüft werden. Die Erfüllbarkeit des Gesamtmodells bildet wiederum die Grundlage für weitere Konsistenzprüfungen, die in diesem Abschnitt betrachtet werden.
Im Folgenden werden verschiedene existierende Konsistenzprüfungen, die zur automatisierten Analyse von Feature-Modellen in Produktlinien verwendet werden, bezüglich der Verwendung in Multiproduktlinien untersucht.
Zur automatisierten Analyse von Feature-Modellen wurden im Laufe der Zeit verschiedene Prüfungen vorgestellt, die unter Anderem von Benavides et al. zusammengetragen wurden [BSRC10]. In diesem Zusammenhang wurden beispielsweise die Konsistenzprüfungen *Dead-Features, False-Optional-Features, Anzahl der Produkte*, *Core-Features* sowie *atomare Mengen* vorgestellt. Im Folgenden werden diese verschiedenen Analysen kurz erläutert und im Zusammenhang mit Multiproduktlinien genauer betrachtet. Des Weiteren werden notwendige Erweiterungen der verschiedenen Konsistenzprüfungen vorgeschlagen.

4.3.1 Dead-Features und False-Optional-Features

Bei Dead-Features und False-Optional-Features handelt es sich um sogenannte Anomalien [BSRC10], die in einem Feature-Modell unerwünscht sind. Diese Anomalien sind Fehler in der Modellierung, die zum Teil schwerwiegende Folgen haben können. Beispielsweise können Anomalien dazu führen, dass bestimmte Features nicht ausgewählt werden können. Demnach stellt das Feature-Modell eine falsche Repräsentation der tatsächlich auswählbaren Features dar. Aus dem gleichen Grund ist vor allem die Analyse zur Aufdeckung von Dead-Features von großer Wichtigkeit bei der Erstellung von Multiproduktlinien.
Benavides et al. definieren Dead-Features wie folgt [BSRC10]:

> „*A feature is dead if it cannot appear in any of the products of the software product line.*“

Demnach handelt es sich bei einem Dead-Feature um ein Feature, das in keiner Variante einer Produktlinie vorkommt. Dead-Features können nur durch eine inkorrekte Verwendung von Bedingungen entstehen. In Abbildung 4.2 auf der nächsten Seite wird eine erweiterte Form der Produktlinie `Liste` mit verschiedenen zusätzlichen Bedingungen vorgestellt. In der dargestellten Produktlinie existieren zwei Sortieralgorithmen (`QuickSort`, `MergeSort`), die sich gegenseitig ausschließen. Durch die zusätzliche Bedingung `Liste -> Sortierung.MergeSort` enthält jede Variante der Produktlinie `Liste` den Sortieralgorithmus `MergeSort`. Dementsprechend handelt es sich bei dem Feature `QuickSort` um ein Dead-Feature, da es in keiner Variante der Produktlinie vorhanden sein kann.
Zur Überprüfung eines bestehenden Feature-Modells auf Dead-Features kann wiederum die aussagenlogische Repräsentation in Verbindung mit einem SAT-Solver verwendet werden. Einen entsprechenden Pseudocode zur Suche nach Dead-Features (siehe Pseudocode 1 auf der nächsten Seite) veröffentlichte Henneberg [Hen11]. Dementsprechend

wird die Variable eines Features `f`, das auf die Eigenschaft Dead-Feature analysiert werden soll, auf `true` gesetzt. Dies entspricht einer Teilkonfiguration des Modells, bei der `f` ausgewählt wurde. Anschließend wird durch den SAT-Solver überprüft, ob das Modell mit dieser Teilkonfiguration immer noch gültig ist. Sollte dies nicht der Fall sein, kommt das Feature `f` in keiner Variante der Produktlinie vor. Somit handelt es sich nach der Definition um ein Dead-Feature.

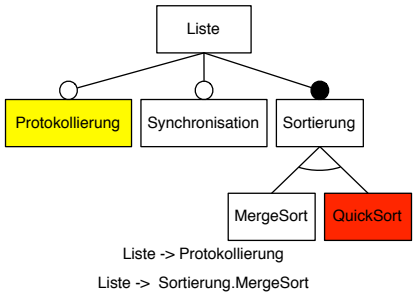


Abbildung 4.2: Dead-Features (rot) und False-Optional-Features (gelb) in Produktlinien.

Einige verfügbare SAT-Solver besitzen die Möglichkeit alle Variablen abzufragen, die entweder immer `Wahr` oder immer `Falsch` sind. Bei der Verwendung solcher SAT-Solver wird die Berechnung für Dead-Features stark vereinfacht, da es sich automatisch bei Features, deren Variablen immer den Wert `Falsch` aufweisen, um Dead-Features handelt. Ein Beispiel für einen SAT-Solver, der solche Anfragen unterstützt, ist Sat4J[1].

Pseudocode 1 Berechnung von Dead-Features [Hen11].

```
1:  function ComputeDeadFeatures(M, F)
2:      F_Dead ← ∅
3:      for all f ∈ F do
4:          M ← addAssumption(M, f = 1)        ▷ Annahme, Feature ist aktiviert
5:          if ¬satisfiable(M) then            ▷ Gibt es gültige Konfigurationen?
6:              F_Dead ← F_Dead ∪ {f}
7:          end if
8:          M ← removeAssumptions(M)           ▷ Entfernen der Annahme
9:      end for
10:     return F_Dead
11: end function
```

Bei False-Optional-Features handelt es sich ebenfalls um eine Anomalie, die von Benavides folgendermaßen definiert wurde [BSRC10]:

> „*A feature is false optional if it is included in all the products of the product line despite not being modelled as mandatory.*“

[1] http://www.sat4j.org/

Dementsprechend ist ein False-Optional-Feature ein Feature, welches zwar als optional modelliert wurde, aber trotzdem in jedem Produkt der Produktlinie vorhanden ist. Beispielsweise wird das Feature `Protokollierung` der Produktlinie `Liste` (siehe Abbildung 4.2 auf der vorherigen Seite) zu einem False-Optional-Feature, wenn zusätzlich die Bedingung `Liste -> Protokollierung` verlangt wird. Da das Feature optional ist, sollte es nicht in jeder Variante der Produktlinie vorhanden sein. Die zusätzliche Bedingung widerspricht jedoch dieser Modellierung.
Zur Berechnung von False-Optional-Features kann wiederum die aussagenlogische Darstellung des Feature-Modells und der SAT-Solver zur Überprüfung der Erfüllbarkeit verwendet werden (siehe Pseudocode 2). Dementsprechend wird jeweils die Variable eines optionalen Features auf `Falsch` gesetzt und anschließend die Erfüllbarkeit des Modells in Bezug auf diese Teilkonfiguration getestet. Wenn das Modell nach dieser Teilkonfiguration nicht mehr erfüllbar ist, handelt es sich um ein False-Optional-Feature, da es nicht deaktiviert werden darf.

Pseudocode 2 Berechnung von False-Optional-Features (Angelehnt an [Hen11]).

```
1:  function ComputeFalseOptionalFeatures(M, F)
2:      F_FalseOpt ← ∅
3:      for all f ∈ F do
4:          if isOptional(f) then
5:              M ← addAssumption(M, f = 0)        ▷ Annahme, Feature ist deaktiviert
6:              if ¬satisfiable(M) then            ▷ Gibt es gültige Konfigurationen?
7:                  F_FalseOpt ← F_FalseOpt ∪ {f}
8:              end if
9:              M ← removeAssumptions(M)           ▷ Entfernen der Annahme
10:         end if
11:     end for
12:     return F_FalseOpt
13: end function
```

Verwendung in Multiproduktlinien

Nachdem bisher die Analyse von False-Optional-Features und Dead-Features in Produktlinien genauer betrachtet wurde, wird nun die Verwendung dieser Analysen in Multiproduktlinien genauer untersucht. In diesem Zusammenhang ergeben sich verschiedene Möglichkeiten die Analysen von False-Optional-Features und Dead-Features in den unterschiedlichen Kompositionen, die von VELVET bereitgestellt werden, zu verwenden. Beispielsweise können die Analysen von Dead-Features und False-Optional-Features wie folgt auf eine Multiproduktlinie angewandt werden: (1) auf die gesamte Multiproduktlinie, (2) auf die Vererbungshierarchie und (3) auf Instanzen. Diese Möglichkeiten werden im Folgenden genauer betrachtet.

(1) Die Analyse der gesamten Multiproduktlinie erfordert die Überführung der einzelnen Feature-Modelle in ein neues Feature-Modell, das die gesamte Multiproduktlinie repräsentiert. Im vorherigen Kapitel wurde die Überführung von Feature-Modellen bei Vererbung und bei Aggregation zu einen Gesamtmodell betrachtet. Dies

bildet die Grundlage für eine Gesamtanalyse der Multiproduktlinie. Da es sich bei VELVET zugleich um eine Konfigurationssprache handelt, ist eine Analyse der gesamten Multiproduktlinie in einem Schritt nicht immer zu empfehlen, da durch die verschiedenen Konfigurationen automatisch Dead-Features und False-Optional-Features entstehen. In Abbildung 4.3 auf der nächsten Seite entsteht bei der Erweiterung der Produktlinie Liste in der Produktlinie `Erweiterte_Liste` durch die Bedingung `Erweiterte_Liste -> MergeSort` unter Anderem das Dead-Feature `QuickSort`. Dabei handelt es sich jedoch um einen gewünschten Konfigurationseffekt und daher um keine Anomalie. Aus diesem Grund sollte im Allgemeinen eine Einzelanalyse mittels (2) und (3) vollzogen werden.

(2) Bei diesem Test wird die Vererbungshierarchie der Multiproduktlinie genauer auf Dead-Features und False-Optional-Features untersucht. Im Gegensatz zu (1) werden dabei False-Optional-Features und Dead-Features erkannt und als Anomalie betrachtet, die nicht durch eine gewünschte Konfiguration entstanden sind. Diese Analyse wird im Folgenden noch genauer betrachtet.

(3) Ähnlich wie die bewusste Konfigurationsentscheidung bezüglich der Vererbung zu Dead-Features führen kann, können auch die verschiedenen Bedingungen bezüglich einer Instanz eine bewusste Einschränkung der Varianten einer Produktlinie bedeuten. Da diese bewussten Konfigurationsentscheidungen bei der Analyse von Dead-Features und False-Optional-Features nicht betrachtetet werden sollten, ist es diesbezüglich lediglich notwendig den Typ der einzelnen Instanzen auf Dead-Features und False-Optional-Features zu untersuchen. Demzufolge können die Instanzen in der Multiproduktlinie `Mail_Client` vom Typ `Liste` auf Dead-Features und False-Optional-Features analysiert werden, indem eine Überprüfung der Produktlinie `Liste` mittels (2) durchgeführt wird.

Bei den verschiedenen Möglichkeiten der Analyse von Dead-Features und False-Optional-Features in Multiproduktlinien wurde eine notwendige Unterscheidung in Bezug auf die Berechnung von Dead-Features und False-Optional-Features in Produktlinien angedeutet. Aus diesem Grund ist es notwendig die Definitionen von Dead-Features und False-Optional-Features für Multiproduktlinien anzupassen. Demnach kann ein Dead-Feature bei der Analyse in Multiproduktlinien wie folgt definiert werden:

> *Ein Feature einer Multiproduktlinie ist ein Dead-Feature, wenn es in keiner Variante der gültigen Kompositionen und einzelnen Produktlinien einer Multiproduktlinie vorkommen kann.*

Dieser Sachverhalt kann beispielsweise durch eine Erweiterung der Produktlinie `Liste` verdeutlicht werden. In Abbildung 4.3 auf der nächsten Seite wird die Produktlinie `Liste` mit dem alternativen Sortieralgorithmen `MergeSort` und `QuickSort` dargestellt. Des Weiteren wird in Quelltext 4.9 auf der nächsten Seite die zugehörige Darstellung in VELVET angegeben. Bei einer Erweiterung der Produktlinie `Liste` zur Produktlinie `Erweiterte_Liste` kann beispielsweise eine Auswahl des Features `MergeSort` durch die Einführung einer zusätzlichen Bedingung stattfinden (siehe Quelltext 4.9 auf der nächsten Seite, Zeile 15). Durch diese Bedingung wird nach Analyse (1) das Feature `QuickSort` automatisch zu einem Dead-Feature, da es nicht mehr ausgewählt werden

kann. Hierbei handelt es sich nach der neuen Definition jedoch um kein Dead-Feature der Multiproduktlinie, da in der Produktlinie `Liste` eine Variante der Liste erzeugt werden kann, in der das Feature `QuickSort` enthalten ist. Im Gegensatz dazu handelt es sich bei dem neu hinzugefügten Feature `HeapSort` (siehe Quelltext 4.9, Zeile 18) nach der neuen Definition bezüglich Multiproduktlinien um ein Dead-Feature der Multiproduktlinie `Erweiterte_Liste`. Dieses Feature kann in keiner Produktlinie der Komposition ausgewählt werden, da es weder in einer Variante der Produktlinie `Liste` noch in einer Variante der Multiproduktlinie `Erweiterte_Liste` vorkommen kann.
In gleicher Weise kann auch die Definition von False-Optional-Features bezogen auf Multiproduktlinien wie folgt angepasst werden:

> *Ein optionales Feature einer Multiproduktlinie ist ein False-Optional-Feature, wenn es in jeder Variante der einzelnen gültigen Kompositionen und in jeder einzelnen Produktlinie der Multiproduktlinie, in der das Feature vorhanden ist, vorkommt.*

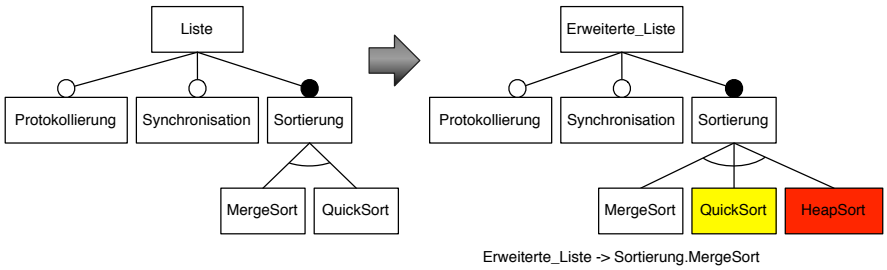


Abbildung 4.3: Dead-Features bei der Analyse des Gesamtmodells: zwei Dead-Features (rot + gelb). Dead-Features bei der Analyse über Vererbungshierarchie: ein Dead-Feature (rot).

```
1  concept Liste{
2    feature Protokollierung;
3    feature Synchronisation;
4
5    mandatory feature Sortierung{
6      oneof{
7        feature QuickSort;
8        feature MergeSort;
9      }
10   }
11 }
```

```
12 concept Erweiterte_Liste : Liste{
13   //kein Dead-Feature -> bewusste
14   //Einschränkung der Varianten
15   constraint Sortierung.MergeSort;
16
17   //Dead-Feature
18   feature Sortierung.HeapSort;
19 }
```

Quelltext 4.9: Dead-Features in Multiproduktlinien bei Vererbung in VELVET.

Nach dieser Definition handelt es sich bei einem optionalen Feature um ein False-Optional-Feature der Multiproduktlinie, wenn es in jeder Produktlinie in der es vorhanden ist, ein False-Optional-Feature darstellt. Diese neue Definition kann wiederum auf die Multiproduktlinie `Erweiterte_Liste` angewandt werden. Durch die Bedingung `Erweiterte_Liste -> MergeSort` wird das Feature `MergeSort` zu einem False-Optional-Feature, wenn lediglich die Analyse (1) auf der gesamten Multiproduktlinie durchge-

führt wird. Wird jedoch die Vererbungshierarchie einzeln untersucht und somit die neue Definition von False-Optional-Features beachtet, so handelt es sich bei dem Feature `MergeSort` nicht um ein False-Optional-Feature, da es in mindestens einer Variante der Produktlinie `Liste` nicht vorkommt.
Zusammenfassend betrachtet, existiert nach der neuen Definition von Dead-Features und False-Optional-Features in der Multiproduktlinie `Erweiterte_Liste` lediglich ein Dead-Feature (siehe Abbildung 4.3 auf der vorherigen Seite). Im Gegensatz zu Analyse (1), in der die gesamte Multiproduktlinie betrachtet wird, bekommt der Modellierer hier eine bessere Übersicht über die Dead-Features und False-Optional-Features. Diese bessere Übersicht ergibt sich aus der Tatsache, dass scheinbare Dead-Features oder False-Optional-Features der gewünschten Konfiguration, nicht betrachtet werden.
Zur Analyse von Multiproduktlinien bezüglich False-Optional-Features und Dead-Features in Verbindung mit Vererbung sind verschiedene Einzelanalysen notwendig. Im Pseudocode 3 wird ein entsprechender Algorithmus abgebildet, der zur Berechnung der einzelnen Dead-Features und False-Optional-Features einer Multiproduktlinie verwendet werden kann. Die Funktion `ComputeAnomaly(M, ComputeDeadF)` verwendet die bestehenden Funktionen zur Berechnung von Dead-Features und False-Optional-Features (Zeile 6, 8), wobei der Parameter `ComputeDeadF` den Berechnungstyp (Dead-Features oder False-Optional-Features) bestimmt. Als weiteren Parameter besitzt die Funktion `ComputeAnomaly` die zu analysierenden Feature-Modelle $M_{AllSpls}$, die nacheinander von der Funktion verarbeitet werden. In diesem Zusammenhang wird das jeweils zu betrachtende Feature-Modell an die eigentliche Funktion zur Berechnung von Dead-Features oder False-Optional-Features übergeben (Zeile 6, 8). Des Weiteren bestimmt der zweite Übergabeparameter die zu betrachtenden Features. Die Resultate der einzelnen Analysen werden jeweils zur Menge F_{An} hinzugefügt und anschließend zurückgegeben.

Pseudocode 3 Berechnung von Dead-Features und False-Optional-Features in Multiproduktlinien.

```
1:  function ComputeAnomaly(M_AllSpls, ComputeDeadF)
2:      F_An ← ∅
3:      for all m ∈ M_AllSpls do
4:          F_DefFeature ← getDefinedFeatures(m)
5:          if ComputeDeadF then
6:              F_An ← F_An ∪ ComputeDeadFeatures(m, F_DefFeature)
7:          else
8:              F_An ← F_An ∪ ComputeFalseOptionalFeatures(m, F_DefFeature)
9:          end if
10:     end for
11:     return F_An
12: end function
```

Zum besseren Verständnis wird die Anwendung von Pseudocode 3 noch einmal bezüglich des Beispiels der Multiproduktlinie `Erweiterte_Liste` betrachtet. Als Eingabe werden an die Funktion `ComputeAnomaly` die Feature-Modelle `Liste` und `Erweiterte_Liste` und `true` zur Berechnung von Dead-Features übergeben. Zunächst wird von dem Algorithmus die Produktlinie `Liste` an die Funktion `ComputeDeadFeatures` mit

den Parametern `Liste` und einer Liste von allen Features aus der Produktlinie `Liste` übergeben. Als Resultat ergibt sich eine leere Menge, da diese Produktlinie kein Dead-Feature enthält. Im nächsten Schritt wird die Produktlinie `Erweiterte_Liste` betrachtet. Diese wird wiederum an die Funktion `ComputeDeadFeature` übergeben. In diesem Fall enthält die Menge der zu betrachtenden Features lediglich ein Element, das Feature `HeapSort`. Dieses Feature wird zugleich von der Funktion `ComputeDeadFeature` als Dead-Feature erkannt und zu der Menge F_{An} hinzugefügt. Die beiden Produktlinien `Liste` und `Erweiterte_Liste` besitzen demnach insgesamt ein Dead-Feature, das Feature `HeapSort`.

4.3.2 Anzahl und Ausgabe von Varianten

Im vorherigen Abschnitt wurden Möglichkeiten zur Analyse von Dead-Features und False-Optional-Features bezüglich der Kompositionsmechanismen Vererbung und Aggregation betrachtet. Eine weitere bedeutende Analyse stellt die Ausgabe und die Bestimmung der Anzahl der verschiedenen Varianten einer Produktlinie dar. Diese Analyse ermöglicht dem Modellierer das erstellte Modell auf Korrektheit zu überprüfen. In diesem Zusammenhang können folgende Fragen beantwortet werden: (1) „Bestehen zuviel oder zuwenig Konfigurationsmöglichkeiten?“ und (2) „Können alle geplanten Varianten generiert werden?“. Durch die Beantwortung dieser Fragen kann der Modellierer prüfen, ob das erstellte Modell den gewünschten Anforderungen entspricht.
Im Pseudocode 4 wird ein Algorithmus zur Bestimmung der Anzahl der Varianten abgebildet. Dieser Algorithmus verwendet einen SAT-Solver zur Berechnung. Einige existierende SAT-Solver (zum Beispiel Sat4J[2]) bieten eine entsprechende Funktion an, die die zusätzliche Berechnung durch den angegebenen Algorithmus überflüssig macht. Wenn diese Funktion jedoch nicht vom vorhandenen SAT-Solver unterstützt wird, kann der abgebildete Algorithmus zur Bestimmung der Anzahl der Varianten verwendet werden.

Pseudocode 4 Berechnung der Anzahl der Varianten.

```
1: function ComputeNumberOfVariants(M, F)
2:     numberOfVar ← 0
3:     while i ≤ 10 do
4:         if satisfiable(M) then
5:             numberOfVar ← numberOfVar + 1
6:         else
7:             return numberOfVar
8:         end if
9:         F_Select ← getSolution(M)
10:        F_Deselect ← F \ F_Select
11:        addClauseToSolver(F_Select, F_Deselect)
12:    end while
13: end function
```

[2] http://www.sat4j.org/

Im Pseudocode 4 auf der vorherigen Seite wird ein bestehendes Modell auf Erfüllbarkeit untersucht. Wenn das Modell erfüllbar ist, wird die Anzahl der Varianten um „1“ erhöht. Andernfalls wird die aktuelle Anzahl der Varianten von der Funktion zurückgegeben. Zur Bestimmung einer weiteren Lösung wird zum SAT-Solver eine weitere Klausel hinzugefügt. In dieser Klausel werden alle Variablen der nicht ausgewählten Features und zusätzlich aller ausgewählten Features in negierter Form hinzugefügt. Anschließend folgen dann weitere Erfüllbarkeitsprüfungen bis der gesamte Ausdruck nicht mehr erfüllbar ist.

Verwendung in Multiproduktlinien

Im vorherigen Abschnitt wurde eine mögliche Berechnung der Anzahl der Varianten in einer Produktlinie vorgestellt. Im Folgenden wird diese Analyse, bezogen auf die Verwendung in Multiproduktlinien, noch einmal genauer betrachtet. In diesem Zusammenhang kann die Analyse bezüglich der Anzahl und Ausgabe der verschiedenen Varianten wiederum auf verschiedenen Ebenen durchgeführt werden: (1) auf die gesamte Multiproduktlinie, (2) auf die Vererbungshierarchie und (3) auf Instanzen.

(1) Die Bestimmung der Anzahl der Varianten für die gesamte Multiproduktlinie erfordert wiederum die Überführung der verschiedenen Einzelmodelle in ein Gesamtmodell. Anschließend kann die Analyse zur Bestimmung der Varianten wie bei gewöhnlichen Produktlinien vorgenommen werden. Die Analyse auf Basis der gesamten Multiproduktlinie verspricht die gleichen Vorteile, wie die Analyse von einer einzelnen Produktlinie. Dementsprechend kann die Ausgabe und Anzahl der verschiedenen Varianten eine Übersicht über die Korrektheit der Modellierung beziehungsweise über die Übereinstimmung der gewünschten Variabilität des Modells liefern.

(2) Im Gegensatz zu der vollständigen Analyse des Gesamtmodells kann eine schrittweise Analyse über die Vererbungshierarchie weitere Aufschlüsse über die Korrektheit der Modellierung geben. Wenn beispielsweise durch die Erweiterung eines bestehenden Modells die Variabilität erhöht werden soll, kann der Vergleich der Anzahl der verschiedenen Varianten, zwischen dem ursprünglichen und dem erweiterten Modell, Rückschlüsse auf eine korrekte Modellierung geben. Dies kann beispielsweise an der Multiproduktlinie `Erweiterte_Liste` (siehe Abbildung 4.3 auf Seite 40) veranschaulicht werden. Die Produktlinie `Liste` enthält 8 mögliche Varianten, wobei die Multiproduktlinie `Erweiterte_Liste` durch die zusätzliche Bedingung lediglich 4 verschiedene Varianten ermöglicht. Die Variantenvielfalt wurde demnach nicht vergrößert sondern, bezogen auf die Produktlinie `Liste`, sogar eingeschränkt.

(3) In einer weiteren Analyse über die Anzahl der Varianten bezüglich der Instanzen einer Multiproduktlinie können weitere Fehler in der Modellierung aufgedeckt werden. Beispielsweise ist oftmals eine vollständige Konfiguration einer Instanz gewünscht, weshalb eine Kontrollausgabe zur Sicherstellung dieser Eigenschaft von großer Bedeutung ist. Die Notwendigkeit und die Durchführung einer solchen Analyse werden im Folgenden noch einmal genauer betrachtet.

In Abbildung 4.4 wird die Multiproduktlinie `Mail_Client` mit den verschiedenen Instanzen der Produktlinie `Liste` dargestellt. Im vorherigen Kapitel (siehe Abschnitt 3.2.2 auf Seite 25) wurde die Überführung der verschiedenen Produktlinien (`Liste` und `Mail_Client`) in ein Feature-Modell dargestellt, das die gesamte Multiproduktlinie repräsentiert. Dieses Modell wird im Folgenden als Grundlage für weitere Beispiele bezüglich der Analysen von Instanzen verwendet. Zur Bestimmung der Anzahl der Varianten einer Instanz gibt es wiederum zwei Möglichkeiten: (2.1) Bestimmung der Anzahl über den Typ der Instanz, oder (2.2) Bestimmung der Anzahl unter Berücksichtigung der Konfigurationen in der Multiproduktlinie.

(2.1) Im Beispiel der Produktlinie `Mail_Client` können die möglichen Varianten der Instanzen `mailListe` und `fehlerListe` durch den jeweiligen Instanz-Typ bestimmt werden. Der Typ der beiden Instanzen ist die Produktlinie `Liste`. Bei der Analyse bezüglich der Anzahl der Varianten dieser Produktlinie werden 2^3 also 8 Varianten ermittelt. Diese Information hat jedoch im Bezug auf Multiproduktlinien wenig Bedeutung, da die jeweiligen Instanzen in der Multiproduktlinie weiter konfiguriert wurden.

(2.2) Im Gegensatz zu (2.1) wird bei dieser Variante die Konfiguration der Instanzen berücksichtigt (siehe Abbildung 4.4). Beispielsweise wird bei der Instanz `mailListe` durch die Bedingung `fehlerListe -> fehlerListe.Synchronisation` das Feature `Synchronisation` ausgewählt und die Anzahl der Variante auf $2^2 = 4$ begrenzt. Weitere Bedingungen in der Multiproduktlinie `Mail_Client` könnten die Anzahl der Varianten bezüglich der Instanz weiter beschränken. Die Angabe, dass die Instanz vier verschiedene Konfigurationen zulässt, gibt dem Modellierer weitere Rückschlüsse über die Korrektheit der Modellierung. Die Realisierung zur Berechnung der verschiedenen Varianten einer Instanz unter Berücksichtigung der verschiedenen Konfigurationen in der Multiproduktlinie wird im Folgenden genauer betrachtet.

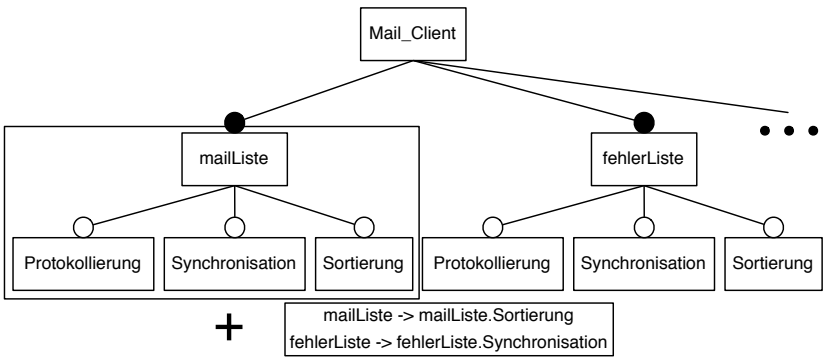


Abbildung 4.4: Anzahl der Varianten einer Instanz in Multiproduktlinien.

Die Bestimmung der Anzahl der Varianten bezogen auf Instanzen und deren Konfigurationen in Multiproduktlinien ist eine komplexe Aufgabe. In diesem Zusammenhang müssen zwar alle Bedingungen, die in der Multiproduktlinie bezüglich der Instanz vorhanden sind, betrachtet werden, zugleich sind aber alle Features außerhalb der Instanz

unbedeutend. Die Anforderungen, bei der Analyse von verschiedenen Varianten einer Instanz, sind äquivalent zu der Analyse von Feature-Modellen, in denen *abstrakte* Features enthalten sind. Zum weiteren Verständnis der Zusammenhänge ist es daher notwendig die Bedeutung von abstrakten Features zu beschreiben. Aus diesem Grund folgt zunächst die Definition eines abstrakten Features von Thüm et al. [TKES11]:

> „*We define a feature as abstract, if and only if it is not mapped to any implementation artifacts.*“

Demnach handelt es sich bei abstrakten Features um Features, für die keine Implementierung erforderlich ist. Zur Bestimmung der Anzahl der verschiedenen Varianten eines Programms sind solche Features daher ebenfalls unbedeutend. Aus diesem Grund entwickelten Thüm et al. eine Möglichkeit bei der die abstrakten Features in einem Feature-Modell vernachlässigt werden. In gleicher Weise müssen auch alle Features der Multiproduktlinie behandelt werden, die nicht der jeweiligen Instanz angehören.
Zur Vernachlässigung eines Features A, das nicht zu einer bestimmten Instanz gehört, muss der aussagenlogische Ausdruck p des Feature-Modells nach Thüm et al. folgendermaßen zu $p^{'}$ angepasst werden [TKES11]:

$$p^{'} = p[A \rightarrow true] \vee p[A \rightarrow false]$$

Das bedeutet, dass der ursprüngliche Ausdruck zunächst so angepasst wird, dass alle Vorkommen des Features A durch `true` ersetzt werden. Des Weiteren wird in gleicher Weise der ursprüngliche Ausdruck so abgewandelt, dass alle Vorkommen des Features A durch `false` ersetzt werden. Diese beiden aussagenlogischen Ausdrücke werden dann durch ein „oder“ verknüpft und bilden zusammen einen Ausdruck, in dem das Feature A nicht mehr vorhanden ist.
Die vorherige Anpassung des aussagenlogischen Ausdrucks bildet die Grundlage für eine weitere Analyse zur Bestimmung der Anzahl der Varianten einer Instanz. Demnach muss die Anpassung des aussagenlogischen Ausdrucks für jedes Feature der Multiproduktlinie `Mail_Client` (siehe Abbildung 4.4 auf der vorherigen Seite) erfolgen, das nicht ein Kind-Feature des jeweiligen Instanz-Features ist. Anschließend kann auf diesem angepassten Ausdruck der Pseudocode 4 auf Seite 42 zur Bestimmung der Anzahl der Varianten angewandt werden.
Im Folgenden wird der aussagenlogische Ausdruck der Multiproduktlinie `Mail_Client` (in verkürzter Form, angelehnt an Abbildung 4.4 auf der vorherigen Seite) dargestellt. Des Weiteren wird exemplarisch die Entfernung des Features `Mail_Client(m)` und die Vereinfachung des aussagenlogischen Ausdrucks verbildlicht. Die Namen der einzelnen Features wurden diesbezüglich abgekürzt. Die Abkürzungen ergeben sich aus den einzelnen Anfangsbuchstaben (außer Feature `Sortierung`: o) der Features in Kombination mit den Anfangsbuchstaben der Eltern-Features. Dementsprechend steht die Abkürzung `mms` für das Feature `Mail_Client.mailListe.Synchronisation`.

$$
\begin{aligned}
&\textbf{Feature-Modell:} && \textbf{Mail_Client}\\
&Wurzel: && m\\
&Instanz\ mailListe: && \wedge\ (mm \rightarrow m) \wedge (m \rightarrow mm) \wedge (mmp \rightarrow mm)\\
& && \wedge\ (mms \rightarrow mm) \wedge (mmo \rightarrow mm) \wedge (mm \rightarrow mmo)\\
&Instanz\ fehlerListe: && \wedge\ (mf \rightarrow m) \wedge (m \rightarrow mf) \wedge (mfp \rightarrow mf)\\
& && \wedge\ (mfs \rightarrow mf) \wedge (mfo \rightarrow mf) \wedge (mf \rightarrow mfs)\\
&\textbf{Entfernung:} && \textbf{Feature Mail_Client (m)}\\
&m \rightarrow true: && \big(true \wedge (mm \rightarrow true) \wedge (true \rightarrow mm) \wedge (mf \rightarrow true)\\
& && \wedge\ (true \rightarrow mf) \vee\\
&m \rightarrow false: && false \wedge (mm \rightarrow false) \wedge (false \rightarrow mm) \wedge (mf \rightarrow false)\\
& && \wedge\ (false \rightarrow mf)\big)\\
& && \\
& && \wedge\ (mmp \rightarrow mm) \wedge (mms \rightarrow mm) \wedge (mmo \rightarrow mm)\\
& && \wedge\ (mm \rightarrow mmo) \wedge (mfp \rightarrow mf) \wedge (mfs \rightarrow mf)\\
& && \wedge\ (mfo \rightarrow mf) \wedge (mf \rightarrow mfs)\\
&\textbf{Vereinfachung:} && \\
& && (mm \wedge mf \vee \neg mm \wedge \neg mf)\\
& && \wedge\ (mmp \rightarrow mm) \wedge (mms \rightarrow mm) \wedge (mmo \rightarrow mm)\\
& && \wedge\ (mm \rightarrow mmo) \wedge (mfp \rightarrow mf) \wedge (mfs \rightarrow mf)\\
& && \wedge\ (mfo \rightarrow mf) \wedge (mf \rightarrow mfs)
\end{aligned}
$$

Die beispielhafte Entfernung des Features `Mail_Client` wurde nach dem Algorithmus von Thüm et al. durchgeführt [TKES11]. Zur Bestimmung der Anzahl der Varianten der Instanz `mailListe` muss der gleiche Vorgang für das Feature `fehlerListe` und dessen Kind-Features wiederholt werden. Anschließend beinhaltet der aussagenlogische Ausdruck lediglich das Feature `mailListe` und die zugehörigen Kind-Features.

$$
\begin{aligned}
&\textbf{Entfernung:} && \textbf{Features mf, mfp, mfs, mfo}\\
&\textbf{und Vereinfachung:} && \\
& && (mmp \rightarrow mm) \wedge (mms \rightarrow mm) \wedge (mmo \rightarrow mm)\\
& && \wedge\ (mm \rightarrow mmo)
\end{aligned}
$$

Zur Bestimmung der Varianten der Instanz `mailListe` muss zusätzlich noch das Wurzel-Feature der Instanz (`mailListe`) in den aussagenlogischen Ausdruck eingefügt werden. Andernfalls würde auch das leere Produkt, in dem kein Feature ausgewählt wurde, als

weitere Variante existieren und das gewünschte Ergebnis verfälschen. Der vollständige aussagenlogische Ausdruck wird im Folgenden noch einmal abgebildet.

$$
\begin{aligned}
&\textbf{Hinzufügen:} && \textbf{Wurzel-Feature der Instanz mailListe} \\
& && mm \\
&\wedge && (mmp \rightarrow mm) \wedge (mms \rightarrow mm) \wedge (mmo \rightarrow mm) \\
&\wedge && (mm \rightarrow mmo) \\
&\textbf{Vereinfachung:} && \\
& && mm \wedge mmo
\end{aligned}
$$

Insgesamt ist der resultierende aussagenlogische Ausdruck über die Features `mm`, `mmp`, `mms` und `mmo` definiert. Die Vereinfachung zeigt, dass die Features `mm` und `mmo` in jeder Variante der Instanz enthalten sind. Die Werte der beiden anderen Features können sowohl „wahr“ als auch „falsch“ sein, um den Ausdruck zu erfüllen. Durch diese zwei optionalen Features ergeben sich daher 2^2 Varianten. Demnach besitzt die Instanz `mailListe` nach der Konfiguration in der Multiproduktlinie `Mail_Client` 4 Varianten.

4.3.3 Core-Features

Die Bestimmung von Core-Features kann ebenfalls zur Überprüfung von Produktlinien verwendet werden und dem Modellierer Rückschlüsse über die Korrektheit des Modells liefern. Ein Core-Feature wird nach Benavides et al. wie folgt definiert [BSRC10]:

> „*Core features are the most relevant features of software product line since they are supposed to appear in all products.*“

Demnach handelt es sich bei Core-Features um Features, die in jedem Produkt einer Produktlinie vorhanden sind. Zur Analyse eines Feature-Modells auf Core-Features wurde von Henneberg ein entsprechender Algorithmus dargestellt (siehe Pseudocode 5) [Hen11].

Pseudocode 5 Berechnung von Core-Features [Hen11].

```
1:  function ComputeCoreFeatures(M, F)
2:      F_Core ← ∅
3:      for all f ∈ F do
4:          M ← addAssumption(M, f = 0)        ▷ Annahme, Feature ist deaktiviert
5:          if ¬satisfiable(M) then            ▷ Gibt es gültige Konfigurationen?
6:              F_Core ← F_Core ∪ {f}
7:          end if
8:          M ← removeAssumptions(M)           ▷ Entfernen der Annahme
9:      end for
10:     return F_Core
11: end function
```

In gleicher Weise wie bei der Bestimmung der False-Optional-Features kann auch die Bestimmung von Core-Features durchgeführt werden. In Pseudocode 5 auf der vorherigen Seite wird nacheinander für jedes Feature die Annahme hinzugefügt, dass das jeweilige Feature deaktiviert ist. Wenn das Modell anschließend nicht mehr erfüllbar ist, handelt es sich um ein Core-Feature. Im Gegensatz zu False-Optional-Features, bei denen lediglich optionale Features betrachtet wurden, werden in diesem Fall alle Features analysiert. Die Berechnung der False-Optional-Features ist demnach ein Spezialfall und kann auch auf Grundlage der Core-Features realisiert werden (siehe Henneberg [Hen11]).

Verwendung in Multiproduktlinien

Nachdem zuvor die Verwendung von Core-Features in gewöhnlichen Produktlinien betrachtet wurde, folgt nun die Analyse einer möglichen Bestimmung von Core-Features in Multiproduktlinien. In diesem Zusammenhang kann die Analyse von Core-Features wiederum auf mehreren Ebenen erfolgen: (1) auf der gesamten Multiproduktlinie, (2) auf die Vererbungshierarchie und (3) auf Instanzen.

(1) Äquivalent zu den vorherigen Analysen, wie beispielsweise die Analyse von Dead-Features, erfolgt die Analyse von Core-Features bezogen auf der gesamten Multiproduktlinie auf Grundlage eines Feature-Modells, das die gesamte Multiproduktlinie repräsentiert. Diese Analyse kann wie bei einer gewöhnlichen Multiproduktlinie vollzogen werden. Dies ergibt beispielsweise für die Multiproduktlinie `Erweiterte_Liste` drei Core-Features, die Features `Erweiterte_Liste`, `Sortierung` und `MergeSort` (siehe Abbildung 4.5 auf der nächsten Seite).

(2) Im Gegensatz zu der Analyse aus (1), kann die Analyse über die Vererbungshierarchie weitere Aufschlüsse über die Korrektheit der Modellierung beziehungsweise der Komposition der einzelnen Produktlinien liefern. Dies kann beispielsweise an der Multiproduktlinie `Erweiterte_Liste` dargestellt werden (siehe Abbildung 4.5 auf der nächsten Seite). Das Basiskonzept enthält lediglich zwei Core-Features, die Features `Liste` und `Sortierung`. Im Gegensatz dazu enthält die Produktlinie `Erweiterte_Liste` drei Core-Features. Durch die zusätzlich eingeführte Bedingung in der Produktlinie `Erweiterte_Liste` wird das Feature `MergeSort` ebenfalls zu einem Core-Feature. Dies ermöglicht einen Vergleich der Core-Features vor und nach der Kombination zu der Produktlinie `Erweiterte_Liste` und bietet somit mehr Kontrollmöglichkeiten. Des Weiteren ermöglicht die schrittweise Analyse dem Modellierer eine erleichterte Fehlersuche. In Quelltext 4.13 auf der nächsten Seite wird in der Produktlinie `Erweiterte_Liste` beispielsweise versucht ein Core-Feature zu deaktivieren. Eine Analyse nach (1) ist nicht möglich, da das gesamte Modell nicht erfüllbar ist. Die Analyse über die Vererbungshierarchie zeigt jedoch, dass das Feature `Sortierung` ein Core-Features ist. Die Deaktivierung des Core-Features `Sortierung` kann demnach schneller als Fehler erkannt werden.

(3) In gleicher Weise wie die Analyse aus (2) kann auch eine schrittweise Analyse über die Instanzen erfolgen. Zunächst wird die Analyse (2) auf die Instanzen angewandt. Anschließend folgt eine Analyse in der die gesamte Multiproduktlinie

auf Core-Features untersucht wird. Dies liefert dem Modellierer weitere hilfreiche Informationen über das Modell. Des Weiteren kann durch die Einführung einer neuen Analyse für Instanzen der Informationsgehalt weiter erhöht werden. Im Folgenden wird die neue Analyse der Core-Instanzen eingeführt und deren Bestimmung genauer betrachtet.

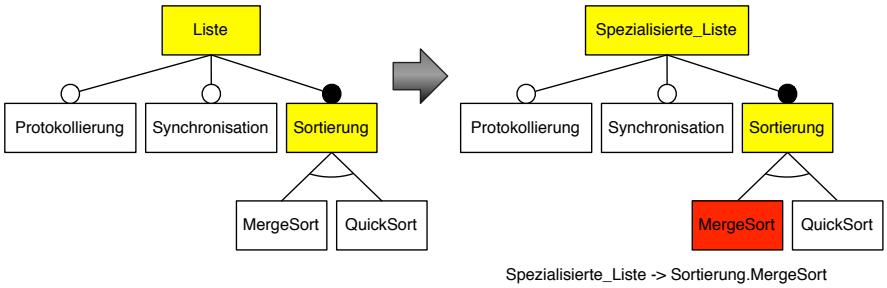


Abbildung 4.5: Core-Features der Produktlinie `Liste` durch Analyse (2) (gelb) und Core-Features der Multiproduktlinie `Spezialisierte_Liste` (gelb + rot).

```
1  concept Liste{
2    ...
3    mandatory feature Sortierung{
4      ...
5    }
6  }
```

```
7  concept Erweiterte_Liste : Liste
8    //Sortierung wird deaktivieren
9    //-> Modell nicht erfüllbar
10   constraint !Sortierung;
11 }
```

Quelltext 4.13: Core-Features in Multiproduktlinien bei Vererbung in VELVET.

Die schrittweise Bestimmung von Core-Features auf den verschiedenen Ebenen der Vererbungshierarchie und bezüglich der verschiedenen Instanzen liefern dem Modellierer detaillierte Informationen zur Kontrolle des Gesamtmodells. Diese Informationen können durch die Einführung einer weiteren Analyse, der Analyse von Core-Instanzen, zusätzlich erweitert werden. Eine Core-Instanz kann wie folgt definiert werden:

> *Eine Instanz einer Multiproduktlinie wird als Core-Instanz bezeichnet, wenn das Wurzel-Feature dieser Instanz in jeder Variante der Multiproduktlinie vorhanden ist.*

Bei der Analyse bezogen auf Core-Instanzen handelt es sich der Definition nach um einen Spezialfall von Core-Features. In diesem Zusammenhang werden von der Multiproduktlinie lediglich Wurzel-Features von Instanzen auf die Eigenschaft der Core-Features untersucht. Wenn beispielsweise die Multiproduktlinie `Mail_Client` auf Core-Instanzen untersucht wird (siehe Abbildung 4.6 auf der nächsten Seite), werden die Instanzen `mailListe` und `fehlerListe` als Core-Instanzen identifiziert. Zur automatischen Bestimmung der Core-Instanzen kann wiederum der Algorithmus zur Bestimmung von Core-Features verwendet werden (siehe Pseudocode 5 auf Seite 47). In diesem Zusammenhang müssen jedoch lediglich das Feature-Modell, das die gesamte Multiproduktlinie repräsentiert, und die beiden Wurzel-Features der Instanzen (`mailListe` und `fehlerListe`) übergeben werden.

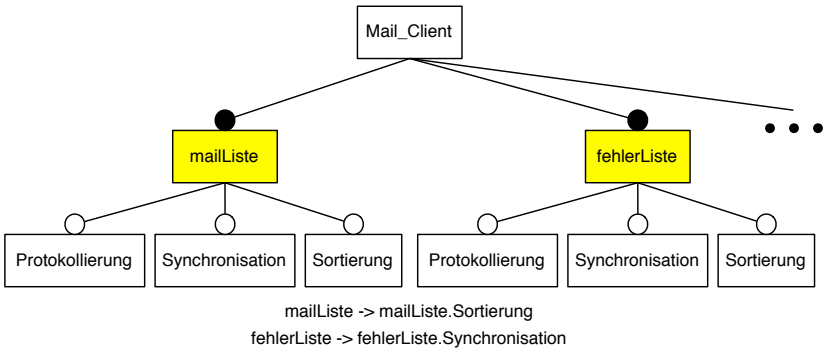


Abbildung 4.6: Core-Instanzen in Multiproduktlinien.

4.3.4 Atomare Mengen

Eine weitere bedeutende Analyse in Produktlinien stellt die Bestimmung von atomaren Mengen dar. Wie bei den vorherigen Analysen wird zunächst die Analyse im Kontext von Produktlinien noch einmal vorgestellt. Anschließend wird die Verwendung dieser Analyse für Multiproduktlinien genauer betrachtet.
Atomare Mengen werden von Benavides et al. folgendermaßen beschrieben [BSRC10]:

> *„An atomic set is a group of features (at least one) that can be treated as an unit when performing certain analyses.“*

Bei atomaren Mengen handelt es sich demnach um eine Gruppe von Features, die als eine Einheit betrachtet werden können. Einerseits kann unter Verwendung dieser Information ein bestehendes Feature-Modell deutlich vereinfacht werden, indem die Features einer Menge zu einem Feature zusammengefasst werden. Oftmals ist die Zusammenfassung von Features jedoch nicht gewünscht, da dies beispielsweise die Übersichtlichkeit des Modells verschlechtert. Andererseits kann diese Analyse auch zur Überprüfung der Korrektheit eines Modells verwendet werden, da gewünschte modellierte Zusammenhänge bestätigt werden können.
In Abbildung 4.7 auf der nächsten Seite wird abermals die Produktlinie `Liste` abgebildet. Ein Beispiel für eine atomare Menge sind die Features `Sortierung` und `MergeSort`. Sie können zu einem Feature zusammengefasst werden, da sie in jeder Variante der Multiproduktlinie immer gemeinsam enthalten oder nicht enthalten sind. Abgesehen von einelementigen Mengen, existieren in dem Beispiel keine weiteren atomaren Mengen.
In Pseudocode 6 auf der nächsten Seite wird ein entsprechender Pseudocode zur Bestimmung der atomaren Mengen aufgeführt. Dieser wurde von Henneberg übernommen [Hen11]. Zusammenfassend werden in dem Algorithmus jeweils Annahmen für je zwei Features zu dem bestehenden Modell hinzugefügt. Ein Feature ist aktiviert, das andere deaktiviert. Wenn das Modell mit den festgelegten Annahmen nicht mehr erfüllbar ist, dann gehören diese beiden Features in die gleiche atomare Menge. Weitere Details zum abgebildeten Algorithmus können den Ausarbeitungen von Henneberg entnommen werden [Hen11].

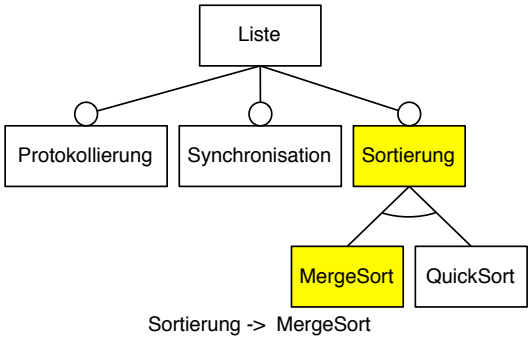


Abbildung 4.7: Atomare Mengen in Produktlinien.

Pseudocode 6 Berechnung von atomaren Mengen [Hen11].

```
1:  function ComputeAtomicSets(M, F)
2:      if |F| = 1 then
3:          simplify(M, F)
4:          return
5:      end if
6:      for all f ∈ F do
7:          G ← F \ {f}
8:          H ← ∅
9:          for all g ∈ G do
10:             M ← addAssumptions(M, f = 1 ∧ g = 0)      ▷ f aktiviert, g deaktiviert
11:             if ¬satisfiable(M) then
12:                 H ← H ∪ {g}
13:             end if
14:             M ← removeAssumptions(M)                  ▷ Entfernen der Annahme
15:         end for
16:         if G ≠ H then
17:             computeAtomicSets(M, H ∪ {f})
18:             computeAtomicSets(M, F \ (H ∪ {f}))
19:             return
20:         end if
21:     end for
22:     simplify(M, F)
23:     return
24: end function
```

Verwendung in Multiproduktlinien

Die Verwendung der Analyse von atomaren Mengen kann sehr ähnlich wie die Analyse von Core-Features in Multiproduktlinien verwendet werden. Zunächst kann die Durchführung der Analyse wieder auf verschiedenen Ebenen der Multiproduktlinie stattfinden: (1) auf der gesamten Multiproduktlinie, (2) auf der Vererbungshierarchie und (3) auf Instanzen.

(1) Wie bei allen anderen vorherigen Analysen kann die gesamte Multiproduktlinie als ein Feature-Modell dargestellt und eine entsprechende Analyse bezüglich der atomaren Mengen vollzogen werden. Wie bei gewöhnlichen Produktlinien werden durch die Analyse von atomaren Mengen Zusammenhänge zwischen Features sichtbar, die zur Kontrolle der korrekten Modellierung verwendet werden können.

(2) Die Analyse der atomaren Mengen über die Vererbungshierarchie ermöglicht dem Modellierer detailliertere Informationen zu erhalten, als es die Analyse (1) ermöglicht. Zu diesem Zweck werden die Produktlinien der Vererbungshierarchie einzeln betrachtetet und vorkommende atomare Mengen ausgegeben. Bei der Analyse der Multiproduktlinie `Spezialisierte_Liste` (siehe Abbildung 4.8) auf atomare Mengen wird daher zunächst die Produktlinie `Liste` untersucht. Diese Produktlinie beinhaltet lediglich eine atomare Menge mit mehr als einem Feature. Die atomare Menge besteht aus den Features `Liste` und `Sortierung`. Anschließend wird die Multiproduktlinie `Spezialisierte_Liste` untersucht, die ebenfalls eine mehrelementige atomare Menge besitzt. Durch die zusätzliche Bedingung besteht diese Menge aus drei Features, da das Feature `MergeSort` zu der Einheit hinzugefügt wird. Durch diese Kontrollausgaben bekommt der Modellierer einen besseren Eindruck über die Auswirkungen der definierten Spezialisierung der Produktlinie `Liste` und kann diese überprüfen.

(3) In gleicher Weise wie bei der Analyse der Vererbungshierarchie bietet es sich an, auch die einzelnen Instanzen der Multiproduktlinie schrittweise zu analysieren. Dementsprechend wird die Analyse bezüglich der atomaren Mengen zunächst auf die einzelnen Produktlinien der Instanzen einer Multiproduktlinie angewandt. Dies ermöglicht wiederum einen Vergleich der ursprünglichen atomaren Mengen mit den atomaren Mengen nach der Komposition in einer Multiproduktlinie.

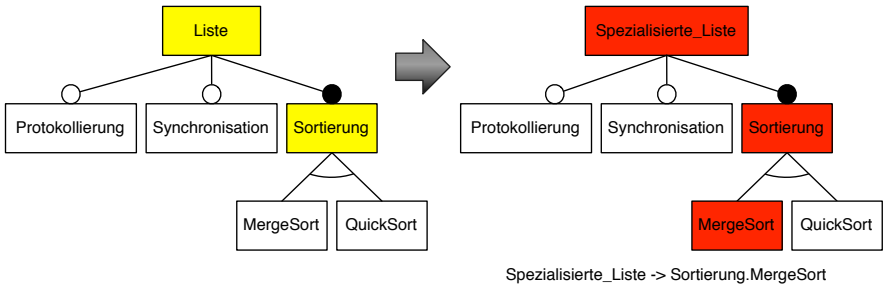


Abbildung 4.8: Mehrelementige atomare Mengen der Produktlinie `Liste` und der Multiproduktlinie `Spezialisierte_Liste`.

4.4 Zusammenfassung

In diesem Kapitel wurde zunächst vorgestellt, wie eine automatisierte Analyse aufgebaut werden kann Anschließend wurden verschiedene notwendige syntaktische Konsistenzprüfungen eingeführt, die zugleich die Grundlage für weitere komplexere Konsistenzprüfungen auf semantischer Ebene bilden. In diesem Zusammenhang wurden verschiedene Konsistenzprüfungen, die aus der Entwicklung von Produktlinien bekannt sind, auf eine mögliche Verwendung in Multiproduktlinien untersucht. Auf diese Weise wurde die Verwendung von Dead-Features, False-Optional-Features, die Anzahl und Ausgabe von Varianten, Core-Features sowie atomare Mengen betrachtet. Die vorgestellten Prüfungen wurden jeweils auf die Verwendung auf unterschiedlichen Ebenen der Multiproduktlinie untersucht. Dabei wurden die gesamte Multiproduktlinie, die Vererbungshierarchie und die einzelnen Instanzen genauer analysiert. Daraus entstanden neue angepasste Definitionen für Dead-Features, False-Optional-Features und Core-Instanzen.

5. Implementierung von Konsistenzprüfungen für VELVET

In diesem Kapitel werden die verschiedenen Konzepte für syntaktische und semantische Konsistenzprüfungen, die im vorherigen Kapitel vorgestellt wurden, noch einmal aufgegriffen und dessen Implementierung genauer betrachtet. Zur Realisierung dieser Konsistenzprüfungen ist eine Werkzeugunterstützung erforderlich, dessen Anforderungen ebenfalls im vorherigen Kapitel dargestellt wurden.
Im folgenden Abschnitt wird zunächst das existierende Werkzeug FmConfig, das als Grundlage zur Erstellung eines neuen Werkzeugs für die Integration der verschiedenen Analysen diente, vorgestellt. Aufgrund dessen wird die Eignung dieses Werkzeugs noch einmal mit den notwendigen Anforderungen zur Integration der vorgestellten Analysen verglichen. Anschließend wird die Implementierung der syntaktischen und der semantischen Analysen genauer betrachtet. In diesem Zusammenhang werden ebenfalls weitere notwendige Implementierungen beschrieben, die zur Grundlage der Integration von Analysen erfüllt werden mussten.

5.1 Bisherige Werkzeugunterstützung für VELVET

Im vorherigen Kapitel wurden verschiedene Analysen bezüglich der Konsistenz von Multiproduktlinien vorgestellt. Zudem wurden verschiedene Aufgaben zur automatisierten Analyse eingeführt, die ein Werkzeug zur Integration der verschiedenen Analysen leisten muss. Hierbei wurden die folgenden Aufgaben herausgestellt: (1) Parsen von Multiproduktlinien in VELVET-Grammatik, (2) Syntaktische Überprüfungen der Modelle, (3) Test auf Erfüllbarkeit der Multiproduktlinien und (4) Realisierung von verschiedenen Konsistenzprüfungen.
Mit der Modellierungssprache VELVET wurden ebenfalls verschiedene Werkzeuge entwickelt, die zumindest einen Teil der notwendigen Funktionalität, zur Unterstützung

von Konsistenzprüfungen, bereitstellen. Bei diesen Werkzeugen handelt es sich um FmView und FmConfig[1]. Unter Anderem ermöglicht FmView bestehende Modelle von anderen Grammatiken in die VELVET-Grammatik zu überführen. Das Kommandozeilen-Werkzeug FmConfig kann hingegen eine gegebene Konfiguration bezüglich einer Multiproduktlinie auf Erfüllbarkeit überprüfen. Bei einer korrekten Konfiguration wird durch FmConfig eine Konfigurationsdatei erzeugt, die eine Variante der Multiproduktlinie beschreibt.
Wie zuvor angedeutet, stellt FmConfig einen Teil der benötigten Funktionalität, die zur Realisierung von automatisierten Konsistenzprüfungen benötigt wird, bereit. Die Kompatibilität von FmConfig, bezogen auf die dargestellten Anforderungen des vorherigen Kapitels, wird im Folgenden genauer betrachtet.

(1) Das Werkzeug FmConfig beinhaltet einen Parser, der die verschiedenen VELVET-Modelle auf syntaktische Korrektheit überprüft. Das Parsen von VELVET-Modellen wird daher vollständig unterstützt, weshalb das Werkzeug diesbezüglich eine gute Grundlage bietet. Im Kapitel 3 auf Seite 17 wurden jedoch verschiedene Anpassungen der VELVET-Grammatik diskutiert, die zur Sicherstellung der Eindeutigkeit der Modellierung durchgeführt wurden. Aus diesem Grund ist eine Anpassung des bestehenden Parsers von FmConfig notwendig.

(2) Nachdem die einzelnen Modelle geparst wurden, überprüft FmConfig die korrekte Referenzierung auf andere Produktlinien. Demnach unterstützt FmConfig auch einen Teil der vorgestellten syntaktischen Prüfungen des vorherigen Kapitels. Weitere syntaktische Prüfungen sind jedoch nicht vorhanden.

(3) Auf der Grundlage der korrekten Ausführung der vorherigen Aufgaben überführt FmConfig die einzelnen Modelle in einen aussagenlogischen Ausdruck, der die gesamte Multiproduktlinie repräsentiert. Anschließend erfolgt eine Überprüfung der Erfüllbarkeit des erstellten Modells beziehungsweise die Überprüfung der Gültigkeit einer gegebenen Konfiguration. Diese Funktionalität wurde in FmConfig bisher nur teilweise umgesetzt, weshalb relativ große Einschränkungen bezüglich der Modellierung bestehen. Einerseits wird die Aggregation mit verschiedenen Instanzen einer Produktlinie nicht unterstützt (siehe Abbildung 3.4 auf Seite 28). Andererseits wird beispielsweise die Erweiterung von bestehenden Konzepten, um ein zusätzliches Feature (Feature `HeapSort` der Multiproduktlinie `Erweiterte_Liste`, siehe Abbildung 5.2 auf Seite 59) nicht korrekt in einen aussagenlogischen Ausdruck überführt. Dies wird in einem späteren Abschnitt noch einmal genauer betrachtet. Eine korrekte Überführung des gesamten Feature-Modells der Multiproduktlinie bildet jedoch die Grundlage zur Integration von Konsistenzprüfungen. Dennoch ist die Verwendung vorhandener Implementierungen sinnvoll, da grundlegende Bestandteile wie der SAT-Solver schon integriert sind.

(4) Da es sich bei FmConfig um ein Kommandozeilen-Werkzeug handelt, das nur zur Konfiguration von Multiproduktlinien entwickelt wurde, sind bisher noch keine semantischen Konsistenzprüfungen integriert.

[1] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/MultiPLe/modeling.htm

Bei der Betrachtung des Funktionalitätsumfangs von FmConfig bietet sich die Wiederverwendung der Infrastruktur von FmConfig zur Integration der vorgestellten Analysen an. Im Gegensatz zur Wiederverwendung der Infrastruktur von FmConfig, wäre eine vollständige Neuentwicklung eines Werkzeugs im Rahmen dieser Masterarbeit in der Komplexität nur schwer realisierbar. Aus diesem Grund wurde auf dieser Basis das Werkzeug *VeAnalyzer* und die vorgestellten Analysen integriert. In Abbildung 5.1 wird noch einmal zusammenfassend die Kompatibilität von FmConfig bezüglich der gegebenen Anforderungen dargestellt. In dieser Darstellung wird der bisherige Stand der integrierten Aufgaben durch einen Ladebalken repräsentiert. Da das Parsen der Modelle fast vollständig wie gewünscht unterstützt wird, ist dieser Ladebalken fast vollständig gefüllt.

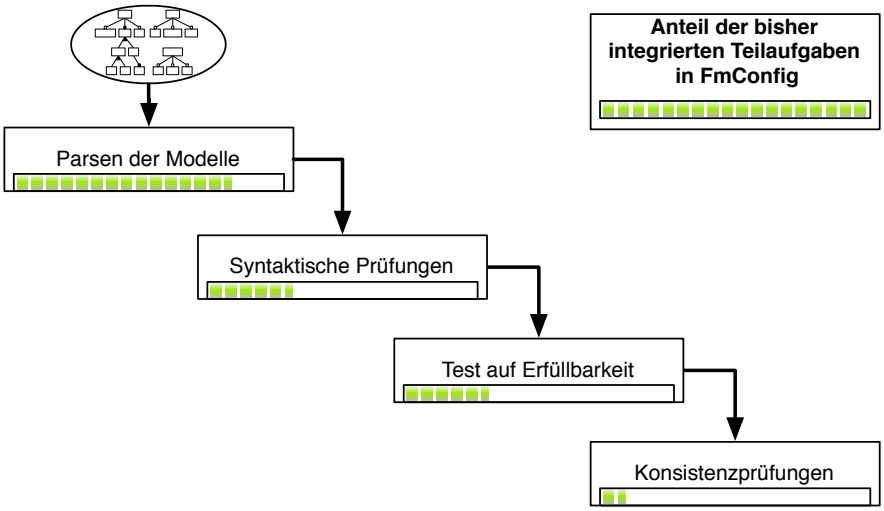


Abbildung 5.1: Anteil der bisher integrierten Teilaufgaben in FmConfig zur Unterstützung von Konsistenzprüfungen.

In den folgenden Abschnitten wird die Integration der einzelnen Analysen in das Werkzeugs VeAnalyzer beschrieben. In diesem Zusammenhang werden auch notwendige Anpassungen für die einzelnen Aufgabenbereiche (siehe Abbildung 5.1) dargelegt.

5.2 Parser und syntaktische Überprüfungen für VELVET-Modelle

Wie im vorherigen Abschnitt beschrieben, wurde das Konfigurations-Werkzeug FmConfig als Basis zur Erstellung des Werkzeugs VeAnalyzer verwendet, um automatisierte Analysen zu realisieren. Des Weiteren wurden im vorherigen Kapitel verschiedene syntaktische Konsistenzprüfungen diskutiert, die zusätzlich zu einem Parser die Konsistenz eines Modells sicherstellen.
Zur Integration der verschiedenen syntaktischen Überprüfungen war es zunächst notwendig den vorhandenen Parser auf die veränderte Grammatik von VELVET anzupassen (siehe Kapitel 3 auf Seite 17). Die Anpassung des Parsers stellte für die Entwicklung

keine Schwierigkeit dar, da der Parser fast vollautomatisch für die gegebene Grammatik generiert und verwendet werden konnte.
Auf Grundlage der korrekt geparsten Modelle, wurden anschließend die verschiedenen syntaktischen Konsistenzprüfungen (siehe Abschnitt 4.2 auf Seite 33) in VeAnalyzer integriert. Zuvor war in FmConfig lediglich ein syntaktischer Test vorhanden, der die Referenzen auf die verschiedenen einzelnen Produktlinien analysiert. Alle Anderen Konsistenzprüfungen wurden im Rahmen dieser Arbeit zu VeAnalyzer hinzugefügt. Eine Übersicht über die bereits vorhandenen und realisierten syntaktischen Konsistenzprüfungen wird in Tabelle 5.1 gegeben.

Syntaktische Prüfung	FmConfig	VeAnalyzer
Referenzen auflösen	✓	✓
Gleicher Name von Attributen und Features	X	✓
Vererbung der gleichen Produktlinie	X	✓
Fehlende Vererbung	X	✓
`mandatory` in Gruppen	X	✓
Modell-Rekursion	X	✓

Tabelle 5.1: Syntaktische Konsistenzprüfungen in FmConfig und VeAnalyzer.

Die verschiedenen syntaktischen Prüfungen wurden teilweise in den Vorgang des Parsens von VeAnalyzer integriert. Dies kann beispielsweise an dem Test *mandatory in Gruppen* veranschaulicht werden. In diesem Fall wird direkt beim Vorgang des Parsens der einzelnen Bestandteile einer Feature-Gruppe getestet, ob ein `mandatory` Feature vorhanden ist. Sollte dies der Fall sein, wird der Vorgang des Parsens abgebrochen und dem Nutzer eine entsprechende Fehlermeldung angezeigt.

5.3 Test auf Erfüllbarkeit für VELVET-Modelle

Nachdem im vorherigen Abschnitt die Integration der verschiedenen syntaktischen Konsistenzprüfungen betrachtet wurde, wird nun auf die Überprüfung der Erfüllbarkeit des Gesamtmodells eingegangen. Wie schon am Anfang des Kapitels erwähnt, unterstützt FmConfig (somit auch VeAnalyzer) den Test auf Erfüllbarkeit bisher nur teilweise. Da eine korrekte Überprüfung der Erfüllbarkeit die Grundlage für weitere Konsistenzprüfungen ist, war eine Verbesserung dieser Funktionalität unumgänglich.
Durch die Anpassung des Parsers für die veränderte VELVET-Grammatik und durch die Integration der syntaktischen Konsistenzprüfungen wurde eine gute Grundlage zur Überprüfung der Erfüllbarkeit des Gesamtmodells geschaffen. Dementsprechend kann bei der Überführung der einzelnen Modelle in einen aussagenlogischen Ausdruck davon ausgegangen werden, dass die Modelle auf syntaktischer Ebene korrekt sind. Nach der Überführung in den aussagenlogischen Ausdruck, kann der in VeAnalyzer integrierte SAT-Solver *MiniSat*[2] zur Überprüfung der Erfüllbarkeit des Modells verwendet werden.

[2] http://minisat.se/

Wie die einzelnen Modelle in einen aussagenlogischen Ausdruck, der die gesamte Multiproduktlinie repräsentiert, bei der Verwendung von Vererbung und Aggregation überführt werden können, wurde in Kapitel 3 auf Seite 17 bereits beschrieben. FmConfig unterstützt bezogen auf Vererbung zumindest teilweise die Übersetzung der einzelnen Modelle in einen aussagenlogischen Ausdruck. Jedoch gibt es Probleme bei der Erkennung von Erweiterungen von bestehenden Features durch Vererbung. Dieses Problem wird an der Abbildung 5.2 beziehungsweise im Quelltext 5.1 noch einmal verdeutlicht, indem das Feature `Sortierung` um ein weiteres Kind-Feature `HeapSort` erweitert werden soll. Bei dem markierten Feature `Sortierung` (Zeile 5, 13) in Quelltext 5.1 handelt es sich um das gleiche Feature. Bei der Erstellung des aussagenlogischen Ausdrucks wird das Feature `Sortierung` (Zeile 13) jedoch als neues Feature erkannt. Aus diesem Grund wird keine Erweiterung der vorhanden Gruppe (Zeile 6-9) vorgenommen. Dies hat zur Folge, dass eine falsche Auswertung des Feature-Modells vollzogen wird. Es waren jedoch nur kleinere Ausbesserungen im Quellcode für VeAnalyzer notwendig.

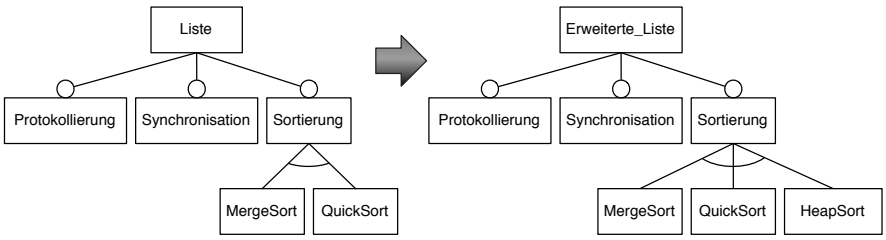


Abbildung 5.2: Probleme bei der Erweiterung von Features.

```
1  concept Liste{
2    feature Protokollierung;
3    feature Synchronisation;
4
5    mandatory feature Sortierung{
6      oneof{
7        feature QuickSort;
8        feature MergeSort;
9      }
10   }
11 }
```

```
12 concept Erweiterte_Liste : Liste{
13   feature  Sortierung.HeapSort;
14 }
```

Quelltext 5.1: Probleme bei der Erweiterung des Features `Sortierung`.

Eine größere Aufgabe stellte die Integration der Überführung von Aggregation in einen aussagenlogischen Ausdruck dar. Diesbezüglich enthielt FmConfig keine Überführung von verschiedenen Instanzen einer Produktlinie in einen gesamten aussagenlogischen Ausdruck der Multiproduktlinie. Die Integration der Überführung der verschiedenen Instanzen zu dem gesamten aussagenlogischen Ausdruck wurde im Rahmen dieser Arbeit in VeAnalyzer integriert. Zusammenfassend wird in Tabelle 5.2 auf der nächsten Seite noch einmal der Funktionsumfang von FmConfig und VeAnalyzer gegenübergestellt.

Mechanismus	FmConfig	VeAnalyzer
Vererbung		
exklusive Feature-Erweiterung	✓	✓
inklusive Feature-Erweiterung	X	✓
Aggregation	X	✓

Tabelle 5.2: Unterstütze Mechanismen zur Bildung eines aussagenlogischen Ausdrucks.

5.4 Konsistenzprüfungen für VELVET-Modelle

Die schon erläuterten Umsetzungen der verschiedenen Anforderungen bilden die Grundlage zur Integration von weiteren Konsistenzprüfungen. Im vorherigen Kapitel wurde die Verwendung der verschiedenen Analysen für Dead-Features, False-Optional-Features, Anzahl und Ausgabe von Varianten, Core-Features und atomaren Mengen bezogen auf Multiproduktlinien diskutiert. Des Weiteren erfolgte die Einführung von neuen Definitionen der False-Optional-Features und der Dead-Features bezogen auf Multiproduktlinien und die Vorstellung einer neuen Analyse, der Core-Instanzen. In diesem Abschnitt wird nun beschrieben, welche Analysen in VeAnalyzer integriert wurden.
Bei der Betrachtung der verschiedenen Analysen, die für Feature-Modelle von Produktlinien entwickelt wurden, folgte nun jeweils eine Diskussion über die Verwendung der Analysen auf verschiedenen Ebenen der Multiproduktlinien. In diesem Zusammenhang wurde die Verwendung der einzelnen Analysen, bezogen auf (1) die gesamte Multiproduktlinie, (2) die Vererbungshierarchie und (3) auf Instanzen, analysiert. Zunächst erfolgte eine Integration der jeweiligen Analysen für die gesamte Multiproduktlinie in VeAnalyzer, um diese als Grundlage für detailliertere Analysen verwenden zu können. Da der integrierte SAT-Solver MiniSat keine besonderen Funktionen zur Bestimmung von Dead-Features und Core-Features anbietet, wurden die vorgestellten Algorithmen des vorherigen Kapitels als Grundlage der Implementierung verwendet.

Dead-Features und False-Optional-Features

Im vorherigen Kapitel wurde ein Algorithmus vorgestellt, der zur Analyse der Vererbungshierarchie auf Dead-Features und False-Optional-Features verwendet werden kann (siehe Pseudocode 3 auf Seite 41). Dieser Algorithmus wurde in einer leicht abgewandelten Form in VeAnalyzer integriert. Dementsprechend werden Dead-Features und False-Optional-Features von VeAnalyzer als Fehler betrachtet, die der neuen Definition bezogen auf Multiproduktlinien entsprechen (siehe Abschnitt 4.3.1 auf Seite 39). Des Weiteren wurden die vorhandenen Basis-Algorithmen für Dead-Features und False-Optional-Features auf die jeweiligen Produktlinien der Instanzen angewandt. Demnach werden alle diskutierten Analysen bezüglich Dead-Features und False-Optional-Features des vorherigen Kapitels unterstützt.

Anzahl und Ausgabe der Varianten

Bezogen auf die Anzahl und Ausgabe der verschiedenen Varianten einer Multiproduktlinie wurde der implementierte Basis-Algorithmus ebenfalls auf die Vererbungshierarchie

der Multiproduktlinie angewandt. Eine Implementierung für die Anzahl der Varianten einer Instanz war im Rahmen dieser Masterarbeit jedoch nicht möglich, da hierzu der komplexe Algorithmus zur Entfernung von abstrakten Features nötig ist (siehe Kapitel 4 auf Seite 31).

Core-Features

Bei der Implementierung der Analyse von Core-Features wurde ebenfalls die schrittweise Analyse der Vererbungshierarchie in VeAnalyzer integriert. Des Weiteren unterstützt VeAnalyzer auch die separate Analyse der verschiedenen Instanzen einer Multiproduktlinie. Zusätzlich zu diesen einzelnen Analysen wurde auch die Möglichkeit in VeAnalyzer integriert, ein gegebenes Feature-Modell einer Multiproduktlinie auf Core-Instanzen zu untersuchen. In diesem Zusammenhang wurde der vorgestellte Basis-Algorithmus zur Analyse von Core-Features auf die Wurzel-Features der jeweiligen Instanzen angewandt.

Atomare Mengen

Wie bei den vorherigen Analysen wurde auch bei der Bestimmung von atomaren Mengen eine Implementierung für die Vererbungshierarchie in VeAnalyzer integriert. Des Weiteren werden auch die atomaren Mengen der Instanzen separat bestimmt. In diesem Fall wird der Basis-Algorithmus auf den Typ der jeweiligen Instanz angewandt.

Überblick zu den integrierten Analysen

Die zuvor betrachteten einzelnen Konsistenzprüfungen, die in VeAnalyzer integriert wurden, werden noch einmal in Tabelle 5.3 dargestellt. Bis auf die Anzahl und die Ausgabe von Varianten für Instanzen wurden alle vorgestellten Analysen umgesetzt.

Analyse	VeAnalyzer		
	Gesamt-Analyse	Vererbungshierarchie	Instanzen
Dead-Features	✓	✓	✓
False-Optional-Features	✓	✓	✓
Anzahl von Varianten	✓	✓	X
Ausgabe von Varianten	✓	✓	X
Core-Features	✓	✓	✓
Core-Instanzen			✓
Atomare Mengen	✓	✓	✓

Tabelle 5.3: Integrierte Konsistenzprüfungen in VeAnalyzer.

5.5 Zusammenfassung

In diesem Kapitel wurde das Werkzeug VeAnalyzer vorgestellt, dass auf der Infrastruktur des Kommandozeilen-Werkzeugs FmConfig basiert. In VeAnalyzer wurden fast alle vorgestellten Analysen des vorherigen Kapitels integriert. Aus diesem Grund unterstützt VeAnalyzer umfassende syntaktische und semantische Analysen. Daher kann es detaillierte Informationen über ein gegebenes Modell liefern und dem Modellierer Rückschlüsse über die Korrektheit des Modells geben.

6. Evaluierung

Im vorherigen Kapitel wurde die Implementierung, der entwickelten und diskutierten Analysen zur Sicherstellung der Konsistenz, für das Konfigurationswerkzeug VeAnalyzer beschrieben. Dieser Prototyp dient in diesem Kapitel als Grundlage für die Evaluierungen, die die Korrektheit der vorgestellten Analysen untermauern.
Im folgenden Abschnitt wird dargestellt, wie durch testgetriebene Entwicklung die Korrektheit der Implementierung sichergestellt werden konnte. Anschließend werden in einer Fallstudie verschiedene Modelle analysiert, die in VELVET von Rosenmüller et al. modelliert wurden. Hierbei wurde auch ein Vergleich der Ergebnisse mit bestehenden Werkzeugen zur Analyse von Produktlinien durchgeführt.

6.1 Testgetriebene Entwicklung

Die vorgestellte Implementierung des vorherigen Kapitels wurde durch testgetriebene Entwicklung realisiert. Hierbei wurden verschiedene kleinere Modelle erstellt, deren Evaluierung Rückschlüsse auf die Korrektheit der Implementierung zulassen. Der genaue Ablauf dieser testgetriebenen Entwicklung und die darin durchgeführten Evaluierungen werden im Folgenden genauer erläutert.

Ablauf der Evaluierung

Wie zuvor angedeutet, handelt es sich bei dieser Evaluierung um viele kleinere, automatisierte Tests, die zur Analyse der entwickelten Konzepte durchgeführt wurden. Die Vorgehensweise der verschiedenen Evaluierungen wird in Abbildung 6.1 auf der nächsten Seite dargestellt.
Wenn beispielsweise eine Analyse für Dead-Features in das bestehende Werkzeug integriert werden sollte, wurden zunächst verschiedene Modelle erstellt, die zum Teil auch Dead-Features enthalten (siehe Abbildung 6.1 auf der nächsten Seite). Diese Modelle wurden dann zu einer Menge von eventuell bestehenden Modellen hinzugefügt. Nach vorheriger Konzeptentwicklung, erfolgte anschließend die Implementierung der Analyse

für Dead-Features. Des Weiteren konnte eine Evaluierung aller vorhandenen Modelle auf der Basis der integrierten Analysen vorgenommen und ausgewertet werden. Diese Vorgehensweise hat den Vorteil, dass schon bestehende Analysen auf neue Modelle angewandt werden können. Des Weiteren werden Seiteneffekte, die durch eine Änderung der Implementierung bezüglich anderer Analysen auftreten, sofort erkannt. Dies ermöglicht es, auftretende Probleme frühzeitig zu beheben. Lieferten alle Tests der neuen und der bereits vorhandenen Analysen korrekte Ergebnisse, wurde die Erstellung von neuen Modellen und der Implementierung einer weiteren Analyse durchgeführt. Wenn jedoch fehlerhafte Ergebnisse aus den Evaluierungen resultierten, erfolgte eine Überarbeitung der aktuellen Implementierung der integrierten Analyse.

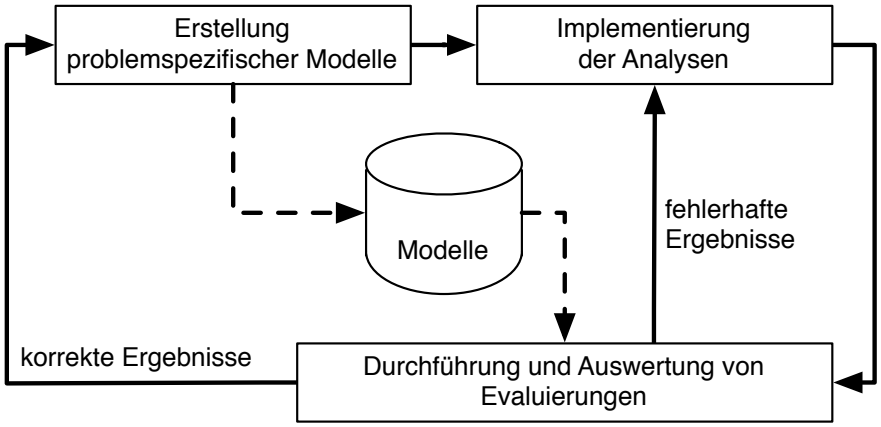


Abbildung 6.1: Ablauf der testgetriebenen Entwicklung.

Durch die Erzeugung verschiedener Modelle konnte die Evaluierung der syntaktischen und semantischen Analysen ermöglicht werden. Im Speziellen wurden für alle vorgestellten Implementierungen aus dem vorherigen Kapitel verschiedene Modelle erstellt und evaluiert. Eine kleine Auswahl der erstellten Modelle zur syntaktischen und semantischen Überprüfung werden im Kapitel B auf Seite 81 abgebildet.
Bei der Implementierung der einzelnen Analysen stellte sich dieses Vorgehen als sehr nützlich heraus, da des Öfteren verschiedene Seiteneffekte auftraten. Dies machte es notwendig, die Implementierung von verschiedenen Analysen noch einmal zu überdenken. Eine Entwicklung ohne dieses Vorgehen hätte eventuell nicht zu dem gewünschten Ergebnis geführt oder Seiteneffekte wären deutlich später aufgefallen und hätten die Entwicklung weiter verzögert.

Aussagekraft der Evaluierung

Bei den durchgeführten Testfällen handelte es sich um kleinere Modelle, die an die zu testende Analyse angepasst wurden. Die geringe Größe der Modelle gewährleistete eine gewisse Übersichtlichkeit die eine manuelle Überprüfbarkeit des korrekten Verhaltens der Implementierung ermöglichte. Außerdem konnten verschiedene Aspekte der Implementierung gezielt analysiert werden. Des Weiteren war es möglich durch die Wiederholung der Evaluierung auf Grundlage der bestehenden Modelle einen möglichst großen Testrahmen zu garantieren. Da es sich bei den Modellen jedoch um kleinere Tests handelt, ist es zusätzlich notwendig die Anwendbarkeit an einer großen Multiproduktlinie zu untersuchen. Eine größere Fallstudie wird im folgenden Abschnitt dargestellt.

6.2 Evaluierung bestehender Multiproduktlinien

In Abbildung 6.2 ist ein weiteres Beispiel einer Multiproduktlinie abgebildet, welches auch schon in einigen Arbeiten von Rosenmüller verwendet wurde [RS10]. Hierbei handelt es sich um ein Sensor-Netzwerk (`SensorNetwork`), das aus verschiedenen Sensor-Knoten besteht. Diese verschiedenen Knoten realisieren unterschiedliche Aufgaben. Dementsprechend existieren Knoten zum Lesen (`SensorNode`), für den Zugriff auf das Sensor-Netzwerk (`AccessNode`), für die Aggregation und die Speicherung der Daten (`DataNode`). Für die Speicherung und für komplexere Anfragen können die verschiedenen Knoten jeweils SQLite und BerkeleyDB verwenden. Die zugehörigen VELVET-Modelle wurden zum Teil mit VELVET veröffentlicht[1].

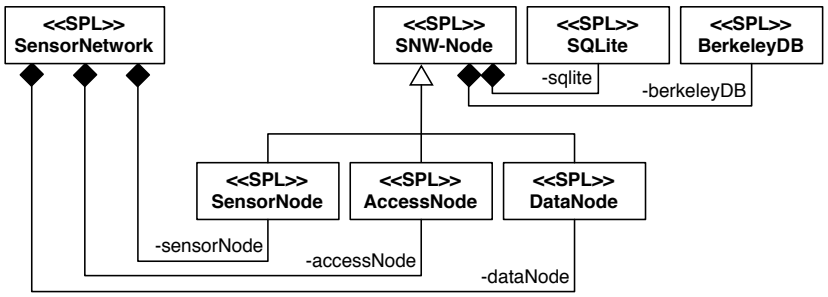


Abbildung 6.2: Kompositionsmodell für ein Sensor-Netzwerk, in dem verschiedene Netzwerk-Knoten existieren [RS10].

Ablauf der Evaluierung

Die Feature-Modelle des dargestellten Klassendiagramms werden nun verwendet, um die korrekte Funktionalität der integrierten Analysen zu überprüfen. Im Gegensatz zu den verschiedenen Tests des vorherigen Abschnitts handelt es sich hierbei um komplexere Modelle, die von Experten entwickelt wurden.
Die einzelnen Feature-Modelle der Produktlinien und Multiproduktlinien aus dem abgebildeten Klassendiagramm existieren bereits in VELVET. Da die Grammatik von VELVET in dieser Arbeit jedoch leicht angepasst wurde, mussten die einzelnen Modelle diesbezüglich korrigiert werden. In diesem Zusammenhang war es beispielsweise nötig verschiedene Gruppen, die parallel nebeneinander auftraten, in ein neues Eltern-Feature zu verschieben (siehe Kapitel 3 auf Seite 17). Die hierfür notwendigen Anpassungen konnten durch die verschiedenen syntaktischen Fehlermeldungen in VeAnalyzer schnell lokalisiert und vollzogen werden. Dieser Aspekt zeigt zugleich die besondere Bedeutung der integrierten, syntaktischen Analysen.
Nachdem alle Modelle angepasst wurden, konnte die Evaluierung mit weiteren Analysen fortgeführt werden. Die Ergebnisse der einzelnen Evaluierungen der Produktlinien `SQLite` und `BerkeleyDB` sowie der Multiproduktlinien wurden in Tabelle 6.1 auf der nächsten Seite dargestellt. Hierbei handelt es sich jeweils um die Analysen für die gesamten Multiproduktlinien (siehe Kapitel 4 auf Seite 31, Analyse (1)).

[1] http://wwwiti.cs.uni-magdeburg.de/itidb/research/MultiPLe/modeling.htm

Multiproduktlinie	Anzahl der Features	Dead-Features	False-Optional-Features	Core-Features	Core-Instanzen/Instanzen	Anzahl atomarer Mengen (mehrelementig)	Anzahl der Varianten
SQLite	86	0	0	8	0/0	1	1.51E23*
BerkeleyDB	14	0	0	1	0/0	0	2048
SnwNode	127	0	0	3	0/2	5	2.67E29*
SensorNode	131	0	0	4	0/2	5	1.52E30*
AccessNode	128	0	0	15	0/2	3	4.46E28*
DataNode	128	0	0	16	1/2	4	1.34E29*
SensorNetwork	393	0	0	23	2/9	12	9.03E88*

Tabelle 6.1: Ergebnisse der Analysen auf den einzelnen Bestandteilen des Sensor-Netzwerks. Die mit (*) dargestellten Werte wurden mittels `S.P.L.O.T.` ermittelt.

Da das dargestellte Modell von Experten modelliert wurde, konnte davon ausgegangen werden, dass wahrscheinlich keine Dead-Features oder False-Optional-Features in den Modellen und deren Kompositionen existieren. Diese Vermutung konnte durch die Evaluierung bestätigt werden. Des Weiteren wurde eine Analyse auf den gesamten Multiproduktlinien bezogen auf Core-Features durchgeführt. Wenn beispielsweise für die Multiproduktlinien `SensorNode`, `AccessNode` und `DataNode` eine erweiterte Analyse über die Vererbungshierarchie (siehe Kapitel 4 auf Seite 31, Analyse (2)) durchgeführt wird, kann der Modellierer weitere Rückschlüsse über die korrekte Modellierung erhalten. Aus diesem Grund wurde die Analyse über die Vererbungshierarchie für diese Multiproduktlinien noch einmal in Abbildung 6.3 auf der nächsten Seite grafisch verdeutlicht. Die verschiedenen Core-Features der Produktlinien beziehungsweise Multiproduktlinien werden in der Abbildung als Mengen dargestellt. In jedem der abgebildeten Knoten existiert `SnwNode` als Basiskonzept mit lediglich 3 Core-Features. Diese Analyse über die Vererbungshierarchie mittels VeAnalyzer zeigt eine Entwicklung der Core-Features. Im Fall von `SensorNode` wird zu der ursprünglichen Menge der Core-Features lediglich ein weiteres Core-Feature hinzugefügt. Bei dem Beispiel `DataNode` und `AccessNode` existieren deutlich mehr Core-Features in den gesamten Multiproduktlinien. Aufgrund dieser zusätzlichen Informationen durch VeAnalyzer bekommt der Modellierer einen Eindruck über die Korrektheit der Modellierung der Multiproduktlinien `DataNode`, `AccessNode` und `SensorNode`.
Eine weitere Analyse, die im Zuge dieser Arbeit eingeführt wurden, ist die Analyse der sogenannten Core-Instanzen. In Tabelle 6.1 wurde ebenfalls die Analyse der Core-Instanzen bezogen auf die gesamten Multiproduktlinien dargestellt. Durch die Ausgabe der Core-Instanzen bekommt der Modellierer einen zusätzlichen Einblick über die

modellierten Abhängigkeiten in der jeweiligen Multiproduktlinie. Betrachtet man beispielsweise die Multiproduktlinie `DataNode` und die Vererbungshierarchie zur Multiproduktlinie `SnwNode`, erhält der Modellierer wiederum besondere Rückschlüsse über das Modell. Durch die Definition weiterer Bedingungen in der Multiproduktlinie `DataNode` wird die Instanz `sqlite` zu einer immer verwendeten Instanz. Im Gegensatz dazu, handelt es sich bei `sqlite` in `SnwNode` um eine optionale Instanz.

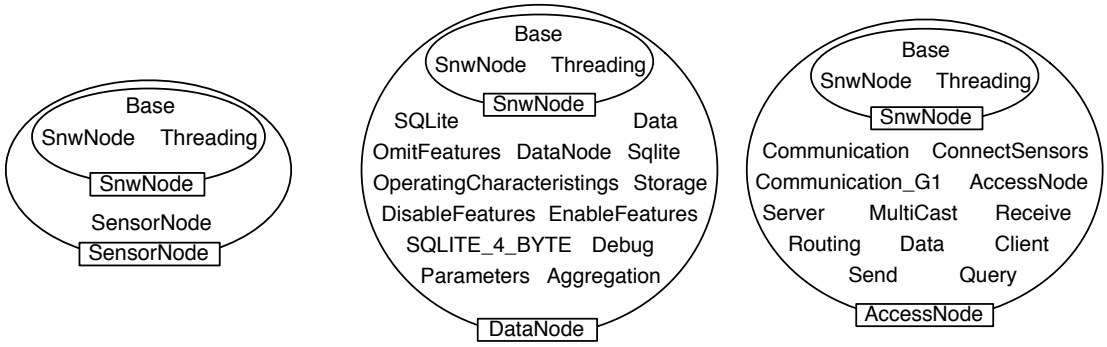


Abbildung 6.3: Analyse von Core-Features über die Vererbungshierarchie.

Die Betrachtung der Vererbungshierarchie bezogen auf atomare Mengen bietet ebenfalls zusätzliche Rückschlüsse auf die Korrektheit der Modellierung. Dies kann beispielsweise an der Multiproduktlinie `AccessNode` veranschaulicht werden (siehe Abbildung 6.4). In diesem Fall besteht das Basiskonzept aus 5 mehrelementigen atomaren Mengen (siehe Abbildung 6.4). Durch die Einführung von Bedingungen in `AccessNode` existieren in dieser Multiproduktlinie anschließend noch 3 mehrelementige atomare Mengen. Demzufolge bewirken die verschiedenen eingeführten Bedingungen eine Vereinigung sowie die Erweiterung von vorhandenen Mengen der Multiproduktlinie `SnwNode`. Einelementige Mengen wurden bei der gesamten Evaluierung nicht betrachtet.

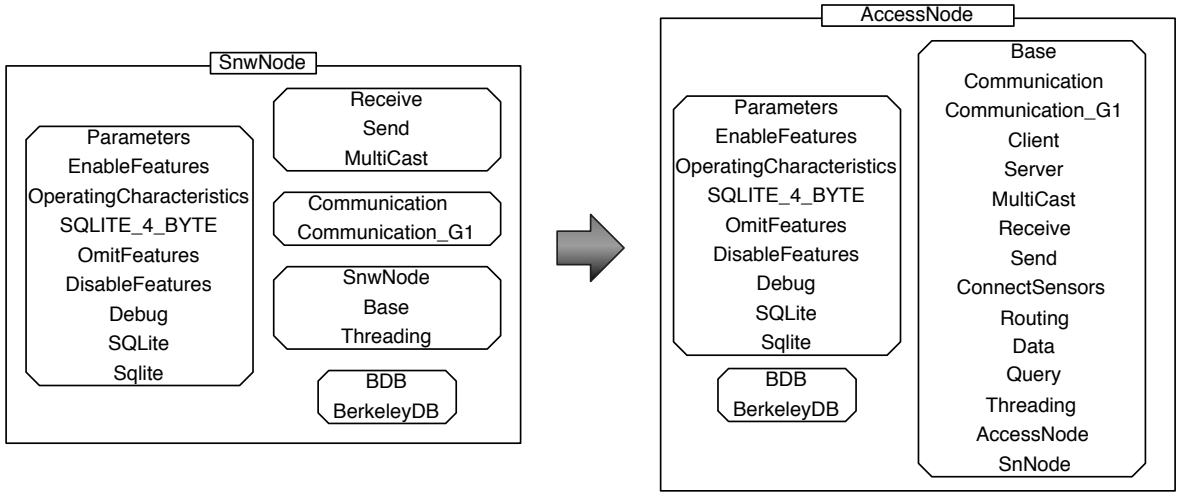


Abbildung 6.4: Analyse von atomaren Mengen über die Vererbungshierarchie.

Abschließend wurde in Tabelle 6.1 auf der vorherigen Seite die Anzahl der verschiedenen Varianten der Multiproduktlinien beziehungsweise Produktlinien dargestellt. Da die verschiedenen semantischen Analysen unter Verwendung eines SAT-Solvers in VeAnalyzer

integriert wurden, konnte diese Analyse nicht vollständig durchgeführt werden. Wie Benavides et al. darstellte [BSTRC06], sind SAT-Solver für solche Anfragen ungeeignet. Der Fokus dieser Arbeit liegt jedoch auf der Entwicklung neuer Konzepte sowie auf der Untersuchung von vorhandenen Konzepten in Verbindung mit Multiproduktlinien und nicht auf der Integration von weiteren Solvern in VeAnalyzer. Die aktuelle Implementierung ist jedoch völlig ausreichend, um konzeptuelle Tests für kleineren Multiproduktlinien beziehungsweise Produktlinien (siehe Tabelle 6.1 auf Seite 66, BerkeleyDB) durchzuführen.
Da es sich hierbei um größere Multiproduktlinien handelt und die Anzahl der Varianten daher nicht mittels VeAnalyzer ermittelt werden konnte, wurden die vorhandenen Modelle noch einmal durch ein weiteres Werkzeug evaluiert. Hierbei handelt es sich um das Werkzeug `S.P.L.O.T.`[2], welches ein eigenes Format zur Darstellung von Feature-Modellen für Produktlinien verwendet. Die vorhandenen Multiproduktlinien wurden daher manuell in ein Gesamtmodell überführt und mittels `S.P.L.O.T.` evaluiert. Auf der einen Seite konnte dadurch die Anzahl der Varianten der Multiproduktlinien ermittelt werden. Auf der anderen Seite wurden aber auch die Werte für Dead-Features, Core-Features und False-Optional-Features bestätigt.

Aussagekraft der Evaluierung

Bei der dargestellten Evaluierung handelte es sich um eine größere Fallstudie, durch die die Notwendigkeit der verschiedenen vorgestellten Analysen bestätigt werden konnte. Im Gegensatz zu den Analysen der testgetriebenen Entwicklung handelte es sich hierbei um größere Modelle die nur schwer überschaubar waren. Durch die gleichzeitige Evaluierung mit dem Werkzeug `S.P.L.O.T.` konnten die Ergebnisse jedoch bestätigt werden. Da die Multiproduktlinien von Experten entwickelt wurden, existierten in diesen Multiproduktlinien leider keine Dead-Features oder False-Optional-Features. Aus diesem Grund sollten in späteren Arbeiten weitere Fallstudien durchgeführt werden.

6.3 Zusammenfassung

In diesem Kapitel erfolgte die Evaluierung der entwickelten und diskutierten Analysen für die Sicherstellung der Konsistenz bei der Modellierung von Multiproduktlinien. In diesem Zusammenhang wurde dargestellt, wie durch eine testgetriebene Entwicklung die Korrektheit der Implementierung von syntaktischen und semantischen Analysen gewährleistet werden konnte. Diesbezüglich konnten verschiedene kleinere Modelle für eine spezielle Analyse erstellt und eine Evaluierung auf deren Basis durchgeführt werden. Anschließend wurde eine komplexere Fallstudie durchgeführt, welche die Notwendigkeit und die Sinnhaftigkeit solcher Analysen untermauert. Die verwendeten Fallstudien enthielten keine Dead-Features oder False-Optional-Features. Dies kann jedoch dadurch begründet werden, dass die Modelle durch Experten erstellt wurden und wenige weitere Zusatzbedingungen enthalten. Dennoch bieten die vorgestellten Analysen eine Hilfestellung, um ein besseres Verständnis über das entwickelte Modell einer Multiproduktlinie zu erhalten.

[2] http://www.splot-research.org/

7. Verwandte Arbeiten

In dieser Arbeit wurden verschiedene Analysen von Multiproduktlinien in VELVET untersucht und diskutiert. Rosenmüller et al. führte die Modellierungssprache VELVET ein, welche verschiedene Mechanismen für die Komposition von Feature-Modellen liefert [RSTS11]. Zusätzlich zu den in dieser Arbeit verwendeten Kompositionsmechanismen wurde von Rosenmüller et al. die Superimposition als weiterer Mechanismus zur Komposition vorgestellt. Die Einführung verschiedener Änderungen an der ursprünglichen Grammatik von VELVET in dieser Masterarbeit ermöglichte, dass nun beispielsweise eine eindeutige Erweiterung von Gruppen durch den Kompositionsmechanismus Vererbung unterstützt wird. Im Gegensatz zu den Ausführungen von Rosenmüller et al. wurde in dieser Arbeit explizit die Überführung der einzelnen Modelle in ein Feature-Modell beschrieben, das die gesamte Multiproduktlinie repräsentiert. Auf dieser Grundlage wurden weitere automatisierte Analysen zur Überprüfung der Konsistenz von Multiproduktlinien vorgestellt.

Modellierung von Multiproduktlinien

Die Idee, Klassen von objektorientierter Programmierung mit Produktlinien zu vergleichen, wurde von Czarnecki et al. veröffentlicht und bildet die Grundlage für die Erweiterung der Modellierung von abhängigen Produktlinien von Rosenmüller et al. [CHE05, RSuRK08]. Rosenmüller et al. verwendeten UML (Unified Modeling Language) Klassendiagramme zur Darstellung der Abhängigkeiten zwischen den einzelnen Produktlinien. Dies ermöglichte auch die grafische Darstellung der Abhängigkeiten von Modellen, die mittels VELVET durch Vererbung und Aggregation erstellt wurden.
In der Ausarbeitung von Czarnecki et al. zum schrittweisen Konfigurationsprozess wird zudem eine Modellreferenz beschrieben, die es ermöglicht in einem Feature-Modell ein anderes Feature-Modell zu referenzieren [CHE05]. Diese Referenzen beschreiben eine mögliche Wiederverwendung von modularen Feature-Modellen. Diese Art der Modellierung ist allerdings nicht so mächtig wie VELVET, da keine komplexen Kompositionsmechanismen wie Vererbung, Aggregation und Superimposition ausgedrückt werden können.

Bei der Sprache FAMILIAR, die von Archer et al. entwickelt wurde [ACLF11], handelt es sich um eine Skriptsprache, in der gleichzeitig verschiedene Produktlinien verwaltet werden können. Diese und auch folgende Arbeiten von Archer et al. stellen zudem verschiedene Operatoren zur Komposition und zur Dekomposition von Produktlinien vor [ACLF12]. So können beispielsweise die Operatoren `Insert`, `Merge` und `Aggregate` zur Komposition von Modellen verwendet werden. Der Operator `Aggregate` ermöglicht die Vereinigung von zwei Modellen unter einer neuen synthetischen Wurzel. Auf diesem gesamten Feature-Modell können dann weitere Bedingungen definiert werden. Der Operator `Insert` ermöglicht es, ein Feature-Modell in ein beliebiges Feature eines anderen Feature-Modells einzufügen. Dieser Operator ist sehr ähnlich zu der Aggregation in VELVET, der eine Definition von verschiedenen Instanzen in ein beliebiges Feature ermöglicht. Der Operator `Slice` unterstützt hingegen die Dekomposition von Modellen.

Analysen für Feature-Modelle

Einen Überblick über die verschiedenen Analysen, die bezogen auf Feature-Modelle für Produktlinien verwendet werden können, stellten Benavides et al. zusammen [BSRC10]. Zusätzlich wurden die verschiedenen Analysen jeweils an einem Beispiel beschrieben. Hierbei wurden deutlich mehr Analysen betrachtet als in dieser Masterarbeit. Beispielsweise wurde auch eine Analyse vorgestellt, die Redundanzen, also doppelt modellierte Aspekte, in einem Feature-Modell erkennt. Des Weiteren erfolgte die Vorstellung verschiedener Berechnungen, die weitere Rückschlüsse über die Variabilität oder auch die Gleichartigkeit des Modells zulassen.
In einer weiteren Arbeit von Archer et al. werden die verschiedenen Operatoren, die im Zusammenhang mit FAMILIAR eingeführt wurden, zur Trennung von Belangen verwendet [ACLF12]. Unter Anderem wird darin beschrieben, wie der Operator `Slice` zum Beispiel zur Entfernung von Dead-Features verwendet werden kann. Nach Archer et al. garantiert dieser Operator, dass ein Feature-Modell, das durch den Operator `Slice` erstellt wurde, keine Anomalien enthält. Des Weiteren werden redundante Darstellungen im Feature-Modell vermieden. Der Operator `Slice` kann also zur Korrektur von bestehenden Modellen verwendet werden. Im Gegensatz dazu zielt diese Arbeit darauf ab, dem Benutzer zusätzliche Information zu geben, die eine korrekte Modellierung gewährleisten. In diesem Fall wird davon ausgegangen, dass ein Dead-Feature unabsichtlich im Modell entstanden ist und dass es sich hierbei um einen Modellierungsfehler handelt. Demnach ist es nicht zwingend erwünscht, dass das Dead-Feature entfernt wird sondern dass eventuell darauf bezogene Bedingungen entfernt werden. Unter Anderem unterstützt FAMILIAR aber auch verschiedene Operatoren die Rückschlüsse über ein Feature-Modell erlauben, wie beispielsweise `counting` (Anzahl der gültigen Konfigurationen), `nbFeatures` (Anzahl der Features) oder auch `isValid` (Test auf Erfüllbarkeit)[1]. In dieser Masterarbeit wurde ebenfalls die Analyse über die Anzahl der vorhandenen Varianten betrachtet. In diesem Zusammenhang wurde das spezielle Wissen über die Komposition (Aggregation und Vererbung) genutzt, um detailliertere Informationen über die Anzahl der Varianten zu erhalten. Dies ermöglicht noch bessere Rückschlüsse auf die Korrektheit der modellierten Multiproduktlinie.

[1] https://nyx.unice.fr/projects/familiar/

Analysen für Produktlinien

Thüm et al. stellten ebenfalls einen Überblick von bestehenden Arbeiten zu verschiedenen Analysen zusammen, mit denen die resultierenden Produkte eines Feature-Modells auf bestimmte Eigenschaften analysiert werden können [TAK+12]. Eine dieser Eigenschaften ist beispielsweise die Sicherstellung, dass alle Produkte einer Produktlinie auch korrekt kompiliert werden können. In diesem Zusammenhang erfolgte eine Klassifizierung der verschiedenen existierenden Arbeiten nach der Strategie zur Bestimmung der jeweiligen Eigenschaften. Im Gegensatz zu dieser Masterarbeit wurden in der Arbeit von Thüm et al. keine Analysen betrachtet die lediglich die Korrektheit des Modells einer Produktlinie sicherstellen sollen. Der Blickpunkt ist eher auf die Betrachtung von Analysen gerichtet, die die Korrektheit einer vollständig entwickelten Produktlinie untersuchen und nicht nur deren Modellierung.

Implementierung von Multiproduktlinien

Wie die vorhandenen Kompositionsmechanismen der Aggregation und Vererbung auf der Ebene der Implementierung umgesetzt werden können, beschreibt Rosenmüller et al. [RSK10]. Bezüglich dessen wurden unterschiedliche Implementierungsmechanismen vorgestellt, die die Verwendung von verschiedenen Instanzen einer Produktlinie in einer anderen Produktlinie ermöglichen. Im Detail handelt es sich bei den betrachteten Implementierungsmechanismen um `Namespaces/Packages`, `Virtual Classes` und um `SPL Interfaces`. Da die verschiedenen Mechanismen sowohl Vor- als auch gewisse Nachteile bieten, wurde speziell diskutiert wann welcher Mechanismus verwendet werden sollte.

Verschiedene Solver zur Analyse von Feature-Modellen

FAMA ist ein Framework, das Benavides et al. entwickelte und das für eine automatisierte Analyse von Feature-Modellen genutzt werden kann [BSTRC07]. Dieses Framework unterstützt verschiedene Solver, die zur Durchführung bestimmter Analysen verwendet werden können. In einer Performance Messung der Solver für `CSP` (Constraint Satisfaction Problem), `SAT` und `BDD` (Binary Decision Diagrams) zeigte Benavides, dass die Wahl des optimalen Solvers von der Analyse abhängig ist [BSTRC06]. Wie auch durch diese Arbeit bestätigt werden konnten, ist beispielsweise ein SAT-Solver für die Berechnung aller Varianten einer Produktlinie ungeeignet. Benavides et al. stellte bei der Messung der Performance fest, das sich `BDD` (Binary Decision Diagrams) für diese Analyse eignen. Des Weiteren zeigte Henneberg wie erweiterte Feature-Modelle (Feature-Modelle, die auch Attribute unterstützen) durch die Verwendung eines Pseudo-Boolean SAT-Solvers analysiert werden können [Hen11]. Dadurch wird zusätzlich die Möglichkeit geboten, Bedingungen auf der Basis von Attributen zu beschreiben. Die aktuelle VELVET-Grammatik unterstützt ebenfalls die Verwendung von Attributen. Bedingungen, in denen Attribute enthalten sind, werden jedoch bisher ignoriert, da der integrierte SAT-Solver diese nicht verarbeiten kann.

8. Zusammenfassung

Die Wiederverwendung von Software und deren Bestandteile ist einer der wichtigsten Aspekte um Kosten und Zeit zur Entwicklung neuer Software einsparen zu können. Durch die Einführung von Produktlinien, zur Entwicklung ähnlicher Programme auf einer gemeinsamen Basis, wurde die Wiederverwendung von Quelltextfragmenten erheblich verbessert. Die Komposition aus verschiedenen Produktlinien wird als Multiproduktlinie bezeichnet und ermöglicht die Wiederverwendung von Produktlinien. Hierbei handelt es sich um ein relativ neues Forschungsgebiet, in dem bisher wenige Modellierungsansätze existieren. Eine Modellierungssprache, die verschiedene Möglichkeiten der Komposition von Produktlinien anbietet, ist VELVET.
Durch die verschiedenen Möglichkeiten, die beispielsweise durch VELVET zur Komposition von Produktlinien angeboten werden, entstehen sehr komplexe Modelle. Dies kann unter Anderem durch die Fallstudie, die in dieser Arbeit betrachtet wurde, geschlussfolgert werden. Hierbei handelte es sich um ein Modell mit 393 Features, aus denen etwa $9 * 10^{88}$ mögliche Varianten abgeleitet werden können. Diese Komplexität macht es einem Modellierer nahezu unmöglich die Übersicht über das Modell zu behalten. Dementsprechend kann ein Modellierer beispielsweise auch nicht mehr überblicken, ob es sich bei dem erstellten Modell, um ein konsistentes Modell handelt. Aus diesem Grund ist es von großer Bedeutung Werkzeuge zu entwickeln, die dem Entwickler Rückschlüsse über die Korrektheit des Modells geben.
Im Bereich der Modellierung von Feature-Modellen für Produktlinien gibt es verschiedene automatisierte Analysen, die zur Sicherstellung der Konsistenz einer Produktlinie verwendet werden können. In dieser Arbeit wurde untersucht, inwieweit diese Analysen auch auf Multiproduktlinien angewandt werden können. In diesem Zusammenhang wurden vier Phasen vorgeschlagen, die zur Analyse der Multiproduktlinien durchgeführt werden sollten: (1) Parsen der Modelle, (2) Syntaktische Prüfungen, (3) Test auf Erfüllbarkeit und (4) Konsistenzprüfungen. Zur Erfüllung der jeweiligen Phase wurden in dieser Arbeit unter Anderem verschiedene einzelne Prüfungen untersucht und vorgeschlagen. Der Beitrag, der durch diese Arbeit geleistet wurde, wird im Folgenden noch einmal zusammengefasst.

(1) Damit die Möglichkeit bestand eine eindeutige Modellierung von Erweiterungen bestehender Gruppen durch den Kompositionsmechanismus der Vererbung in VELVET zu unterstützen, wurde zunächst die Grammatik von VELVET angepasst. Dies bildete die Grundlage zur eindeutigen Interpretation der erstellten Modelle und für weitere automatisierte Analysen.

(2) Es wurden verschiedene syntaktische Überprüfungen vorgestellt, die für eine korrekte Interpretation der einzelnen Produktlinien einer Multiproduktlinie notwendig sind. Hierbei handelte es sich um kleinere Prüfungen, die beispielsweise die Vergabe des gleichen Namens für unterschiedliche Features unterbinden.

(3) Zur Überprüfung der gesamten Multiproduktlinien auf Korrektheit wurde explizit eine Überführung der einzelnen beteiligten Feature-Modelle einer Multiproduktlinie in ein Feature-Modell vorgestellt, das die gesamte Multiproduktlinie repräsentiert. Dies ermöglicht zunächst einen Test auf Erfüllbarkeit der gesamten Multiproduktlinie. Des Weiteren bildete die korrekte Interpretation der einzelnen Feature-Modelle die Grundlage für zusätzliche automatisierte Analysen.

(4) In einem weiteren Abschnitt wurde diskutiert, wie existierende Analysen für Feature-Modelle auf Multiproduktlinien angewandt werden können. In diesem Zusammenhang wurde eine neue Definition von Dead-Features und False-Optional-Features für Multiproduktlinien eingeführt. Des Weiteren erfolgte eine Einführung einer neuen Analyse für Instanzen einer Multiproduktlinie. Die Analyse der Core-Instanzen ermöglicht es, alle Instanzen zu ermitteln, die immer in der Multiproduktlinie verwendet werden.

(5) Anschließend wurde gezeigt, dass es unter Anderem sehr hilfreich sein kann, wenn Analysen nicht nur auf die gesamte Multiproduktlinie sondern auch auf die Vererbungshierarchie und auf die einzelnen Instanzen einer Multiproduktlinie angewandt werden können. Daraus ergeben sich detailliertere Informationen über die gesamte Multiproduktlinie, weshalb der Modellierer besser Rückschlüsse über die Korrektheit des Modells erlangt.

(6) Die vorgestellten Analysen bezüglich der syntaktischen und semantischen Korrektheit der Multiproduktlinien wurden prototypisch implementiert. Hierbei handelte es sich um eine testgetriebene Entwicklung, die die Korrektheit der neu implementierten Analysen sicherstellte. Die Evaluierung an einer größeren Fallstudie hat gezeigt, dass die Durchführung von automatisierten Analysen in Multiproduktlinien den Entwickler bei der Modellierung unterstützten können.

9. Zukünftige Arbeiten

In dieser Arbeit wurden Analysen von Multiproduktlinien für die verschiedenen Kompositionsmechanismen Vererbung und Aggregation der Modellierungssprache VELVET entwickelt und die Verwendung von bestehenden Analysen von Produktlinien in Multiproduktlinien untersucht. VELVET unterstützt jedoch nicht nur die Kompositionsmechanismen Vererbung und Aggregation, sondern auch die Superimposition. In weiteren Arbeiten sollten ebenfalls verschiedene Analysen bezogen auf Multiproduktlinien, die durch Superimposition erzeugt wurden, untersucht werden.
Bei der Untersuchung bezüglich der Verwendung von bestehenden Analysen von Produktlinien in Multiproduktlinien, handelte es sich bei den betrachteten Analysen lediglich um eine kleinere Auswahl. Beispielsweise können in weiteren Arbeiten auch Analysen betrachtet werden, die Redundanzen, also doppelt modellierte Aspekte, in Feature-Modellen erkennen. Des Weiteren können auch Berechnungen integriert werden, die eine Aussage über die Gleichheit oder die Variabilität der Modelle zulassen.
Im Rahmen dieser Masterarbeit konnten leider nicht alle vorgestellten Analysen vollständig in das Werkzeug VeAnalyzer integriert werden. Beispielsweise wurde aufgrund der beschränkten Zeit die Analyse bezüglich der Anzahl der Varianten einer Instanz nicht in VeAnalyzer integriert. Dieser Aspekt bietet weitere Möglichkeiten für zukünftige Arbeiten, in denen diese Analyse integriert und evaluiert werden kann.
Speziell bei der Evaluierung einer größeren Fallstudie wurden die Grenzen der aktuellen Implementierung deutlich. In dieser Arbeit lag der Fokus vor allem auf der Entwicklung von Konzepten, um die Konsistenz des Gesamtmodells zu gewährleisten. In weiteren Arbeiten kann die Implementierung durch zusätzliche Solver erweitert werden, die beispielsweise eine effizientere Berechnung der Anzahl der Varianten einer Multiproduktlinie ermöglichen. Beispielsweise kann ein Solver für `BDD` (Binary Decision Diagrams) integriert werden, der die effizientere Berechnung ermöglicht.
Die Unterstützung von zusätzlichen Solvern in VeAnalyzer hätte ebenfalls noch weitere Vorteile. Da in VELVET in jedem Feature zusätzlich Attribute angegeben werden können, ermöglicht die Integration von einem Pseudo-Booleschen SAT-Solver (PBSAT) in VeAnalyzer die Verwendung und Auswertung von Attributen in Bedingungen [Hen11].

Des Weiteren könnte auch ein Solver für `CSP` (Constraint Satisfaction Solver) integriert werden, der ebenfalls solche komplexeren Bedingungen auswerten kann.
Die Fallstudie, die in dieser Arbeit betrachtet wurde, liefert weitere Aufschlüsse über die untersuchten und entwickelten Konzepte. In diesem Zusammenhang wurde eine Multiproduktlinie mit 393 Features untersucht. In späteren Arbeiten könnten weitere Fallstudien betrachtet werden, die die Ergebnisse der durchgeführten Fallstudie untermauern.
Des Weiteren gibt es die Möglichkeit eine empirische Fallstudie durchzuführen. Hierbei könnte an verschiedenen Probanden die Aufgabe gestellt werden, eine Multiproduktlinie anhand einer textuellen Beschreibung mittels VELVET zu modellieren. Anschließend kann VeAnalyzer zur Analyse der erstellten Modelle verwendet werden. Dies kann weitere Aufschlüsse über die entwickelten Konzepte geben.
Im Bereich der grammatikalischen Ausdruckskraft von VELVET existieren verschiedene Möglichkeiten, mit denen sich zukünftige Arbeiten auseinander setzen könnten. In der Ausarbeitung zu VELVET von Rosenmüller et al. (siehe [RSTS11]) wurde beispielsweise eine Subtyp-Beziehung zwischen Produktlinien und deren Erweiterungen (Spezialisierung und Generalisierung) definiert. Dies ermöglicht es die verschiedenen Produktlinien in polymorpher Weise zu verwenden. Dementsprechend kann ein Subtyp an Stellen verwendet werden, an denen ein Supertyp gefordert ist. Aus diesem Grund könnte die Grammatik zur Beschreibung der Aggregation dahingehend erweitert werden, dass Zuweisungen von Instanzen unterstützt werden. Diese grammatikalische Veränderung hat ebenfalls weitreichende Folgen für die jeweiligen Analysen in VeAnalyzer. In diesem Zusammenhang muss sichergestellt werden, dass eine Instanzzuweisung gültig und die Zusammenstellung der Multiproduktlinie konsistent ist.

A. VELVET-Grammatik

Im Zusammenhang mit der Bearbeitung des Themas dieser Arbeit wurde die Grammatik der verwendeten Modellierungssprache VELVET verändert. Die aktuelle und die angepasste Version der Grammatik wird im Folgenden noch einmal zusammenfassend dargestellt (siehe Quelltext A.1, Quelltext A.2). Die Veränderungen werden dabei noch einmal farbig hinterlegt.

```
1  VelvetModel: Import* Concept ;
2  Import: "import" ImportPath ";" ;
3  ImportPath: ID ("." ID)* ;
4
5  //////// concept ////////
6  Concept: ["refines"] "concept" ConceptName [ConceptBases] ConceptBody ;
7  ConceptName: ID ;
8  ConceptBases: ":" ID ("," ID)* ;
9  ConceptBody: "{" Definition* "}" ;
10 Definition: (Feature | FeatureGroup | Constraint | Instance |
11                                                        Attribute) ;
12
13 //////// feature ////////
14 Feature: [MandatoryFlag] [AbstractFlag] "feature" FeatureName (
15                                                    FeatureDef | ";") ;
16 MandatoryFlag: "mandatory" ;
17 AbstractFlag: "abstract" ;
18 FeatureName: FeatureUnqualName ("." FeatureUnqualName)* ;
19 FeatureUnqualName: ID | "string" | "bool" | "float" ;
20 FeatureDef: "{" Definition* "}" ;
21
22 //////// feature group (oneof, someof) ////////
23 FeatureGroup: GroupType GroupChilds ;
24 GroupType: (SomeOfGroup | OneOfGroup) ;
25 SomeOfGroup: "someof" ;
26 OneOfGroup: "oneof" ;
27 GroupChilds: "{" Feature+ "}" ;
28
29 //////// constraint ////////
```

```
30 Constraint: "constraint" [ConstraintName "="] ConstraintDefinition ";" ;
31 ConstraintName: ID ;
32
33 ConstraintDefinition: (BinaryConstraint | UnaryConstraint |
34                                                    AttribConstraint) ;
35 BinaryConstraint: UnaryConstraint BinaryOp ConstraintDefinition ;
36 UnaryConstraint: (UnaryOp UnaryConstraint | ParensConstraint |
37                                                        FeatureName) ;
38 ParensConstraint: "(" ConstraintDefinition ")" ;
39 AttribConstraint: AttribInstance BinaryAttribOp AttribInstance;
40 AttribInstance: (INT | FLOAT | String | BoolValue | AttributeName) ;
41
42 UnaryOp: "!" ;
43 BinaryOp: ("&&" | "||" | "xor" | "->" | "<->") ;
44 BinaryAttribOp: ("==" | "!=" | ">" | "<" | "<=" | ">=") ;
45
46 //////// instances ////////
47 Instance: ConceptName InstanceName ";" ;
48 InstanceName: ID ;
49
50 //////// attributes ////////
51 Attribute: (IntAttribute | FloatAttribute | StringAttribute |
52                                                    BoolAttribute) ;
53 AttributeName : ID ("." ID)* ;
54
55 IntAttribute: "int" AttributeName ["=" INT] ";" ;
56 FloatAttribute: "float" AttributeName ["=" FLOAT] ";" ;
57 StringAttribute: "string" AttributeName ["=" String] ";" ;
58 BoolAttribute: "bool" AttributeName ["=" BoolValue] ";" ;
59
60 BoolValue: ("true" | "false") ;
```

Quelltext A.1: Vollständige, aktuelle VELVET-Grammatik.

```
1  VelvetModel: Import* Concept ;
2  Import: "import" ImportPath ";" ;
3  ImportPath: ID ("." ID)* ;
4
5  //////// concept ////////
6  Concept: ["refines"] "concept" ConceptName [ConceptBases] Body ;
7  ConceptName: ID ;
8  ConceptBases: ":" ID ("," ID)* ;
9
10
11 Name: ID ("." ID)* ;
12 Body: "{" [Definition] "}" ;
13 Definition:  (ConstInstAttr* [((FeatureGroup ConstInstAttr*) |
14                                    (ConstInstAttr | Feature)*)]) ;
15
16 //////// feature ////////
17 Feature: [MandatoryFlag] [AbstractFlag] "feature"  Name ( Body | ";") ;
18 MandatoryFlag: "mandatory" ;
19 AbstractFlag: "abstract" ;
20
21
```

```
22 //////// feature group (oneof, someof) ////////
23 FeatureGroup: GroupType GroupChilds ;
24 GroupType: (SomeOfGroup | OneOfGroup) ;
25 SomeOfGroup: "someof" ;
26 OneOfGroup: "oneof" ;
27 GroupChilds: "{" Feature+ "}" ;
28
29 //////// constraint ////////
30 Constraint: "constraint" [ConstraintName "="] ConstraintDefinition ";" ;
31 ConstraintName: ID ;
32
33 ConstraintDefinition: (BinaryConstraint | UnaryConstraint |
34                                                    AttribConstraint) ;
35 BinaryConstraint: UnaryConstraint BinaryOp ConstraintDefinition ;
36 UnaryConstraint: (UnaryOp UnaryConstraint | ParensConstraint |  Name) ;
37 ParensConstraint: "(" ConstraintDefinition ")" ;
38 AttribConstraint: AttribInstance BinaryAttribOp AttribInstance;
39 AttribInstance: (INT | FLOAT | String | BoolValue | AttributeName) ;
40
41 UnaryOp: "!" ;
42 BinaryOp: ("&&" | "||" | "xor" | "->" | "<->") ;
43 BinaryAttribOp: ("==" | "!=" | ">" | "<" | "<=" | ">=") ;
44
45 //////// instances ////////
46 Instance: ConceptName  Name ";" ;
47
48 //////// attributes ////////
49 Attribute: (IntAttribute | FloatAttribute | StringAttribute |
50                                                    BoolAttribute) ;
51 AttributeName: ID ("." ID)* ;
52
53 IntAttribute: "int" AttributeName ["=" INT] ";" ;
54 FloatAttribute: "float" AttributeName ["=" FLOAT] ";" ;
55 StringAttribute: "string" AttributeName ["=" String] ";" ;
56 BoolAttribute: "bool" AttributeName ["=" BoolValue] ";" ;
57
58 BoolValue: ("true" | "false") ;
```

Quelltext A.2: Vollständige, angepasste VELVET-Grammatik.

B. Testgetriebene Entwicklung

Im Folgenden werden verschiedene Modelle für die beschriebene testgetriebene Entwicklung der einzelnen Konzepte abgebildet (siehe Abschnitt 6.1 auf Seite 63). Hierbei handelt es sich nur um eine kleine Auswahl der verwendeten Modelle, um einen Eindruck über die Größe und die Komplexität dieser zu erhalten. Die Modelle besitzen eine Hierarchie, bei der das Modell `A` jeweils die oberste Produktlinie darstellt, von der keine andere Produktlinie abhängig ist.

B.1 Modelle für syntaktische Analysen

Zunächst wird eine Auswahl an Modellen angegeben auf deren Basis die syntaktischen Analysen durchgeführt wurden. In diesem Zusammenhang werden Modelle für die Analysen *mandatory in Gruppen* (siehe Quelltext B.1, Quelltext B.2), *Doppelte Vererbung* (siehe Quelltext B.3) und *Doppelte Namensvergabe* (siehe Quelltext B.4 bis Quelltext B.8 auf den Seiten 82–83) abgebildet.

```
1 concept A {
2     feature g{
3         oneof{
4             mandatory feature g1;
5             feature g2;
6         }
7     }
8 }
```

Quelltext B.1: Mandatory in Gruppen I.

```
1 concept A {
2     feature h{
3         someof{
4             mandatory feature h2;
5         }
6     }
7 }
```

Quelltext B.2: Mandatory in Gruppen II.

```
concept A : B,B{
    feature g;
}

concept B {
    feature f;
}
```

Quelltext B.3: Doppelte Vererbung.

```
concept A : B{
    feature f.g; // g schon vorhanden
}

concept B{
    feature f.g;
}
```

Quelltext B.4: Doppelte Namensvergabe I.

```
concept A : B{
    feature f.f1; // f1 schon vorhanden
}

concept B{
    feature f{
        oneof{
            feature f1;
            feature f2;
        }
    }
}
```

Quelltext B.5: Doppelte Namensvergabe II.

```
concept A{
    feature f;
    B f; // f schon vorhanden
}

concept B{
    feature g;
}
```

Quelltext B.6: Doppelte Namensvergabe III.

```
concept A{
    feature f;
    feature f; // f schon vorhanden
}
```

Quelltext B.7: Doppelte Namensvergabe IV.

```
concept A : B{
    feature f.f1; // f1 schon vorhanden
}

concept B{
    feature f{
        C f1;
    }
}

concept C{
    feature l;
}
```

Quelltext B.8: Doppelte Namensvergabe V.

B.2 Modelle für semantische Analysen

Nachdem im vorherigen Abschnitt einige Modelle aufgeführt wurden, die einige syntaktische Analysen untersuchten, werden nun einige Beispiele für Modelle der semantische Analyse abgebildet. In diesem Zusammenhang werden einige kleinere Modelle dargestellt, die zum Teil Dead-Features beinhalten (siehe Quelltext B.9 bis Quelltext B.13 auf den Seiten 83–84). Diese müssen von VeAnalyzer korrekt erkannt werden.

```
concept A {
    oneof {
        feature f1;
        feature f2; //Dead-Feature
    }
    constraint f1;
}
```

Quelltext B.9: Dead-Features I.

```
concept A : B {
    oneof{
        feature g1;
        feature g2;
    }
}

concept B{
    feature h{
        oneof {
            feature h1;
            feature h2; //Dead-Feature
        }
    }
    constraint h.h1;
}
```

Quelltext B.10: Dead-Features II.

```
concept A : B{
    feature f.f3; //Dead-Feature
}

concept B {
    feature f
    {
        oneof{
            feature f1;
            feature f2; //Dead-Feature
        }
        constraint f1;
    }
}
```

Quelltext B.11: Dead-Features III.

```
concept A {
    B b1;
}

concept B {
    feature f{
        oneof{
            feature f1;
            feature f2; //Dead-Feature
        }
    }
    constraint f.f1;
}
```

Quelltext B.12: Dead-Features IV.

```
concept A {
    C b1;
}

concept B : C {
    constraint f.f1;
}

concept C {
    feature f{
        oneof{
            feature f1;
            feature f2;
        }
    }
}
```

Quelltext B.13: Dead-Features V.

Literaturverzeichnis

[ACLF11] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. A Domain-Specific Language for Managing Feature Models. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pages 1333–1340. ACM, 2011. (zitiert auf Seite 70)

[ACLF12] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. Separation of Concerns in Feature Modeling: Support and Applications. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 1–12. ACM, 2012. (zitiert auf Seite 70)

[Bat05] Don Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 7–20. Springer, 2005. (zitiert auf Seite 2, 6, 7, and 8)

[BKPS04] Günter Böckle, Peter Knauber, Klaus Pohl, and Klaus Schmid. *Software-Produktlinien: Methoden, Einführung und Praxis*. Dpunkt Verlag, 2004. (zitiert auf Seite 6)

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–708, 2010. (zitiert auf Seite 2, 36, 37, 47, 50, and 70)

[BSTRC06] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. A First Step Towards a Framework for the Automated Analysis of Feature Models. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 39–47. IEEE Computer Science, 2006. (zitiert auf Seite 68 and 71)

[BSTRC07] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 129–134, 2007. (zitiert auf Seite 71)

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM/Addison-Wesley, 2000. (zitiert auf Seite 2 and 6)

[CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Staged Configuration through Specialization and Multi-Level Configuration of Feature Models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005. (zitiert auf Seite 6, 10, 11, 12, 13, 14, and 69)

[CN01] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, USA, 2001. (zitiert auf Seite 5)

[CW07] Krzysztof Czarnecki and Andrzej Wasowski. Feature Diagrams and Logics: There and Back Again. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 23–34. IEEE Computer Science, 2007. (zitiert auf Seite 6, 7, and 8)

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. (zitiert auf Seite 1)

[Hen11] Sebastian Henneberg. Next-Generation Feature Models with Pseudo-Boolean SAT Solvers. Bachelor thesis, Department of Informatics and Mathematics, University of Passau, 2011. (zitiert auf Seite 36, 37, 38, 47, 48, 50, 51, 71, and 75)

[JKB08] Mikoláš Janota, Joseph Kiniry, and Goetz Botterweck. Formal Methods in Software Product Lines: Concepts, Survey, and Guidelines. Technical Report Lero-TR-SPL-2008-02, Lero, University of Limerick, 2008. (zitiert auf Seite 1)

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990. (zitiert auf Seite 2, 5, 6, and 7)

[Kru06] Charles W. Krueger. New Methods in Software Product Line Development. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 95–102, 2006. (zitiert auf Seite 2 and 11)

[Mey00] Bertrand Meyer. *Object-Oriented Software Construction (Book/CD-ROM) (2nd Edition)*. Prentice Hall, 2nd edition, 2000. (zitiert auf Seite 1)

[MWC09] Marcilio Mendonca, Andrzej Wąsowski, and Krzysztof Czarnecki. SAT-Based Analysis of Feature Models is Easy. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 231–240. Carnegie Mellon University, 2009. (zitiert auf Seite 9)

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering : Foundations, Principles and Techniques*. Springer, 2005. (zitiert auf Seite 2 and 6)

[RS10] Marko Rosenmüller and Norbert Siegmund. Automating the Configuration of Multi Software Product Lines. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 123–130. University of Duisburg-Essen, 2010. (zitiert auf Seite 2, 11, 12, and 65)

[RSK10] Marko Rosenmüller, Norbert Siegmund, and Martin Kuhlemann. Improving Reuse of Component Families by Generating Component Hierarchies. In *Workshop on Feature-oriented Software Development (FOSD)*, pages 57–64. ACM Press, 2010. (zitiert auf Seite 71)

[RSTS11] Marko Rosenmüller, Norbert Siegmund, Thomas Thüm, and Gunter Saake. Multi-Dimensional Variability Modeling. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 11–22. ACM, 2011. (zitiert auf Seite 3, 17, 18, 21, 22, 29, 69, and 76)

[RSuRK08] Marko Rosenmüller, Norbert Siegmund, Syed Saif ur Rahman, and Christian Kästner. Modeling Dependent Software Product Lines. In *Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE)*, pages 13–18. Department of Informatics and Mathematics, University of Passau, 2008. (zitiert auf Seite 11, 12, 13, 14, 15, and 69)

[TAK+12] Thomas Thüm, Sven Apel, Christian Kästner, Martin Kuhlemann, Ina Schaefer, and Gunter Saake. Analysis Strategies for Software Product Lines. Technical Report FIN-004-2012, School of Computer Science, University of Magdeburg, Germany, 2012. (zitiert auf Seite 71)

[TBK09] Thomas Thüm, Don Batory, and Christian Kästner. Reasoning about Edits to Feature Models. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 254–264. IEEE Computer Science, 2009. (zitiert auf Seite 9)

[TKES11] Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. Abstract Features in Feature Modeling. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 191–200. IEEE Computer Science, 2011. (zitiert auf Seite 45 and 46)

[vdLSR07] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, 2007. (zitiert auf Seite 1 and 2)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 11. April 2012