Konsistenzerhaltung von Feature-Modellen durch externe Sichten

Bachelorarbeit von

Atilla Ateş

an der Fakultät für Informatik
Institut für Informationssicherheit und Verlässlichkeit

Erstgutachter:	Prof. Dr. Ralf H. Reussner
Zweitgutachter:	Prof. Dr.-Ing. Anne Koziolek
Betreuender Mitarbeiter:	M.Sc. Timur Sağlam
Zweiter betreuender Mitarbeiter:	M.Sc. Sofia Ananieva

20. September 2021 – 20. Januar 2022

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

This document is licensed under a Creative Commons Attribution 4.0 International License (CC BY 4.0): https://creativecommons.org/licenses/by/4.0/deed.en

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Änderungen entnommen wurde.

Schömberg, den 20.01.2022

. . . . . .

(Atilla Ateş)

Zusammenfassung

Bei der Produktlinienentwicklung werden *Software-Produktlinien* (SPLs) meistens Feature-orientiert strukturiert und organisiert. Um die gemeinsamen und variablen Merkmale der Produkte einer Produktlinie darzustellen, können *Feature-Modelle* verwendet werden. Ein Software-Werkzeug zum Erstellen und Editieren von Feature-Modellen ist *FeatureIDE*, welche die Zustände der Feature-Modelle als Dateien der *Extensible Markup Language* (XML) persistiert. Bei der Entwicklung von Software-Systemen existieren allerdings mehrere unterschiedliche Artefakte. Diese können sich Informationen mit den Feature-Modellen teilen. Um diese Artefakte und Modelle gemeinsam automatisch evolvieren zu können, werden Konsistenzerhaltungsansätze benötigt. Solche Ansätze sind jedoch nicht mit den persistierten XML-Dateien kompatibel.

In dieser Arbeit implementieren wir eine bidirektionale Modell-zu-Text-Transformation, welche die als XML-Dateien persistierten Zustände der FeatureIDE-Modelle in geeignete Modellrepräsentationen überführt, um daraus feingranulare Änderungssequenzen abzuleiten. Diese können zur *deltabasierten Konsistenzerhaltung* verwendet werden. Für die Modellrepräsentation verwenden wir ein bestehendes Metamodell für Variabilität. Zur Ableitung der Änderungssequenzen wird ein existierendes Konsistenzerhaltungsframework eingesetzt.

Wir validieren die Korrektheit der Transformation mithilfe von Round-Trip-Tests. Dabei zeigen wir, dass die in dieser Arbeit implementierte Transformation alle geteilten Informationen zwischen FeatureIDE und dem Variabilitäts-Metamodell korrekt transformiert. Zudem können mithilfe der in dieser Arbeit implementierten Transformation und mit dem verwendeten Konsistenzerhaltungsframework zu 94,44% korrekte feingranulare Änderungssequenzen aus den als XML-Datei persistierten Zuständen der FeatureIDE-Modelle abgeleitet werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Variabilitätsmodelle spielen in der Produktlinienentwicklung von Software-Systemen eine wichtige Rolle [26]. Bei der Produktlinienentwicklung werden sogenannte *Software-Produktlinien* (SPLs) erzeugt. Diese Produktlinien werden meistens Feature-orientiert strukturiert und organisiert [5]. Ein Feature ist dabei ein für den Nutzer sichtbares Merkmal oder Verhalten des Software-Systems. Eine Produktlinie enthält mehrere individuelle Software-Produkte, die sich darin unterscheiden, welche Features sie enthalten. Um die gemeinsamen und variablen Merkmale oder Features der Produkte einer Produktlinie darzustellen, werden Variabilitätsmodelle verwendet. Eines der häufigsten Variabilitätsmodelle ist das Feature-Modell. Mithilfe des Feature-Modells können die Features einer Produktlinie und deren Beziehungen dokumentiert werden [4]. Ein weitverbreitetes Werkzeug zum Erstellen und Editieren von Feature-Modellen, welches häufig in der Forschung eingesetzt wird, ist *FeatureIDE* [33].

Bei der Entwicklung eines Software-Systems werden allerdings nicht nur Feature-Modelle verwendet. Es kommen mehrere Modelle mit unterschiedlichen Metamodellen zum Einsatz. Die Systeme sind oft sehr komplex, da viele Abhängigkeiten zwischen den Modellen existieren. Die Modelle können sich Informationen teilen, wodurch es zu Inkonsistenzen kommen kann. Voneinander abhängige Modelle müssen demnach bei Änderungen konsistent gehalten werden. Diese Konsistenzerhaltung manuell durchzuführen ist sehr zeitaufwendig und fehleranfällig. Deshalb gibt es Systeme zur automatisierten Konsistenzerhaltung. Ein solches System ist *Vitruvius* [21], welches auf dem Ansatz von Atkinson, Stoll und Bostan [7] basiert und einen sogenannten *Virtual-Single-Underlying-Modell* (VSUM) verwendet. Dieses enthält alle Informationen und Modelle des Systems. Bei dem Vitruvius Ansatz werden Modelle mithilfe *deltabasierter Modellkonsistenzerhaltung*, also auf Basis von feingranularen Änderungssequenzen konsistent gehalten. Dabei ist eine feingranulare Änderungssequenz eine Menge von atomaren Änderungen, bei der die Reihenfolge relevant ist. Sollen also Modelle des Systems in einem externen Werkzeug editiert werden, wird zur Konsistenzerhaltung die feingranulare Änderungssequenz benötigt. Da aber viele externe Software-Werkzeuge wie FeatureIDE, keine feingranularen Änderungssequenzen zur Verfügung stellen, können diese Modelle nicht ohne Weiteres mit anderen Modellen im System konsistent gehalten werden. Um dieses Problem zu lösen, stellen Sağlam und Kühn [28] einen Ansatz vor. Dabei verwenden sie *externe Sichten*, welche die persistierten Zustände des externen Werkzeugs in geeignete Modellrepräsentationen transformiert, um daraus dann die feingranulare Änderungssequenz abzuleiten. Um demnach eine externe Sicht nach Sağlam und Kühn [28] zu konstruieren, müssen zwei Eigenschaften erfüllt werden. Die Eigenschaft, dass die persistieren Zustände einer expliziten Modellrepräsentation entsprechen (**E1**) und die Eigenschaft, dass die Änderungen als feingranulare Änderungssequenzen verfügbar sind (**E2**). Diesen Ansatz wollen wir nun für Feature-Modelle der FeatureIDE anwenden.

1.1. Ziele und Beiträge der Arbeit

Das Hauptziel dieser Arbeit ist die Konsistenzerhaltung von Feature-Modellen durch externe Sichten (**Z**). Um dieses Ziel zu erfüllen, implementieren wir eine *bidirektionale Model-zu-Text-Transformation* (M2T-Transformation). Damit transformieren wir Dateien im Format der *EXtensible-Markup-Language* (XML) der FeatureIDE-Modelle in Feature-Modelle des Varibilitätsmodells *VaVe* [3] und leiten mithilfe von Vitruvius feingranulare Änderungssequenzen aus den transformierten FeatureIDE-Modellen ab. Diese können dann in Vitruvius zur Konsistenzerhaltung verwendet werden. Wir verwenden das *VaVe-Metamodell*, welches der Beschreibung eines variablen Systems dient und Konstrukte für Feature Modellierung beinhaltet. Konzeptuell kann aber auch statt dem VaVe-Metamodell, ein anderes beliebiges explizites Metamodell für Feature-Modelle verwendet werden sowie ein anderes Software-Werkzeug an der Stelle von FeatureIDE, welches die Zustände seiner Feature-Modell als XML-Dateien persistiert.

Für die Implementierung der bidirektionalen M2T-Transformation wird das *Eclipse-Modeling-Framework* (EMF) [31] verwendet. Da FeatureIDE die Zustände der Modelle als XML-Dateien persistiert und kein explizites Metamodell enthält, wird eine entsprechende *XML-Schemadefinition* (XML-Schema) implementiert. Mithilfe dieser generieren wir automatisch durch EMF ein Zwischenmetamodell, welches wir in unserer Transformation verwenden. Nach der Implementierung der Transformation führen wir für die Evaluation *Round-Trip-Tests* durch. Dabei zeigen wir, dass die in dieser Arbeit implementierte Transformation alle geteilten Informationen zwischen FeatureIDE und dem VaVe-Metamodell, korrekt transformiert. Zudem evaluieren wir, wie gut sich mit unserer implementierten Transformation und mithilfe von Vitruvius feingranulare Änderungssequenzen ableiten lassen und vergleichen die *Evolutionsklassen* von Feature-Modellen nach Thüm, Batory und Kästner [32]. Wir zeigen, dass mithilfe der in dieser Arbeit implementierten Transformation und mit Vitruvius zu 94,44% korrekte feingranulare Änderungssequenzen aus den als XML-Datei persistierten Zuständen der FeatureIDE-Modelle abgeleitet werden können.

Zusammengefasst leistet die Arbeit zwei primäre Beiträge:

Bidirektionale M2T-Transformation (B1) Der erste Beitrag ist die Implementierung einer bidirektionalen M2T-Transformation, welche die als XML-Dateien persistierten Zustände der Feature-Modelle der FeatureIDE in VaVe-Modelle transformiert und umgekehrt. Damit wird die erste Eigenschaft (**E1**) erfüllt.

Initiale Untersuchung der abgeleiteten Änderungssequenzen (B2) Der zweite Beitrag ist eine initiale Untersuchung der aus den transformierten Feature-Modellen der FeatureIDE abgeleiteten Änderungssequenzen. Damit soll die Erfüllbarkeit der zweiten Eigenschaft (**E2**) untersucht werden.

Beide Beiträge unterliegen dem gemeinsamen Ziel der Konsistenzerhaltung von Feature-Modellen durch externe Sichten. Um dieses gemeinsame Ziel zu erreichen, nutzen wir den Ansatz der externen Sichten [28].

Mit diesen beiden Beiträgen möchten wir zwei Fragen beantworten: Die Frage, wie gut sich Änderungssequenzen aus Feature-Modellen der FeatureIDE mithilfe der in dieser Arbeit implementierten Transformation und mit Vitruvius ableiten lassen (**RQ1**) und

somit die Frage, wie gut sich FeatureIDE durch eine externe Sicht an Vitruvius anbinden lässt (**RQ2**).

Es existieren vier Vorteile, die sich durch diese Arbeit ergeben. Der erste Vorteil ist die Möglichkeit, FeatureIDE-Projekte über VaVe-Modellinstanzen mit anderen beliebigen Metamodellen konsistent halten zu können. Ein weiterer Vorteil ist die Erstellung einer Liste von Transformationsregeln, um zwischen XML-Dateien der FeatureIDE und VaVe-Modellinstanzen transformieren zu können. Der dritte Vorteil ist die Möglichkeit VaVe-Feature-Modelle, durch die bidirektionale M2T-Transformation, in FeatureIDE bearbeiten zu können. Außerdem ist es möglich FeatureIDE-Modelle, durch die bidirektionale M2T-Transformation, in VaVe-Modellinstanzen zu transformieren, welches der vierte Vorteil ist.

1.2. Aufbau der Ausarbeitung

Kapitel 1 bietet eine Einleitung in die Arbeit und Kapitel 2 legt die erforderlichen Grundlagen dar, auf welchen diese Arbeit basiert. In Kapitel 3 wird die Konzeption diskutiert und beschrieben. Dabei werden die Beiträge, welche diese Arbeit leistet, erläutert. Zudem wird diskutiert, wie die Transformation umgesetzt und Feature-Modelle der FeatureIDE in VaVe-Modelle transformiert werden können. Kapitel 4 gibt einen Einblick in die Implementierungsdetails der bidirektionalen M2T-Transformation sowie die während der Implementierung aufgetretenen Probleme. Das Evaluationskapitel 5 beschreibt und diskutiert die Evaluation. Es wird die Idee und der GQM-Plan (5.1.1) der Evaluation vorgestellt. Des Weiteren werden die verwendeten Modelle sowie die Umsetzung der Evaluation erklärt. Zudem werden die Ergebnisse der Evaluation erläutert und diskutiert. Kapitel 6 setzt sich mit den Erkenntnissen dieser Arbeit auseinander. Kapitel 7 schlägt vor, welche Themen in zukünftigen Arbeiten behandelt werden können. Die Ergebnisse der Arbeit werden in Kapitel 9 zusammengefasst und ein Fazit gegeben.

2. Grundlagen

Das folgende Kapitel zeigt die für diese Arbeit notwendigen Grundlagen auf. Es werden Grundlagen der modellgetriebenen Softwareentwicklung und der automatisierten Konsistenzerhaltung in der sichtenbasierten Entwicklung erläutert. Des Weiteren werden die in dieser Arbeit verwendeten Metamodelle, Modelle sowie Software-Werkzeuge erklärt.

2.1. Modellgetriebene Softwareentwicklung

Bei der modellgetriebenen Softwareentwicklung sind Modelle zusammen mit dem Quellcode die primären Artefakte. Sie sind damit ein unverzichtbarer Teil des Entwicklungsprozesses. Sie reduzieren die Komplexität und verbessern die Kommunikationseffizienz [30].

Zudem lassen sich auch Quelltexte aus ihnen generieren [29]. Die Transformationen von Modellen sind ein grundlegendes Element in der modellgetriebenen Softwareentwicklung. Sie sind deshalb wichtig, da man mithilfe der Transformationen zum einen voneinander abhängige Modelle konsistent halten kann und zum anderen aus einem Modell beispielsweise Quelltext generieren kann. Es gibt zwei grundlegende Arten von Modelltransformationen: Die *Modell-zu-Modell-Transformation* (M2M-Transformation) und die Generierung von Quelltext auch bekannt als *Modell-zu-Text-Transformation* (M2T-Transformation). Bei der Rückrichtung wird von einer *Text-zu-Modell-Transformation* (T2M-Transformation) gesprochen. Transformationen werden auf der Metamodellebene definiert. Ein Metamodell definiert die verwendbaren Modellelemente und deren Zusammenhänge. Unterscheiden sich die Metamodelle des Quell- und Zielmodells liegt eine *exogene Transformation* vor [25]. Sind die Metamodelle identisch wird sie als *endogene Transformation* [25] bezeichnet. Ersteres trifft auf die M2T-bzw. T2M-Transformation zu, jedoch kann es sich auch bei der M2M-Transformation um eine exogene Transformation handeln.

Die Neugenerierung des Zielmodells bei einer kleinen Veränderung des Quellmodells ist sehr aufwendig [18]. Deshalb gibt es verschiedene Ansätze zur *inkrementellen Transformation* [23]. Bei der inkrementellen Transformation werden, bei der Veränderung des Quellmodells, nur die veränderten Elemente transformiert [23].

2.2. Automatisierte Konsistenzerhaltung in der sichtenbasierten Entwicklung

Software-Systeme werden immer größer und komplexer. Sie enthalten zahlreiche Modelle, Dateien und Artefakte unterschiedlicher Domänen. Um diese Komplexität zu reduzieren wird die *sichtenbasierte Entwicklung* verwendet[7]. Bei der sichtenbasierten Entwicklung wird das System in mehrere meist domänenspezifische *Sichten* aufgeteilt. Eine Sicht enthält Informationen basierend auf Modellen oder anderen Sichten, es ist demnach selbst ein Modell. Das Metamodell einer Sicht wird als *Sichttyp* bezeichnet. Das System kann von verschiedenen Perspektiven betrachtet werden. Diese Perspektiven werden als *Sichtpunkte* bezeichnet. [9].

2.2.1. Der Vitruvius Ansatz

Da mehrere Sichten die selbe Information enthalten können, kann es hierbei zu Redundanzen kommen. Dies kann zu Inkonsistenzen im System führen, da sich die redundanten Informationen durch Veränderungen unterscheiden können. Um diese Inkonsistenzen zu vermeiden, gibt es verschiedene Ansätze für die *automatisierte Konsistenzerhaltung* [34]. *Vitruvius* ist ein solches automatisiertes System zur Konsistenzerhaltung [21]. Vitruvius basiert auf dem Ansatz von Atkinson, Stoll und Bostan [7] und hat einen sogenannten *Virtual-Single-Underlying-Model* (VSUM). Dieser enthält alle Informationen und Artefakte des Systems. Die verschiedenen Sichten werden dabei mithilfe der *deltabasierten Konsistenzerhaltung* [14] konsistent gehalten.

2.2.2. Sequenzen von atomaren Änderungen

Die Operationen zum Ändern eines Feature-Modells können nach Bürdek u. a. [10] in *atomare* und *komplexe Änderungsoperationen* unterteilt werden. Eine Änderungsoperation, die nicht weiter sinnvoll in weitere Operationen reduziert werden kann, wird als *atomare Änderungsoperation* (atomare Änderung) bezeichnet. *Komplexe Änderungsoperationen* (komplexe Änderungen) sind Kompositionen von atomaren oder anderen komplexen Änderungsoperationen [10]. Beispiele für eine atomare Änderung wären das Löschen eines Elements oder das Umbenennen eines Attributes. Feingranulare Änderungssequenz sind eine Menge von atomaren Änderungen deren Reihenfolge relevant ist. Eine Feingranulare Änderungssequenz stellt komplexe Änderungen durch mehrere atomare Änderungen dar. Eine Änderungssequenz mit der es möglich ist, von dem in Abbildung 2.1 gezeigten linken Zustand des Modells zum rechten Zustand zu gelangen wäre: Element erstellen, Attribut *Name* auf *Diesel* setzen, Referenz des *Diesel Elements* dem *Motor Element* hinzufügen.

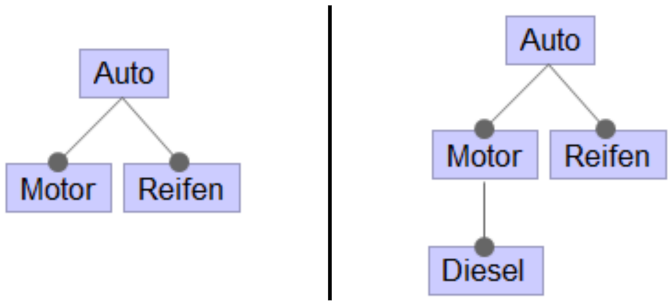


Abbildung 2.1.: Zu sehen sind zwei Zustände eines Modells. Links ist das unveränderte Modell zu sehen, rechts das veränderte.

2.2.3. Deltabasierte und zustandsbasierte Konsistenzerhaltung

Bei der *deltabasierten Konsistenzerhaltung* werden die Änderungsoperationen die auf ein Modell angewendet werden aufgezeichnet. Dadurch erhält man eine feingranulare Änderungssequenz von atomaren Änderungen [24]. Der Nachteil dabei ist die Notwendigkeit der feingranularen Änderungssequenz welcher bei vielen weit verbreiteten Software-Werkzeugen nicht immer vorhanden ist, da die Änderungen am Modell nicht aufgezeichnet werden.

Bei der *zustandsbasierten Konsistenzerhaltung* wird eine von vielen möglichen Änderungssequenzen anhand der zwei Zuständen des Modells berechnet [14]. Ein Zustand ist dabei das unveränderte Modell, der andere Zustand stellt das veränderte Modell dar. Der Nachteil bei der zustandsbasierten Konsistenzerhaltung ist allerdings, dass sich die berechnete Änderungssequenz von den eigentlich ausgeführten Änderungsoperationen unterscheiden kann. Das Umbenennen eines Elements ist nicht dieselbe Änderungssequenz wie das Löschen des Elements und das Erstellen eines neuen Elements mit dem neuen Namen [3].

2.2.4. Verschiedene Ableitungsstrategien

Eine Ableitungsstrategie erhält als Eingabe zwei Zustände eines Modells. Der erste Zustand ist das originale Modell ohne Veränderungen, in Abbildung 2.1 wäre dies das linke Modell. Der zweite Zustand ist der veränderte Zustand des gleichen Modells, dies wäre das rechte Modell in der Abbildung 2.1. Aus diesen beiden Zuständen leitet die Strategie feingranulare Änderungssequenz ab. Es gibt verschiedene Arten von Ableitungsstrategien zur Ableitung von feingranularen Änderungssequenzen die sich hauptsächlich darin unterscheiden, wie übereinstimmende Elemente in den Modellen gefunden werden (*Model-Matching*) [8]. Bei der *identitätsbasierten Vergleichsstrategie* werden übereinstimmende Elemente, zwischen den beiden Zuständen, mithilfe von eindeutigen Identifikatoren der Elemente gefunden. Die *änhlichkeitsbasierte Vergleichsstrategie* hingegen identifiziert die Elemente anhand ihrer Ähnlichkeit [20, 22].

2.3. Feature-Modelle

Ein Feature-Modell dokumentiert die Features einer Produktlinie und deren Beziehungen [4]. Dabei stellt ein Feature einen variablen Teil der Produktlinie dar [10]. Das Feature-Modell definiert, welche Kombinationen an Features gültig sind. Dies wird auch als Konfigurationsraum bezeichnet [10]. Ein Produkt basiert auf einer gültigen Kombination von Features. Eine mögliche graphische Darstellung eines Feature-Modells ist ein Baum, dessen Knoten die Features darstellen [4]. Ein Beispiel hierfür ist in Abbildung 2.7 zu sehen. Es gibt verschiedene *Eltern-Kind-Beziehungen* zwischen Features [4]:

- *Kind-Feature:* Kann nur ausgewählt werden, wenn das Eltern-Feature ausgewählt wird.
- *Optionale Features:* Sind optional, müssen nicht gewählt werden.
- *Obligatorische Features:* Müssen gewählt werden, wenn das Eltern-Feature ausgewählt wird.
- *Oder-Gruppe:* Mindestens ein Kind-Feature muss gewählt werden.
- *Alternativ-Gruppe:* Es muss genau ein Kind-Feature gewählt werden.
- *Abstrakte-Features:* Sind nur strukturell. Erhalten im Gegensatz zu den oben aufgezählten konkreten Features keine Implementierung.
- *Cross-Tree-Bedingungen:* Sind logische Ausdrücke von Features.

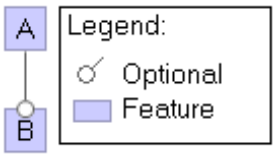


Abbildung 2.2.: Optionales Feature

Abbildung 2.3.: Obligatorisches Feature

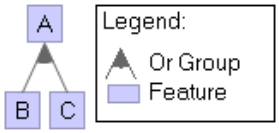


Abbildung 2.4.: Oder-Gruppe

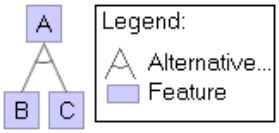


Abbildung 2.5.: Alternativ-Gruppe

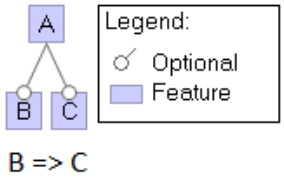


Abbildung 2.6.: Cross-Tree-Bedingung

Die möglichen Cross-Tree-Bedingungen sind *Implikation*, *Konjunktion*, *Disjunktion*, *Negation* und *Äquivalenz*. Mithilfe der Beziehungen und Cross-Tree-Bedingungen können, die als valide gewünschten Konfigurationen, definiert werden. In dem in Abbildung 2.7 gezeigten Feature-Modell kann ein Auto konfiguriert werden. Es muss dabei genau eine Art von Sitzen ausgewählt werden (Alternativ-Gruppe). Ein Radio ist immer enthalten (obligatorisch). Wählt man das optionale Sportpaket, müssen die Sportsitze gewählt werden (Cross-Tree-Bedingung). Eine Navigation ist optional, dabei muss mindestens eines der Kartenpakete Europa, Asien oder Amerika gewählt werden (Oder-Gruppe).

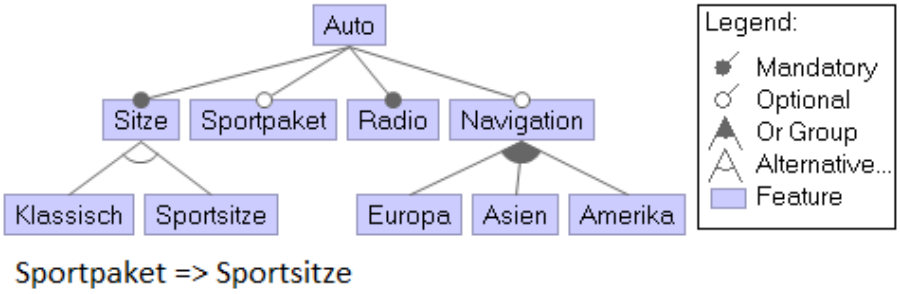


Abbildung 2.7.: Ein Feature-Modell welches die Kundenkonfiguration eines Autos beschreibt.

Feature-Modell	XML-Schlüsselwort
Optionales oder obligatorisches Feature ohne Kinder	feature
Optionales oder obligatorisches Feature mit Kinder	and
Alternativ-Gruppe	alt
Oder-Gruppe	or
Implikation	imp
Konjunktion	conj
Disjunktion	disj
Negation	not
Äquivalenz	eq

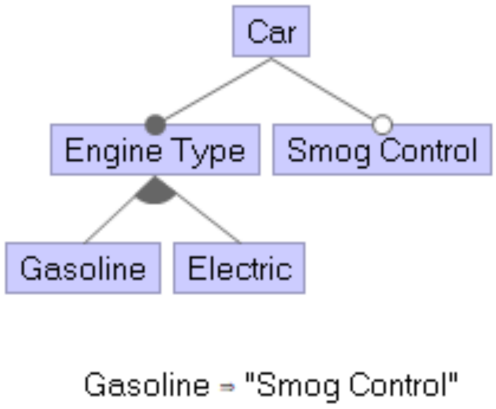
Tabelle 2.1.: Elemente in den XML-Dateien der FeatureIDE in Beziehung zu den Eltern-Kind-Beziehungen und Cross-Tree-Bedingungen im Feature-Modell.

2.4. Evolutionsklassen von Feature-Modellen

In der Literatur wird, je nachdem, welche Änderungsoperation oder Änderungsoperationen auf ein Feature-Modell angewendet werden, das daraus resultierende Feature-Modell klassifiziert. Diese Klassen werden als *Evolutionsklassen* bezeichnet. Es gibt einige verschiedene Evolutionsklassen für Feature-Modelle. Thüm, Batory und Kästner [32] unterscheiden zwischen *Refaktorisierung*, *Spezialisierung*, *Generalisierung* und *willkürlichem Editieren*. Bei der Refaktorisierung werden keine Elemente dem Konfigurationsraum hinzugefügt oder gelöscht. Werden existierende Elemente gelöscht und keine neuen hinzugefügt spricht man von Spezialisierung. Bei einer Generalisierung werden neue Elemente hinzugefügt und bereits existierende Elemente nicht gelöscht. In allen anderen Fällen wird von willkürlichem Editieren gesprochen. In dieser Arbeit werden für die Evaluierung der Änderungssequenzen die Evolutionsklassen von Thüm, Batory und Kästner [32] verwendet.

2.5. FeatureIDE

FeatureIDE ist ein auf *Eclipse* basiertes, weit verbreitetes Software-Werkzeug für Feature-orientierte Softwareentwicklung. Es bietet einen Editor zum Erstellen von Feature-Modellen an. FeatureIDE unterstützt die Entwickler bei den verschiedenen Implementierungstechniken der Feature-orientierten Softwareentwicklung [33]. Die Feature-Modelle werden in FeatureIDE als XML-Dateien persistiert. In Abbildung 2.8 ist links ein Feature-Modell und rechts die dazugehörige XML-Datei zu sehen. Die Abbildung 2.9 zeigt die für diese Arbeit relevanten Elemente und deren Beziehungen in den XML-Dateien der FeatureIDE-Modelle. Das Element *featureModel* enthält das *struct* und das *constraints* Element. Das *struct* Element enthält die Features und das *constraints* Element enthält die Cross-Tree-Bedingungen. Tabelle 2.1 zeigt, welche Elemente in der XML-Datei für welche Elemente im Feature-Modell stehen. Ist das Feature optional wird das Attribute *mandatory* auf *false* gesetzt. Ist das Feature obligatorisch wird *mandatory* auf *true* gesetzt.




```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<featureModel>
    <struct>
        <and mandatory="true" name="Car">
            <graphics key="collapsed" value="false"/>
            <or mandatory="true" name="Engine Type">
                <feature name="Gasoline"/>
                <feature name="Electric"/>
            </or>
            <feature name="Smog Control"/>
        </and>
    </struct>
    <constraints>
        <rule>
            <imp>
                <var>Gasoline</var>
                <var>Smog Control</var>
            </imp>
        </rule>
    </constraints>
</featureModel>
```

Abbildung 2.8.: Links ist ein Feature-Modell der FeatureIDE zu sehen, rechts die entsprechende XML-Datei.

Eltern-Kind-Beziehung	GroupType
optionales Feature	XORNONE
obligatorisches Feature	XOR
Oder-Gruppe	OR
Alternativ-Gruppe	XOR

Tabelle 2.2.: GroupTypes in Beziehung zu den Eltern-Kind-Beziehungen.

2.6. VaVe

VaVe ist ein Werkzeug für konsistentes sichtbasiertes Management von variablen Systemen [2, 3]. Es umfasst ein Metamodell zur Beschreibung von Feature Modellen. In Abbildung 2.10 sind die für diese Arbeit relevanten Klassen und Beziehungen des VaVe-Metamodells dargestellt. Die Klasse *System* enthält dabei alle Features und Cross-Tree-Bedingungen des Feature-Modells. Möchte man ein Feature dem *System* hinzufügen muss man zuerst ein *Feature-Objekt* erstellen. Je nach der Eltern-Kind-Beziehung wird dann ein *Tree-Constraint-Objekt* mit entsprechendem *Group-Type* erstellt. Tabelle 2.2 zeigt, für welche Eltern-Kind-Beziehung welcher *Group-Type* steht. Soll das erstellte Feature ein optionales oder obligatorisches Feature sein, wird die Referenz des Features dem *Tree-Constraint* Objekt hinzugefügt und die *Tree-Constraint* Instanz dem Eltern-Feature. Bei Alternativ- oder Oder-Gruppen werden die Referenzen der Kinder dem *Tree-Constraint* hinzugefügt und dieses dann dem Eltern-Feature.

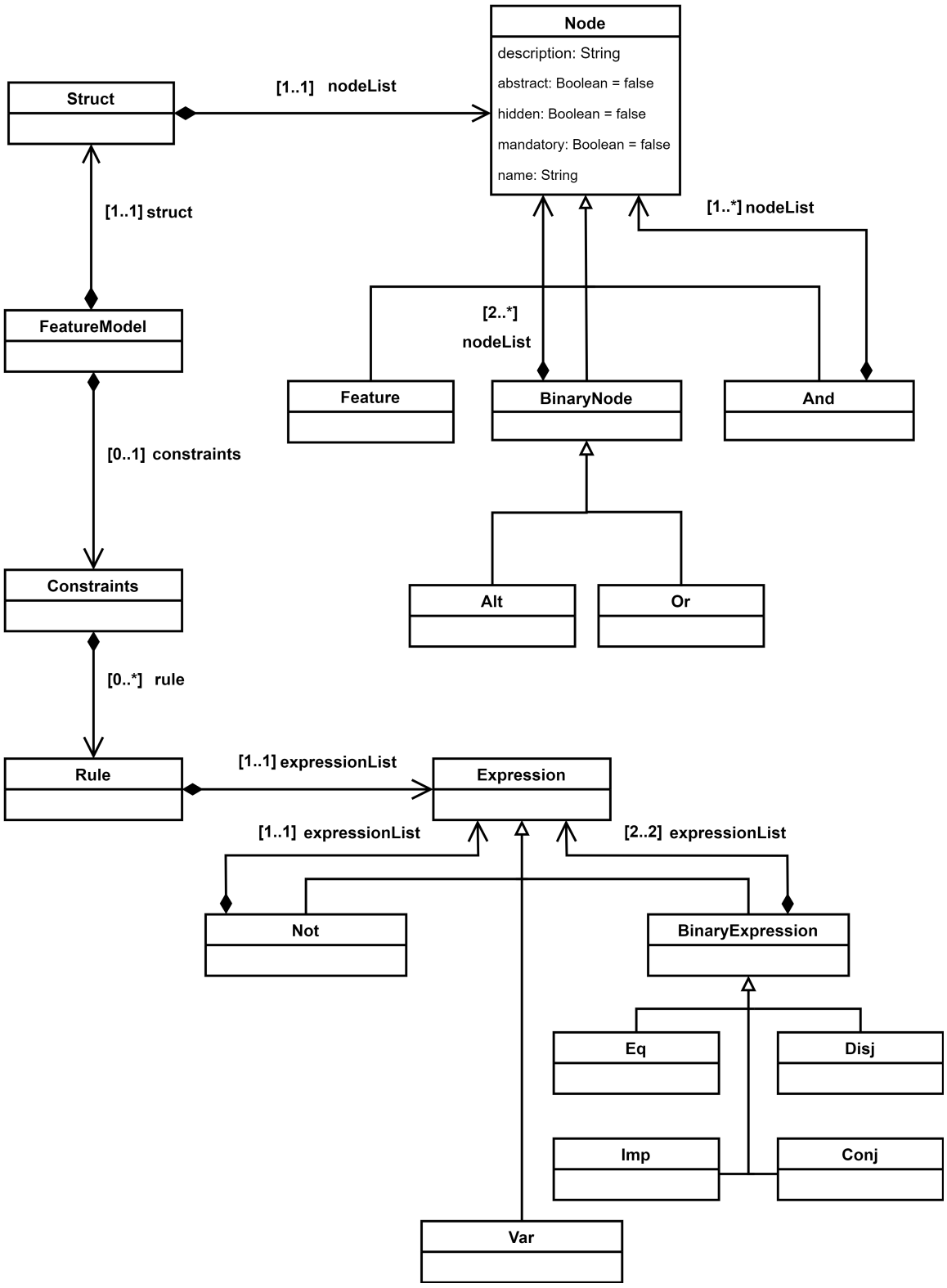


Abbildung 2.9.: Die Elemente und deren Beziehungen in XML-Dateien der FeatureIDE.

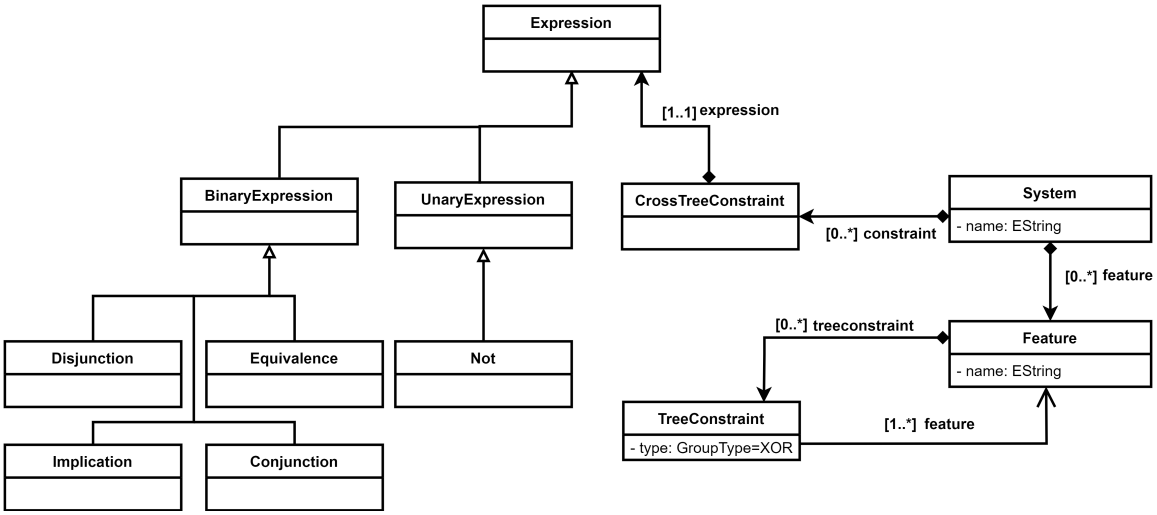


Abbildung 2.10.: Die für diese Arbeit relevanten Klassen und Beziehungen des Variabilitätsmodells VaVe.

2.7. Externe Sichten

Sağlam und Kühn [28] präsentieren einen Ansatz, wie es möglich ist Modelle eines externen Werkzeugs mithilfe einer externen Sicht konsistent halten zu können. Dabei definieren sie die Begriffe wie folgt:

Definition 2.1 (Modellierungssystem) *Beinhaltet alle Modelle des Systems.*

Definition 2.2 (Externes Werkzeug) *Das externe Werkzeug beinhaltet die industrielle Anwendung und seine Artefakte, welche vom Modellierungssystem abhängig sind.*

Definition 2.3 (Externe Sicht) *Beschreibt die Sicht auf eines oder mehrere Modelle im Modellierungssystem. Es beinhaltet auch eine Transformation für die persistierten Zustände des externen Werkzeugs.*

Wir verwenden für diese Begriffe ebenfalls die gleichen Definitionen. Da viele externe Werkzeuge keine Änderungssequenzen zur Verfügung stellen und die Modelle nicht der Modellrepräsentation des Modellierungssystems entsprechen, können die Modelle des externen Werkzeugs nicht mit den Modellen im Modellierungssystem konsistent gehalten werden. Die Notwendigkeit einer geeigneten Modellrepräsentation der persistierten Zustände des externen Werkzeugs bezeichnen sie als *erste Eigenschaft*. Die Notwendigkeit der feingranularen Änderungssequenzen bezeichnen sie als *zweite Eigenschaft*. Diese beiden Eigenschaften müssen für die Konsistenzerhaltung von persistierten Zuständen der externen Werkzeuge erfüllt werden [28]. Denn erst, wenn diese beiden Eigenschaften erfüllt sind kann eine externe Sicht konstruiert und diese zur Konsistenzerhaltung der persistierten Zustände verwendet werden. Um die erste Eigenschaft zu erfüllen wird ein Modell des externen Werkzeugs in eine Sicht transformiert, welcher der Modellrepräsentation des Modellierungssystems entspricht. Anschließenden werden dann die Änderungssequenzen

mithilfe einer Ableitungsstrategie aus den verschiedenen Zuständen der Sicht abgeleitet. Damit wird die zweite Eigenschaft erfüllt. Durch die abgeleitete Änderungssequenz können die Modelle deltabasiert konsistent gehalten werden.

2.8. Eclipse Modeling Framework

Das *Eclipse Modeling Framework* (EMF) [31] ist ein Framework zum Modellieren und Integrieren von Daten in Eclipse. Das Modell zum Repräsentieren von Metamodellen in EMF ist das *Ecore-Meta-Metamodell*. Mithilfe von EMF können aus Modellen, die als Java-Code, XML-Dokumente oder XML-Schemas definiert sind, Ecore-Modellinstanzen erstellt werden. Aus den Ecore-Modellinstanzen ist es wiederum möglich, automatisch Java-Code zu generieren. Zudem ist es möglich, einen Modelleditor als Eclipse-Plugin mithilfe der Ecore-Modellinstanz zu erstellen. In diesem können dann Modellinstanzen, des als Java-Code, XML-Dokument oder XML-Schema definierten Modells, erstellt werden [31].

3. Konzeption

In diesem Kapitel wird der Ansatz dieser Arbeit zur Konsistenzerhaltung von Feature-Modellen durch externe Sichten (**Z**) im Detail erörtert. Im Vordergrund stehen dabei die beiden Hauptbeiträge (**B1** und **B2**) die diese Arbeit leistet, um die Konsistenzerhaltung von Feature-Modellen zu ermöglichen. Des Weiteren wird die bidirektionale M2T-Transformation, welche im Rahmen des ersten Beitrags (**B1**) dieser Arbeit implementiert wird, diskutiert.

3.1. Konsistenzerhaltung von Feature-Modellen durch externe Sichten

Das Hauptziel dieser Arbeit ist die Konsistenzerhaltung von Feature-Modellen (**Z**). Zum Erstellen und Bearbeiten von Feature-Modellen verwenden wir in dieser Arbeit dabei FeatureIDE, welche seine Modelle als XML-Dateien persistiert (2.5). Wie bereits von Sağlam und Kühn [28] diskutiert, existieren bei der Konsistenzerhaltung von Modellen externer Software-Werkzeuge mittels deltabasierter Konsistenzerhaltungsansätzen [7, 21, 35], zwei Probleme. Das erste Problem ist die Notwendigkeit von feingranularen Änderungssequenzen bei Veränderung von Modellen. Denn wird ein Modell in einem externen Software-Werkzeug editiert, erhält man keine feingranulare Änderungssequenz, da viele weitverbreitete Software-Werkzeuge diese nicht zur Verfügung stellen. Das zweite Problem ist die Notwendigkeit einer geeigneten Modellrepräsentation, da externe Software-Werkzeuge keine expliziten Metamodelle verwenden. Um diese beiden Probleme zu lösen, verwenden Sağlam und Kühn [28] externe Sichten, welche die persistierten Zustände des externen Werkzeugs in geeignete Modellrepräsentationen transformieren, um daraus dann die feingranulare Änderungssequenz abzuleiten. Um also eine externe Sicht nach Sağlam und Kühn [28] zu konstruieren, müssen zwei Eigenschaften erfüllt werden. Die Eigenschaft, dass die persistieren Zustände einer expliziten Modellrepräsentation entsprechen (**E1**) und die Eigenschaft, dass die Änderungen als feingranulare Änderungssequenzen verfügbar sind (**E2**) [28].

Auch die in der FeatureIDE persistierten Zustände der Feature-Modelle, bringen diese beiden Probleme mit sich. Deshalb nutzen wir, zum Lösen dieser Probleme, den Ansatz der externen Sichten von Sağlam und Kühn [28]. Um die Konsistenzerhaltung der Feature-Modelle durch externe Sichten zu ermöglichen, leisten wir in dieser Arbeit zwei Beiträge (**B1** und **B2**).

3.1.1. Überführung in eine Modellrepräsentation

Der erste Beitrag (**B1**) dieser Arbeit ist die Implementierung einer bidirektionalen M2T-Transformation, welche die als XML-Dateien persistierten Zustände der FeatureIDE-Modelle in VaVe-Modelle transformiert und umgekehrt. Damit sollen die von der FeatureIDE persistierten Zustände in einer geeigneten Modellrepräsentation zur Verfügung stehen und so die erste Eigenschaft für die Konstruktion einer externen Sicht (**E1**) erfüllt werden. Wir verwenden das VaVe-Metamodell, da es ein explizites Metamodell für Feature-Modelle ist. In dieser Arbeit beschäftigen wir uns primär mit dem ersten Beitrag (**B1**).

Für die Implementierung der bidirektionalen M2T-Transformation stellen wir Transformationsregeln auf. Da FeatureIDE die Zustände der Feature-Modelle als XML-Dateien persistiert und kein explizites Metamodell enthält, definieren wir eine entsprechende XML-Schemadefinition (XML-Schema). Mithilfe dieser generieren wir automatisch in EMF ein Zwischenmetamodell welches wir in unserer Transformation verwenden.

Der in dieser Arbeit vorgestellte Ansatz funktioniert auch mit einem beliebigen anderen externen Software-Werkzeug, welches seine Modelle als XML-Dateien persistiert und einem anderen EMF konformen Metamodell. Dabei müssen jedoch die Transformationsregeln angepasst werden.

3.1.2. Bereitstellung von Änderungssequenzen

Der zweite Beitrag dieser Arbeit ist eine initiale Untersuchung der aus den transformierten Feature-Modellen der FeatureIDE abgeleiteten Änderungssequenzen (**B2**). Damit soll die Erfüllbarkeit der zweiten Eigenschaft (**E2**) untersucht werden.

Für die Ableitung der feingranularen Änderungssequenzen aus den persistierten Zuständen der FeatureIDE-Modelle verwenden wir in dieser Arbeit die Ableitungsstrategie von Vitruvius, welche die ähnlichkeitsbasierte Vergleichsstrategie verwendet. In dieser Arbeit bezeichnen wir diese Ableitungsstrategie als *ähnlichkeitsbasierte Ableitungsstrategie*.

Wir verwenden die ähnlichkeitsbasierte Ableitungsstrategie, da diese zum Ableiten von Änderungssequenzen aus zwei Zuständen keine eindeutigen Identifikatoren für die Elemente in den Modellen benötigt. Denn viele externe Software-Werkzeuge wie auch FeatureIDE verwenden keine eindeutigen Identifikatoren für die Elemente in den Modellen. Eine Ableitungsstrategie mit identitätsbasierter Vergleichsstrategie benötigt hingegen zum Ableiten von Änderungssequenzen eindeutige Identifikatoren, weshalb sie in dieser Arbeit nicht verwendet werden kann.

Wie bereits im Unterkapitel 3.1.1 erwähnt, soll im Rahmen des ersten Beitrags (**B1**) dieser Arbeit eine bidirektionale M2T-Transformation implementiert werden. Diese Transformation verwenden wir dafür, um aus Feature-Modellen der FeatureIDE feingranulare Änderungssequenzen abzuleiten, welche dann in Konsistenzerhaltungsframeworks wie Vitruvius verwendet werden können.

Um dies zu ermöglichen, transformieren wir die Feature-Modelle der FeatureIDE mithilfe der von uns implementierten bidirektionalen M2T-Transformation in VaVe Feature-Modelle. Dadurch wird die erste Eigenschaft (**E1**) erfüllt. Die Ableitungsstrategie von Vitruvius kann dann die VaVe-Feature-Modelle für die Ableitung der Änderungssequenzen verwenden. Abbildung 3.1 verdeutlicht unseren Ansatz genauer. Wir bedienen uns daran,

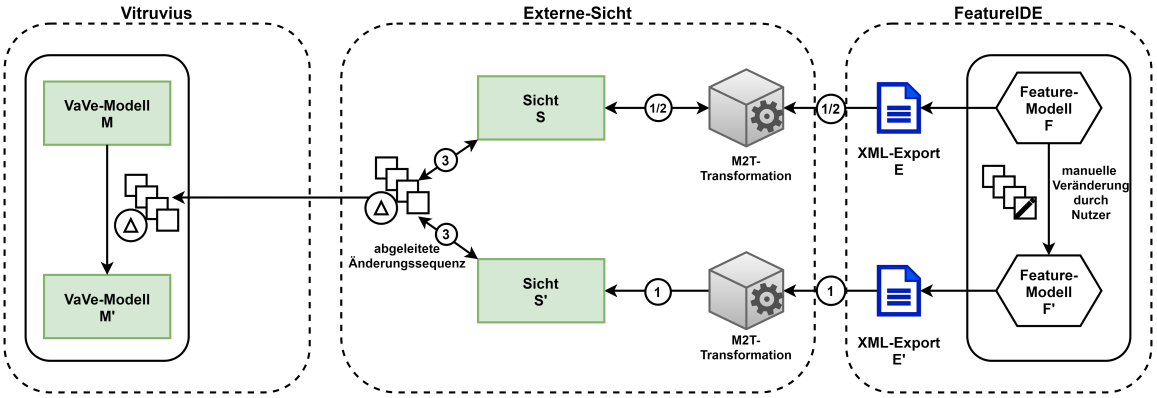


Abbildung 3.1.: Vereinfachte Darstellung des Ansatzes basierend auf dem Konzept von Sağlam und Kühn [28].

dass FeatureIDE seine Feature-Modelle als XML-Dateien persistiert und implementieren eine bidirektionale M2T-Transformation. Wird nun ein Feature-Modell F in FeatureIDE zu einem Feature-Modell F' verändert, können wir die XML-Dateien E und E', der beiden Feature-Modelle F und F', in VaVe-Modelle S und S' transformieren (1). Anschließend kann dann die feingranulare Änderungssequenz der VaVe-Modelle S und S' abgeleitet werden (3). Da die M2T-Transformation bidirektional ist, kann zudem ein VaVe-Modell S in eine XML-Datei E der FeatureIDE transformiert werden (2). Dadurch kann man dann das VaVe-Modell in der FeatureIDE zu einem Feature-Modell F' editieren und die XML-Datei E' des Feature-Modell F' in ein VaVe-Model S' transformieren (1). Mithilfe der Ableitungsstrategie wird dann die feingranulare Änderungssequenz der Modelle S und S' abgeleitet.

3.2. Transformation für Feature-Modelle

In diesem Abschnitt wird die bidirektionale M2T-Transformation, welche im Rahmen des ersten Beitrags (**B1**) dieser Arbeit implementiert wird erklärt.

3.2.1. Zwischenmetamodell für die Transformation

In Abbildung 3.2 wird das Konzept der bidirektionalen M2T-Transformation graphisch dargestellt. Bei der bidirektionalen M2T-Transformation machen wir uns zunutze, dass FeatureIDE seine Modelle als XML-Dateien persistiert und implementieren einmalig ein XML-Schema. Ein XML-Schema definiert, wie die Elemente und Attribute einer Klasse von XML-Dateien kombiniert werden können. Das von uns erstellte XML-Schema für die XML-Dateien der FeatureIDE-Modelle können wir in EMF einlesen und daraus ein Ecore-Metamodell generieren (1). Mithilfe des Ecore-Metamodells generieren wir dann einmalig ein Zwischenmetamodell (2) für die bidirektionale Transformation, um die beiden Seiten VaVe und FeatureIDE voneinander zu entkoppeln und so die Evolvierbarkeit der Transformation zu gewähren. Sind diese beiden Schritte erst einmal umgesetzt, können wir anhand des Zwischenmetamodells in Java automatisch XML-Dateien von Feature-Modellen der FeatureIDE einlesen und schreiben (3), da EMF dies out-of-the-box ermöglicht. Lesen wir die XML-Datei der FeatureIDE-Modelle ein, bekommen wir dann die eingelesene XML-Datei als Modellinstanz des Zwischenmetamodells.

3.2.2. Zwischentransformation

Als Zwischenschritt der bidirektionalen M2T-Transformation wird die Modellinstanz des Zwischenmetamodells, die wir durch das Einlesen der XML-Datei des FeatureIDE-Modells erhalten, durch eine M2M-Transformation (4) in eine VaVe-Modellinstanz transformiert. Wollen wir ein VaVe-Feature-Modell in ein Feature-Modell der FeatureIDE überführen, transformieren wir zuerst mit der M2M-Transformation die VaVe-Modellinstanz in eine Modellinstanz des Zwischenmetamodells (4) und erhalten dann die entsprechende XML-Datei des Feature-Modells der FeatureIDE. Die beiden Richtungen der bidirektionalen Zwischentransformation werden jeweils separat in Java beziehungsweise XTend implementiert. Für die Implementierung werden die von EMF in Java generierten Klassen und Interfaces des Zwischenmetamodells und VaVe verwendet.

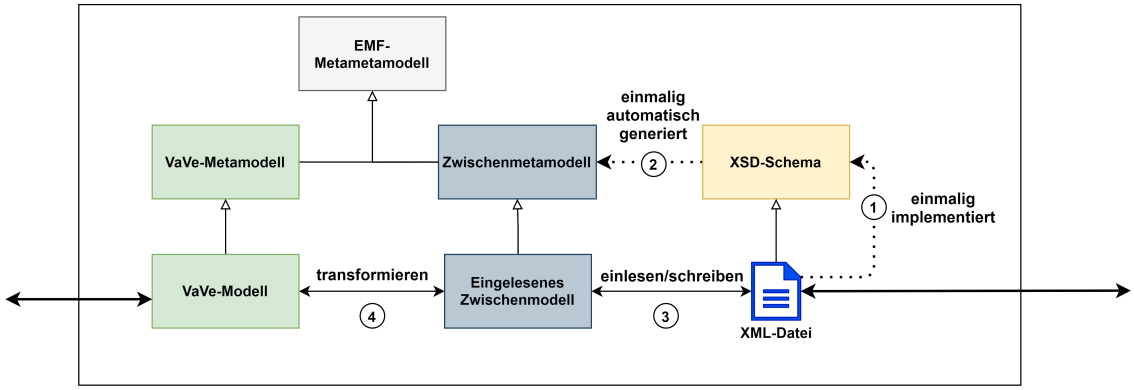


Abbildung 3.2.: Die bidirektionale M2T-Transformation um zwischen XML-Dateien der Feature-Modelle der FeatureIDE und den VaVe-Feature-Modellen transformieren zu können.

3.2.3. Transformationsregeln

Für die bidirektionale M2T-Transformation gelten die in Tabelle 3.1 aufgestellten Transformationsregeln. In FeatureIDE sind die Features im *struct* Element enthalten und die einzelnen Cross-Tree-Bedingungen mit den logischen Ausdrücken in jeweils einem *rule* Element. Die *rule* Elemente sind im *constraints* Element enthalten. Das *Struct* und *constraints* Element wiederum ist im *featureModel* Element enthalten. Abbildung 2.9 im Grundlagenkapitel zeigt die Struktur und Beziehungen der Elemente in den FeatureIDE XML-Dateien noch einmal graphisch.

Im VaVe-Metamodell sind sowohl die Features als auch die Cross-Tree-Bedingungen in der Klasse *System* enthalten. Sie werden in der Liste *feature* beziehungsweise *constraints* verwaltet. Für jede Cross-Tree-Bedingung wird jeweils ein *CrossTreeConstraint* Element erstellt und der *constraints* Liste hinzugefügt. Für die Features wird jeweils ein *feature* Element erstellt und je nachdem, welche Eltern-Kind-Beziehung sie haben, ein *TreeConstraint* mit entsprechendem *GroupType*. Abbildung 2.10 im Grundlagenkapitel zeigt die Klassen des VaVe-Metamodells und deren Beziehungen.

In FeatureIDE gibt es die Möglichkeit, *Abstract*- oder *Hidden*-Features zu erstellen. Ein *Abstract*-Feature wird anders als ein normales Feature nicht implementiert und *Hidden*-Features werden im Konfigurationseditor der FeatureIDE nicht angezeigt. Beide haben jedoch keine Auswirkung auf den Konfigurationsraum des Feature-Modells. Im VaVe-Metamodell können diese Art von Features nicht erstellt werden, da sie aber den Konfigurationsraum des Feature-Modells nicht verändern, werden sie bei der Transformation als normale Features transformiert. FeatureIDE persistiert den Zustand der Benutzeroberfläche sowie die des Feature-Modells in der selben XML-Datei. Beim Transformieren wird der Zustand der Benutzeroberfläche nicht transformiert. Zudem werden auch die Dokumentationen der Features nicht transformiert, weil das VaVe-Metamodell diese nicht unterstützt.

Da das VaVe-Metamodell ausdrucksmächtiger als FeatureIDE ist, werden bei der Transformation von VaVe-Modellen in Feature-Modelle nur die Elemente transformiert, welche auch von FeatureIDE ausgedrückt werden können.

FeatureIDE	VaVe
featureModel	System
struct	System:Feature
alt (mandatory/optional)	Feature:TreeConstraint:GroupType:XOR/XORNONE
or (mandatory/optional)	Feature:TreeConstraint:GroupType:XOR/XORNONE
and (mandatory/optional)	Feature:TreeConstraint:GroupType:XOR/XORNONE
feature (mandatory/optional)	Feature:TreeConstraint:GroupType:XOR/XORNONE
constraints	System:Constraint
rule	CrossTreeConstraint
imp	Implication
conj	Conjunction
disj	Disjunction
eq	Equivalence
not	Not
var	Variable

Tabelle 3.1.: Die Transformationsregeln für die bidirektionale M2T-Transformation. Die Regeln gelten in beide Richtungen.

4. Implementierung

Dieses Kapitel thematisiert die Implementierungsdetails der bidirektionalen M2T-Transformation sowie Probleme, die während der Implementierung aufgetreten sind.

4.1. Implementierungsdetails der Transformation

Die bidirektionale M2T-Transformation wurde in Java und Xtend implementiert. Wie in Abschnitt 3.2 erwähnt, wird für die Transformation ein XML-Schema für die XML-Dateien der FeatureIDE-Modelle benötigt. XML-Schemas können automatisch generiert werden, indem man eine oder mehrere XML-Dateien in einen entsprechenden Reverse-Engineering-Werkzeug importiert. Dieser extrahiert dann die Gemeinsamkeiten der XML-Dateien und generiert daraus automatisch ein XML-Schema. Es gibt einige solcher Reverse-Engineering-Werkzeuge, jedoch benötigten wir einen der das Extrahieren von Gemeinsamkeiten aus mehr als nur einer XML-Datei erlaubt. Ein einziges Modell, das alle möglichen Elemente gleichzeitig enthält, stand nicht zur Verfügung.

In unserem Fall kam das Reverse-Engineering-Werkzeug XMLSpy [37] zum Einsatz. Das generierte XML-Schema war jedoch nicht vollständig, wodurch viele XML-Dateien der FeatureIDE-Modelle beim Validieren mit dem XML-Schema fehlschlugen. Deshalb musste das XML-Schema manuell implementiert werden. Dieses ist im Anhang (A.1) zu finden.

Nach der Implementierung des XML-Schemas wurde dieses in EMF eingelesen und ein entsprechendes Ecore-Metamodell erstellt. Aus diesem Ecore-Metamodell wurde dann das Zwischenmetamodell *FeatureIDEXSD* in Java automatisch generiert. Danach wurde die bidirektionale M2T-Transformation in Java und XTend implementiert. XTend bietet den Vorteil *Dispatch-Methoden* implementieren zu können. Das Klassendiagramm des Zwischenmetamodells ist in Abbildung 4.1 zu sehen.

Das XML-Schema wurde so definiert, dass beim Generieren des Zwischenmetamodells, mithilfe von EMF, Vererbungshierachien entstehen. um die Implementierung der Transformation zu erleichtern. Zudem kann das XML-Schema Erweiterte (Extended)-Feature-Modelle der FeatureIDE darstellen. Diese sind in dieser Arbeit nicht relevant, können aber in zukünftigen Arbeiten relevant sein, in denen das in dieser Arbeit definierte XML-Schema verwendet werden kann. Außerdem wurde das XML-Schema so definiert, dass die Transformation strukturerhaltend implementiert werden konnte.

Wir haben unsere Transformation mit Java Version 11, Xtend Version 2.26.0 sowie EMF Version 2.27.0 für FeatureIDE Version 3.8.0 erstellt.

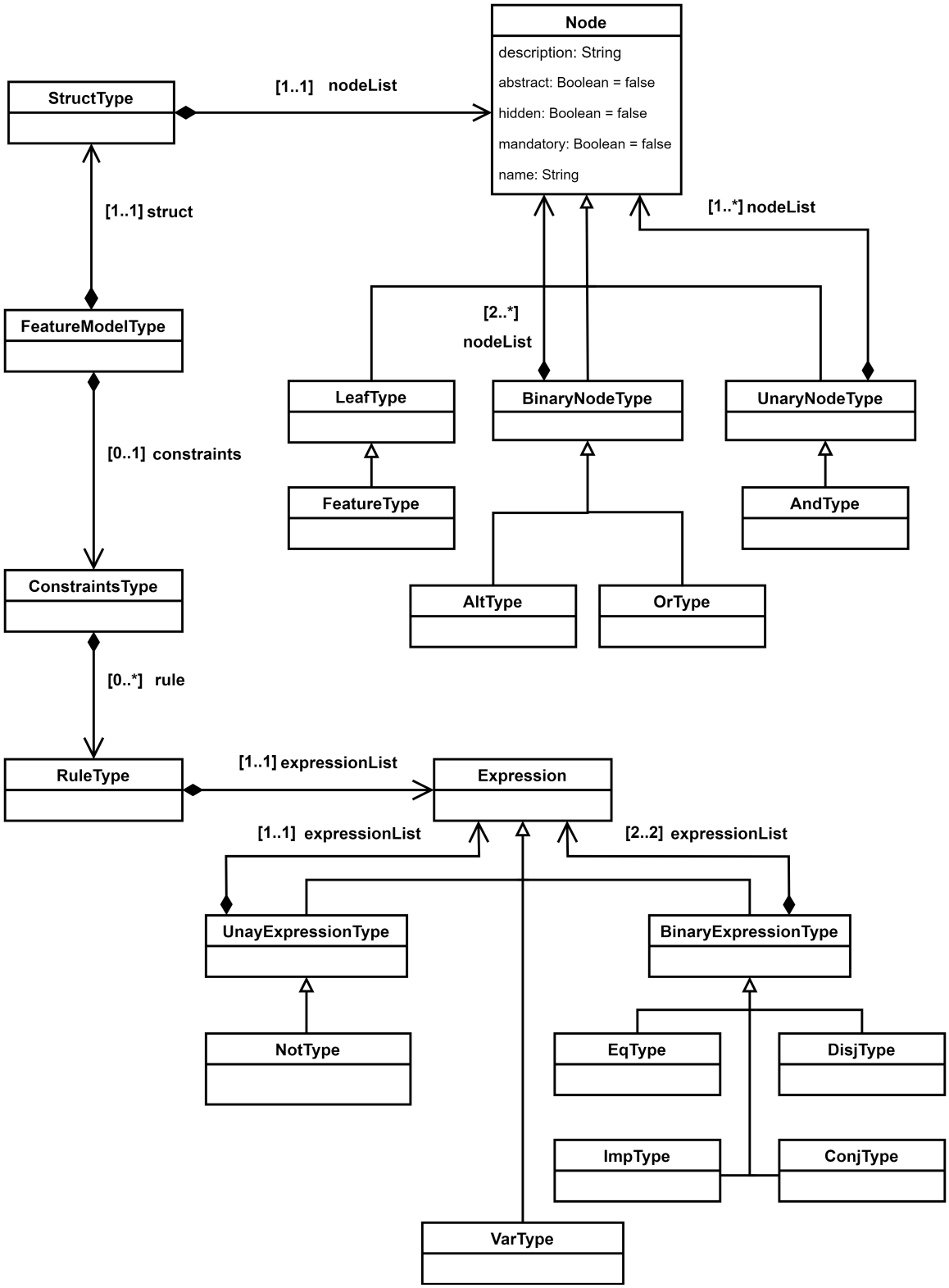


Abbildung 4.1.: Klassendiagramm des Zwischenmetamodells.

4.2. Probleme des automatisch generierten XML-Schemas

Wie bereits erwähnt war das größte Problem während der Implementierung, dass das automatisch generierte XML-Schema nicht alle Elemente der FeatureIDE-Modelle darstellen konnte. Es wurden 87 Beispielmodelle der FeatureIDE für die Generierung des XML-Schemas verwendet. Vermutlich haben diese verwendeten Feature-Modelle nicht alle möglichen Kombinationen der Elemente abgedeckt. Beispielsweise wurden Feature-Modelle, welche mehrere Alternativ- oder Oder-Features hintereinander enthalten haben, nicht erkannt. Außerdem waren die Struktur und die Beziehung der Elemente im XML-Schema nicht für die Implementierung einer leicht wartbaren und strukturerhaltenden Transformation geeignet.

Um dieses Problem zu lösen, wurde das XML-Schema auf Basis des automatisch generierten XML-Schemas manuell definiert. Dabei wurde darauf geachtet, das Schema einfach zu halten, aber auch für die Implementierung der Transformation, wie beispielsweise mittels Vererbungshierachien, vorteilhaft zu sein. Das manuell definierte XML-Schema hat mit dem automatisch generierten XML-Schema nur noch die Namen der Elemente gemeinsam.

5. Evaluation

Dieses Kapitel diskutiert die Evaluation dieser Arbeit. Dabei werden zuerst die Idee und der GQM-Plan vorgestellt. Anschließend wird die Umsetzung der Evaluation für die bidirektionale M2T-Transformation und die Ableitungsstrategie von Vitruvius erläutert. Zudem werden die Modelle welche bei den jeweiligen Tests verwendet werden gezeigt und erklärt. Des Weiteren werden die Ergebnisse und die Validität der Evaluation besprochen.

5.1. Idee

Die Evaluation besteht aus zwei Teilen. Im ersten Teil zeigen wir die Korrektheit der bidirektionalen M2T-Transformation, welches für den ersten Beitrag (**B1**) relevant ist. Im zweiten Teil beschäftigen wir uns damit, wie gut man mit der von uns implementierten Transformation und der Ableitungsstrategie von Vitruvius feingranulare Änderungssequenzen aus Feature-Modellen der FeatureIDE ableiten kann. Dazu vergleichen wir die verschiedenen Evolutionsklassen von Thüm, Batory und Kästner [32]. Der zweite Evaluationsteil ist für den zweiten Beitrag (**B2**) dieser Arbeit relevant. Mithilfe der Ergebnisse der zwei Evaluationsteile können wir die beiden Forschungsfragen (**RQ1** und **RQ2**) dieser Arbeit beantworten. Die beiden Teile der Evaluation adressieren die entsprechenden Ziele des GQM-Plans 5.1.1.

5.1.1. GQM-Plan

G.1: Korrektheit der Transformation zwischen der als XML-Datei persistierten Zustände der FeatureIDE-Modellinstanz und der VaVe-Modellinstanz.

Q.1.1: Wie stark unterscheidet sich das Feature-Modell vor der Transformation von dem Feature-Modell nach der Transformation?

M.1.1.1: Die Anzahl an strukturellen und semantischen Unterschieden der Feature-Modelle vor und nach der Transformation.

G.2: Ableitung korrekter feingranularer Änderungssequenzen der als XML-Datei persistierten Zustände der FeatureIDE-Modelle mithilfe der Externen-Sicht.

Q.2.1: Wie viele abgeleitete Änderungssequenzen sind nach der Definition 5.3 korrekt?

M.2.1.1: Anteil an korrekten Änderungssequenzen (in Prozent).

Q.2.2: Wie viele der korrekt abgeleiteten Änderungssequenzen sind nach der Definition 5.5 redundanzfrei?

M.2.2.1: Die Anzahl der redundanten atomaren Änderungen in einer korrekten Änderungssequenz (Grad der Redundanz).

Q.2.3: Wie sehr unterscheiden sich die abgeleiteten Änderungssequenzen in den jeweiligen Evolutionsklassen?

M.2.3.1: Die Anzahl von korrekten und redundanzfreien Änderungssequenzen in jeder Evolutionsklasse (in Prozent).

M.2.3.2: Die Anzahl der redundanten atomaren Änderungen in jeder Evolutionsklasse (in Prozent).

Q.2.4: Spielt es eine Rolle, ob das Ausgangsmodell am Anfang eine VaVe-Modellinstanz oder eine FeatureIDE-Modellinstanz ist?

M.2.4.1: Die prozentuale Differenz der Anzahl an korrekten und redundanzfreien Änderungssequenzen sowie der atomaren redundanten Änderungen in beiden Evaluationsfällen.

5.1.2. Evaluation der Transformation

Das Ziel des ersten Evaluationsteils ist die korrekte Transformation zwischen der als XML-Datei persistierten Zustände der FeatureIDE-Modellinstanz und einer VaVe-Modellinstanz (**G.1**). Dazu evaluieren wir die bidirektionale M2T-Transformation in beide Richtungen. Dabei stellen wir uns die Frage, wie sehr sich die Feature-Modelle vor und nach der Transformation unterscheiden (**Q.1.1**). Um diese Frage beantworten zu können, messen wir die Anzahl an *strukturellen* und *semantischen* Unterschieden nach einem Round-Trip-Test. Wir definieren die Begriffe wie folgt:

Definition 5.1 (Struktureller Unterschied) *Zwei Feature-Modelle F1 und F2 unterscheiden sich strukturell, wenn die graphische Darstellung, also auf Ebene der konkreten Syntax, unterschiedlich ist.*
Ein Beispiel hierfür ist die Reihenfolge von Features.

Definition 5.2 (Semantischer Unterschied) *Zwei Feature-Modelle unterscheiden sich semantisch, wenn sie sich in ihrer Bedeutung unterscheiden. Das sind demnach alle Unterschiede, die nicht strukturell sind.*

Dabei ist eine Transformation korrekt, wenn sich das Modell vor der Transformation mit dem Modell nach der Transformation nicht semantisch unterscheidet. Unterscheiden sie sich auch nicht strukturell, dann ist die Transformation strukturerhaltend.

5.1.3. Evaluation der abgeleiteten Änderungssequenzen

Der zweite Teil der Evaluation basiert auf dem zweiten Ziel des GQM-Plans 5.1.1. Dies ist die korrekte Ableitung feingranularer Änderungssequenzen, der als XML-Datei persistierten Zustände, der FeatureIDE-Modelle mithilfe der Externen- Sicht (**G.2**). Mit dem zweiten Teil der Evaluation weisen wir nach, dass dieses Ziel erreicht werden kann. Das Erreichen dieses Ziels würde bedeuten, dass es möglich ist Feature-Modelle durch eine externe Sicht mit anderen Modellen im System konsistent zu halten.

Denn durch die implementierte bidirektionale M2T-Transformation können zwei verschiedene Zustände eines Feature-Modells der FeatureIDE in VaVe-Feature-Modelle transformiert werden. Mit diesen zwei VaVe-Feature-Modellen können wir dann, mithilfe der Ableitungsstrategie von Vitruvius, eine Änderungssequenz ableiten. Diese Änderungssequenz kann zur Konsistenzerhaltung der Modelle verwendet werden. Zur Ableitung der Änderungssequenzen der FeatureIDE-Modelle nach der Transformation verwenden wir die Ableitungsstrategie von Vitruvius mit ähnlichkeitsbasierter Vergleichsstrategie. Um Nachzuweisen, dass das Ziel erreicht wurde beantworten wir zwei Fragen. Die Frage wie viele der abgeleiteten Änderungssequenzen *korrekt* sind (**Q.2.1**) und die Frage wie viele davon *redundanzfrei* sind (**Q.2.1**). Dabei überprüfen wir zuerst ob die Änderungssequenzen *korrekt* (**M.2.1.1**) sind und danach, ob sie *redundanzfrei* (**M.2.2.1**) sind.

Dafür definieren wir zuerst einige Begriffe.

Definition 5.3 (Korrekte Änderungssequenz) *Sei F1 ein Feature-Modell und F1' das von einem Nutzer veränderte Feature-Modell von F1. Sei weiter F2 das Feature-Modell, welches nach dem Anwenden der abgeleiteten Änderungssequenz auf F1 entsteht. Dann ist eine Änderungssequenz korrekt, wenn F1' und F2 identisch sind.*

Definition 5.4 (Redundante Änderungen) *Redundante Änderungen sind eine Teilmenge einer Änderungssequenz, die aus der Sequenz entfernt werden können, ohne dass die Sequenz ihre Korrektheit verliert.*

Definition 5.5 (Redundanzfreie Änderungssequenz) *Eine Änderungssequenz ist redundanzfrei, wenn sie keine redundanten Änderungen enthält.*

In Abbildung 5.2 ist eine redundante Änderung in der oberen Änderungssequenz das Erstellen und das Löschen des Features F2. Würden diese Änderungen nicht angewendet werden, würde die Änderungssequenz immer noch das selbe Modell ergeben. Die untere Änderungssequenz ist hingegen redundanzfrei. Die Frage nach der Redundanzfreiheit einer Änderungssequenz ist deshalb interessant, da eine redundanzfreie Änderungssequenz bei der Prävention von Nebeneffekten während der Konsistenzerhaltung helfen kann. Die Korrektheit und Redundanzfreiheit der abgeleiteten Änderungssequenzen überprüfen wir händisch.

Es ist zu beachten, dass unterschiedliche Änderungssequenzen beim Anwenden auf ein Modell am Ende das gleiche veränderte Modell ergeben können. Beispielsweise würde eine Änderungssequenz bei der zuerst das Element *F1* gelöscht und danach ein neues Element *F2* dem Modell hinzugefügt wird und eine Änderungssequenz bei der zuerst das neue Element *F2* hinzugefügt und dann das Element *F1* gelöscht wird das gleiche Modell ergeben. In Abbildung 5.1 ist dies graphisch dargestellt. Es kann auch sein, dass eine Änderungssequenz weniger atomare Änderungen enthält als eine andere Änderungssequenz, sie aber am Ende das gleiche Modell ergeben. Ein Beispiel dafür ist in Abbildung 5.2 zu sehen.

Solange eine Änderungssequenz *korrekt* (5.3) ist, ergibt eine Unterscheidung und Bewertung durch die Reihenfolge der atomaren Änderungen in der Änderungssequenz, wie bei Wittler [36], in unserem Fall wenig Sinn, da die eigentliche Änderung an einem Feature-Modell der FeatureIDE getätigt wird. Die beiden Zustände des FeatureIDE-Modells

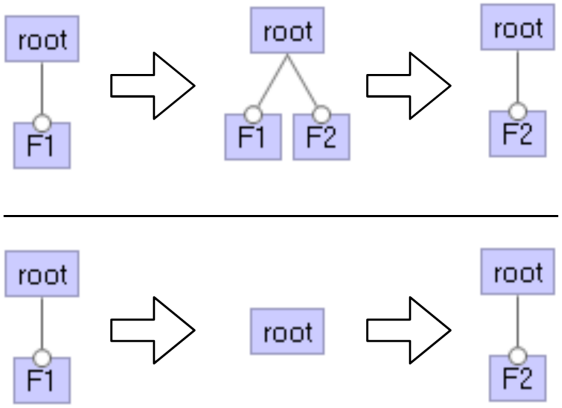


Abbildung 5.1.: Unterschiedliche Änderungssequenzen mit **gleicher** Anzahl an atomaren Änderungen, welche am Ende das selbe Modell ergeben. In der oberen Hälfte wird zuerst das Feature *F2* erstellt und danach das Feature *F1* gelöscht. In der unteren Hälfte wird zuerst das Feature *F1* gelöscht und danach das Feature *F2* hinzugefügt.

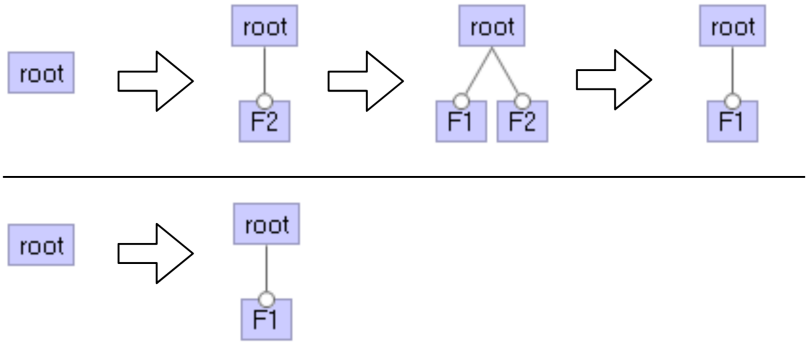


Abbildung 5.2.: Unterschiedliche Änderungssequenzen mit **unterschiedlicher** Anzahl an atomaren Änderungen, welche am Ende das selbe Modell ergeben. In der oberen Hälfte wird zuerst das Feature *F2* erstellt, danach das Feature *F1*. Das Feature *F2* wird danach wieder gelöscht. In der unteren Hälfte wird nur das Feature *F1* erstellt.

werden in VaVe-Feature-Modelle transformiert und die Änderungssequenz anhand dieser VaVe-Feature-Modelle abgeleitet.

Um Änderungssequenzen ableiten zu können, müssen wir Testmodelle editieren. Wir editieren die Testmodelle entsprechend den Evolutionsklassen von Thüm, Batory und Kästner [32]. Dazu stellen wir uns zusätzlich im zweiten Evaluationsteil die Frage, wie sich die abgeleiteten Änderungssequenzen in den verschiedenen Evolutionsklassen unterscheiden (**Q.2.3**). Um die Frage beantworten zu können, ob es eine Rolle spielt, wenn das Ausgangsmodell bei der Ableitung der Änderungssequenz ein VaVe-Feature-Modell ist, (**Q.2.4**) führen wir den zweiten Teil der Evaluation mit zwei Fällen durch. Die beiden *Evaluationsfälle* sind in Abbildung 5.3 und 5.4 dargestellt.

1. Fall: Wir erstellen ein Feature-Modell *F* in FeatureIDE und ändern dieses in FeatureIDE zu einem Feature-Modell *F'* (1). Danach transformieren wir die XML-Dateien *E* und *E'* mit der T2M-Transformation in VaVe-Modelle *S* und *S'* (2) und verwenden diese bei der Ableitungsstrategie (3).

2. Fall: Wir erstellen ein VaVe-Modell *S*. Dieses transformieren wir mit der M2T-Transformation in eine XML-Datei *E* der FeatureIDE (1). Ändern dann *F* in FeatureIDE zu *F'* (2). Anschließend transformieren wir die XML-Datei *E'* mit der T2M-Transformation in ein VaVe-Modell *S'* (3). Bei der Ableitungsstrategie verwenden wir dann *S* und *S'* (4).

5.2. Umsetzung

In diesem Unterkapitel diskutieren wir die Umsetzung der beiden Evaluationsteile. Als erstes besprechen wir die Umsetzung der Evaluation für die implementierte bidirektionale M2T-Transformation. Danach die Umsetzung der Evaluation für die abgeleiteten Änderungssequenzen der ähnlichkeitsbasierten Ableitungsstrategie von Vitruvius.

5.2.1. Bidirektionale M2T-Transformation

Die Evaluation der implementierten bidirektionalen M2T-Transformation soll die Frage beantworten, ob und wie sehr sich die Modelle nach der Transformation unterscheiden (**Q1.1**). Dafür führen wir zwei verschiedene Round-Trip-Tests durch, um beide Richtungen der Transformation zu evaluieren. Die erste Art bezeichnen wir als *FeatureIDE-Test*. Die zweite Art bezeichnen wir als *VaVe-Test*. Sie unterscheiden sich darin, mit welcher Modellinstanz begonnen wird. Der Ablauf des FeatureIDE-Test ist in Abbildung 5.5 dargestellt. Bei einem FeatureIDE-Test wird ein Feature-Modell der FeatureIDE F1 in ein VaVe-Feature-Modell V transformiert (1), wobei dieses dann wieder in ein Feature-Modell der FeatureIDE F2 transformiert wird (2). Danach werden F1 und F2 miteinander verglichen (3).

Der VaVe-Test ist die Umkehrung des FeatureIDE-Test. Bei einem VaVe-Test wird ein VaVe-Feature-Modell V1 in ein Feature-Modell der FeatureIDE F transformiert (1), wobei dieses dann wieder in ein VaVe-Feature-Modell V2 transformiert wird (2). Danach werden V1 und V2 miteinander verglichen (3). Der Ablauf ist in Abbildung 5.6 dargestellt.

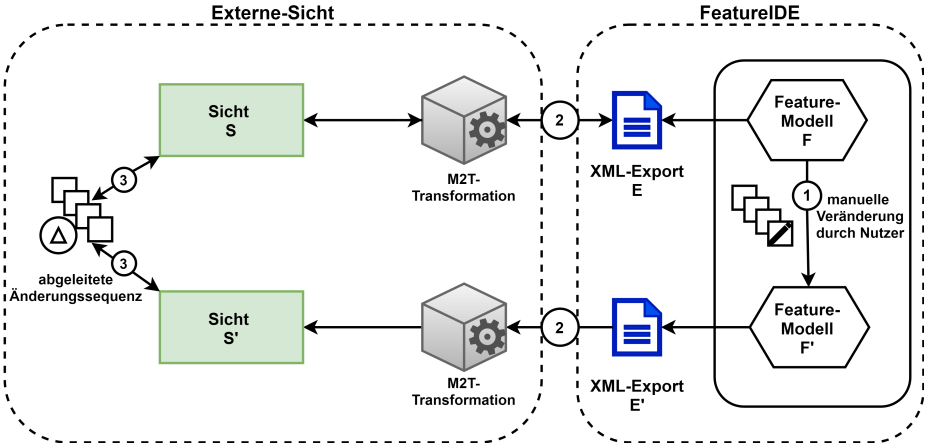


Abbildung 5.3.: Ablauf des 1. Falls für die Evaluation der Ableitungsstrategie.

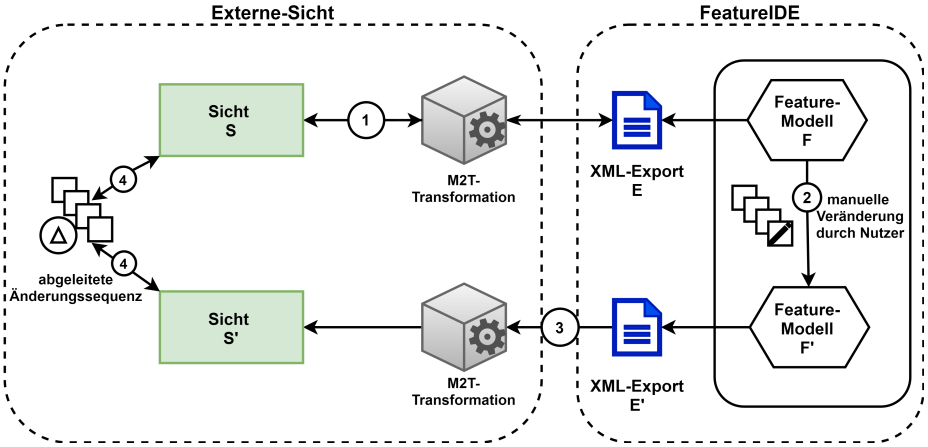


Abbildung 5.4.: Ablauf des 2. Falls für die Evaluation der Ableitungsstrategie.

Bei beiden Round-Trip-Tests kommt für den Vergleich *XMLUnit* [38] zum Einsatz. Mithilfe von XMLUnit können wir die Unterschiede, welche durch die in Kapitel 4 erwähnten, nicht transformierbaren Modellelementen auftreten, ignorieren, da dadurch weder die Struktur noch der Konfigurationsraum verändert wird.

5.2.2. Standardableitungsstrategie von Vitruvius

Das zweite Ziel des GQM-Plans ist die Evaluation der abgeleiteten Änderungssequenzen durch die ähnlichkeitsbasierte Ableitungsstrategie von Vitruvius mithilfe der externen Sicht (**G.2**). Um die Strategie zu Evaluieren editieren wir Testmodelle der FeatureIDE und leiten dann mithilfe der in dieser Arbeit implementierten Transformation und der ähnlichkeitsbasierten Ableitungsstategie von Vitruvius Änderungssequenzen ab. Dazu editieren wir die Testmodelle entsprechend der Evolutionsklassen von Thüm, Batory und Kästner [32]. Dabei werden die Testmodelle so editiert, dass sie jeweils einen neuen Zustand für jede Evolutionsklasse enthalten. Thüm, Batory und Kästner [32] unterscheiden dabei zwischen *Refaktorisierung*, *Spezialisierung*, *Generalisierung* und *willkürlichem Editieren*.

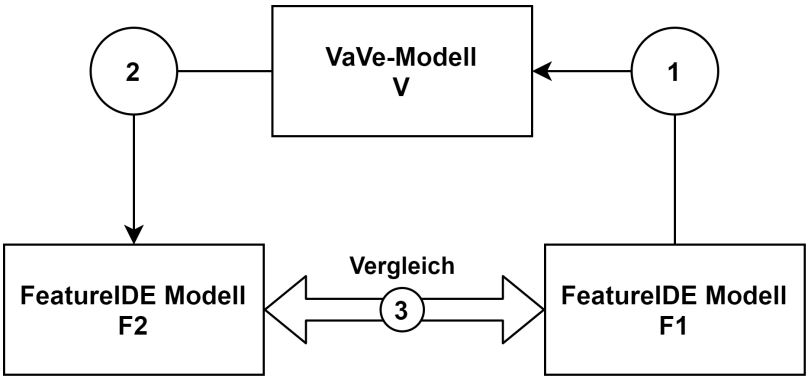


Abbildung 5.5.: Ablauf des FeatureIDE-Tests.

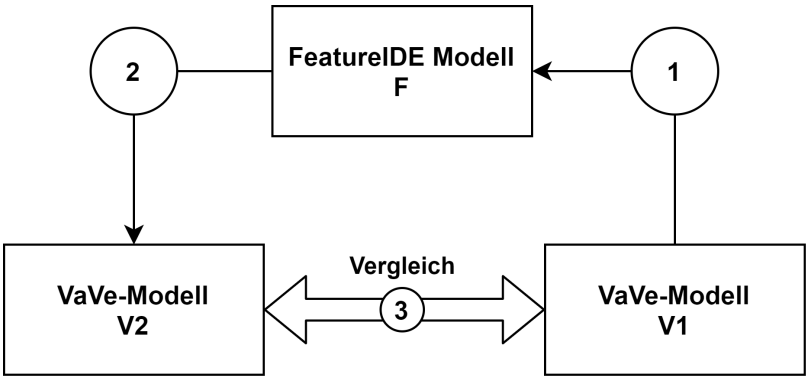


Abbildung 5.6.: Ablauf des VaVe-Tests.

Änderungsoperation	Evolutionsklasse
Feature entfernen	Spezialisierung
optionales Feature zu obligatorischem Feature	Spezialisierung
Oder-Gruppe zu Alternativ-Gruppe	Spezialisierung
Feature hinzufügen	Generalisierung
obligatorisches Feature zu optionalem Feature	Generalisierung
Alternativ-Gruppe zu Oder-Gruppe	Generalisierung
Positionswechsel von Features mit gleichem Eltern-Feature	Refaktorisierung

Tabelle 5.1.: Die bei der Evaluation verwendeten Änderungsoperationen mit den jeweiligen Evolutionsklassen.

Bei der Refaktorisierung werden keine Elemente dem Konfigurationsraum hinzugefügt oder gelöscht. Werden existierende Elemente gelöscht und keine neuen hinzugefügt, spricht man von Spezialisierung. Bei einer Generalisierung werden neue Elemente hinzugefügt und bereits existierende Elemente nicht gelöscht. In allen anderen Fällen wird von willkürlichem Editieren gesprochen. Die Tabelle 5.1 zeigt welche Änderungsoperationen, die wir in unserer Evaluation verwenden, welcher Klasse zugehören. Für die Evaluation der Ableitungsstrategie werden für jedes Testmodell, bei beiden Evaluierungsfällen (5.3 und 5.4) vier Kopien des entsprechenden Feature-Modells der FeatureIDE erstellt. Bei der ersten Kopie werden Änderungsoperationen, welche in die Evolutionsklasse Spezialisierung fallen, angewendet und bei der zweiten Strategie Änderungsoperationen, die in die Evolutionsklasse Generalisierung fallen. Wir nennen diese beiden Kopien *Spezialisierungs-* beziehungsweise *Generalisierungskopie*. Die dritte Kopie deckt dann die Evolutionsklasse willkürliches Editieren ab. Dabei wenden wir bei Testmodellen, bei denen es möglich ist, die Änderungsoperationen, die bei den anderen beiden Kopien angewendet wurden, an. Bei Testmodellen, bei denen dies nicht möglich ist, werden Änderungsoperationen sowohl aus der Evolutionsklasse Spezialisierung als auch Generalisierung angewendet. Diese Kopie nennen wir *Willkürlichekopie*. Die vierte Kopie ist für die Evolutionsklasse Refaktorisierung, welche wir als *Refaktorisierungskopie* bezeichnen. Beim Editieren der Kopien werden manuell die vom Nutzer durchgeführten atomaren Änderungen notiert, um diese dann mit der abgeleiteten Änderungssequenz zu vergleichen. Die notierten atomaren Änderungen entsprechen den Änderungen, die der Nutzer in den entsprechenden VaVe-Feature-Modellen machen müsste. Es werden im zweiten Evaluationsteil jeweils einmal die Spezialisierungs-, Generalisierungs-,Refaktorisierungs- und Willkürlichekopie als neuer Zustand und das originale Feature-Modell der FeatureIDE als alter Zustand der Ableitungsstrategie zum Ableiten der Änderungssequenz übergeben. Da die verwendete Ableitungsstrategie von Vitruvius beim Editieren von Cross-Tree-Bedingungen (CTCs) eine Exception wirft, können für die Evolutionsklasse Refaktorisierung als Änderungsoperation nur die Positionen von Features mit gleichem Eltern-Feature getauscht werden.

5.3. Testmodelle

Für die Evaluation der bidirektionalen M2T-Transformation werden 85 Feature-Modelle aus den FeatureIDE Beispielprojekten verwendet. Alle 85 Feature-Modelle der FeatureIDE durchlaufen den FeatureIDE-Test. Die Beispielprojekte enthalten Feature-Modelle unterschiedlicher Größen. In Abbildung 5.7 ist ein FeatureIDE-Modell mit 13 Features und ohne CTC zu sehen. Das Feature-Modell in Abbildung A.1 hingegen enthält 23 Features und 8 CTCs. Für den VaVe-Test, werden für alle 85 Feature-Modelle der FeatureIDE identische VaVe-Feature-Modelle händisch erstellt. Diese 85 VaVe-Feature-Modelle durchlaufen dann den VaVe-Test.

Für den zweiten Teil der Evaluation, verwenden wir neun Feature-Modelle der FeatureIDE, von denen drei selbst erstellt sind und sechs aus den FeatureIDE Beispielprojekten stammen. Bei allen neun Testmodellen werden die angesprochenen vier Kopien erstellt. Insgesamt werden also 36 Änderungssequenzen abgeleitet und analysiert.

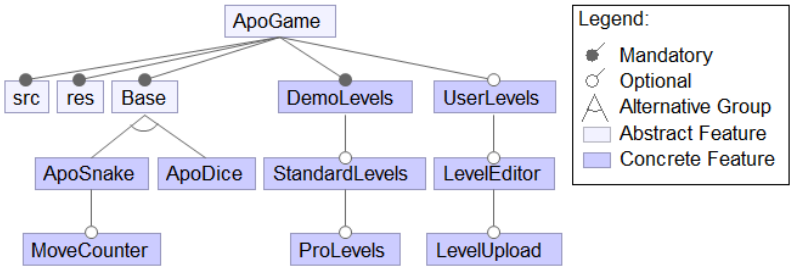


Abbildung 5.7.: Feature-Modell der FeatureIDE mit 13 Features ohne CTC.

Round-Trip-Test	korrekt transformiert	Unterschiede
FeatureIDE-Test	85/85	0
VaVe-Test	85/85	0

Tabelle 5.2.: Bei beiden Round-Trip-Testarten transformiert die bidirektionale M2T-Transformation alle 85 Testmodelle korrekt. Keine Unterschiede vor und nach der Transformation (**M.1.1.1**).

5.4. Ergebnisse

Im Folgenden diskutieren wir die Ergebnisse der Evaluation. Zuerst werden die Ergebnisse der bidirektionalen M2T-Transformation erläutert. Anschließend diskutieren wir die Ergebnisse der Ableitungsstrategie von Vitruvius. Dabei zeigen wir bei welchen Evaluationsklassen die Ableitungsstrategie Schwierigkeiten hat.

5.4.1. Ergebnisse der Transformation

Die Ergebnisse der Round-Trip-Tests in Tabelle 5.2 zeigen, dass die bidirektionale M2T-Transformation in 100% der 85 Testfällen bei beiden Round-Trip-Testarten alle transformierbaren Modellelemente korrekt transformiert. Es gibt keine strukturellen oder semantischen Unterschiede zwischen dem Ausgangsmodell und dem Modell nach dem Round-Trip-Test (**M.1.1.1**). Entsprechend der Spezifikationen ist damit die Transformation korrekt, denn alle geteilten Informationen zwischen FeatureIDE und dem VaVe-Metamodell können korrekt transformiert werden. Da ebenfalls keine strukturellen Unterschiede existieren, ist die bidirektionale M2T-Transformation somit in den 85 Testfällen bei beiden Round-Trip-Testarten strukturerhaltend.

Evaluationsfall	Änderungsseq. korrekt	Änderungsseq. redundanzfrei	Anzahl gesamter redundanter atomarer Änd.
1. Fall	34/36	0/34	159/546
2. Fall	34/36	0/34	159/546

Tabelle 5.3.: Ergebnisse der ähnlichkeitsbasierten Ableitungsstrategie (Änd. = Änderungen). Bei der insgesamt 36 Änderungssequenzen abgeleitet wurden. Die Metriken in der Tabelle von links nach rechts (**M.2.1.1**), (**M.2.2.1**) und (**M.2.4.1**).

Evolutionsklasse	Änderungsseq. vorhanden	Änderungsseq. korrekt	Änderungsseq. redundanzfrei
Spezialisierung	9/9	9/9	0/9
Generalisierung	9/9	9/9	0/9
Willkürliches Editieren	9/9	9/9	0/9
Refaktorisierung	7/9	5/7	0/7

Tabelle 5.4.: Test Ergebnisse der Ableitungsstrategie in den verschiedenen Evolutionsklassen (1. Fall **M.2.3.1**)

.

5.4.2. Ergebnisse der Ableitungsstrategie von Vitruvius

In den Tabellen 5.3, 5.4 und 5.5 sind die Ergebnisse der ähnlichkeitsbasierten Ableitungsstrategie von Vitruvius zu sehen. Die Ergebnisse in der Tabelle 5.3 zeigen, dass von den 36 abgeleiteten Änderungssequenzen 94,44% korrekt sind, davon sind 0% redundanzfrei (**M.2.1.1**). Der Grad der Redundanz beträgt 29,12% (**M.2.2.1**). Es existieren keine Unterschiede zwischen den beiden Evaluationsfällen.

Die Tabellen 5.4 und 5.5 vergleichen die verschiedenen Evolutionsklassen im ersten und zweiten Evaluationsfalls. Die Ergebnisse der Tests zeigen, dass die Ableitungsstrategie in den Evolutionsklassen Spezialisierung, Generalisierung und willkürliches Editieren in 100% der Testfälle eine Änderungssequenz ableitet. Dabei sind in diesen drei Evolutionsklassen 100% davon korrekte Änderungssequenzen und 0% der korrekten Än-

Evolutionsklasse	Änderungsseq. vorhanden	Änderungsseq. korrekt	Änderungsseq. redundanzfrei
Spezialisierung	9/9	9/9	0/9
Generalisierung	9/9	9/9	0/9
Willkürliches Editieren	9/9	9/9	0/9
Refaktorisierung	7/9	5/7	0/5

Tabelle 5.5.: Test Ergebnisse der Ableitungsstrategie in den verschiedenen Evolutionsklassen (2. Fall **M.2.3.1**)

	1.Fall		2.Fall	
Evolutionsklasse	Anzahl gesamter Änd.	Anzahl redundanter Änd.	Anzahl gesamter Änd.	Anzahl redundanter Änd.
Spezialisierung	88	18	88	18
Generalisierung	171	60	171	60
Willkürliches Editieren	227	63	227	63
Refaktorisierung	60	18	60	18

Tabelle 5.6.: Anzahl der redundanten atomaren Änderungsoperationen (Änd.) im Vergleich zu der Anzahl der gesamten Änd. in den korrekten Änderungssequenzen der verschiedenen Evolutionsklassen (**M.2.3.2**).

derungssequenzen sind redundanzfrei. In der Evolutionsklasse Refaktorisierung kann die Ableitungsstrategie in 77,78% der Testfälle eine Änderungssequenz ableiten. Von den abgeleiteten Änderungssequenzen sind 71,43% korrekt und davon 0% redundanzfrei. Die Evolutionsklasse Refaktorisierung ist die einzige Klasse, bei der die Ableitungsstrategie in den neun Testfällen nicht korrekte oder keine Änderungssequenzen ableitet. Tabelle 5.6 zeigt die Anzahl der gesamten atomaren Änderungen im Vergleich zu der Anzahl an redundanten atomaren Änderungen in den korrekt abgeleiteten Änderungssequenzen der verschiedenen Evolutionsklassen (**M.2.3.2**). Dabei kann man erkennen, dass die Ableitungsstrategie in der Evolutionsklasse Spezialisierung einen *Grad der Redundanz* von 20,46% aufweist. In der Evolutionsklasse Generalisierung beträgt dieser 35,09% und in der Evolutionsklasse willkürliches Editieren 27,75%. Die Evolutionsklasse Refaktorisierung hat dabei einen Anteil von 30% an redundanten atomaren Änderungen in den korrekten Änderungssequenzen. Die Ergebnisse zeigen, dass die Evolutionsklasse Generalisierung die meisten redundanten atomaren Änderungen in den korrekten Änderungssequenzen enthält. Die Evolutionsklasse Spezialisierung hingegen hat die wenigsten redundanten atomaren Änderungen in den korrekten Änderungssequenzen. Da in der Tabelle 5.3 bei beiden Evaluationsfällen keine Unterschiede zu erkennen sind und die Tabellen 5.4 und 5.5 identisch sind, spielt es demnach keine Rolle, ob das Ausgangsmodell ein Feature-Modell der FeatureIDE oder ein VaVe-Feature-Modell ist (**M.2.4.1**).

5.5. Validität

In diesem Unterkapitel diskutieren wir die Validität der Evaluation. Es gibt sehr viele verschiedene Feature-Modelle mit unterschiedlicher Anzahl an Features oder Cross-Tree-Bedingungen und verschiedenen Strukturen. Deshalb können wir bei der Evaluierung der Transformation nicht alle möglichen Feature-Modelle testen. Es wurden allerdings 85 existierende Testfälle mit möglichst verschiedener Struktur sowie Anzahl an Features und Cross-Tree-Bedingungen aus den Beispielmodellen der FeatureIDE verwendet. Bei der Evaluation der bidirektionalen M2T-Transformation durch die Round-Trip-Tests können außerdem Fehler übersehen werden, die bei der Richtung von FeatureIDE zu VaVe oder

VaVe zu FeatureIDE entstehen, aber bei der Rückrichtung aufgehoben werden. Um dem zu entgegnen, wurden während der Implementierung der Transformation allerdings einige Feature-Modelle der FeatureIDE in VaVe-Feature-Modelle transformiert und umgekehrt. Dabei wurde manuell die Korrektheit überprüft.

Da sehr viele Möglichkeiten existieren, ein Feature-Modell zu editieren, können wir bei der Evaluation der Änderungssequenz nicht alle Fälle abdecken. Deshalb wurden gängige Evolutionsszenarien aus der Literatur genutzt, um typische Fälle abzudecken. Darüber hinaus wurden die Ableitungssequenzen händisch evaluiert und für die Korrektheit mit den vom Nutzer ausgeführten atomaren Änderungen verglichen. Zudem wurden die redundanten Änderungen ebenfalls händisch identifiziert. Idealerweise sollte dies automatisch überprüft werden, jedoch fehlen dafür im Moment die technischen Grundlagen in Vitruvius, da Änderungssequenzen zwischen zwei Modellzuständen zum Zeitpunkt der Evaluation nicht auf beliebige andere Modelle angewandt werden können.

6. Diskussion

In diesem Kapitel werden die Forschungsfragen (**RQ1** und **RQ2**) beantwortet. Des Weiteren werden die Ableitungsmuster und Schwächen der Strategie diskutiert, die während der Evaluation festgestellt wurden.

6.1. Forschungsfragen

Die zwei Beitrage die diese Arbeit leistet sind die Implementierung der bidirektionalen M2T-Transformation (**B1**) und die initiale Untersuchung der abgeleiteten Änderungssequenzen (**B2**). Der erste Beitrag erfüllt die erste Eigenschaft (**E1**) zur Konstruktion einer externen Sicht. Der zweite Beitrag untersucht dabei die Erfüllung der zweiten Eigenschaft (**E2**). Mit diesen beiden Beiträgen beantworten wir zwei Fragen: Die Frage, wie gut sich Änderungssequenzen aus Feature-Modellen der FeatureIDE mithilfe der in dieser Arbeit implementierten Transformation und mit Vitruvius ableiten lassen (**RQ1**) und somit die Frage, wie gut sich FeatureIDE durch eine externe Sicht an Vitruvius anbinden lässt (**RQ2**).

Die Ergebnisse der Evaluation dieser Arbeit zeigen, dass die in dieser Arbeit implementierte Transformation alle geteilten Informationen zwischen FeatureIDE und dem VaVe-Metamodell, korrekt transformiert. Damit wird die erste Eigenschaft (**E1**) zur Konstruktion einer externen Sicht erfüllt. Des Weiteren zeigen die Ergebnisse, dass die ähnlichkeitsbasierte Ableitungsstrategie von Vitruvius mithilfe der in dieser Arbeit implementierten Transformation in 94,44% der Fälle eine korrekte Änderungssequenz aus den Feature-Modellen der FeatureIDE ableitet (**M.2.2.1**). Die zweite Eigenschaft (**E2**) kann demnach auch erfüllt werden. Da beide Eigenschaften zur Konstruktion einer externen Sicht erfüllbar sind, ist die Ableitung feingranularer Änderungssequenzen aus Feature-Modellen der FeatureIDE mit der in dieser Arbeit implementierten Transformation und mit Vitruvius möglich (**RQ1**). Damit ist auch die Anbindung der FeatureIDE durch eine externe Sicht an Vitruvius realisierbar (**RQ2**).

6.2. Muster in abgeleiteten Änderungssequenzen

Bei der Evaluation mit Feature-Modellen sind drei Ableitungsmuster in den abgeleiteten Änderungssequenzen der ähnlichkeitsbasierten Ableitungsstrategie von Vitruvius zu erkennen. Diese zeigen, dass abgeleitete Änderungssequenzen einem eigenen, nicht direkt nachvollziehbaren Schema folgen können. Sie sind deshalb nicht direkt nachvollziehbar, da diese für einen Menschen nicht immer als intuitiv erscheinen. Denn es ist nicht zu erwarten, dass ein Mensch, um zum selben Modell wie die abgeleitete Änderungssequenz zu gelangen, dieselbe Änderungssequenz ausführen würde.

6.2.1. Referenz des Wurzel-Elements löschen und wieder hinzufügen

Bei allen abgeleiteten Änderungssequenzen ist zu beobachten, dass die ersten beiden atomaren Änderungen der Ableitungsstrategie immer identisch sind. Die Referenz des Wurzel-Elements *System* wird gelöscht und wieder hinzugefügt. Dies ist nach Definition 5.4 eine redundante Änderung. Darum sind auch keine der in den Testfällen abgeleiteten Änderungssequenzen redundanzfrei.

6.2.2. Ändern der Eltern-Kind-Beziehung

Wird die Eltern-Kind-Beziehung eines Features im Feature-Modell der FeatureIDE geändert, wird statt der Änderung des Group-Typ des entsprechenden Tree-Constraints, bei der abgeleiteten Änderungssequenz in allen Testfällen ein neuer Tree-Constraint mit dem entsprechenden Group-Typ erstellt. Die Referenzen vom alten Tree-Constraint werden gelöscht und in den neuen Tree-Constraint hinzugefügt. Danach wird der alte Tree-Constraint gelöscht. Dadurch hat die abgeleitete Änderungssequenz mehr atomare Änderungen als eine äquivalente Änderungssequenz wie sie ein Nutzer manuell durchführen würde.

6.2.3. Erstellen von Tree-Constraints

Wird nur ein Feature gelöscht und ein neues Feature mit der selben Eltern-Kind-Beziehung erstellt und hinzugefügt, wird das Tree-Constraint vom gelöschten Feature dem neu erstellten Feature hinzugefügt. Dadurch wird das Erstellen eines Tree-Constraints und das Ändern des Group-Typs gespart, wodurch es möglich ist, dass die abgeleitete Änderungssequenz weniger atomare Änderungen enthält als die atomaren Änderungen, die der Nutzer ausgeführt hat. Beim Löschen von Features ohne Kinder im Feature-Modell ist die abgeleitete Änderungssequenz mit der Änderungssequenz des Nutzers in den Testfällen immer identisch.

6.2.4. Wiederverwenden von Tree-Constraints

Ein weiteres Ableitungsmuster ist beim Hinzufügen von obligatorischen oder optionalen Features in mehrere Zweige auf der selben Ebene eines Feature-Modells mit nur optionalen oder obligatorischen Features zu erkennen. Die Abbildung 6.1 verdeutlicht das Muster. Das linke Modell in der Abbildung stellt das Ausgangsmodell dar. Diesem Modell werden den Blättern jeden Zweigs jeweils ein neues Feature hinzugefügt (*NewFeature*, *NewFeature2* und *NewFeature3*). Das rechte Modell in der Abbildung stellt das veränderte Modell dar. Die Kanten zwischen den Features stellen die Tree-Constraints (TC) im VaVe-Feature-Modell dar. In der abgeleiteten Änderungssequenz der ähnlichkeitsbasierten Ableitungsstrategie erhält das Feature, welches im linkesten Zweig hinzugefügt wird, die Kante des Features einer Ebene höher und ein Zweig rechts. In der Abbildung ist das Feature welches dem linkesten Zweig hinzugefügt wird das Feature *NewFeature*. Dieser erhält die Kante des Features einer Ebene höher und ein Zweig rechts. In der Abbildung 6.1 ist das die Kante *TC1*. Danach werden die Kanten in der höheren Ebene nach links verschoben. Die Kante *TC0* wird in der Abbildung ein Zweig nach links verschoben. Für das rechteste Feature wird dann

eine neue Kante erstellt, in der Abbildung ist das *TC5*. Für die anderen hinzugefügten neuen Features wird auch jeweils ein TC erstellt. Die ähnlichkeitsbasierte Ableitungsstrategie von Vitruvius verwendet demnach die Kanten (Tree-Constraints) zwischen den bereits existierenden Features für die neu erstellten Features.

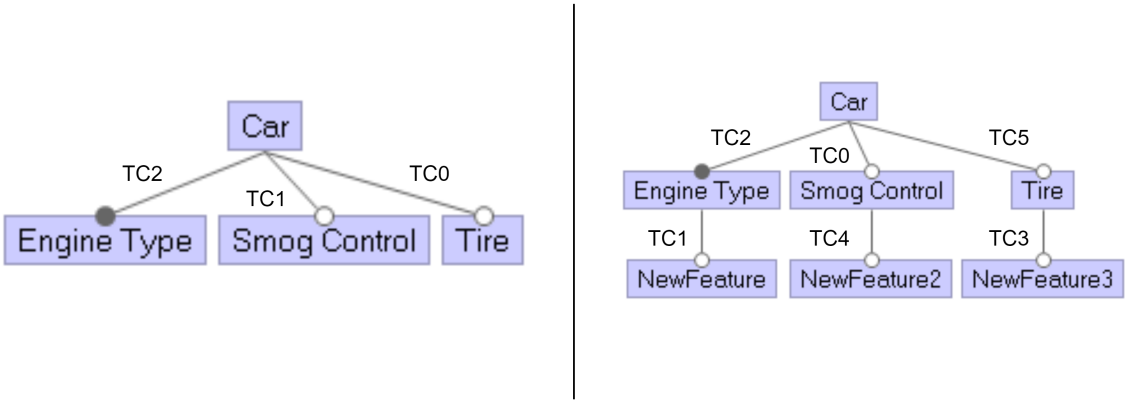


Abbildung 6.1.: Ableitungsmuster der Ableitungsstrategie bei der Tree-Constraints wiederverwendet werden. Links ist das Ausgangsmodell zu sehen, rechts das veränderte Modell.

6.3. Schwächen der Ableitungsstrategie

Die Ableitungsstrategie kann keine Änderungssequenz für Veränderungen in den Cross-Tree-Bedingungen ableiten. Dadurch können wir bei der Evolutionsklasse Refaktorisierung als Änderungsoperation nur die Position der Features auf der selben Ebene tauschen. Auch bei den restlichen Evolutionsklassen können wir dadurch die Cross-Tree-Bedingungen nicht verändern und die Ableitungsstrategie evaluieren. Dieses Problem der ähnlichkeitsbasierten Ableitungsstrategie von Vitruvius ist bekannt. Es wird bereits an einer Lösung des Problems gearbeitet. Eine weitere Schwäche ist beim Tauschen der Position der Kinder in einer Alternativ- oder Oder-Gruppe zu beobachten. Die Ableitungsstrategie leitet dabei keine Änderungssequenz ab. Er erkennt also nicht, dass sich die Struktur des Feature-Modells geändert hat. Eine Änderungssequenz ohne atomare Änderungen ist demnach falsch, da sich die Struktur des Feature-Modells beim Tauschen der Positionen ändert. Ein Beispiel dafür ist in Abbildung 6.2 zu sehen. Das linke Modell ist das unveränderte Modell, wo hingegen im rechten Modell die Position der Kind-Features *Diesel* und *Benzin* in der Alternativ-Gruppe getauscht wurden. Im linken Modell sieht die Liste der Referenzen demnach wie folgt aus: [Referenz zum Kind-Feature *Diesel*, Referenz zum Kind-Feature *Benzin*]. Eine mögliche Änderungssequenz dafür wäre: Lösche Referenz des Kind-Features *Diesel* aus dem Eltern-Feature *Motor*, füge Referenz des Features *Diesel* am Ende der Referenzliste des Eltern-Feature *Motor* hinzu. Die Liste der Referenzen würde im Eltern-Feature *Motor* dadurch so aussehen: [Referenz zum Kind-Feature *Benzin*, Referenz zum Kind-Feature *Diesel*]. Eine Änderungssequenz ohne atomare Änderungen wäre falsch, da die Referenzliste sich nicht ändern würde und damit auch nicht die Struktur.

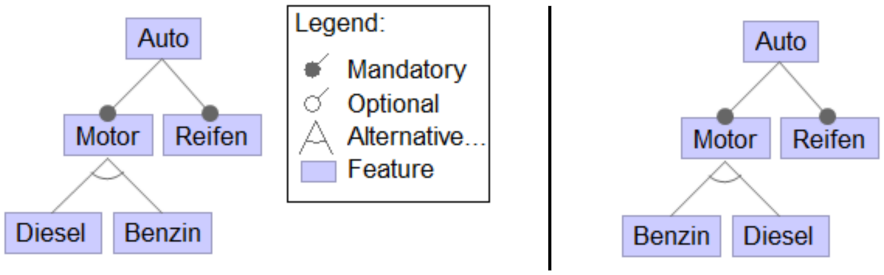


Abbildung 6.2.: Zwei verschiedene Zustände des Feature-Modells. Links das unveränderte Modell und rechts das veränderte Modell bei welchem die Postionen des *Benzin* und *Diesel* Features getauscht wurden.

7. Zukünftige Arbeiten

Wie in Abschnitt 6.3 erwähnt, hat die Ableitungsstrategie das Problem, bei Veränderungen von Cross-Tree Bedingungen in den Feature-Modellen keine Änderungssequenz ableiten zu können. Sobald dieses Problem behoben wird, ist es möglich, die Evaluation zu erweitern und die Ableitungsstrategie für Änderungsoperationen mit Cross-Tree-Bedingungen zu evaluieren. Außerdem kann in zukünftigen Arbeiten die Überprüfung der Korrektheit der abgeleiteten Änderungssequenz automatisiert werden. Ein Ansatz dafür wäre die abgeleitete Änderungssequenz auf das Ausgangsmodell in VaVe anzuwenden und das resultierende Modell mit dem VaVe-Feature-Modell, das durch die Änderungsoperationen des Nutzers entsteht, zu vergleichen. Existieren keine Unterschiede, ist die abgeleitete Änderungssequenz korrekt. Da die Automatisierung die Evaluierung vereinfacht, könnte diese mit mehr Testmodellen durchgeführt werden, um dann die Ableitungsmuster und Ergebnisse genauer zu studieren.

Wittler [36] hat im Rahmen seiner Masterarbeit die Ableitungsstrategie von Vitruvius modifiziert. In zukünftigen Arbeiten könnte diese modifizierte Strategie, mithilfe der in dieser Arbeit implementierten Transformation, ebenfalls untersucht und mit den Ergebnissen der Ableitungsstrategie von Vitruvius verglichen werden.

Der wichtigste Punkt ist allerdings, dass im zweiten Beitrag (**B2**) dieser Arbeit nur initial die Erfüllbarkeit der zweiten Eigenschaft (**E2**) untersucht wird. Nur, wenn die erste und zweite Eigenschaft (**E1**, **E2**) erfüllt ist kann eine externe Sicht konstruiert werden, welche die Feature-Modelle konsistent hält. In zukünftigen Arbeiten müsste also an der Erfüllung der zweiten Eigenschaft gearbeitet werden, um Feature-Modelle vollständig durch externe Sichten konsistent halten zu können.

8. Verwandte Arbeiten

Dieses Kapitel gibt einen Überblick über die verwandten Arbeiten in den verschiedenen Themengebieten dieser Arbeit. Die Themen, mit denen sich diese Arbeit beschäftigt, können in drei Kategorien unterteilt werden. Die Modelltransformation, insbesondere die Transformation von Variabilitätsmodellen sowie die Evolution von Feature-Modellen und die Modellkonsistenz. Dementsprechend werden verwandte Arbeiten in diesen Themengebieten aufgelistet und von dieser Arbeit abgegrenzt.

8.1. Modelltransformation

Die Modelltransformation ist eine grundlegende Methode in der modellgetriebenen und sichtenbasierten Softwareentwicklung um voneinander abhängige Sichten und Modelle konsistent halten zu können. Aranega, Etien und Mosser [6] verwenden in ihrem Paper Feature-Modelle, um Modelltransformationen nach bestimmten Geschäftsdomänen zu klassifizieren. Krzysztof Czarnecki und Simon Helsen [11] klassifizieren die verschiedenen Ansätze zur Modelltransformation. Bei den M2T-Transformationsansätzen unterscheiden sie zwischen *visitor-based* und *template-based* Ansätzen. Die M2M-Transformationsansätze unterteilen sie in fünf Klassen. In die *direct-manipulation*, *relational*, *graph-transformation-based*, *structure-driven* und *hybrid* Ansätze. Direct-Manipulation Ansätze sind gewöhnlich als object-orientiertes Framework implementiert. Die Transformationsregeln müssen vom Nutzer implementiert werden. Nach Krzysztof Czarnecki und Simon Helsen ist die M2M-Transformation, die wir als Zwischenschritt verwenden, ein direct-manipulation Ansatz.

Tom Mens und Pieter Van Gorp [25] adressieren in ihrem Paper die Verifizierung und Korrektheit einer Transformation an. Sie unterscheiden unter anderem zwischen *syntaktischer Korrektheit* und *semantischer Korrektheit*. Die *syntaktische Korrektheit* soll bei einem wohlgeformten Quellmodell garantieren, dass das durch die Transformation erzeugte Zielmodell ebenfalls wohlgeformt ist. Bei einer *semantisch korrekten* Transformation muss das Zielmodell die erwarteten semantischen Eigenschaften haben. In dieser Arbeit überprüfen wir die Korrektheit unserer implementierten Transformation in dem wir das Feature-Modelle vor der Transformation mit dem Feature-Modell nach der Transformation vergleichen. Die Transformation ist dabei korrekt, wenn sich die Feature-Modelle nach Definition 5.2 nicht semantisch Unterscheiden.

8.2. Transformation von Feature-Modellen

Feichtinger u. a. [17] stellen in ihrem Paper den *TRAVART* (TRAnsforming Variability ARTifacts) Ansatz vor. Dieser Ansatz soll es ermöglichen, zwischen bekannten Variabilitätsmodellen, wie *Feature-Modelle* oder *Decision-Modelle*, aber auch in der Industrie entwickelte kundenspezifische Variabilitätsdarstellungen transformieren zu können [15]. Damit sollen die verschiedenen Variabilitätsmodelle leichter miteinander verglichen werden. Der TRAVART Ansatz basiert auf drei Hauptkomponenten [16]. Den Variabilitätsmetamodellen, den Transformationsoperationen und den Transformationsalgorithmen. Dabei müssen für kundenspezifische Variabilitätsmodelle wie, in der Industrie, zuerst entsprechende Metamodelle definiert werden. In unserem Ansatz definieren wir ein XML-Schema für die XML-Dateien der FeatureIDE-Modelle, mit welchem wir in EMF ein Zwischenmetamodell erstellen. Dieses verwenden wir dann in unserer bidirektionalen M2T-Transformation. Für die Evaluation des TRAVART Ansatzes transformieren Feichtinger u. a. [17] Feature-Modelle der FeatureIDE in *DOPLER* Decision-Modelle [13] und in *Orthogonal-Variability-Modelle* [27] und wieder zurück. Die Transformationen implementieren sie in Java, basierend auf der API der FeatureIDE-Bibliothek. Anders als unser Ansatz soll der TRAVART Ansatz nicht die Konsistenzerhaltung von Variabilitätsmodellen, wie von FeatureIDE-Modellen, ermöglichen, sondern die automatische Transformation zwischen beliebigen Metamodellen.

8.3. Evolution von Feature-Modellen

Die Evolution von Feature-Modellen und Software-Produktlinien ist ein gut erforschtes Forschungsfeld [1, 10, 12, 19, 32]. Bürdek u. a. [10] stellen einen Ansatz vor, welcher automatisch die Unterschiede nach komplexen Änderungsoperationen an Feature-Modellen erkennt und dokumentiert. Zudem untersuchen sie die semantische Auswirkungen von den Änderungen an den Feature-Modellen. Dabei verwenden sie, wie auch wir, die Evolutionsklassen von Thüm, Batory und Kästner [32]. Anders als diese Arbeit, die die Konsistenzerhaltung von Feature-Modellen ermöglichen soll, basiert der Ansatz von Bürdek u. a. [10] darauf, einen Überblick über die Evaluation zu erlangen und die Auswirkung auf die Produktlinie zu verstehen.

Der Ansatz von Hoff u. a. [19] ermöglicht das Erstellen eines Feature-Modell Evolutionsplans und das rückwirkende Adaptieren dieses Plans in die Software-Produktlinie. Anders als der Ansatz dieser Arbeit soll nicht die Konsistenzerhaltung von Feature-Modellen ermöglicht werden, sondern die Prävention von unerwünschten Auswirkungen auf die strukturelle und logische Konsistenz der Software-Produktlinie.

In der Literatur werden Evolutionen von Feature-Modellen in Klassen unterteilt. Für die Evaluation dieser Arbeit verwenden wir die Evolutionsklassen von Thüm, Batory und Kästner [32]. Sie unterscheiden dabei zwischen *Refaktorisierung*, *Spezialisierung*, *Generalisierung* und *willkürliches Editieren*. Bei der Refaktorisierung werden keine Elemente dem Konfigurationsraum hinzugefügt oder gelöscht. Werden existierende Elemente gelöscht und keine neuen hinzugefügt spricht man von Spezialisierung. Bei einer Generalisierung

werden neue Elemente hinzugefügt und bereits existierende Elemente nicht gelöscht. In allen anderen Fällen spricht man von willkürlichem Editieren.

Czarnecki, Helsen und Eisenecker [12] bezeichnen, wie Thüm, Batory und Kästner [32] und somit auch wir, ein Feature-Modell X als eine Spezialisierung eines Feature-Modell Y, wenn der Konfigurationsraum von X eine Teilmenge des Konfigurationsraums von Y ist.

Alves u. a. [1] bezeichnen Operationen, die den Konfigurationsraum nicht verändern als *bi-refactorings* und Operationen die dem Konfigurationsraum des Feature-Modells neue Elemente hinzufügen als *refactorings*. In den Evolutionsklassen von Thüm, Batory und Kästner [32], welche wir in dieser Arbeit verwenden, entspricht bi-refactorings der Evolutionsklasse Refaktorisierung und refactorings der Evolutionsklasse Generalisierung.

Für die Evaluation erzeugen wir Testmodelle die hauptsächlich den Evaluationsklassen von Thüm, Batory und Kästner [32] entsprechen, da diese den Großteil der vorhandenen Klassen abdecken.

8.4. Konsistenzerhaltung von Modellen

Anders als in dieser Arbeit untersucht Wittler [36] externe Sichten nicht im Rahmen von Feature-Modellen, sondern im Rahmen der *Unified Modeling Language* (UML). Wittler [36] analysiert in seiner Arbeit speziell die Auswirkung der Konsistenzerhaltung mit anderen Modellen. In dieser Arbeit wird der Fokus auf die Überführung der FeatureIDE-Modelle in geeignete Modellrepräsentationen gelegt und die Auswirkung der Konsistenzerhaltung nicht untersucht.

In unserer Arbeit validieren und untersuchen wir im zweiten Beitrag (**B2**), die abgeleiteten Änderungssequenzen der Ableitungsstrategie von Vitruvius, welche zur Konsistenzerhaltung der FeatureIDE-Modelle benötigt werden. Die abgeleiteten Änderungssequenzen können zur deltabasierten Konsistenzerhaltung verwendet werden. Im Folgenden werden deshalb drei existierende Ansätze zur deltabasierten Konsistenzerhaltung vorgestellt. Der *Orthographic-Software-Modeling Ansatz* (OSM) von Atkinson, Stoll und Bostan [7] basiert auf dem *Single-Underlying-Model* (SUM) Prinzip und ermöglicht die Konsistenzerhaltung in der sichtenbasierten Entwicklung. Das SUM enthält dabei alle Informationen und Artefakte des Systems. Die verschiedenen Sichten werden dabei durch Transformationen und Änderungssequenzen konsistent gehalten. Der *Vitruvius* Ansatz von Klare u. a. [21] enthält ein *Virtual-Single-Underlying-Modell* (VSUM). Auch beim Vitruvius Ansatz werden die Sichten mithilfe der *deltabasierten Konsistenzerhaltung* [14] konsistent gehalten. Ein alternativer Ansatz zur automatisierten Konsistenzerhaltung der ebenfalls ein SUM verwendet ist der *MoConseMI* (MOdel-CONSistency-Ensured-by-Metamodel-Integration) Ansatz. Dieser Ansatz sowie der Ansatz von Vitruvius ermöglichen zusätzlich das Integrieren von existierenden Modellen oder Metamodellen. Alle erwähnten Ansätze verwenden die deltabasierte Konsistenzerhaltung. Der Ansatz von Sağlam und Kühn [28] basiert auf dem Vitruvius Ansatz. Um das Ziel, der Konsistenzerhaltung von Feature-Modellen, dieser Arbeit zu erreichen (**Z**), verwenden wir den Ansatz der externen Sichten [28]. Damit basiert auch der Ansatz dieser Arbeit auf dem Vitruvius Ansatz [21].

9. Fazit

Das Hauptziel dieser Arbeit ist die Konsistenzerhaltung von Feature-Modellen durch externe Sichten. Um dieses Ziel zu erreichen, werden in dieser Arbeit zwei Beiträge geleistet.

Der erste Beitrag (**B1**) ist die Implementierung einer bidirektionalen M2T-Transformation, welche die als XML-Dateien persistierten Zustände der Feature-Modelle der FeatureIDE in VaVe-Modelle transformiert und umgekehrt. Nach der Implementierung der Transformation wurden für die Evaluierung Round-Trip-Tests durchgeführt. Diese haben gezeigt, dass die in dieser Arbeit implementierte Transformation alle geteilten Informationen der beiden Modelle (FeatureIDE und VaVe) korrekt transformiert. Damit wird die erste Eigenschaft (**E1**) zur Konstruktion einer externen Sicht erfüllt.

Der zweite Beitrag (**B2**) dieser Arbeit ist eine initiale Untersuchung der abgeleiteten Änderungssequenzen aus den transformierten Feature-Modellen der FeatureIDE. Damit wird die Erfüllbarkeit der zweiten Eigenschaft untersucht (**E2**). Dazu wurden die als XML-Dateien persistierten Zustände der FeatureIDE-Modelle durch die in dieser Arbeit implementierte bidirektional M2T-Transformation in VaVe-Modelle transformiert. Aus diesen VaVe-Modellen wurden dann, mithilfe der ähnlichkeitsbasierten Ableitungsstrategie von Vitruvius, feingranulare Änderungssequenzen abgeleitet.

Diese Änderungssequenzen wurden bei der Evaluierung auf Korrektheit (Definition 5.3) und Redundanzfreiheit (Definition 5.5) untersucht. Zudem wurden die abgeleiteten Änderungssequenzen in den Evaluationsklassen von Thüm, Batory und Kästner [32] verglichen. Es wurden insgesamt 36 Änderungssequenzen aus neun FeatureIDE-Testmodellen mit jeweils vier veränderten FeatureIDE-Testmodellen abgeleitet. Jede Änderung entsprach dabei einer Evolutionsklasse von Thüm, Batory und Kästner [32]. Damit gab es jeweils neun modifizierte FeatureIDE-Modelle für alle vier Evolutionsklassen. Zudem wurden FeatureIDE-Testmodelle händisch in VaVe erstellt und der oben beschriebene Prozess auch für diese angewendet. Die Ergebnisse waren jedoch identisch.

Die Tests haben ergeben, dass die ähnlichkeitsbasierte Ableitungsstrategie von Vitruvius mithilfe der in dieser Arbeit implementierten Transformation in 94,44% der Fälle eine korrekte Änderungssequenz ableitet. Keine der korrekt abgeleiteten Änderungssequenzen sind dabei redundanzfrei. Der prozentuale Anteil der redundanten atomaren Änderungen in den korrekt abgeleiteten Änderungssequenzen beträgt insgesamt 29,12%. In den Evolutionsklassen Spezialisierung, Generalisierung und willkürliches Editieren sind die abgeleiteten Änderungssequenzen alle korrekt. Die abgeleiteten Änderungssequenzen in der Evolutionsklasse Refaktorisierung sind zu 77,78% korrekt. In allen vier Evolutionsklassen gibt es keine redundanzfreie Änderungssequenz.

Zusammenfassend zeigen die oben erwähnten Ergebnisse, dass die Ableitung feingranularer Änderungssequenzen aus Feature-Modellen der FeatureIDE mit der in dieser Arbeit

implementierten Transformation und mit Vitruvius möglich ist (**RQ1**). Damit ist auch die Anbindung von FeatureIDE durch eine externe Sicht an Vitruvius realisierbar (**RQ2**).

Da in dieser Arbeit die Erfüllbarkeit der zweiten Eigenschaft (**E2**) nur analysiert wurde, muss in zukünftigen Arbeiten an der Erfüllung der zweiten Eigenschaft gearbeitet werden, um die Konsistenzerhaltung von Feature-Modellen durch externe Sichten zu ermöglichen.

Literatur

[1] Vander Alves u. a. „Refactoring Product Lines“. In: *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*. GPCE '06. Portland, Oregon, USA: Association for Computing Machinery, 2006, S. 201–210. ISBN: 1595932372. DOI: 10.1145/1173706.1173737. URL: https://doi.org/10.1145/1173706.1173737.

[2] Sofia Ananieva. „Consistent Management of Variability in Space and Time“. In: *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume B*. New York, NY, USA: Association for Computing Machinery, 2021, S. 7–12. ISBN: 9781450384704. URL: https://doi.org/10.1145/3461002.3473067.

[3] Sofia Ananieva u. a. „Variants and Versions Management for Models with Integrated Consistency Preservation“. In: *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems*. VAMOS 2018. Madrid, Spain: Association for Computing Machinery, 2018, S. 3–10. ISBN: 9781450353984. DOI: 10.1145/3168365.3168377. URL: https://doi.org/10.1145/3168365.3168377.

[4] Sven Apel u. a. „A Development Process for Feature-Oriented Product Lines“. In: *Feature-Oriented Software Product Lines: Concepts and Implementation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, S. 17–44. ISBN: 978-3-642-37521-7. DOI: 10.1007/978-3-642-37521-7_2. URL: https://doi.org/10.1007/978-3-642-37521-7_2.

[5] Sven Apel u. a. *Feature-Oriented Software Product Lines*. Jan. 2013. ISBN: 978-3-642-37520-0. DOI: 10.1007/978-3-642-37521-7.

[6] Vincent Aranega, Anne Etien und Sebastien Mosser. „Using Feature Model to Build Model Transformation Chains“. In: *Model Driven Engineering Languages and Systems*. Hrsg. von Robert B. France u. a. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, S. 562–578. ISBN: 978-3-642-33666-9. DOI: 10.1007/978-3-642-33666-9_36.

[7] Colin Atkinson, Dietmar Stoll und Philipp Bostan. „Orthographic Software Modeling: A Practical Approach to View-Based Development“. In: *Evaluation of Novel Approaches to Software Engineering*. Hrsg. von Leszek A. Maciaszek, César González-Pérez und Stefan Jablonski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, S. 206–219. ISBN: 978-3-642-14819-4. DOI: 10.1007/978-3-642-14819-4_15.

[8] Petra Brosch u. a. „An Introduction to Model Versioning“. In: *Formal Methods for Model-Driven Engineering: 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures*. Hrsg. von Marco Bernardo, Vittorio Cortellessa und Alfonso Pierantonio. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, S. 336–

398. ISBN: 978-3-642-30982-3. DOI: 10.1007/978-3-642-30982-3_10. URL: https://doi.org/10.1007/978-3-642-30982-3_10.

[9] Hugo Bruneliere u. a. „A feature-based survey of model view approaches“. In: *Software & Systems Modeling* 18.3 (2019), S. 1931–1952. DOI: 10.1007/s10270-017-0622-9. URL: https://doi.org/10.1007/s10270-017-0622-9.

[10] Johannes Bürdek u. a. „Reasoning about product-line evolution using complex feature model differences“. In: *Automated Software Engineering* 23.4 (2015), S. 687–733. DOI: 10.1007/s10515-015-0185-3.

[11] Krzysztof Czarnecki und Simon Helsen. „Classification of model transformation approaches“. In: *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*. Bd. 45. 3. USA. 2003, S. 1–17.

[12] Krzysztof Czarnecki, Simon Helsen und Ulrich Eisenecker. „Staged configuration through specialization and multilevel configuration of feature models“. In: *Software Process: Improvement and Practice* 10 (Apr. 2005), S. 143–169. DOI: 10.1002/spip.225.

[13] Deepak Dhungana, Paul Grünbacher und Rick Rabiser. „The DOPLER Meta-Tool for Decision-Oriented Variability Modeling: A Multiple Case Study“. In: *Automated Software Engg.* 18.1 (März 2011), S. 77–114. ISSN: 0928-8910. DOI: 10.1007/s10515-010-0076-6. URL: https://doi.org/10.1007/s10515-010-0076-6.

[14] Zinovy Diskin u. a. „From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case“. In: *Model Driven Engineering Languages and Systems*. Hrsg. von Jon Whittle, Tony Clark und Thomas Kühne. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, S. 304–318. ISBN: 978-3-642-24485-8. DOI: 10.1007/978-3-642-24485-8_22.

[15] Kevin Feichtinger und Rick Rabiser. „How Flexible Must a Transformation Approach for Variability Models and Custom Variability Representations Be?“ In: *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume B*. New York, NY, USA: Association for Computing Machinery, 2021, S. 69–72. ISBN: 9781450384704. DOI: 10.1145/3461002.3473945. URL: https://doi.org/10.1145/3461002.3473945.

[16] Kevin Feichtinger und Rick Rabiser. „Towards Transforming Variability Models: Usage Scenarios, Required Capabilities and Challenges“. In: *Proceedings of the 24th ACM International Systems and Software Product Line Conference - Volume B*. SPLC '20. Montreal, QC, Canada: Association for Computing Machinery, 2020, S. 44–51. ISBN: 9781450375702. DOI: 10.1145/3382026.3425768. URL: https://doi.org/10.1145/3382026.3425768.

[17] Kevin Feichtinger u. a. „TRAVART: An Approach for Transforming Variability Models“. In: *15th International Working Conference on Variability Modelling of Software-Intensive Systems*. VaMoS'21. Krems, Austria: Association for Computing Machinery, 2021. ISBN: 9781450388245. DOI: 10.1145/3442391.3442400. URL: https://doi.org/10.1145/3442391.3442400.

[18] Holger Giese und Robert Wagner. „From model transformation to incremental bidirectional model synchronization“. In: *Software & Systems Modeling* 8.1 (2009), S. 21–43. DOI: https://doi.org/10.1007/s10270-008-0089-9.

[19] Adrian Hoff u. a. „Consistency-Preserving Evolution Planning on Feature Models“. In: *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A*. SPLC '20. Montreal, Quebec, Canada: Association for Computing Machinery, 2020. ISBN: 9781450375696. DOI: 10.1145/3382025.3414964. URL: https://doi.org/10.1145/3382025.3414964.

[20] Timo Kehrer u. a. „Adaptability of Model Comparison Tools“. In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ASE 2012. Essen, Germany: Association for Computing Machinery, 2012, S. 306–309. ISBN: 9781450312042. DOI: 10.1145/2351676.2351731. URL: https://doi.org/10.1145/2351676.2351731.

[21] Heiko Klare u. a. „Enabling consistency in view-based system development — The Vitruvius approach“. In: *Journal of Systems and Software* 171 (2021), S. 110815. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2020.110815. URL: https://www.sciencedirect.com/science/article/pii/S0164121220302144.

[22] Dimitrios S. Kolovos u. a. „Different models for model matching: An analysis of approaches to support model differencing“. In: *2009 ICSE Workshop on Comparison and Versioning of Software Models*. 2009, S. 1–6. DOI: 10.1109/CVSM.2009.5071714.

[23] Angelika Kusel u. a. „A survey on incremental model transformation approaches“. In: *ME 2013–Models and Evolution Workshop Proceedings*. Bd. 1090. Citeseer. 2013, S. 4–13.

[24] Johannes Meier u. a. „Single Underlying Models for Projectional, Multi-View Environments“. In: *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development*. MODELSWARD 2019. Prague, Czech Republic: SCITEPRESS - Science und Technology Publications, Lda, 2019, S. 117–128. ISBN: 9789897583582. DOI: 10.5220/0007396401170128. URL: https://doi.org/10.5220/0007396401170128.

[25] Tom Mens und Pieter Van Gorp. „A Taxonomy of Model Transformation“. In: *Electronic Notes in Theoretical Computer Science* 152 (2006). Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005), S. 125–142. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2005.10.021. URL: https://www.sciencedirect.com/science/article/pii/S1571066106001435.

[26] Klaus Pohl, Günter Böckle und Frank Van Der Linden. *Software product line engineering: foundations, principles, and techniques*. Bd. 1. Springer. ISBN: 978-3-540-28901-2.

[27] Klaus Pohl und Andreas Metzger. „Variability Management in Software Product Line Engineering“. In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. Shanghai, China: Association for Computing Machinery, 2006, S. 1049–1050. ISBN: 1595933751. DOI: 10.1145/1134285.1134499. URL: https://doi.org/10.1145/1134285.1134499.

[28] Timur Sağlam und Thomas Kühn. „Towards the Co-Evolution of Models and Artefacts of Industrial Tools Through External Views“. In: *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, Okt. 2021. DOI: 10.1109/models-c53483.2021.00064. URL: https://doi.org/10.1109%2Fmodels-c53483.2021.00064.

[29] B. Selic. „The pragmatics of model-driven development“. In: *IEEE Software* 20.5 (2003), S. 19–25. DOI: 10.1109/MS.2003.1231146.

[30] Thomas Stahl, Markus Völter und Krzysztof Czarnecki. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, Inc., 2006. ISBN: 0470025700.

[31] Dave Steinberg u. a. *EMF: eclipse modeling framework*. Pearson Education, 2008. ISBN: 9780-321-33188-5.

[32] Thomas Thüm, Don Batory und Christian Kästner. „Reasoning about Edits to Feature Models“. In: *Proceedings of the 31st International Conference on Software Engineering*. ICSE '09. USA: IEEE Computer Society, 2009, S. 254–264. ISBN: 9781424434534. DOI: 10.1109/ICSE.2009.5070526. URL: https://doi.org/10.1109/ICSE.2009.5070526.

[33] Thomas Thüm u. a. „FeatureIDE: An Extensible Framework for Feature-Oriented Software Development“. In: *Sci. Comput. Program.* 79 (Jan. 2014), S. 70–85. ISSN: 0167-6423. DOI: 10.1016/j.scico.2012.06.002. URL: https://doi.org/10.1016/j.scico.2012.06.002.

[34] Weslley Torres, Mark GJ Van den Brand und Alexander Serebrenik. „A systematic literature review of cross-domain model consistency checking by model management tools“. In: *Software and Systems Modeling* 20.3 (2021), S. 897–916. DOI: 10.1007/s10270-020-00834-1. URL: https://doi.org/10.1007/s10270-020-00834-1.

[35] Christian Tunjic und Colin Atkinson. „Synchronization of Projective Views on a Single-Underlying-Model“. In: *Proceedings of the 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-Based Software-Engineering*. MORSE/VAO '15. L'Aquila, Italy: Association for Computing Machinery, 2015, S. 55–58. ISBN: 9781450336147. DOI: 10.1145/2802059.2802066. URL: https://doi.org/10.1145/2802059.2802066.

[36] Jan Willem Wittler. „Derivation of Change Sequences from State-Based File Differences for Delta-Based Model Consistency“. Englisch. Magisterarb. Karlsruher Institut für Technologie (KIT), 2021. 89 S. DOI: 10.5445/IR/1000141972.

[37] XMLSpy. URL: https://www.altova.com/de/xmlspy-xml-editor (besucht am 18. 01. 2022).

[38] XMLUnit. URL: https://www.xmlunit.org/ (besucht am 18. 01. 2022).

A. Anhang

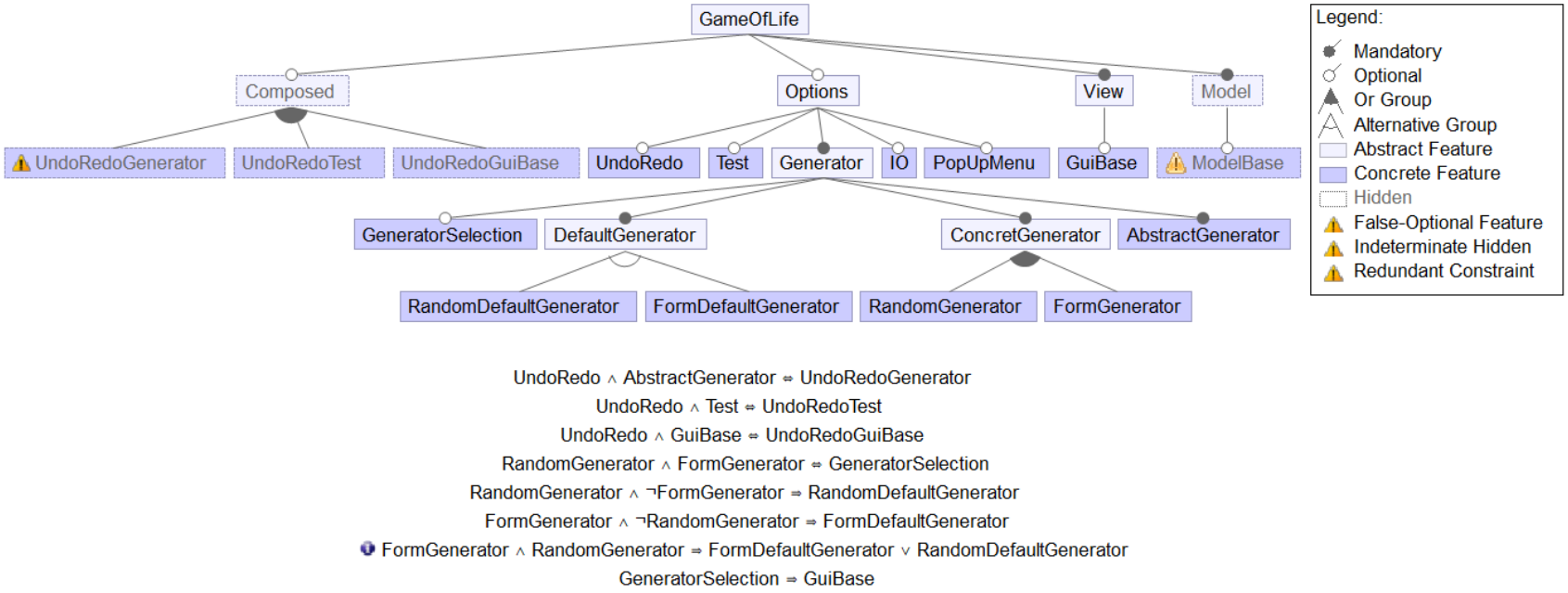


Abbildung A.1.: Feature-Modell der FeatureIDE mit 23 Features und 8 CTCs.

Listing A.1: Implementiertes XML-Schema

```
1
2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3    <xs:complexType name="unaryNodeType">
4       <xs:complexContent>
5          <xs:extension base="node">
6             <xs:sequence>
7                <xs:group ref="nodeGroup" maxOccurs="unbounded"/>
8             </xs:sequence>
9          </xs:extension>
10      </xs:complexContent>
11   </xs:complexType>
12   <xs:complexType name="leafType">
13      <xs:complexContent>
14         <xs:extension base="node"/>
15      </xs:complexContent>
16   </xs:complexType>
17   <xs:complexType name="binaryExtendedNodeType">
18      <xs:complexContent>
19         <xs:extension base="extendedNode">
20            <xs:sequence>
21               <xs:group ref="extendedNodeGroup" minOccurs="2"
                     maxOccurs="unbounded"/>
22            </xs:sequence>
23         </xs:extension>
24      </xs:complexContent>
25   </xs:complexType>
26   <xs:element name="extendedFeatureModel">
27      <xs:complexType>
28         <xs:sequence>
29            <xs:element ref="properties" minOccurs="0"/>
30            <xs:element name="struct">
31               <xs:complexType>
32                  <xs:group ref="extendedNodeGroup"/>
33               </xs:complexType>
34            </xs:element>
35            <xs:element ref="constraints" minOccurs="0"/>
36            <xs:element ref="calculations" minOccurs="0"/>
37            <xs:element ref="comments" minOccurs="0"/>
38            <xs:element ref="featureOrder" minOccurs="0"/>
39         </xs:sequence>
40      </xs:complexType>
41   </xs:element>
42   <xs:group name="extendedNodeGroup">
43      <xs:choice>
44         <xs:element name="feature" type="extendedLeafType"/>
45         <xs:element name="alt" type="binaryExtendedNodeType"/>
46         <xs:element name="or" type="binaryExtendedNodeType"/>
47         <xs:element name="and" type="unaryExtendedNodeType"/>
48      </xs:choice>
49   </xs:group>
50   <xs:complexType name="extendedLeafType">
51      <xs:complexContent>
```

```
52          <xs:extension base="extendedNode"/>
53       </xs:complexContent>
54    </xs:complexType>
55    <xs:complexType name="unaryExtendedNodeType">
56       <xs:complexContent>
57          <xs:extension base="extendedNode">
58             <xs:sequence>
59                <xs:group ref="extendedNodeGroup" maxOccurs="
                     unbounded"/>
60             </xs:sequence>
61          </xs:extension>
62       </xs:complexContent>
63    </xs:complexType>
64    <xs:complexType name="extendedNode">
65       <xs:complexContent>
66          <xs:extension base="node">
67             <xs:sequence>
68                <xs:element name="attribute" minOccurs="0" maxOccurs=
                     "unbounded"/>
69             </xs:sequence>
70          </xs:extension>
71       </xs:complexContent>
72    </xs:complexType>
73    <xs:complexType name="binaryNodeType">
74       <xs:complexContent>
75          <xs:extension base="node">
76             <xs:sequence>
77                <xs:group ref="nodeGroup" minOccurs="2" maxOccurs="
                     unbounded"/>
78             </xs:sequence>
79          </xs:extension>
80       </xs:complexContent>
81    </xs:complexType>
82    <xs:group name="nodeGroup">
83       <xs:sequence>
84          <xs:element ref="nodeList"/>
85       </xs:sequence>
86    </xs:group>
87    <xs:element name="c" type="xs:string"/>
88    <xs:element name="calculations">
89       <xs:complexType>
90          <xs:attribute name="key" type="xs:string" use="required"/>
91          <xs:attribute name="value" type="xs:boolean" use="required"
               />
92       </xs:complexType>
93    </xs:element>
94    <xs:element name="comments">
95       <xs:complexType>
96          <xs:sequence>
97             <xs:element ref="c" minOccurs="0" maxOccurs="unbounded"/
                  >
98          </xs:sequence>
99       </xs:complexType>
100   </xs:element>
```

```
101     <xs:element name="constraints">
102        <xs:complexType>
103           <xs:sequence>
104              <xs:element ref="rule" minOccurs="0" maxOccurs="
                    unbounded"/>
105           </xs:sequence>
106        </xs:complexType>
107     </xs:element>
108     <xs:element name="description" type="xs:string"/>
109     <xs:complexType name="disjType">
110        <xs:complexContent>
111           <xs:extension base="binaryExpressionType"/>
112        </xs:complexContent>
113     </xs:complexType>
114     <xs:element name="conj" type="conjType" substitutionGroup="
           expressionList"/>
115     <xs:complexType name="notType">
116        <xs:complexContent>
117           <xs:extension base="unaryExpressionType"/>
118        </xs:complexContent>
119     </xs:complexType>
120     <xs:element name="alt" type="altType" substitutionGroup="nodeList
           "/>
121     <xs:element name="or" type="orType" substitutionGroup="nodeList"/
           >
122     <xs:element name="and" type="andType" substitutionGroup="nodeList
           "/>
123     <xs:element name="feature" type="featureType" substitutionGroup="
           nodeList"/>
124     <xs:element name="eq" type="eqType" substitutionGroup="
           expressionList"/>
125     <xs:complexType name="impType">
126        <xs:complexContent>
127           <xs:extension base="binaryExpressionType"/>
128        </xs:complexContent>
129     </xs:complexType>
130     <xs:complexType name="altType">
131        <xs:complexContent>
132           <xs:extension base="binaryNodeType"/>
133        </xs:complexContent>
134     </xs:complexType>
135     <xs:complexType name="orType">
136        <xs:complexContent>
137           <xs:extension base="binaryNodeType"/>
138        </xs:complexContent>
139     </xs:complexType>
140     <xs:complexType name="andType">
141        <xs:complexContent>
142           <xs:extension base="unaryNodeType"/>
143        </xs:complexContent>
144     </xs:complexType>
145     <xs:complexType name="featureType">
146        <xs:complexContent>
147           <xs:extension base="leafType"/>
```

```
148         </xs:complexContent>
149     </xs:complexType>
150     <xs:element name="nodeList" type="node" abstract="true"/>
151     <xs:element name="var" abstract="false" substitutionGroup="
            expressionList">
152         <xs:complexType mixed="true">
153             <xs:complexContent>
154                 <xs:extension base="expression"/>
155             </xs:complexContent>
156         </xs:complexType>
157     </xs:element>
158     <xs:complexType name="eqType">
159         <xs:complexContent>
160             <xs:extension base="binaryExpressionType"/>
161         </xs:complexContent>
162     </xs:complexType>
163     <xs:element name="disj" type="disjType" substitutionGroup="
            expressionList"/>
164     <xs:element name="expressionList" type="expression" abstract="
            true"/>
165     <xs:group name="expressionGroup">
166         <xs:sequence>
167             <xs:element ref="expressionList"/>
168         </xs:sequence>
169     </xs:group>
170     <xs:complexType name="unaryExpressionType">
171         <xs:complexContent>
172             <xs:extension base="expression">
173                 <xs:sequence>
174                     <xs:group ref="expressionGroup"/>
175                 </xs:sequence>
176             </xs:extension>
177         </xs:complexContent>
178     </xs:complexType>
179     <xs:element name="not" type="notType" substitutionGroup="
            expressionList"/>
180     <xs:complexType name="conjType">
181         <xs:complexContent>
182             <xs:extension base="binaryExpressionType"/>
183         </xs:complexContent>
184     </xs:complexType>
185     <xs:element name="imp" type="impType" substitutionGroup="
            expressionList"/>
186     <xs:complexType name="binaryExpressionType">
187         <xs:complexContent>
188             <xs:extension base="expression">
189                 <xs:sequence>
190                     <xs:group ref="expressionGroup" minOccurs="2"
                            maxOccurs="2"/>
191                 </xs:sequence>
192             </xs:extension>
193         </xs:complexContent>
194     </xs:complexType>
195     <xs:complexType name="node">
```

```
196         <xs:sequence>
197             <xs:element ref="description" minOccurs="0"/>
198             <xs:element ref="graphics" minOccurs="0"/>
199         </xs:sequence>
200         <xs:attribute name="name" type="xs:string" use="required"/>
201         <xs:attribute name="abstract" type="xs:boolean"/>
202         <xs:attribute name="mandatory" type="xs:boolean"/>
203         <xs:attribute name="hidden" type="xs:boolean"/>
204     </xs:complexType>
205     <xs:element name="featureModel">
206         <xs:complexType>
207             <xs:sequence>
208                 <xs:element ref="properties" minOccurs="0"/>
209                 <xs:element name="struct">
210                     <xs:complexType>
211                         <xs:sequence>
212                         <xs:element ref="nodeList"/>
213                         </xs:sequence>
214                     </xs:complexType>
215                 </xs:element>
216                 <xs:element ref="constraints" minOccurs="0"/>
217                 <xs:element ref="calculations" minOccurs="0"/>
218                 <xs:element ref="comments" minOccurs="0"/>
219                 <xs:element ref="featureOrder" minOccurs="0"/>
220             </xs:sequence>
221         </xs:complexType>
222     </xs:element>
223     <xs:element name="featureOrder">
224         <xs:complexType>
225             <xs:sequence>
226                 <xs:element name="feature" type="leafType" minOccurs="0"
                        maxOccurs="unbounded"/>
227             </xs:sequence>
228             <xs:attribute name="userDefined" type="xs:boolean" use="
                    required"/>
229         </xs:complexType>
230     </xs:element>
231     <xs:element name="graphics">
232         <xs:complexType>
233             <xs:attribute name="key" type="xs:string" use="required"/>
234             <xs:attribute name="value" type="xs:string" use="required"/
                    >
235         </xs:complexType>
236     </xs:element>
237     <xs:element name="properties">
238         <xs:complexType>
239             <xs:choice minOccurs="0" maxOccurs="unbounded">
240                 <xs:element ref="graphics"/>
241                 <xs:element ref="calculations"/>
242             </xs:choice>
243         </xs:complexType>
244     </xs:element>
245     <xs:complexType name="expression" abstract="false"/>
246     <xs:element name="rule">
```

```
247         <xs:complexType>
248             <xs:sequence>
249                 <xs:element ref="description" minOccurs="0"/>
250                 <xs:sequence>
251                     <xs:group ref="expressionGroup"/>
252                 </xs:sequence>
253             </xs:sequence>
254         </xs:complexType>
255     </xs:element>
256 </xs:schema>
```