Universität Bayreuth
Fakultät für Mathematik, Physik und Informatik
Institut für Informatik
Lehrstuhl Angewandte Informatik I
Software Engineering

Master-Thesis

Mapping-basierte Modellierung von Softwareproduktlinien

Felix Schwägerl

30. Mai 2012

Prüfer:
Prof. Dr. Bernhard Westfechtel
Prof. Dr.-Ing. Stefan Jablonski

Zusammenfassung

Die *modellgetriebene Softwareentwicklung* erlaubt die Beschreibung von Softwaresystemen auf höherem Abstraktionsgrad. Neben der Dokumentation dienen Modelle der automatisierten Generierung von Quelltext in einer höheren Programmiersprache. Auf Programmiersprachen-Ebene erlauben Konstrukte wie Vererbung oder Typparametrisierung die *Wiederverwendung im Kleinen*. Adäquate Konstrukte stehen auf Modell-Ebene zur Verfügung.

Softwareproduktlinien beschreiben Gemeinsamkeiten und Unterschiede verwandter Software. Durch sie kann *Wiederverwendung im Großen* betrieben werden, um aus einer gemeinsamen Basis ähnliche Produkte zu erzeugen. Einzelne Produkte unterscheiden sich in der Implementierung spezifischer *Softwaremerkmale*, die in einem *Featuremodell* festgehalten werden. *Featurekonfigurationen* beschreiben hingegen deren Ausprägung je Produkt.

Die Kombination des Softwareproduktlinien-Ansatzes mit der modellgetriebenen Entwicklung ist keine neue Idee. Produkte werden hierbei durch Modelle repräsentiert. In dieser Arbeit wird der Sonderfall der *negativen Variabilität* betrachtet: Ein *Multivarianten-Domänenmodell* beinhaltet sämtliche Artefakte, die in Mitgliedern der Produktfamilie obligatorisch oder optional enthalten sein können. Ein Produkt entsteht durch das Löschen derjenigen Modell-Elemente, die nicht in seiner Konfiguration enthaltenen Features zugeordnet sind.

In der vorliegenden Master-Thesis wird ein Ansatz zur Abbildung von Elementen eines Multivarianten-Domänenmodells auf ein Featuremodell vorgestellt. Die Abbildung selbst wird vom Modellierer in einem sog. *Mapping-Modell* erzeugt. Es erlaubt die Annotation von Domänenmodell-Artefakten mit sog. *Feature-Ausdrücken*, welche sich wiederum auf das Featuremodell beziehen. Die Auswertung von Feature-Ausdrücken weist einem Mapping einen *Selektionszustand* zu. Die Arbeit liefert Beiträge in den folgenden vier Bereichen:

Konsistenz Selektionszustände voneinander abhängiger Mappings widersprechen sich unter bestimmten Voraussetzungen. Die hierbei entstehenden Inkonsistenzen werden nicht nur erkannt; in dieser Thesis ausgearbeitete Strategien wie die *Propagation* oder *Surrogate* erlauben die *automatische Reparatur* derselben. Für die Formulierung domänenspezifischer Abhängigkeitsbedingungen ist eine eigene Sprache vorgesehen.

Synchronität Feature- und Domänenmodell unterliegen einer kontinuierlichen *Evolution*. Durch sie können existierende Mappings ungültig werden. Gegenstand eines ausgearbeiteten Konzepts ist die weitestgehend automatische Synchronisation der Modelle, wobei ein möglicher *Informationsverlust* minimiert wird.

Agilität Die modellgetriebene Entwicklung von Softwareproduktlinien erfolgt nicht ausschließlich *plangetrieben*. Man will etwa produktspezifische Änderungen an einem existierenden Mapping-Modell vornehmen. Die eingeführten *Alternativen-Mappings* ermöglichen darüber hinaus eine konzeptionelle Erweiterung durch *positive Variabilität*.

Manifestation der Variabilität In dieser Thesis wird untersucht, inwieweit sich in Featuremodellen festgehaltene Variabilität auf Produkte niederschlagen kann. Hierbei wird die starre m:n-Beziehung zwischen Features und Domänenmodell-Artefakten aufgelöst, um neue Expressionsmittel wie *Attribut-Constraints* zur Verfügung zu stellen.

In einem vorbereitenden Abschnitt werden die erwähnten Aspekte zunächst theoretisch untersucht. Anschließend wird mit *F2DMM* eine Modellierungsumgebung für Softwareproduktlinien vorgestellt. Schließlich erfolgt eine Evaluierung anhand eines konstruierten Beispiels sowie eine Abgrenzung zu verwandten Ansätzen.

Abstract

Model driven software engineering allows for the description of software systems at a higher level of abstraction. Models are not only suitable for documentation, but also for an automated generation of source code in a modern programming language. At language level, constructs such as inheritance or type parametrization enable *reuse in the small*. Adequate constructs exist at modeling level.

Software product lines describe commonalities and differences of related software. They enable *reuse in the large* to derive similar products from a common software basis. Distinct products differ in the implementation of specific *software features* which can be recorded in a *feature model*. Contrastingly, a *feature configuration* describes a product's characteristics.

The combination of both the software product line approach and model driven software engineering is not a novel idea. Products are represented by models in this discipline. This thesis considers the special case of *negative variability*: a *multi-variant domain model* contains the complete set of optional or mandatory software artifacts which can occur in some or all of the members of a software family. Deriving a product means deleting model artifacts which are not assigned to any feature that is included in the feature configuration describing it.

The present Master thesis introduces an approach for mapping elements of a multi-variant domain model onto elements of a feature model. The mapping itself is created by the user in a so called *mapping model*. It allows for the annotation of domain model artifacts with *feature expressions* that refer to the feature model of the product line. Evaluating its feature expression, a distinct *selection state* is determined for each mapping. The thesis makes contributions in four areas:

Consistency Selection states of mutually depending mappings may be contradicting for some conditions, resulting in inconsistencies. Strategies elaborated in this thesis, e.g. *propagation* and *surrogates*, do not only identify them, but also allow for an *automatic repair*. A dedicated language has been developed to enable the phrasing of domain specific consistency conditions.

Synchronicity Feature and domain model are affected by continuous *evolution*, which can turn mappings into an invalid state. Another concept elaborated in this thesis is a mostly automatic synchronization of involved models, minimizing the potential *loss of information*.

Agility Model driven software product line engineering is not supposed to be purely plan-driven. Considering mapping models, product specific changes may be intended to occur lately. This requirement is covered by the concept of *alternative mappings* introduced to even support *positive variability*.

Manifestation of variability The thesis includes an investigation of how variability captured in feature models can influence products. Generally, features are supposed to be related to domain model artifacts by a *m*:*n* relationship only. New concepts try to enhance this by introducing flexible constructs such as *attribute constraints*.

A preparatory section covers the concepts described above in theory. Next, the software product line modeling environment *F2DMM* is introduced that implements these concepts. Finally, the approach is evaluated using an artificial case study with limited extent. The thesis is concluded by presenting related work.

Inhaltsverzeichnis

1 Einleitung

1.1 Modelle als Abstraktion von Softwaresystemen

Das Abstraktionsniveau bei der Entwicklung von Softwaresystemen steigt seit jeher an: In den 1960er Jahren wurden Assemblersprachen als vermeintlich menschenlesbare Metaphern für Maschinenbefehle entworfen. Die steigende Komplexität von zu entwickelnder Software erforderte jedoch schnell die Einführung einer weiteren Abstraktionsebene: Höhere Programmiersprachen lassen sich wiederum mit Hilfe von Compilern in Assemblerprogramme übersetzen.

Seit den 1980er Jahren vollzog sich bei der Entwicklung neuer Sprachen ein ständiger Paradigmenwechsel: Angefangen bei der strukturierten Programmierung, bis hin zu den heute gängigen Konzepten der objektorientierten oder funktionalen Programmierung. Eine Gemeinsamkeit aller in höheren Programmiersprachen formulierten Programme ist der zusätzliche Bedarf an technischer Dokumentation, welche den Sprung von der Konzept- auf die Implementierungsebene erleichtern soll.

Zur abstrakten Beschreibung von Anwendungssystemen haben sich im Bereich des Software Engineering verschiedene Diagrammarten etabliert. Die Sprache **UML** (*Unified Modeling Language*, vgl. Hitz und Kappel [32] im Literaturverzeichnis) vereint alle zur Dokumentation von Software notwendigen Diagrammarten in einer Diagrammfamilie. Standardisierte Diagramme dienen der Kommunikation von Ideen vor, bzw. der Beschreibung von erzeugter Software nach der Implementierung. Als Hauptvorteil von UML wird der gesteigerte Abstraktionsgrad gesehen. Während UML einen generellen Modellierungsansatz für Softwaresyteme darstellt, existieren zusätzlich eine Vielzahl *domänenspezifischer Sprachen*, die es erlauben, auf bestimmte Anwendungszwecke zugeschnittene Modelle zu entwerfen.

Die *modellgetriebene Softwareentwicklung* (**MDSE**) hat zum Ziel, Diagramme bzw. Modelle nicht weiterhin begleitend zur Programmierung eines Systems als zusätzliches, nur der Dokumentation dienendes Artefakt zu erzeugen, sondern sie präskriptiv einzusetzen, um Quelltext aus ihnen abzuleiten. Modelle werden hierbei als Abstraktion der Software auf höherer Ebene verstanden — sie verhalten sich also zum Quelltext wie der Quelltext zu einem entsprechenden Assemblerprogramm.

1.2 Konzepte der Wiederverwendung von Software

Software wird in den seltensten Fällen von Grund auf neu entwickelt: In größeren Projekten wird häufig von gemeinsamen *Bibliotheken* Gebrauch gemacht, die wiederverwendbare Softwareartefakte bereitstellen. Die *objektorientierte Programmierung* (**OOP**) in einer Sprache wie *Java* (s. Ullenboom [51]) bietet die Voraussetzungen hierfür in Form geeigneter Sprachkonstrukte.

1.2.1 Wiederverwendung im Kleinen: Spezialisierung und Typparametrisierung

Das zentrale Konzept der objektorientierten Programmierung sind *Klassen* (vgl. [51, Abschnitt 3]): Sie fassen *Attribute* und *Operationen* zusammen, um Eigenschaften und Verhaltensweisen einer Menge von *Objekten* zu beschreiben. Objekte wiederum lassen sich aus eine Klasse erzeugen (*instanziieren*); ihr Zustand wird durch die Werte ihrer Attribute bestimmt. Eine Klasse lässt sich folglich als *Bauplan* für Objekte desselben Typs betrachten. **OOP** erlaubt zusätzlich, Klassen miteinander in Beziehung zu setzen und sieht hierfür die Konstrukte *Spezialisierung* und *Typparametrisierung* vor, die im Folgenden erläutert werden:

Spezialisierung (auch *Vererbung*, vgl. [51, Abschnitt 5.8]): Klassen können bei ihrer Definition die Attribute und Operationen einer vorhandenen Klasse (*Oberklasse*) erweitern oder neu definieren. Vorhandener Quellcode kann auf diese Weise wiederverwendet werden. Instanzen der spezialisierten Klasse sind immer kompatibel mit ihrer Oberklasse. In Java wird die Spezialisierung durch das Schlüsselwort `extends` notiert (s. Quelltext 1.1). Auch **UML**-Klassendiagramme sehen für die Darstellung von Vererbungsbeziehungen spezielle Diagrammelemente vor (vgl. Abbildung 1.1).

```
1  public class Vehicle {
2      public double speed;
3  }
4
5  public class Car extends Vehicle {
6      public int doors;
7  }
```

Quelltext 1.1: Notation der Vererbung in Java.

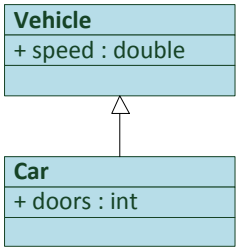

Abbildung 1.1: Darstellung von Vererbung in UML-Klassendiagrammen. Ein Pfeil mit flächiger Spitze stellt eine *Generalisierung* (Gegenrichtung der *Spezialisierung*) dar.

Typparametrisierung (auch *Generizität*, vgl. [51, Abschnitt 9]): Eine Klasse enthält optional einen oder mehrere *Typparameter*, die als Platzhalter für konkrete Typen fungieren. Bei der Instanziierung einer typparametrisierten Klasse in Java muss je ein konkreter Typ angegeben werden, der für das erzeugte Objekt den Platz eines Typparameters einnimmt (s. Quelltext 1.2). In der **UML** erfolgt hingegen das Binden von Typparametern an konkrete Typen zunächst auf Diagrammebene, bevor Instanzen erzeugt werden können (vgl. Abbildung 1.2).

```
1  public class List<T> {
2      protected T[] elements;
3      public T get(int i) { /*..*/ }
4      public void add(int i, T element) { /*..*/ }
5  }
6
7  public class ListTest {
8      public static void main(String[] args) {
9          List<Vehicle> vehicleList = new List<Vehicle>();
10         Car c = new Car();
11         c.speed = 180.0;
```

```
12          c.doors = 5;
13          vehicleList.add(c);
14          Vehicle first = vehicleList.get(0);
15      }
16  }
```

Quelltext 1.2: Deklaration einer parametrisierten Klasse `List` in Java. Der Typparameter `T` wird im Hauptprogramm (`main`) für `vehicleList` an die Klasse `Vehicle` gebunden.

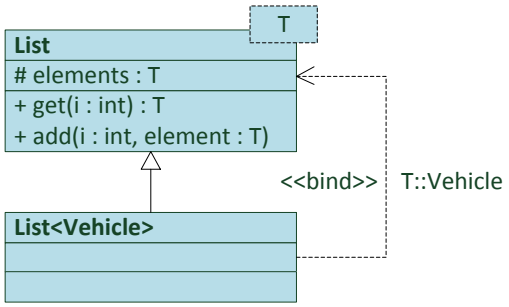

Abbildung 1.2: Parametrisierte Klassen werden in UML-Klassendiagrammen bereits auf Diagrammebene an konkrete Typen gebunden.

Durch geeignete Verwendung dieser beiden Konstrukte kann *Wiederverwendung im Kleinen* erzielt werden, um Redundanz zu vermeiden und die Wartbarkeit zu erhöhen. Typparametrisierung und Spezialisierung sind häufig Ergebnisse so genannter *Refactoring*-Schritte, in welchen existierende Software systematisch auf redundante Quelltextfragmente durchsucht wird, um Angriffspunkte für Konstrukte wie Vererbung oder Typparametrisierung zu identifizieren.

1.2.2 Wiederverwendung im Großen: Modularisierung und Komponenten

Bei der Entwicklung größerer Softwaresysteme, insbesondere bei der Zusammenarbeit in verteilten Entwicklerteams, spielt die *Modularisierung* von zu entwickelnder Software eine zunehmende Rolle [6, Kapitel 4.5]. Die wesentlichen Gründe für die Gliederung von Softwaresysteme in Module sind wie folgt:

- Trennung wiederverwendbarer Software in *Schnittstelle* und *Implementierung*: Lediglich die Schnittstelle eines Moduls ist nach außen hin sichtbar. Dies erleichtert die Wartung der nach außen nicht sichtbaren Implementierung eines Moduls und verringert die Komplexität aus Anwendersicht.
- *Austauschbarkeit* von Modulen: Mehrere Module können ähnliche Funktionalität unter derselben Schnittstelle implementieren. Für bestimmte Anwendungsfälle eignen sich austauschbare Module unterschiedlich gut. Das finale Softwareprodukt entsteht durch die Festlegung auf eines der austauschbaren Module.
- *Zerlegung* eines komplexen Problems in kleinere Teilprobleme: Dieses Grundprinzip der Informatik lässt sich auf der Entwicklungsebene ebenfalls durch die Verwendung von Modulen erzielen. Ein Modul implementiert hierbei die Lösung eines Teilproblems, wofür wiederum auf weitere Module zugegriffen werden kann.

Die **UML** sieht für die Beschreibung von Modulen keine eigene Diagrammart vor [32]. Vielmehr werden Module aus verschiedenen Perspektiven betrachtet: *Anwendungsfalldiagramme* beschreiben die Interaktion zwischen Systemkomponenten aus Anwendersicht. *Paketdiagramme* bieten einen Überblick über die Architektur eines Softwaresystems und beschreiben Abhängigkeiten zwischen *Paketen*. Für die Beschreibung der Modulschnittstelle selbst kommen *Klassendiagramme* in Frage, die auf die von außen sichtbaren Artefakte, reduziert werden können.

1.2.3 Organisierte Wiederverwendung: Produktlinien und Softwarefamilien

Während die genannten Konzepte der Wiederverwendung von Software im Kleinen (Spezialisierung und Typparametrisierung) sowie im Großen (Modularisierung und Komponenten) häufig *re-aktiv* als Folge von Refactoring-Schritten zum Einsatz kommen, zielt die Entwicklung von *Softwareproduktlinien* (**SPL**, [45]) auf die *pro-aktive* bzw. *organisierte* Wiederverwendung von Softwareartefakten ab: Die Absicht der Wiederverwendung soll im Vorhinein die Entwicklung wiederverwendbarer Komponenten bedingen, um sich ähnelnde Softwareprodukte mit geringem Aufwand produzieren zu können [5, Kapitel 1.1]. Im Folgenden werden einige Begriffe eingeführt, die in den weiteren Abschnitten dieser Arbeit genauer untersucht werden.

Softwareproduktlinie (**SPL**, auch *Softwarefamilie*) Eine Gruppe von sich strukturell sowie funktional ähnelnden Softwaresystemen, denen eine gemeinsame Quelltext-Basis zugrunde liegt.

Die strukturelle Ähnlichkeit von Softwareproduktlinien lässt sich beispielsweise durch den Vergleich von Klassendiagrammen, die deren Architektur beschreiben, feststellen [50]. Funktional ähneln sich Produkte, wenn sie einem ähnlichen Zweck dienen bzw. für die gleiche Zielgruppe entworfen wurden, sich jedoch in Details der Bedienung oder des Umfangs unterscheiden.

Die *modellgetriebene Softwareentwicklung* (**MDSE**, vgl. engl. *model driven software engineering*), wie sie in Abschnitt 2.1 dieser Arbeit definiert wird, verspricht eine Steigerung der Produktivität bei der Entwicklung von Softwaresystemen durch Erhöhung des Abstraktionsgrades sowie einer eventuellen Spezialisierung auf die Zieldomäne. Durch die Kombination der beiden Ansätze **MDSE** und **SPL** kann sich die erwartete Produktivitätssteigerung potenzieren [10].

Modellgetriebene Entwicklung von Softwareproduktlinien (**MDPLE**, vgl. engl. *model driven product line engineering*) Prozess der Erzeugung einer Softwareproduktlinie auf dem Abstraktionsniveau der modellgetriebenen Softwareentwicklung.

Auf die Grundlagen der **MDPLE** wird in Abschnitt 3 eingegangen: Sie sieht die Identifikation von *Softwaremerkmalen* einer **SPL** in sog. *Featuremodellen* vor; die Ausprägung dieser Merkmale ist in jeweils einer *Featurekonfiguration* pro Produkt festgehalten. Die gemeinsamen Komponenten der Softwarefamilie werden – betrachtet man den Spezialfall der *negativen Variabilität* – einem sog. *Multivarianten-Domänenmodell* festgehalten. Produkte entstehen durch das Entfallen von Elementen des Domänenmodells (**DM**), deren zugeordnete Features in der entsprechenden Konfiguration fehlen. Verschiedene **MDPLE**-Ansätze unterscheiden sich in der Art und Weise der *Abbildung* von Softwaremerkmalen auf Elemente des **DM**.

1.3 Beitrag der Arbeit unter bestimmten Aspekten

In dieser Arbeit wird mit **F2DMM** (s. Abschnitt 4) ein Werkzeug zur Abbildung (vgl. engl. *mapping*) von Features auf Domänenmodell-Elemente vorgestellt. Die Ausdrucksmächtigkeit dieser Abbildung unterscheidet sich von existierenden **MDPLE**-Ansätzen (vgl. Abschnitt 6) in einigen Aspekten, die im Folgenden definiert werden. Wie im vorhergehenden Abschnitt werden hierbei Begriffe eingeführt, die in den Abschnitten 3 und 4 ausführlicher behandelt werden. In Abschnitt 7.2 werden die formulierten Aspekte wieder aufgenommen, um den beschriebenen Beitrag mit konkreten ausgearbeiteten Konzepten zu belegen.

1.3.1 Konsistenz

Modelle beschreiben Softwaresysteme auf einem höheren Abstraktionsgrad. Ein Ziel der **MDSE** ist die Generierung von ausführbarem Quellcode aus Modellen. Hierfür ist die *Validität* von Modellen eine notwendige Voraussetzung: Ihre Struktur muss bestimmten Randbedingungen bzw. *Constraints* genügen.

Die Abbildung eines Feature- auf ein Domänenmodell muss demnach so erfolgen, dass jedes Produkt aus der Softwareproduktlinie valide ist. Im Folgenden wird eine Abbildung als *konsistent* bezeichnet, wenn diese Voraussetzung zutrifft. Hierbei gilt die Annahme, dass das Multivarianten-Domänenmodell, welches die gemeinsamen Software-Artefakte aller Produkte beschreibt, sich selbst in einem validen Zustand befindet.

In dieser Thesis werden mit *Propagationsstrategien* und *Surrogaten* zwei Konzepte eingeführt, die die automatische *Reparatur* inkonsistenter Modelle erlauben.

1.3.2 Synchronität

Feature- und Domänenmodell unterliegen einer kontinuierlichen *Evolution*: Die Gesamtheit wiederverwendbarer Softwarekomponenten wächst mit der Entwicklung von Produkten. Ebenso kann es zur Identifikation neuer Softwaremerkmale kommen. Beide Entwicklungen müssen in der Abbildung berücksichtigt werden, ohne bereits erzeugte Verknüpfungen zwischen den Modellen zu zerstören.

Die Anforderung der Synchronität erfordert ein Konzept zur Berücksichtigung der *Koevolution* der beteiligten Modelle. Dies betrifft unter anderem die Modellpaare Featuremodell/Featurekonfiguration und Featuremodell/Domänenmodell. Weiterhin sollte eine Synchronisation weitestgehend automatisch erfolgen und den Benutzer nur im Falle nicht auflösbarer Konflikte in die Wiederherstellung der Synchronität einbeziehen.

Der Beitrag dieser Arbeit unter diesem Aspekt besteht im Entwurf und in der Implementierung zweier Synchronisationsmodule für die erwähnten Modellpaare. Die Wiederherstellung der Synchronität erfolgt wenn möglich *automatisch*; nicht synchronisierbare Änderungen erfolgen benutzergesteuert.

1.3.3 Agilität

Der pro-aktive Charakter von Softwareproduktlinien wurde bereits beschrieben. **MDPLE** ist demnach eine Methode zur organisierten Wiederverwendung von Softwareartefakten. Im Zeitalter der *agilen Softwareentwicklung* [9] wird jedoch dem *Reagieren* auf sich ändernde Anforderungen eine zunehmende Bedeutung zugeschrieben.

Werkzeuge zur Modellierung von Softwareproduktlinien sollten demnach auch nachträgliche Änderungen im Entwurf von Softwaresystemen berücksichtigen können: Die Abbildung des Featuremodells auf das Domänenmodell sollte so flexibel sein, dass etwa Eigenschaften einzelner Produkte flexibel verändert werden können, ohne dass andere Mitglieder der Softwarefamilie davon betroffen sind.

Die in das in dieser Arbeit vorgestellte Werkzeug integrierten *Alternativen-Mappings* erlauben kurzfristige Änderungen in begrenztem Umfang, ohne sich auf das zugrundeliegende Domänenmodell auszuwirken.

1.3.4 Manifestation der Variabilität

Die Identifikation von Softwaremerkmalen ist der wesentliche Arbeitsschritt bei der Erzeugung des Featuremodells. Einem Merkmal entsprechende Artefakte des Multivarianten-Domänenmodells verhalten sich häufig orthogonal dazu: Ein Merkmal kann sich auf verschiedene Softwarekomponenten abbilden; umgekehrt kann ein Element aus dem Domänenmodell mehrere Features repräsentieren.

Wesentlich für eine adäquate Abbildungsvorschrift ist die Unterstützung verschiedener Arten der Abbildung von *Variabilität*, also den durch das Featuremodell dargestellten Gemeinsamkeiten und Unterschieden von Mitgliedern einer Softwarefamilie. Das Vorhandensein eines Softwaremerkmals in einer Featurekonfiguration kann sich dabei durch Eigenschaften äußern, die über das Vorhandensein bzw. Nichtvorhandensein eines Elements aus dem Multivarianten-Domänenmodell hinausgehen.

Der Begriff der *Manifestation der Variabilität* wird in diesem Kontext in der vorliegenden Thesis untersucht. Hierbei werden neue Möglichkeiten der Abbildung von Merkmalen erarbeitet, welche in F2DMM entsprechend umgesetzt wurden, etwa in Form von *mehrwertigen* Features oder *Attribut-Constraints*.

1.4 Aufbau der Arbeit

Die vorliegende Arbeit ist wie folgt gegliedert: In Abschnitt 2 werden einige Grundlagen zur *modellgetriebenen Softwareentwicklung* erläutert. Darüber hinaus sollen einige auf der *Eclipse*-Plattform basierende Werkzeuge und Rahmenwerke vorgestellt werden, die für den praktischen Teil der Arbeit relevant sind. Dies betrifft unter anderem das Modellierungsrahmenwerk **EMF**, sowie Werkzeuge zur textuellen Repräsentation, Validierung und Modell-zu-Modell-Transformationen (**M2M**).

Abschnitt 3 stellt einige Vorüberlegungen an, die zu Entwurfsentscheidungen im Hinblick auf das zu entwickelnde Werkzeug beigetragen haben. Hierzu werden zunächst einige Begriffe aus den Bereichen *modellgetriebene Entwicklung von Softwareproduktlinien* (**MDPLE**) formalisiert. Außerdem wird diskutiert, wie sich Abbildungen zwischen Feature- und Domänenmodellen äußern können und unter welchen Voraussetzungen die Konsistenz dieser Abbildungen sichergestellt werden kann. Zuletzt wird erörtert, inwiefern sich **MDPLE** als *Softwareentwicklungsprozess* definieren lässt.

Die Realisierung des Mapping-Editors **F2DMM** wird in Abschnitt 4 zunächst aus der konzeptionellen Perspektive betrachtet. Zur Abbildung von Merkmalen, die in Featuremodellen bzw. -konfigurationen beschrieben werden, auf Elemente eines Domänenmodells, über dessen Metamodell keine Annahmen zugrunde gelegt werden, wird eine textuelle Modellierungssprache (**FEL**) definiert. Eine weitere Sprache (**SDIRL**) erlaubt die Formulierung

domänenspezifischer *Abhängigkeitsbedingungen*, die zur Ableitung von *Reparaturaktionen* verwendet werden. In diesem Abschnitt werden auch Mechanismen zur Wiederherstellung der Konsistenz, der Synchronität sowie zur Unterstützung der Agilität erläutert. Schließlich wird der implementierte Editor aus der Anwenderperspektive betrachtet.

Im darauf folgenden Abschnitt 5 werden das Werkzeug und die zugrundeliegenden Konzepte anhand eines konstruierten Anwendungsbeispiels evaluiert. Hierbei wird die Abbildung eines exemplarischen Featuremodells und mehrerer -konfigurationen auf ein **UML2**-Domänenmodell beschrieben. Zunächst wird die Formulierung von Abhängigkeitsbedingungen betrachtet, um diese schließlich hinsichtlich der Auflösung von Konsistenzverletzungen zu untersuchen. Anschließend wird die Ableitung von Produkten aus dem Mapping-Modell thematisiert.

Abschnitt 6 ordnet die beschriebene Arbeit in den Kontext verwandter **MDPLE**-Ansätze ein und beleuchtet Gemeinsamkeiten und Unterschiede in Modellierungsansatz, Realisierung und Bedienung. Auch wird der Bezug zum vorausgehenden Lehrstuhlprojekt **MODPL** hergestellt. Eine Zusammenfassung und kritische Betrachtung der Arbeit sowie mögliche zukünftige Verbesserungsansätze sind Gegenstand des abschließenden Abschnitts 7.

1.5 Begleitende Ressourcen

Der dieser Arbeit beiliegende Datenträger enthält folgende Software-Ressourcen, die begleitend zu den Ausführungen der nachfolgenden Abschnitte eingesehen werden können:

- Im Wurzelverzeichnis befindet sich die vorliegende Ausarbeitung im PDF-Format. Abbildungen sowie LaTeX-Quelldateien können im Verzeichnis `thesis` eingesehen werden.
- Das Verzeichnis `literature` enthält einige der frei zugänglichen Artikel, die im Literaturverzeichnis referenziert werden, ebenfalls im PDF-Format.
- `eclipse` enthält eine *Eclipse*-Installation, in der sämtliche zum Nachvollziehen der Beispiele nötigen Plugins vorinstalliert sind. Zur Ausführung wird eine 64-Bit *Windows*-Installation sowie ein entsprechendes *Java*-SDK[1] vorausgesetzt. Der Inhalt des Verzeichnisses sollte vor der Ausführung auf ein lokales Laufwerk kopiert werden.
- `development-workspace` beinhaltet den Entwicklungs-Workspace, in welchem die in Abschnitt 4 beschriebenen Artefakte (Ecore-Metamodelle, Java-Quellcode und Konfigurationsdateien) in mehreren Eclipse-Projekten zusammengefasst sind. Dieser Workspace muss nicht zum Nachvollziehen der Beispiele geladen sein.
- Der Laufzeit-Workspace in Verzeichnis `runtime-workspace` beinhaltet die in Abschnitt 5 vorgestellten Beispiele sowie von diesen vorausgesetzte Artefakte, etwa das Domänenmodell. Das Verzeichnis sollte ebenfalls auf ein lokales Laufwerk kopiert werden, um ihn aus der bereitgestellten Eclipse-Installation zu öffnen.

[1] `http://www.oracle.com/technetwork/java/javase/downloads/jdk-7u4-downloads-1591156.html`

2 Modellgetriebene Softwareentwicklung im Eclipse-Kontext

In diesem Abschnitt werden Grundlagen und Werkzeuge der *modellgetriebenen Softwareentwicklung* (**MDSE**) vermittelt. Die *Object Management Group* (**OMG**) definiert einige Standards für Modelle und deren Repräsentation. Auf der Werkzeugseite stellt das *Eclipse Modeling Project* (**EMP**) den De-Facto-Standard sowohl für die Unterstützung bei der Werkzeugentwicklung für Modell-Editoren als auch als Plattform für diese dar. Nach der theoretischen Definition von **MDSE** in Unterabschnitt 2.1 werden das Modellierungsrahmenwerk **EMF** (s. Abschnitte 2.2 und 2.3) sowie das zugehörige *Validation Framework* (2.4) vorgestellt. Außerdem werden einige Rahmenwerke vorgestellt, die für die Entwicklung des in Abschnitt 4 beschriebenen entwickelten Werkzeugs relevant sind: *Xtext* (2.5) erlaubt die Definition textueller Repräsentationen für Modelle. Der **OMG**-Standard **OCL** (2.6) erlaubt die Formulierung von sog. *Queries*, um den Zustand von Teilmodellen abzufragen. Modell-zu-Modell-Transformationen sind Gegenstand von Abschnitt 2.7.

2.1 Grundlagen der modellgetriebenen Softwareentwicklung

2.1.1 Ziele

Wie schon in der Einleitung erwähnt, bezeichnet der Begriff *modellgetriebene Softwareentwicklung* den Prozess der Erzeugung ausführbarer Modelle, welche wiederum Abstraktionen von Softwaresystemen darstellen [20]. Die Beschreibung von Software auf einem höheren Abstraktionsniveau birgt eine Reihe von Vorteilen, die im Folgenden erläutert werden.

Fokus auf Entwurfsentscheidungen Während sich grundlegende Entscheidungen bezüglich der Architektur von Softwaresystemen nur schwer auf der Quellcode-Ebene formulieren lassen, bieten Diagramme als Repräsentationsform von abstrahierenden Modellen den hierzu geeigneten Auflösungs- und Detailgrad.

Identifikation von Risiken Viele Softwareentwicklungsprozesse [6] legen den Fokus auf eine frühestmögliche Identifikation und Elimination von Risiken bei der Entwicklung von Softwaresystemen. Durch einen Entwurf auf dem Abstraktionsniveau von Modellen kann die Gefahr konzeptioneller Fehler im Vorhinein gesenkt werden. Auch etwaigen Laufzeitrisiken wie zeit- oder sicherheitskritischer Datenverarbeitung kann durch adäquate Entwurfsentscheidungen frühzeitig begegnet werden.

Organisierte Wiederverwendung im Großen Die modellgetriebene Softwareentwicklung betrachtet Modelle als wiederverwendbare Artefakte. Wie eingangs dargestellt, stehen dem Entwickler auf Quelltextebene beispielsweise im Falle der objektorientierten Programmierung Konstrukte wie Vererbung oder Typparametrisierung zur *Wiederverwendung im Kleinen* zur Verfügung. *Wiederverwendung im Großen* kann auf Modellebene etwa durch Modularisierung erzielt werden (vgl. Abschnitt 1.2).

Dokumentation Modelle dienen der technischen Dokumentation von Softwaresystemen, wenn sie in geeignetem Auflösungs- und Detailgrad vorliegen. Auf diese Weise kann der Bedarf an nachträglicher Erzeugung von Dokumentationsartefakten gesenkt werden. Zudem entfällt die Gefahr des Veraltens von Dokumentation.

2.1.2 Modellierungsebenen

Voraussetzung für die Erzeugung wohlgeformter Modelle ist eine *Modellierungssprache*, die verwendbare Modell-Elemente definiert. In Analogie zu den formalen Sprachen [1, 52] kann eine solche durch die Menge verwendbarer Modell-Elemente in einer *Grammatik* definiert werden. Die Definition einer Grammatik kann wiederum durch ein Modell erfolgen: Ein konkretes, auf der zuvor definierten Modellsprache basierendes Modell wird somit zur Instanz eines *Metamodells* [7]. Abbildung 2.1 veranschaulicht diesen Zusammenhang: Auf Ebene **M1** werden konkrete Modelle beschrieben, beispielsweise UML-Klassendiagramme. Im (UML-)Metamodell (**M2**) werden verfügbare Modellierungskonstrukte beschrieben. Typischerweise werden in einem Metamodell verwendbare Elemente (*Klassen*), deren Eigenschaften (*Attribute*) und ihr Wertebereich, sowie mögliche Beziehungen zwischen Klassen (*Referenzen*) definiert.

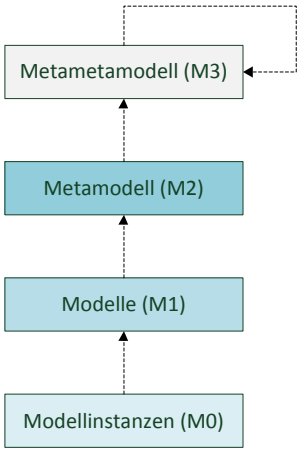

Abbildung 2.1: Modellierungsebenen: Ein Modell kann jeweils als Instanz eines Modells aus der nächsthöheren Ebene dargestellt werden. Modelle auf der M3-Ebene sind in der Lage, sich selbst zu beschreiben.

Die höchste Modellierungsebene, **M3**, erlaubt die Beschreibung eines *Metametamodells*, welches wiederum Elemente des Metamodells definiert. Es ist meist so formuliert, dass es in der Lage ist, sich selbst zu definieren; in der Theorie wären jedoch beliebig viele Meta-Ebenen erlaubt. Die *Object Management Group* (**OMG**) schlägt als **M3**-Modell den Standard **MOF** (*Meta Object Facility*, [42]) vor, in dem die kleinstmögliche Elementmenge für Metamodelle identifiziert wird. Der MOF-Standard wird vom *Ecore*-Metamodell (s. Abschnitt 2.2.1) implementiert.

Über die Bedeutung der Ebene **M0** existieren in der Modelltheorie unterschiedliche Annahmen [8]: Sie beinhaltet entweder die „Objekte der Realität“, welche durch das Modell repräsentiert werden, oder im Falle eines ausführbaren Modells die Gesamtheit aller vorhandenen Laufzeitobjekte, welche durch das Modell erzeugt wurden.

2.1.3 Abstrakte und konkrete Syntax

Bisher wurden die Begriffe „Modell“ und „Diagramm“ als Synonyme betrachtet. Tatsächlich stehen beide Begriffe in einer anderen Beziehung zueinander: Ein Diagramm ist eine mögliche, grafische *Repräsentation* eines Modells. Eine weitere Form der Repräsentation eines Modells ist beispielsweise die *textuelle Syntax*. Letztere hilft auch bei der Herstellung der Analogie zu

den erwähnten formalen Sprachen, bei denen man ebenfalls zwischen abstrakter und konkreter Syntax unterscheidet. Im Folgenden werden die Zusammenhänge anhand eines Beispiels erläutert. Es basiert auf einem Teil des **UML2**-Metamodells, der in Abbildung 2.2 eingeführt wird.

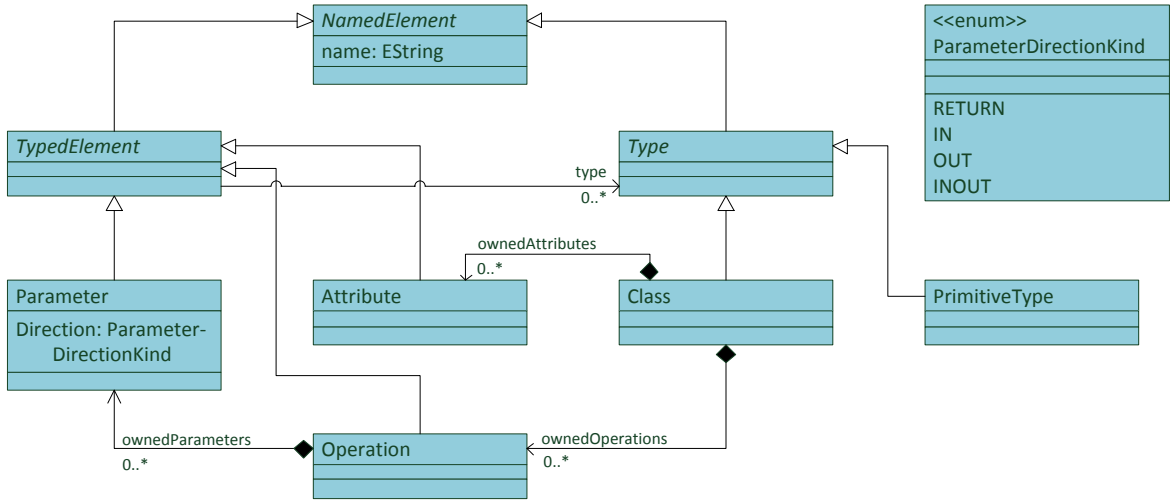

Abbildung 2.2: Ein auf die für die nachfolgenden Ausführungen relevanten Elemente reduzierter Auszug aus dem UML2-Metamodell. Es definiert eine Untermenge der in UML2-Klassendiagrammen verwendbaren Elemente: Klassen setzen sich aus Attributen und Operationen zusammen, letztere wiederum beinhalten Parameter. Die Metaklassen `Attribute` und `Parameter` erben die Referenz auf `Type` von der abstrakten Klasse `TypedElement`.

Die *Instanziierung* ermöglicht, aus einer Modellsprache, etwa einem Metamodell, ein Modell zu erzeugen. Sie beschreibt also den Übergang von einer Modellierungsebene auf die nächstniedrigere. In der objektorientierten Programmierung findet eine Instanziierung beispielsweise durch Aufruf eines Konstruktors statt: Aus einer Klasse wird ein Objekt als Instanz der Klasse abgeleitet. Analog dazu stellt das MOF-Objektdiagramm in Abbildung 2.3 eine Instanz des Klassendiagramms aus Abbildung 2.2 dar.

Das Objektdiagramm repräsentiert ein UML-Klassendiagramm[2] in *abstrakter Syntax*. Diese Art der Repräsentation entzieht sich jedoch häufig der Interpretierbarkeit innerhalb der Zieldomäne: Die grafische Repräsentation von Modellen soll, im Gegensatz zur logischen Repräsentation, vom Metamodell abhängen. Die Repräsentation eines Modells durch Diagramme bzw. Text wird durch die Definition einer *konkreten Syntax* (s. Abbildung 2.4 und Quellcode 2.1) erzielt.

Während Referenzen, also Beziehungen zwischen Modell-Elementen, in grafischer Notation durch entsprechende Verbindungslinien dargestellt werden können, ist das mehrfache Auftreten desselben Elements als Ziel mehrerer Referenzen in textueller Notation schwieriger darzustellen. Meist erfolgt der Rückbezug über einen eindeutigen *Bezeichner*. Unabhängig von der konkreten Repräsentation müssen folgende Arten des Auftretens von Objekten unterschieden werden:

[2] korrekterweise müsste man hier von einem Klassen*modell* sprechen. Im Kontext von UML-Modellen hat sich jedoch der Begriff Klassen*diagramm* auch für die abstrakte Repräsentation etabliert.

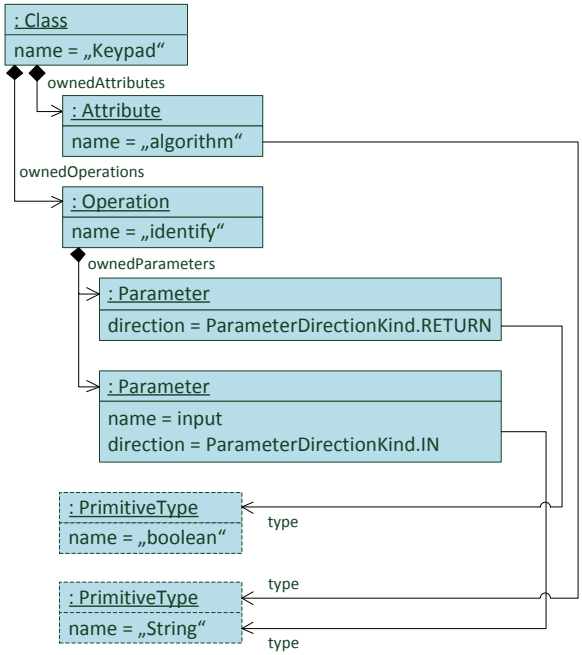

Abbildung 2.3: Beispiel für ein UML2-Modell als Instanz des UML2-Metamodells aus Abbildung 2.2, dargestellt als MOF-Objektdiagramm. Es stellt in abstrakter Syntax eine UML-Klasse `Keypad` dar, welche ein Attribut `algorithm` vom Typ `String` sowie eine Operation namens `identify` beinhaltet. Die über `ownedParameters` enthaltenen Operationsparameter definieren den Rückgabetyp (`boolean`) sowie den Formalparameter `input` vom Typ `String`.

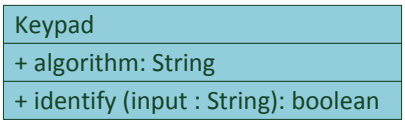

Abbildung 2.4: Darstellung des UML2-Modells aus Abbildung 2.3 in konkreter grafischer Syntax, definiert durch den UML2-Standard [44]. Das Symbol „+“ definiert eine Sichtbarkeit (`public`), welche aus Gründen der Übersichtlichkeit nicht im vorliegenden Ausschnitt des Metamodells berücksichtigt wurde.

Deklarierendes Auftreten Hierdurch wird ein Objekt in seiner Struktur definiert und innerhalb eines bestimmten *Sichtbarkeitsbereiches* als Ziel einer Referenz (in textueller Notation über einen eindeutigen Bezeichner) verfügbar gemacht.

Angewandtes Auftreten Es erfolgt Rückbezug auf ein durch ein deklarierendes Auftreten erzeugtes Objekt (in textueller Notation über dessen Bezeichner). So wird sein mehrfaches Auftreten als Referenzziel in unterschiedlichen Bereichen des Modells ermöglicht.

```
1 primitive String
2 primitive boolean
3
4 class Keypad {
5    property public algorithm : String
6    operation public identify (input : String) : boolean
7 }
```

Quelltext 2.1: Darstellung des UML2-Modells aus Abbildung 2.3 in einer konkreten, textuellen Notation. Der Text enthält dieselbe Information wie das Diagramm in Abbildung 2.4.

In Quelltext 2.1 findet das deklarierende Auftreten der Elemente `String` bzw. `boolean` innerhalb entsprechender `primitive`-Anweisungen statt. Im Sichtbarkeitsbereich des deklarierenden Auftretens sind diese beiden Elemente als Referenzziel verfügbar (s. Ende der Zeilen 6 und 7).

2.2 Das Eclipse Modeling Framework

2.2.1 Das Ecore-Metamodell

Das von der Eclipse Community[3] hervorgebrachte Modellierungsrahmenwerk *Eclipse Modeling Framework* (**EMF**) [48] stellt eine Java-basierte Plattform und Werkzeuge für die modellgetriebene Softwareentwicklung zur Verfügung. Das *Ecore*-Metamodell implementiert den von der **OMG** vorgeschlagenen **MOF**-Standard und unterstützt das **XML**-basierte Austauschformat **XMI** (*XML Metadata Interchange*) für Modellinstanzen. Ecore kann entweder auf **M2**-Ebene zur Modellierung von Klassendiagrammen, oder auf **M3**-Ebene zur Definition eines eigenen Metamodells für eine domänenspezifische Sprache verwendet werden. Im Rahmen des *Eclipse Modeling Project* (**EMP**) wird etwa ein Ecore-basiertes UML-Metamodell gepflegt. Im Folgenden wird auf Spezifika des Ecore-Metamodells eingegangen.

Hierarchisierung von Datentypen Abbildung 2.5 zeigt einen für die weiteren Ausführungen relevanten Auszug aus dem Ecore-Metamodell: Ineinander verschachtelte Pakete (repräsentiert durch `EPackage`) beinhalten wiederum Datentypen (`EClassifier`). Pakete werden global durch eine eindeutige *Package-URI* qualifiziert. Beispiele für Datentypen sind Klassen (`EClass`), die strukturelle Eigenschaften (`EStructuralFeature`) und Operationen (`EOperation`) beinhalten. Instanzen von `EDataType` bilden in Ecore primitive Java-Datentypen wie Integer oder String ab.

[3] `http://www.eclipse.org`

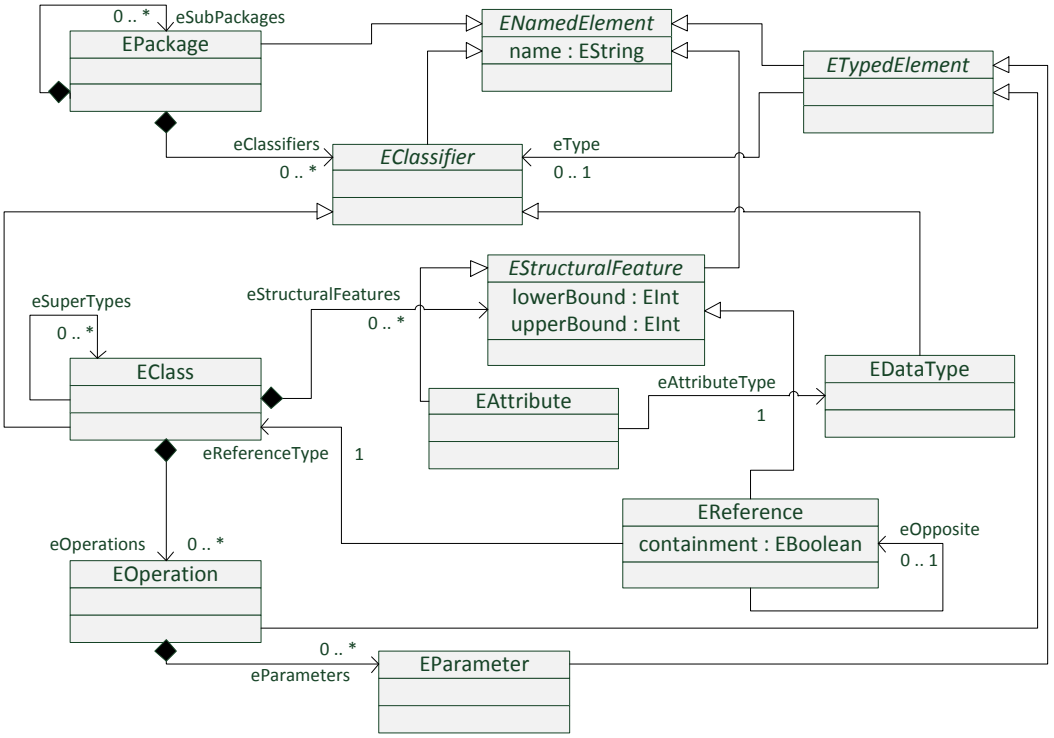

Abbildung 2.5: Das Ecore-Metamodell in Klassendiagramm-Notation (vereinfachte Darstellung) [48].

Strukturelle Eigenschaften Die konkreten Metaklassen `EAttribute` und `EReference` repräsentieren strukturelle Eigenschaften. Durch deren Attribute `lowerBound` und `upperBound` wird die *Wertigkeit* von Attributen bzw. Referenzen angegeben: Ist die Obergrenze größer eins, so bezeichnet man eine strukturelle Eigenschaft als *mehrwertig*. Während Attribute primitive Datentypen abbilden, sind die Typen von Referenzen immer Klassen. Ist das boolesche Attribut `containment` gesetzt, so handelt es sich um eine sog. *Containment-Referenz*: Referenzziele sind in diesem Fall *existenzabhängig* von ihrem `eContainer`. Unabhängig davon können bidirektionale Referenzen durch gegenseitiges Setzen von `eOpposite` eines Referenzpaars erzeugt werden.

Modellierung von Verhalten Analog zu den strukturellen Eigenschaften stellen Instanzen von `EOperation` die sog. *verhaltensbezogenen Eigenschaften* einer Klasse dar. Operationen beinhalten eine Menge von Parametern (`EParameter`), welche genauso wie die Operation selbst einen *Typ* haben (im Falle einer Operation der Rückgabetyp). Das Verhalten einer Methode selbst – deren *Ausführungssemantik* – kann nicht mit Hilfe des Ecore-Metamodells ausgedrückt werden. Stattdessen werden in generiertem Java-Quellcode (s. Abschnitt 2.2.2) leere Methodenrümpfe erzeugt, die manuell implementiert werden müssen, um dem Modell dynamisches Verhalten hinzuzufügen.

2.2.2 Generierung von Java-Quellcode

In der Einleitung wurde bereits erwähnt, dass die Modellierung auf einer höheren Abstraktionsebene wie die Programmierung in einer modernen Programmiersprache wie Java anzusiedeln ist. Steinberg u. a. [48, Abschnitt 2.2] behaupten jedoch:

> „To model or to program, that is not the question."

(„Modellieren oder Programmieren, das ist nicht die Frage.", Übers. d. Autors). Sie stützen diese Behauptung auf das Argument, **EMF** biete für jeden Zweck den geeigneten Abstraktionsgrad: Paket- und Klassenstrukturen eines Metamodells lassen sich leicht durch Ecore beschreiben. Das dynamische Verhalten hingegen sei durch Modelle weniger einfach zu spezifizieren als auf Quelltext-Ebene. Die **EMF**-Codegenerierung schlägt die Brücke zwischen Modellierung und Programmierung, indem sie die Übersetzung eines Ecore-Modells in Java-Quelltext ermöglicht. In diesem Abschnitt liegt der Fokus auf der Codegenerierung für durch ein Ecore-Modelle beschriebene Klassendiagramme. Der nachfolgende Abschnitt 2.3 befasst sich darüber hinaus mit der Generierung von *Editoren*.

Das Interface EObject Zur Laufzeit werden alle Objekte, die durch ein auf Ecore basierendes Metamodell erzeugt wurden, als Instanzen des Interface `EObject` repräsentiert. Die EMF-Codegenerierung sorgt bei der Abbildung von Ecore- auf Java-Klassen dafür, dass alle Klassen direkt oder indirekt von `EObject` erben. Diese erlaubt den *generischen* Zugriff auf jede Ecore-basierte Modellinstanz: Strukturelle Eigenschaften können etwa auch dann abgefragt oder gesetzt werden, wenn generierte Java-Klassen nicht bekannt sind[4]. `EObject` bietet auf Java-Ebene u.A. folgende Methoden an, die durch die EMF-Codegenerierung automatisch implementiert werden:

- `eClass()`: Liefert die das Objekt definierende `EClass` zurück und erlaubt so den *reflexiven* Zugriff auf das Metamodell.
- `eResource()`: Gibt die EMF-Ressource an, die das vorliegende Objekt beinhaltet. Ressourcen können beispielsweise in **XMI** serialisiert werden.
- `eContainer()`: Gibt dasjenige `EObject` an, von welchem das vorliegende Objekt unmittelbar existentiell abhängt[5].
- `eContents()`: Gibt diejenigen Instanzen von `EObject` zurück, die durch eine Containment-Referenz vom vorliegenden Objekt abgebildet werden.
- `eGet(EStructuralFeature)`: Liefert alle Objekte zurück, welche vom vorliegenden Objekt über eine gegebene strukturelle Eigenschaft abhängen. Diese können ein- oder mehrwertig, sowie durch primitive Datentypen oder Klassen definiert sein.
- `eSet(EStructuralFeature, Object)`: Setzt den durch die übergebene strukturelle Eigenschaft abgebildeten Wert neu.

[4] In der generischen Variante *Dynamic EMF* wird vollständig auf Codegenerierung verzichtet; alle Zugriffe geschehen über die generische Schnittstelle. Dadurch verbietet sich zunächst die Angabe von Ausführungssemantik, da keine ausfüllbaren Methodenrümpfe existieren.

[5] Die **MOF**-Spezifikation verlangt, dass dieses Objekt, falls vorhanden, eindeutig sein muss. Ansonsten ist es das Wurzelelement einer Ressource.

Klassen, Attribute und Referenzen Wie bereits erwähnt, werden Ecore-Klasen während der Codegenerierung auf Java-Klassen abgebildet. Genau genommen erfolgt die Abbildung auf ein Paar von Klasse und Interface, wobei die (Implementierungs-) Klasse jeweils das Suffix `Impl` erhält.

Attribute werden durch private Objektfelder mit dem entsprechenden Java-Datentyp repräsentiert; außerdem werden entsprechende Zugriffsmethoden (`get`/`set`) erzeugt. Mehrwertige Attribute werden in speziellen Datenstrukturen, jeweils von `java.util.Collection` erbend, abgelegt.

Referenzen werden ebenfalls durch private Felder mit entsprechenden Zugriffsmethoden repräsentiert. Einen Sonderfall stellen über das Setzen von `eOpposite` definierte *bidirektionale* Referenzen dar: Hier sorgt der generierte Quellcode für den Erhalt der Konsistenz beim Setzen der Referenzziele.

Operationen und geschützte Bereiche Wie bereits erwähnt, erzeugt die EMF-Codegenerierung für `EOperation`s lediglich leere Methodenrümpfe; lediglich Formal- und Rückgabeparameter werden aus dem Ecore-Modell übernommen. Die Implementierung des durch eine Operation beschriebenen Verhaltens nach der Codegenerierung ist Aufgabe des Modellierers, bzw. des Programmierers.

Die Generierung von Quellcode erfolgt nicht inkrementell, sondern vollständig für jeweils ein Ecore-Modell. Um manuell hinzugefügten Quellcode vor einem erneuten Überschreiben durch die Codegenerierung zu schützen, muss die *Javadoc*-Annotation `@generated NOT` vor dem entsprechenden Quelltextfragment eingefügt werden.

2.3 EMF-Baumeditoren

Die EMF-Codegenerierung unterstützt nicht nur die Übersetzung eines Ecore-Modells in Java-Quellcode, welcher das Modell repräsentiert, sondern auch die Erzeugung eines prototypischen Baumeditors, in dem Modellinstanzen in deren *abstrakter Syntax* eingesehen und manipuliert werden können. Die Generierung von Quellcode wird prinzipiell über ein sog. **EMF**-Generatormodell gesteuert, in dem die Codegenerierung zusätzlich angepasst werden kann (vgl. auch Abschnitt 2.3.3). Das Generatormodell erlaubt die Erzeugung von vier unterschiedlichen Eclipse-Plugins [48, Kapitel 2.4]:

Model Erzeugt das Datenmodell in seiner Java-Repräsentation, wie im vorhergehenden Unterabschnitt beschrieben.

Edit Generiert wiederverwendbaren, von der Benutzerschnittstelle unabhängigen Code, der Darstellung und Manipulation des von Modellen beschreibt (vgl. Abschnitt 2.3.1).

Editor Erzeugt einen prototypischen Baumeditor auf Basis der **GUI**-Bibliothek *JFace*[6], der den *Edit*-Quellcode wiederverwendet (vgl. Abschnitt 2.3.2).

Test Stellt einen Rahmen zur Implementierung von Testfällen zur Verfügung. Dieses Plugin ist für die weiteren Ausführungen nicht relevant.

[6] `http://wiki.eclipse.org/index.php/JFace`

2.3.1 Der UI-unabhängige Teil: Item-Provider

Das generierte *Edit*-Plugin stellt pro nicht-abstrakter Modellklasse einen sog. *Item-Provider* zur Verfügung. Dieser beschreibt die Darstellung und Manipulation von Modell-Elementen, ohne auf die Spezifika von Benutzeroberflächen Bezug zu nehmen. Die generierten `ItemProvider`-Klassen übernehmen für ihre entsprechenden Modellinstanzen mehrere Rollen, was sich jeweils durch Implementierung eines oder mehrerer Interfaces ausdrückt:

- `ItemProviderAdapter`: Erlaubt die Registrierung des *Item-Providers* an jeweils ein oder mehrere Instanzen einer Modellklasse über den Ecore-internen *Adapter*-Mechanismus[7].
- `Structured-` bzw. `TreeItemContentProvider`: Definiert die Methoden `getElements()` bzw. `getChildren()` und `getParents()`, die jeweils Kind- oder Eltern-Elemente der durch den Item-Provider beschriebenen Modellinstanz bestimmen. Diese Mechanismen werden beispielsweise von hierarchischen Baumeditoren genutzt.
- `IItemLabelProvider`: Verlangt die Implementierung von `getText()` und `getImage()`, die einen beschreibenden Text bzw. ein repräsentatives Symbol zurückgeben.
- `IItemPropertySource`: Stellt den Bezug zum sog. *Property Sheet* her, in dem Eigenschaften selektierter Objekte angezeigt bzw. manipuliert werden können. Der implementierende Item-Provider registriert sich hierfür beim Eclipse-weiten *Selection Service*.
- `IEditingDomainItemProvider`: Definiert Kind-Objekte, welche unter einem vom jeweiligen Item-Provider abgebildeten Element erstellt werden können und integriert den Item-Provider in das **EMF** *Command Framework*, welches die Durchführung elementarer Manipulations-Operationen steuert.

Zusätzlich wird pro Ecore-Modell eine Klasse mit dem Suffix `ItemProviderAdapter-Factory` generiert, die die Erzeugung eines geeigneten Item-Providers für eine Modellklasse delegiert. Dieser Mechanismus erlaubt die Wiederverwendung des *Edit*-Codes auch wenn die konkrete Item-Provider-Klasse nicht bekannt ist.

2.3.2 Der Editor als Benutzerschnittstelle

Der Baumeditor, der im Generator-Modell erzeugt werden kann, ist nicht als vollwertige Anwenderschnittstelle, sondern vielmehr als Prototyp zu betrachten, der demonstrieren soll, wie der generierte *Edit*-Code zur manuellen Implementierung eines – grafischen oder textuellen – Editors verwendet werden kann, etwa um die Unterstützung für konkrete Syntax zu implementieren. Abbildung 2.6 zeigt einen generierten Baumeditor ohne weitere Anpassungen[8].

Im Hauptfenster wird das Modell in seiner durch die Containment-Hierarchie definierten Baumstruktur angezeigt. Wird ein Eintrag ausgewählt, so werden dessen strukturelle Eigenschaften in der Properties-Ansicht angezeigt. Dort wird auch Unterstützung bei der Manipulation von Werten von Attributen und Zielen von Nicht-Containment-Referenzen gegeben. Die durch `IEditingDomainItemProvider` gesteuerte Erzeugung von Kind-Elementen wird über einen Rechtsklick auf das entsprechende Element im Baum erzielt. Auch das Löschen

[7] Das Entwurfsmuster *Adapter* ist in [21, Abschnitt 4.1] beschrieben.

[8] Der hier dargestellte generierte Editor entstammt einem frühen Prototypen des in Abschnitt 4 vorgestellten **F2DMM**-Editors.

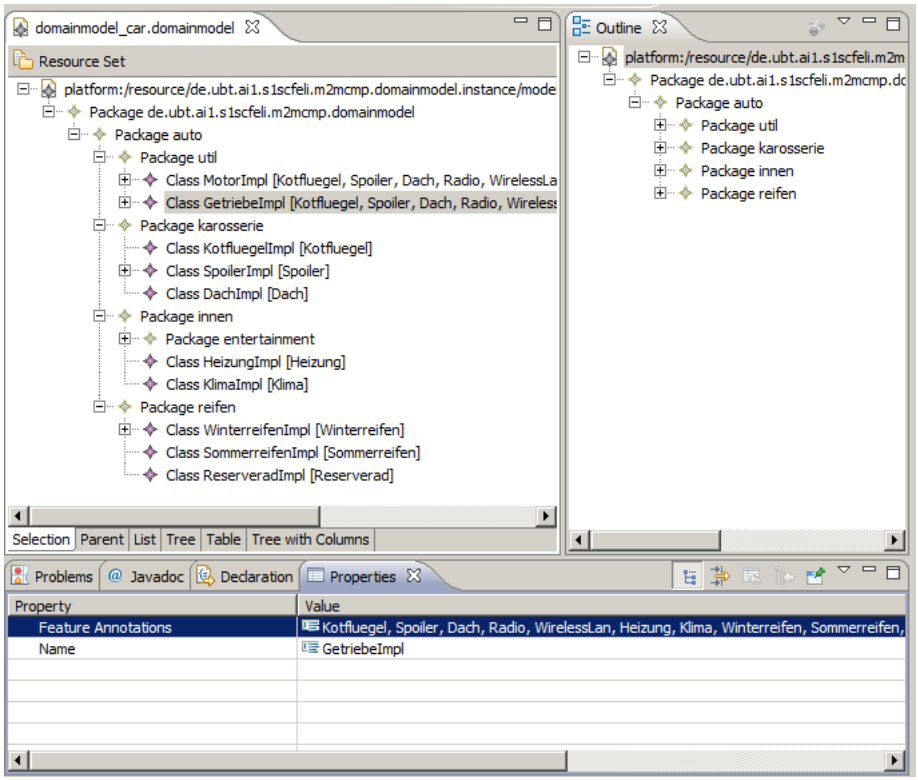

Abbildung 2.6: Beispiel für einen generierten EMF-Baumeditor.

von Elementen oder das Rückgängigmachen bzw. Wiederholen von Aktionen ist per Kontextmenü-Eintrag möglich.

2.3.3 Anpassungsmöglichkeiten

Im praktischen Teil der vorliegenden Masterarbeit wurde die Generierung von EMF-Baumeditoren genutzt, um diese im Nachhinein manuell anzupassen. **EMF** bietet an vielen Stellen die Möglichkeit, entweder in die Codegenerierung einzugreifen oder den generierten Quelltext zu manipulieren, um benutzerdefiniertes Editor-Verhalten zu erzielen. Im Folgenden sollen einige Anpassungsmöglichkeiten vorgestellt werden, die bei der Generierung und Implementierung der in Abschnitt 4 vorgestellten Baumeditoren Anwendung gefunden haben.

Anpassungen im Generatormodell Das **EMF**-Generatormodell (auch *Genmodel*) stellt Einstellungen zur Anpassung des zu generierenden Quelltextes zur Verfügung. Diese betreffen sowohl den Modell- als auch den *Edit*-Code. Während auf Modellebene etwa die Namen erzeugter Java-Pakete oder die Basisklasse (standardmäßig `EObject`) gewählt werden können, können für den UI-unabhängigen Teil folgende Einstellungen für die Darstellung jeder strukturellen Eigenschaft des definierten Metamodells getroffen werden:

- Children: Anzeige von Referenzzielen als Kind-Elemente im Baum.
- Create Child: Möglichkeit zur Erzeugung von Kind-Elementen per Kontextmenü-Eintrag.

- Property Category und `Property Description`: Kategorie bzw. Beschreibung der Eigenschaft im generierten Property-Sheet.
- Property Type: Die strukturelle Eigenschaft soll editierbar (Editable), lesbar (Readonly) oder ausgeblendet (None) sein.

Label und Icon Nach der Generierung können die von den entsprechenden `ItemProvider`-Klassen definierten Methoden `getText()` und `getImage()` überschrieben werden, um die Darstellung von Elementen im Baum zu beeinflussen.

Command-Framework Die von `IEditingDomainItemProvider` geerbten Methoden `createSetCommand`, `createCopyCommand` und `createRemoveCommand` können überschrieben werden, um beispielsweise benutzerdefinierte Konsistenzprüfungen vorzunehmen. Nicht ausführbare *Commands* werden durch die Klasse `UnexecutableCommand` repräsentiert, um beispielsweise das Löschen oder Kopieren von Elementen zu verhindern.

Plugin-Erweiterungspunkte Eclipse stellt einen Plugin-Mechanismus zur Verfügung, der die Implementierung spezifischer Plugin-Schnittstellen, sog. *Extension Points*, erlaubt. Auf diese Weise können etwa Aktionen für das Kontext-Menü, weitere Modell-Wizards, benutzerdefinierte Ansichten (*Views*) oder sog. *Preference Pages*, Workspace-weite Einstellungsmasken, deklarativ innerhalb der Datei `plugin.xml` definiert werden.

2.4 Das EMF Validation Framework

Das **EMF** *Validation Framework* [48, Kapitel 18] ist Teil des Eclipse Modeling Framework. Es stellt einen Rahmen zur *Validierung* von Instanzen Ecore-basierter Modelle zur Verfügung, die über die durch die Struktur des Metamodells definierten Bedingungen wie Typkonformität oder Multiplizität hinausgeht. Das Validation Framework unterscheidet zwei Arten von Bedingungen auf `EObject`-Instanzen:

Constraints Bedingungen, die zu einem bestimmten Zeitpunkt zutreffen müssen, etwa vor oder nach dem Setzen des Wertes eines Attributs.

Invarianten Bedingungen, die zu jedem Zeitpunkt zutreffen müssen.

2.4.1 Definition von Constraints und Invarianten

Das Plugin des Validation Framework sieht den *Extension Point* `org.eclipse.emf.validation.constraintProviders` zur Definition von Constraints und Invarianten vor. Ein *Constraint Provider* bezieht sich auf jeweils ein Ecore-Paket, das das Metamodell der zu validierenden Modelle beinhaltet. Die Definition von Constraints und Invarianten erfolgt zunächst in der Datei `plugin.xml` des Plugins, welches die Modellvalidierung enthält. Pro Constraint bzw. Invariante sind unter anderem folgende Eigenschaften anzugeben:

- Name Ein eindeutiger Bezeichner des Constraints bzw. der Invariante.
- Message Eine Fehlermeldung, die dem Benutzer bei Verletzung des Constraints angezeigt werden soll.

- Lang Die Sprache, in welcher die Bedingung formuliert ist. Derzeit werden vom Validation Framework *Java* und **OCL** (s. Abschnitt 2.6) unterstützt. Im Folgenden wird lediglich auf die Java-basierte Constraint-Definition eingegangen.
- Class Wurde Java als Constraint-Sprache angegeben, ist hier die jeweilige *Constraint-Klasse* anzugeben (s. unten).
- Severity Eine *Fehlerstufe* (*Info*, *Warning*, *Error* oder *Cancel*).
- Target Eine *Kontext-Klasse* des Metamodells, auf deren Instanzen die definierte Bedingung geprüft werden soll.
- Mode *Batch-* oder *Live*-Modus (s. nächster Unterabschnitt)

Bei der Java-basierten Constraint-Definition muss die entsprechende Constraint-Klasse von der in der Validation-**API** definierten abstrakten Klasse `AbstractModelConstraint` erben. Diese deklariert eine Objektmethode `validate(IValidationContext)` mit Rückgabetyp `IStatus`. In der Implementierung des Constraints kann die im Formalparameter zu extrahierende Kontext-Klasse auf Validität geprüft und ein entsprechender Fehlerstatus zurückgegeben werden (vgl. Quelltext 2.2).

```
1  public class NoSyntaxErrorsConstraint extends AbstractModelConstraint {
2
3      @Override
4      public IStatus validate(IValidationContext ctx) {
5          if (ctx.getTarget() instanceof Annotatable) {
6              Annotatable annotatable = (Annotatable) ctx.getTarget();
7              if (annotatable.isSetFeatureExprStr() &&
                   !annotatable.isSetFeatureExpr()) {
8                  return ctx.createFailureStatus(annotatable.getFeatureExprStr());
9              }
10         }
11         return ctx.createSuccessStatus();
12     }
13 }
```

Quelltext 2.2: Definition eines Java-basierten Constraints im **EMF** Validation Framework. Zur Erzeugung entsprechender `IStatus`-Rückgabewerte stellt `IValidationContext` Hilfsmethoden (`createFailureStatus` bzw. `createSuccessStatus`) zur Verfügung.

2.4.2 Batch- und Live-Validierung

Definierte Constraints und Invarianten können entweder in einem *Batch-* oder einem *Live-*Modus geprüft werden, sobald entsprechende Modelle in einem unterstützten Editor geöffnet wurden. Im Folgenden wird darauf eingegangen, durch welche Mechanismen die Prüfung der Bedingungen in beiden Modi erfolgt.

Batch-Modus Die Validierung wird über den im Kontextmenü vom Validation Framework erzeugten Eintrag Validate angestoßen. Der durch die Containment-Hierarchie definierte Modellbaum wird traversiert[9] und die im Batch-Modus befindlichen Constraints, falls deren

[9] Hierbei kann der Modellierer zwischen mehreren *Traversierungsstrategien* wählen oder eigene definieren.

Kontext-Klasse mit dem jeweiligen Element übereinstimmt, geprüft. Verletzungen werden in der Problems-Ansicht hinterlegt. Außerdem wird dem Benutzer eine Validierungs-Statistik angezeigt.

Live-Modus Hier erfolgt die Validierung *inkrementell*, nachdem das Modell durch den Benutzer manipuliert wurde. Hierzu müssen bei den im Constraint-Provider definierten Bedingungen entsprechende Validierungs-Auslöser definiert werden. Die Validierung eines Objekts kann beispielsweise nach dem Setzen einer bestimmten strukturellen Eigenschaft angestoßen werden. Verstöße werden ebenfalls in der Problems-Ansicht angezeigt.

Rückmeldung von Constraint-Verletzungen Pro Constraint-Verletzung wird in der Problems-Ansicht ein sog. *Marker* erzeugt, über den per Doppelklick zum betroffenen Kontext-Objekt navigiert werden kann (s. Abbildung 2.7).

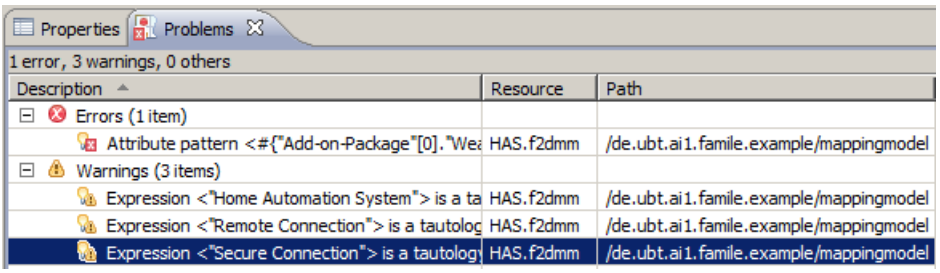

Abbildung 2.7: Die Problems-Ansicht der Eclipse-Workbench.

2.5 Textuelle Syntax mit Xtext

Xtext [19] ist ein Rahmenwerk zur Erzeugung einer konkreten textuellen Syntax für Ecore-basierte Modelle (vgl. Abschnitt 2.1.3). Die Syntax ist – wie bei den aus den formalen Sprachen [52] bekannten *kontextfreien Grammatiken* (**KFG**) – durch eine Menge von *Produktionsregeln* definiert.

2.5.1 Aufbau einer Xtext-Grammatik

Der Header Eine Xtext-Grammatik beginnt mit dem sog. *Header*, in dem auf das Schlüsselwort `grammar` ein Bezeichner für die Grammatik folgt. Die Wiederverwendung vorhandener Grammatiken ist durch Angabe derer Bezeichner in einem `with`-Statement möglich. Die Grammatik `Terminals` aus der Xtext-Standardbibliothek definiert etwa Terminalsymbole für elementare Ecore-Datentypen. Soll von der Grammatik zusätzlich ein Ecore-Modell generiert werden (s. nächster Unterabschnitt), so ist dies ebenfalls im Header notiert: Dem Schlüsselwort `generate` folgt die *Package-URI* des zu generierenden Modells.

Produktionsregeln Auf den Header folgt die Auflistung der die Grammatik definierenden Produktionsregeln. Per Konvention ist die erste Produktionsregel die *Startregel* und repräsentiert das Wurzel-Element eines in abstrakter Syntax vorliegenden Modells. Weitere Regeln definieren jeweils ein *Terminal-* oder *Nichtterminalsymbol* auf der *linken Seite* und die Regelanwendung auf der *rechten Seite*. Während Terminalsymbole durch das Schlüsselwort

`terminal` eingeleitet und auf Attribute des Metamodells abgebildet werden, entsprechen Nichtterminalsymbole ineinander über Containment-Beziehungen verschachtelte Objekten. Wie bei **KFG**s sind in einer Xtext-Grammatik folgende Konstrukte erlaubt:

- *Option*: Ausdruck kommt entweder vor oder nicht (gekennzeichnet durch das Suffix „`?`“).
- *Alternative*: Genau einer von mehreren Ausdrücken trifft zu (durch „`|`“ voneinander getrennt).
- *Wiederholung*: Ein Ausdruck kann mehrmals vorkommen (Suffix „`*`“, bzw. „`+`“ für mindestens einmalig).
- *Gruppierung*: Mehrere Ausdrücke können für die Anwendung der vorigen Konstrukte zusammengefasst werden (Einschluss des gruppierten Ausdrucks in runde Klammern).

Semantische Aktionen Während das Ergebnis einer Textanalyse durch eine **KFG** im Allgemeinen ein *Ableitungsbaum* ist, aus dem in einem weiteren Schritt ein *abstrakter Syntaxbaum* (**AST**) abgeleitet wird, beschreibt eine Xtext-Grammatik den direkten Aufbau von Modell-Instanzen in deren durch das Metamodell definierter abstrakter Syntax. Eine Produktionsregel entspricht einer gleichnamigen Klasse im Metamodell[10]. Der Aufbau der Containment-Hierarchie einer Modellinstanz wird durch sog. *semantische Aktionen* realisiert, die der Grammatik angefügt werden. Hierfür sind folgende Symbole reserviert:

- „`=`“: Einer einwertigen strukturelle Eigenschaft, deren Bezeichner links des Gleichheitszeichens steht, wird der Wert, der sich durch die Regelanwendung rechts neben dem Symbol ergibt, zugewiesen.
- „`+=`“: Einer mehrwertigen strukturellen Eigenschaft wird der ermittelte Wert angefügt.
- „`?=`“: Ein boolesches Attribut erhält den Wert `true`, falls sich die auf der rechten Seite stehende Option anwenden lässt, ansonsten `false`.

Angewandtes Auftreten Eine Xtext-Grammatik hat sowohl kontextfreie als auch *kontextsensitive* Bestandteile. Letztere werden immer dann benötigt, wenn beim Parsen der Textrepräsentation Nicht-Containment-Referenzen aufgebaut werden sollen. Wie in Abschnitt 2.1.3 beschrieben, muss zwischen *deklarierendem* und *angewandtem* Auftreten unterschieden werden: Referenzen entstehen durch einen *Rückbezug* auf ein bereits deklariertes Element, etwa durch dessen Bezeichner[11]. In der Grammatik wird ein solcher Rückbezug durch Einschluss des deklarierenden Auftretens in ein Paar eckiger Klammern (`[]`) realisiert.

Beispiel Quelltext 2.3 listet eine Xtext-Grammatik für ein Beispiel aus der offiziellen Dokumentation[12] auf.

[10] Abweichende Typen können durch eine `returns`-Anweisung innerhalb der linken Seite einer Regel angegeben werden.

[11] Die genauen Regeln für den Rückbezug sowie Sichtbarkeitsbereiche können im *Linking*-Modul (s. nächster Unterabschnitt) festgelegt werden.

[12] `http://www.eclipse.org/Xtext/documentation/2_1_0/020-domainmodel-step-by-step.php#DomainmodelWalkThrough_5`

```
1  grammar org.example.domainmodel.DomainModel
2  with org.eclipse.xtext.common.Terminals
3  generate domainModel "http://www.example.org/domainmodel/DomainModel"
4
5  Domainmodel : elements += Type* ;
6
7  Type : DataType | Entity ;
8
9  DataType : 'datatype' name = ID ;
10
11 Entity : 'entity' name = ID ('extends' superType = [Entity])? '{'
12     features += Feature* '}' ;
13
14 Feature: many?='many'? name = ID ':' type = [Type] ;
```

Quelltext 2.3: Beispiel für eine Xtext-Grammatik. Ein Domänenmodell (`Domainmodel`) setzt sich aus Datentypen und Entitäten zusammen, die jeweils einen Namen (`name`) haben. Supertypen können optional durch *angewandtes Auftreten* (`[Entity]`) angegeben werden. Eigenschaften von Entitäten (repräsentiert durch `Feature`), sind optional mehrwertig (`many?`) und referenzieren wiederum einen Typen durch angewandtes Auftreten.

2.5.2 Generierung von Laufzeitmodulen

Abbildung 2.8 stellt den gängigen Ablauf bei der Erzeugung eines Xtext-basierten Texteditors für Modelle dar: Aus einer Xtext-Grammatik werden sog. *Laufzeitmodule* generiert, die die Funktionalität des Texteditors in verschiedene Belange trennen. Die Konfiguration der Module erfolgt deklarativ in einem **MWE**-Workflow-Dokument (*Modeling Workflow Engine*)[13]. Die Erzeugung von Quelltext erfolgt, ähnlich wie bei der EMF-Codegenerierung, in verschiedene Eclipse-Plugins: Das *Laufzeitplugin* enthält die generierten wiederverwendbaren Laufzeitmodule, während das *UI*-Plugin Komponenten der Editor-Benutzerschnittstelle enthält. Im Folgenden werden die Funktionen einiger Laufzeitmodule erläutert [19, Abschnitt 5]:

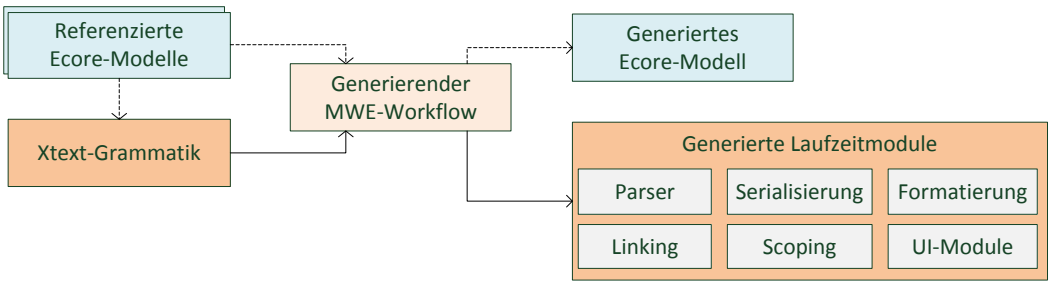

Abbildung 2.8: Erzeugung von Xtext-Laufzeitmodulen aus der Grammatik und einem **MWE**-Workflow.

Parser Erlaubt das Überführen eines textuell repräsentierten Modells in seine abstrakte Syntax (im Folgenden auch *Modellrepräsentation*).

[13] `http://wiki.eclipse.org/Modeling_Workflow_Engine_(MWE)`

Serialisierung Implementiert umgekehrt die (Wieder-)Erzeugung der Textrepräsentation eines Modells aus dessen Modellrepräsentation.

Formatierung Stellt automatische Formatierung (*Pretty Formatting*) der Textrepräsentation nach benutzerdefinierten Regeln zur Verfügung.

Linking Ermöglicht das *angewandte Auftreten* von Modell-Objekten, indem Regeln für das Referenzieren deklarierter Objekte aufgestellt werden. Das Standardverhalten sieht vor, dass Objekte durch ein Attribut namens `name` oder `id` qualifiziert werden.

Scoping Definiert den *Sichtbarkeitsbereich* deklarierter Objekte.

UI-Module Stellt erweiterte Editierunterstützung in Form eines Texteditors zur Verfügung. Dieser umfasst unter anderem *Syntax Highlighting*, automatische Quelltextvervollständigung (*Code Completion*), Reparaturvorschläge (*Quick Fixes*) und die Einbettung der Baumrepräsentation von Modellen in der Outline-Ansicht der *Eclipse*-Workbench.

2.5.3 Neuerzeugung oder Wiederverwendung eines Ecore-Modells

Xtext ist, wie bereits erwähnt, in der Lage, Ecore-Modelle aus einer Grammatik abzuleiten (vgl. Generiertes Ecore-Modell in Abbildung 2.8). Hierzu ist eine `generate`-Anweisung nötig. Die Generierung erfolgt jedoch nicht inkrementell; etwaige manuelle Änderungen am abgeleiteten Modell werden bei einer Neugenerierung überschrieben.

Um einem solchen Informationsverlust vorzubeugen, unterstützt Xtext den *Import* existierender Ecore-Modelle über eine gleichnamige Anweisung, auf welche die *Package-URI* eines existierenden Ecore-basierten Modells folgt (z.B. `import "http://www.eclipse.org/emf/2002/Ecore"`). Folgt man diesem Ansatz, wird das referenzierte Modell nicht manipuliert; die in den linken Seiten der Produktionsregeln definierten Symbole müssen aber als Klassennamen vorhanden sein, ebenso referenzierte strukturelle Eigenschaften.

Ein möglicher Hybrid-Ansatz sieht zunächst die Ableitung eines Ecore-Modells aus einer existierenden Xtext-Grammatik ab. Anschließend wird dieses per `import` (anstatt von `generate`) referenziert, um nachträglich manuelle Änderungen zu ermöglichen. Dieser Ansatz wurde auch in dieser Arbeit verfolgt (vgl. Abschnitte 4.4 und 4.5).

2.6 OCL - Object Constraint Language

Die textuelle Anfragesprache **OCL** (*Object Constraint Language*) [53] wurde ursprünglich als Teil der **UML**-Spezifikation verabschiedet und liegt mittlerweile in Version 2.2 vor [43]. Sie stellt Sprachkonstrukte zur Verfügung, die Anfragen auf Instanzen UML- oder Ecore-basierter Modelle erlauben. Der ursprüngliche Verwendungszweck – die Definition von *Constraints* – spielt inzwischen nicht mehr die übergeordnete Rolle: OCL wird innerhalb vieler textueller Sprachen, etwa zur Beschreibung von Modell-zu-Modell-Transformationen (vgl. Abschnitt 2.7) eingesetzt, um den *Zustand* von Modellen in deklarativer Weise abzufragen. OCL stellt keine Konstrukte für die Manipulation von Modellen zur Verfügung, ist also *seiteneffektfrei* [51, Abschnitt 5.1.2].

2.6.1 Essential OCL

Der **OCL**-Standard definiert für Anfragen auf Modellinstanzen die Sprache *Essential OCL*. Sie stellt Sprachkonstrukte zur Verfügung, die ausgehend von einem sog. *Kontext-Objekt* über strukturelle Eigenschaften navigieren können. Jede OCL-*Anfrage* (vgl. engl. *query*) hat einen definierten Rückgabewert. Auf eine Zusammenstellung aller Sprachkonstrukte von Essential OCL soll an dieser Stelle verzichtet werden; hierfür sei auf die entsprechende Hilfe-Seite von Eclipse verwiesen[14]. Stattdessen formuliert Quelltext 2.4 einige Beispiel-Anfragen auf dem in Abbildung 2.9 dargestellten UML-Objektdiagramm, um die gängigen Konstrukte zu demonstrieren. Kontext-Objekt ist hierbei die Instanz `e1`.

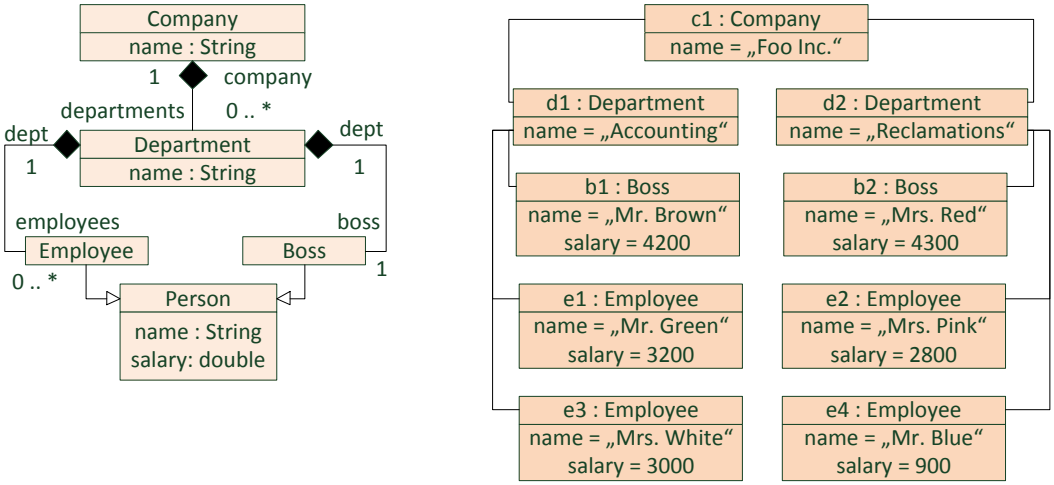

Abbildung 2.9: UML-Klassendiagramm und ein zugehöriges Objektdiagramm, welches die Daten für die folgenden OCL-Beispielanfragen definiert. Auf die Darstellung von Operationen wurde aus Platzgründen verzichtet.

```
1 self
2 dept
3 dept.employees
4 salary
5 dept.boss.salary
6 dept.employees->collect(e : Employee | e.name)
7 dept.company.departments.employees->select(e | e.salary >= 3000)
```

Quelltext 2.4: Beispiel-OCL-Anfragen auf das in Abbildung 2.9 eingeführte UML-Objektdiagramm.

Der Beispiel-Quelltext enthält eine OCL-Query je Zeile. Folgende Aufzählung erläutert die jeweils zugrundeliegenden Essential-OCL-Konstrukte und nennt die ermittelten Rückgabewerte.

1. Mit `self` wird auf das Kontext-Objekt selbst verwiesen, der Rückgabewert ist in diesem Fall also das Objekt `e1`.

2. Die Angabe des Bezeichners einer Referenz, in diesem Fall `dept`, liefert den durch sie

[14] `http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.ocl.doc%2Fhelp%2FEssentialOCL.html`

abgebildeten – unter Umständen auch mehrwertigen – Wert, im Beispiel `d1`. Auch die Schreibweise `self.dept` wäre hier möglich gewesen.

3. Referenzen lassen sich über Punkte zu *Pfaden* konkatenieren. In diesem Fall werden die Kanten `dept` und `employees` durchlaufen, um das mehrwertige Referenzziel, die Menge {`e1, e3`}, zu bestimmen.

4. Werte von Attributen lassen sich auf gleiche Weise wie Referenzziele bestimmen. `salary` wertet sich in diesem Fall zu dem Wert *3200* aus.

5. Auch zur Berechnung von Attributen können Pfade gebildet werden. Das Einkommen des Abteilungsleiters `b1` ist in diesem Beispiel *4200*.

6. Der *Mengenoperator* `collect` bindet Elemente (in diesem Fall `e`) einer Menge sequenziell an einen auszuwertenden Ausdruck (`e.name`) und liefert dessen Ergebnisse wiederum in einer Menge zurück. Das Ergebnis lautet hier {`Mr. Green, Mrs. White`}.

7. Der Mengenoperator `select` bindet wiederum Elemente einer Menge an einen OCL-Ausdruck, um diejenigen Elemente zu bestimmen, welche dem Ausdruck genügen. Werden in einem Pfad mehrere mehrwertige Referenzen durchlaufen, werden diese während der Auswertung zu einer Menge zusammengefasst (*implizite Iteration*).

Xtext-Unterstützung für OCL Essential OCL eignet sich zur Einbettung von OCL als Anfragesprache für domänenspezifische Sprachen. Hierfür wird vom *Eclipse-OCL*-Projekt[15] eine *Xtext*-Grammatik zur Verfügung gestellt, welche eine Wiederverwendung der Query-Konstrukte und deren Interpretation durch die Eclipse-OCL-Ausführungsumgebung erlaubt. Dieser Ansatz wurde – unter gewissen Einschränkungen – in der vorliegenden Arbeit verfolgt, um domänenspezifische *Constraints* auf Basis von OCL zu formulieren (s. Abschnitt 4.5.2).

2.6.2 Complete OCL

Der ursprüngliche Anwendungsbereich von OCL – die Definition von *Constraints* – wird von der Sprache *Complete OCL*, die Essential OCL erweitert, abgedeckt. Sie verwendet die von Essential OCL bereitgestellten Konstrukte zur Abfrage des Modellzustands, um die Formulierung von Constraints auf UML- oder Ecore-basierten Modellen zu erlauben. Das in Abschnitt 2.4 vorgestellte **EMF** *Validation Framework* bietet beispielsweise eine Complete-OCL-Schnittstelle zur deklarativen Formulierung von Validierungs-Constraints bzw. Invarianten an.

Complete OCL erweitert den Sprachumfang von Essential OCL an einigen Stellen. Queries können auf verschiedene Weisen mit einem Kontext-Objekt in Zusammenhang stehen: Für Operationen können etwa *Vor-* oder *Nachbedingungen* angegeben werden, für Klassen können *Invarianten* definiert werden. Die Werte von abgeleiteten Attributen oder Operationen können über OCL-Ausdrücke – unter der Annahme der Seiteneffektfreiheit – ermittelt werden. Die Syntax von Essential OCL ist für die weitere Arbeit nicht relevant.

[15] `http://www.eclipse.org/projects/project.php?id=modeling.mdt.ocl`

2.7 Modell-zu-Modell-Transformationen

Ein weiterer Bestandteil des Eclipse Modeling Project sind *Modell-zu-Modell-Transformationen* (**M2M**). Im Eclipse-Umfeld existieren viele M2M-Ansätze, die den gemeinsamen Zweck haben, ein *Quellmodell* nach einer bestimmten *Vorschrift* in ein *Zielmodell* übersetzen.

In-Place-Transformationen Einen Spezialfall stellen sog. *In-Place*-Transformationen dar: Anstatt ein neues Modell zu erzeugen, manipulieren sie ein vorhandenes. Anwendungsgebiet ist unter anderem die Modellierung von dynamischem Verhalten — etwa der modellbasierten Beschreibung der Semantik von in Ecore-Klassendiagrammen spezifizierten Operationen. Hierzu kommen häufig *Graphtransformationsregeln* bzw. *Graphersetzungsregeln* zum Einsatz (vgl. *ModGraph* [13] und *EMF Henshin* [4]). Diese Art der Modell-zu-Modell-Transformation ist jedoch für die weitere Arbeit nicht von Belang.

Out-Place-Transformationen Einen allgemeineren Fall stellen *Out-Place*-Modelltransformationen dar: Hierbei bleibt das Quellmodell während der Ausführung unverändert, stattdessen wird entsprechend einer Transformationsvorschrift ein Zielmodell neu erzeugt. Exogene Modell-zu-Modell-Transformationen kommen häufig zur Dokumentenkonvertierung zum Einsatz [37]. Verschiedene M2M-Ansätze stellen dem Anweder beim Formulieren der Transformationsvorschrift ein unterschiedlich hohes Abstraktionsniveau zur Verfügung. Weiterhin lassen sie sich in *imperative* und *deklarative* Ansätze einteilen.

2.7.1 Imperative Ansätze: Java, ATL und QVT Operational

Imperative **M2M**-Ansätze fassen eine Transformationsvorschrift als eine Sequenz von *Seiteneffekten* zum Aufbau eines Zielmodells auf. Dem Anwender werden hierfür Konstrukte wie die Erzeugung einer Modellklasse oder die Manipulation von Attributen oder Referenzzielen zur Verfügung gestellt. Zuweisungen, welche sich auf zwei korrespondierende Elementpaare beziehen, werden meist zu *Regeln* zusammengefasst, die sich gegenseitig aufrufen können. Für die Aufrufreihenfolge von Regeln ist der Modellierer selbst verantwortlich.

Java-basierte Transformationen Auf niedrigem Abstraktionsniveau befindet sich eine Transformationsbeschreibung in einer gängigen Programmiersprache wie Java: Eine mögliche Darstellung wäre eine Funktion mit der Signatur `public Zielmodell transformiere(Quellmodell)`. Steinberg u. a. [48, Abschnitt 16.4] beschreiben das *bedingte Kopieren* eines Modells als Sonderfall einer **M2M**-Transformation: Durch Spezialisierung der Klasse `EcoreUtil.Copier` kann der Kopiervorgang von Objekten, Attributen und Referenzen selektiv an Bedingungen geknüpft werden. Dieser Ansatz wird in Abschnitt 4.8 zur Ableitung von Produkten aus der Softwareproduktlinie verfolgt.

ATL Die von der *AtlanMod Group*[16] entwickelte Modelltransformationssprache **ATL** [2] ist derzeit der im Eclipse-Umfeld am weitesten verbreitete **M2M**-Ansatz. Die textuelle Sprache stellt sowohl imperative als auch deklarative Mittel zur Verfügung. ATL-Transformationen

[16] `http://www.emn.fr/z-info/atlanmod/index.php/Main_Page`

bestehen aus einer Menge von Regeln, die die Erzeugung von Zielmodell-Elementen aus korrespondierenden Elementen des Quellmodells beschreiben. Zusätzlich können für eine Transformation **OCL**-Hilfsfunktionen definiert werden, um Werte von Attributen oder Referenzen zu ermitteln. Quellcodeausschnitt 2.5 zeigt den Aufbau einer ATL-Transformationsregel.

```
1  lazy rule Class2Class {
2      from
3          featList : FEATURELIST!FeatureList,
4          pimClass : DOMAINMODEL!Class (
5              pimClass.isRequiredBy(featList)
6          )
7      to
8          psmClass : DOMAINMODEL!Class (
9              name <- pimClass.name,
10             attributes <- pimClass.transformedAttributes(featList)
11         )
12 }
```

Quelltext 2.5: Eine in **ATL** formulierte Transformationsregel. Quell- und Zielmodell werden direkt in der Transformationsvorschrift ausgezeichnet: Der `from`-Teil definiert den Kontext des Quellmodells. Im `to`-Teil werden Elemente des Zielmodells erzeugt und deren Attribute mit Hilfe des Zuweisungsoperators („`<-`“) initialisiert. Auf der rechten Seite können **OCL**-Ausdrücke zur Ermittlung von Attributwerten oder Referenzzielen verwendet werden.

QVT Operational Die **QVT**-Spezifikation (*Query View Transformation*) [41] wurde im Jahr 2005 als offener Standard zur Beschreibung von Modell-zu-Modell-Transformationen von der **OMG** verabschiedet. Sie umfasst mit QVT *Operational* [18] einen imperativen und mit QVT *Relations* [24] einen deklarativen Dialekt.

QVT Operational sieht Sprachkonstrukte für die Erzeugung und Manipulation eines Zielmodells aus einer Menge von Quellmodellen vor. Einstiegspunkt einer Transformation ist die Funktion `main()`. Meist wird in dieser Funktion genau ein *Mapping* aufgerufen, welches jeweils eine Menge von Manipulationsoperationen zusammenfasst. Mappings können einander referenzieren und **OCL**-Ausdrücke enthalten.

Obwohl es sich um einen OMG-Standard handelt, findet QVT Operational in der Eclipse-Community derzeit weniger Zuspruch als ATL[17].

2.7.2 Deklarative Ansätze: QVT Relations und Tripel-Graph-Grammatiken

Deklarative **M2M**-Ansätze basieren auf der Beschreibung der *Korrespondenz* von Elementen aus beteiligten Metamodellen [24]. Ebendiese abstrakte Beschreibung erlaubt die Ableitung der eigentlichen Transformation in beliebige Richtungen (Stichwort *Bidirektionalität*, [25]) aus einer *Korrespondenzbeschreibung*. Eine Gemeinsamkeit deklarativer **M2M**-Ansätze ist die Unterstützung verschiedener *Ausführungsmodi* beteiligter Modelldomänen[18]:

[17] Von den ersten 50 Beiträgen des Eclipse-Modeling-Forums beziehen sich 19 auf ATL und 9 auf QVT Operational. Stand: 21. Mai 2012, `http://www.eclipse.org/forums/index.php/f/23/`.

[18] Als *Domänen* werden im **M2M**-Kontext die jeweiligen Metamodelle von Quell- und Zielmodell bezeichnet.

Check-Only Ein Modell, das sich in diesem Modus befindet, wird während der Transformation nicht verändert. Es wird lediglich geprüft, ob bestimmte Muster auf Teile des Modells anwendbar sind. Ist dies der Fall, werden Korrespondenzen festgehalten und auf etwaige Entsprechungen in Modellen der anderen Domäne geprüft.

Enforce Dieser Modus erlaubt die Manipulation eines Modells, um die Korrespondenz mit anderen Modellen sicherzustellen. Eine gängige Strategie ist „*check before enforce*“: Zunächst wird wie beim Enforce-Modus auf Anwendbarkeit von Mustern geprüft. Treffen bestimmte Entsprechungen nicht zu, werden sie erzwungen (vgl. engl. für *to enforce*).

QVT Relations Der **OMG**-Standard *QVT Relations* [41] nimmt sich der deklarativen Beschreibung von Modell-zu-Modell-Transformationen an und definiert hierfür eine konkrete textuelle Syntax. Bei einem QVT-Relations-Dokument handelt es sich um eine Beschreibung von Beziehungen zwischen Modell-Elementen beliebig vieler Domänen. Den Hauptbestandteil stellen sog. *Relationen* dar, die miteinander korrespondierende Klassen auszeichnen. Innerhalb einer Relation können eine Reihe von *Relationsvariablen* definiert werden, um korrespondierende strukturelle Eigenschaften verschiedener Domänen zu identifizieren.

Ebenfalls innerhalb einer Relation findet sich ein *Domain-Pattern* pro beteiligtem Metamodell, gekennzeichnet durch das Schlüsselwort `domain`. Einem Domain-Pattern geht eines der Schlüsselwörter `checkonly` oder `enforce` voraus, wodurch der Ausführungsmodus pro Modell innerhalb der Relation explizit bestimmt wird. Ein Domain-Pattern kann – möglicherweise verschachtelte – *Pattern-Ausdrücke* enthalten, die Attribute oder Referenzen einer Domänenklasse mittels OCL-Ausdrücken mit Relationsvariablen vergleichen. Eine Relation ist für eine Menge von Domänenmodell-Elementen gültig, wenn alle Pattern-Ausdrücke von im Check-Only-Modus befindlichen Domänen als wahr evaluieren sowie die Werte beteiliger Relationsvariablen übereinstimmen. Ist dies der Fall, ist die Relation gültig und die Elemente von Enforce-Domänen werden entsprechend erzeugt.

Innerhalb einer Relation dürfen zusätzlich zwei Arten von Bedingungen formuliert werden: Während der `when`-Teil in **OCL** definierte Ausdrücke lediglich auswertet, und die Relation nicht zustande kommen lässt, sobald einer dieser Ausdrücke als falsch evaluiert, ist die Semantik des `where`-Teils restriktiver: Falls dessen Bedingungen nicht gelten, wird die Transformation abgebrochen. Ein Beispiel für eine QVT-Relations-Regel findet sich in Quelltext 2.6.

```
1  relation FeatureToFeature {
2
3      featureName : String;
4
5      checkonly domain absFM af : featuremodel::Feature {
6          name = featureName,
7          parent = afParent : featuremodel::FeatureGroup{}
8      };
9
10     checkonly domain confFM cf : featuremodel::Feature {
11         name = featureName,
12         parent = cfParent : featuremodel::FeatureGroup{}
13     };
14
15     enforce domain list fl : featurelist::FeatureList {
16         names = fn : featurelist::FeatureName {
17             name = featureName
```

```
18          }
19      };
20
21      when {
22          cf.state = FeatureState::active;
23          (TopFeatureGroupToTopFeatureGroup(afParent, cfParent, fl) or
                FeatureGroupToFeatureGroup(afParent, cfParent, fl));
24      }
25
26      where {
27          af.state->oclIsUndefined() or af.state = FeatureState::incomplete;
28          cf.state = FeatureState::active or cf.state = FeatureState::inactive;
29      }
30  }
```

Quelltext 2.6: Eine Relation aus einer in QVT Relations formulierte Transformationsvorschrift, welche Instanzen einer Modellklasse `Feature` auf Korrespondenz überprüft. Im `when`-Teil wird vorausgesetzt, dass übergeordnete `FeatureGroup`-Elemente bereits korrespondieren. Der `where`-Teil definiert eine notwendige Bedingung für Teile des Modells.

Tripel-Graph-Grammatiken Tripel-Graph-Grammatiken (**TGG**s) wurden 1994 von Schürr [47] als Ansatz zur deklarativen Beschreibung von Modell-zu-Modell-Transformationen vorgestellt. Sie beschreiben den Aufbau von *Tripel-Graphen* mit Hilfe einer Menge von *Produktionsregeln*. **TGG**s operieren grundsätzlich auf drei Domänen: Neben einem Quell- und einem Zielgraphen spielt der Aufbau des *Korrespondenzgraphen* die Rolle der Anwendung von Relationen in QVT Relations: Er referenziert korrespondierende Elemente.

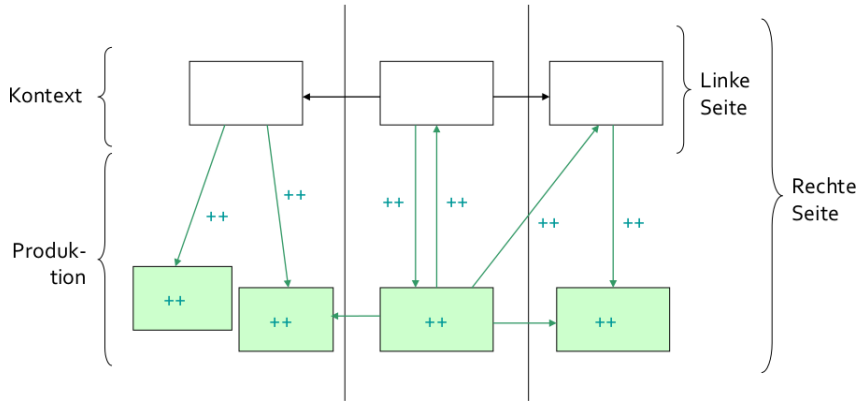

Abbildung 2.10: Produktionsregeln von TGGs erlauben die Beschreibung korrespondierender Elemente zweier Graphen in einer grafischen Notation.

Eine Produktionsregel beschreibt, wie Teilgraphen aller drei Domänen simultan erweitert werden können. Abbildung 2.10 beschreibt exemplarisch den Aufbau einer solchen: Die Notation unterscheidet zwischen *Kontext*knoten und -kanten, welche in weiß dargestellt werden, und *produzierten* Konten bzw. Kanten in grün und mit zusätzlicher Beschriftung „++“. Kontextelemente definieren ein sog. *Pattern*: Kommt diese Teilgraphenkonstellation während der Transformation auf bereits produzierten Elementen zustande, wird die Regel angewendet, wodurch alle produzierten Knoten wiederum gebunden (*Check-Only-Modus*) oder erzeugt (*Enforce-Modus*) werden.

Aufgrund ihres deklarativen Charakters eignen sich **TGG**s besonders für sog. *Synchronisationstransformationen*: Hierbei gilt die Annahme, dass bereits eine *Basistransformation* durch die Anwendung der TGG auf einen Graphen in eine bestimmte Transformationsrichtung stattgefunden hat. Die TGG bleibt auch dann anwendbar, wenn beide beteiligten Modelle seit der Basistransformation manipuliert wurden: Durch die erneute Ausführung werden Teilgraphen gebunden, wenn dies möglich ist, bzw. neu erzeugt, falls dies nötig ist. Spezielle Algorithmen befassen sich mit der Auflösung von Konflikten, falls bereits korrespondierende Elemente konkurrierenden Änderungen unterliegen [22, 26].

Modelltransformationen durch **TGG**s werden von verschiedenen Werkzeugen unterstützt, wie etwa *Fujaba*[19] oder dem **EMF**-basierten *TGG Interpreter*[20]. Bei letzterem Ansatz kann kann der Benutzer zur Formulierung von Korrespondenzbedingungen von **OCL** Gebrauch machen [24]. Außerdem existiert für Produktionsregeln ein Vererbungskonzept. Das Werkzeug kam in einem frühen Prototypen des Werkzeugs **F2DMM** zur Lösung des Abbildungsproblems zwischen Features und Artefakten eines Multivarianten-Domänenmodells zum Einsatz (vgl. Abschnitt 7.4.1).

[19] `http://www.fujaba.de`

[20] `http://www.cs.uni-paderborn.de/fachgebiete/fachgebiet-softwaretechnik/forschung/projekte/tgg-interpreter.html`

3 Vorüberlegungen und Formalisierung der Problemstellung

Die Disziplin *modellgetriebene Entwicklung von Softwareproduktlinien* (**MDPLE**, vgl. engl. *Model Driven Product Line Engineering*) vereint die in der Einleitung beschriebenen Konzepte der Abstraktion von Softwaresystemen durch Modelle und der organisierten Wiederverwendung durch Softwareproduktlinien (vgl. Abschnitte 1.1 und 1.2). Ein Beitrag der vorliegenden Arbeit liegt in der Ausarbeitung eines neuen, Mapping-basierten Modellierungsansatzes für diese Disziplin, um ein entsprechendes Werkzeug zu entwickeln, welches in Abschnitt 4 vorgestellt wird. Zuvor sollen einige Vorüberlegungen angestellt werden, welche zu Entwurfsentscheidungen hinsichtlich des erstellten Werkzeugs beigetragen haben. Außerdem soll in diesem Abschnitt eine Formalisierung der Problemstellung erfolgen, um den vorgestellten Ansatz mit verwandten Ansätzen vergleichen zu können (s. Abschnitt 6).

3.1 Modellierung von Softwaremerkmalen

3.1.1 Identifikation von Merkmalen der Variabilität

Im Alltag begegnen uns an vielen Stellen Phänomene, die sich unter dem Begriff *Variabilität* zusammen fassen lassen. Ein anschauliches Beispiel hierfür liefert die Automobilindustrie: Jeder PKW aus einer bestimmten Serie besitzt beispielsweise ein Schaltgetriebe. Er kann mit Sommer- oder Winterreifen, als Drei- oder Fünftürer ausgeliefert werden. Zur Sonderausstattung der Serie zählen optional elektrische Fensterheber, ein integriertes Navigationssystem oder eine Einparkhilfe. Der Bordcomputer kann je nach Verkaufsregion in unterschiedlichen Sprachen installiert sein.

Variabilität ist immer dann gegeben, wenn sich zwischen ähnlichen Objekten Gemeinsamkeiten oder Unterschiede erkennen lassen. Pohl u. a. [45, Abschnitt 4.2] unterscheiden im Rahmen der Formalisierung des Variabilitätsbegriffs zwischen

- dem *Subjekt der Variabilität*: „Ein variierbarer Gegenstand der realen Welt oder eine variierbare Eigenschaft eines Gegenstands“ [45, Definition 4-1], sowie
- dem *Objekt der Variabilität*: „Eine bestimmte Ausprägung eines Subjekts der Realität“ [45, Definition 4-2].

Das Subjekt der Realität ist gemäß dieser Definition als Antwort auf die Frage: „Was variiert?“ zu identifizieren; verschiedene Objekte der Variabilität können durch Beantwortung der Frage „Wie variiert es?“ ermittelt werden. Die Autoren empfehlen zudem zur Identifikation von Softwaremerkmalen die Frage „Warum variiert etwas?“, um etwaige *Dimensionen* der Variabilität eines Softwaremerkmals zu identifizieren: Diese können beispielsweise Größe, Farbe, Sprache, oder voneinander abweichende technische Voraussetzungen sein.

3.1.2 Merkmale der Variabilität

Vor der formellen Definition von Variabilität im Bezug auf Softwareproduktlinien bedarf es der Unterscheidung verschiedener *Merkmale der Variabilität*, die sich in der Art und Weise, wie sie sich in Produkten niederschlagen, unterscheiden können [56]:

Gemeinsamkeit Ein identifiziertes Softwaremerkmal ist obligatorischer Bestandteil eines jeden Produkts aus der Softwareproduktlinie.

Optionalität Das Softwaremerkmal kommt in einigen Produkten aus der Produktlinie vor, ist aber nicht obligatorisch.

Alternativität Ein Softwaremerkmal, das ein Subjekt der Realität beschreibt, kann durch verschiedene, sich gegenseitig ausschließende, Objekte der Realität repräsentiert werden. In jedem Produkt aus der Produktlinie muss genau eine Alternative aus einer vorgegebenen Menge ausgewählt werden. Das Subjekt der Realität wird auf Modellebene häufig als *Variationspunkt* bezeichnet[21].

Multiplizität Ein Softwaremerkmal kann sich in einem Produkt in mehreren Instanzen ausdrücken. In diesem Fall bildet ein Subjekt der Realität mehrere gleichartige Objekte der Realität ab.

Bezieht man sich zurück auf das oben genannte Beispiel aus der Automobilindustrie, stellt beispielsweise das Schaltgetriebe ein gemeinsames Merkmal dar. Das integrierte Navigationssystem ist optional, da es nur Bestandteil der Sonderausstattung ist. Die Merkmale Drei- bzw. Fünftürer stellen verschiedene Alternativen für den Variationspunkt „Anzahl der Türen“ dar und schließen sich gegenseitig aus; eine der beiden Alternativen muss jedoch gewählt werden, um ein vollständiges Produkt zu definieren. Als ein Beispiel für Multiplizität seien die vier jeweils gleichartigen Räder eines PKW zu nennen.

3.1.3 Softwaremerkmale und Abhängigkeiten: Das Featuremodell von Kang

Featuremodelle (**FM**, auch *Feature-Bäume*) sind ein Formalismus zur Erfassung identifizierter Softwaremerkmale und wurden von Kang u. a. [35] im Kontext der *Feature-orientierten Domänenanalyse* (**FODA**) begründet. Sie dienen der Beschreibung von Softwaremerkmalen (vgl. engl. *software features*) einer Produktfamilie und deren Variabilität. Featuremodelle sind baumartig strukturiert. Features stehen in einer der folgenden beiden Beziehungen mit ihrem unmittelbar beinhaltenden Eltern-Feature:

UND-Dekomposition Ist das Eltern-Feature in einem Produkt realisiert, so muss auch das Kind-Feature enthalten sein.

ODER-Dekomposition Höchstens ein Kind-Feature aus der Menge der mit dem Eltern-Feature in ODER-Beziehung stehenden Features darf in einem Produkt enthalten sein, in welchem das Eltern-Feature realisiert ist.

Entsprechend der obigen Definition lässt sich die *Gemeinsamkeit* von Softwaremerkmalen durch verschachtelte, UND-dekomponierte Teilbäume realisieren, während die *Alternativität* sowie die *Optionalität* mittels ODER-Dekomposition realisiert werden können.

Abbildung 3.1 stellt ein Featuremodell dar, welches das PKW-Beispiel in grafischer Notation im Sinne des von Kang u. a. [35] definierten FODA-Modells formalisiert. Die UND-Dekomposition wird hier durch eine durchgezogene, die ODER-Dekomposition durch eine gestrichelte Linie dargestellt.

[21] In [56] und weiterer Literatur wird anstatt *Alternativität* häufig der Terminus *Variation* verwendet.

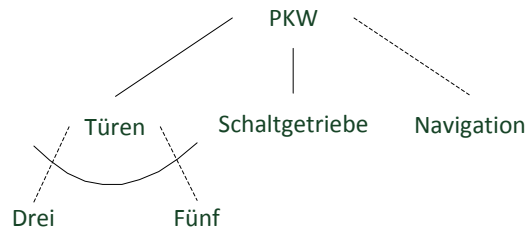

Abbildung 3.1: Beispiel-Featuremodell in der Notation von Kang u. a. [35].

3.1.4 Gruppierung und Kardinalität von Features

Das Featuremodell von Kang ist jedoch in seiner Ausdrucksmächtigkeit eingeschränkt: Festgehaltene Variablität kann nicht mit einer eindeutigen Semantik verknüpft werden. Pohl u. a. [45] merken an, dass der Unterschied zwischen Alternativität und Optionalität im Featuremodell aufgrund der Implikation „mindestens“ nicht explizit darstellbar ist. Außerdem wird das Merkmal der Multiplizität nicht berücksichtigt.

Weiterführende Ansätze versuchen diese Uneindeutigkeiten durch explizite Modellierung von *Feature-Gruppen* im Featuremodell zu lösen: Wie bereits in Abbildung 3.1 durch einen Verbindungsbogen dargestellt, können ODER-dekomponierte Features zusätzlich gruppiert werden. Aus dieser Gruppe muss *genau* ein (und nicht *mindestens* ein) Feature von einem konkreten Produkt realisiert werden (*Alternativität* im Sinne des gegenseitigen Ausschlusses). Nicht gruppierte, ODER-dekomponierte Features modellieren entsprechend die *Optionalität* und können je nach Produkt entweder realisiert oder nicht realisiert werden.

Eine weitere Generalisierung ist die von Czarnecki und Kim [16] hervorgebrachte *Kardinalitätsbasierte Featuremodellierung* (**CBFM**, vgl. engl. *Cardinality Based Feature Modeling*). Sie berücksichtigt das zuvor formulierte Variabilitätsmerkmal der *Multiplizität*. Ein Feature wird zusätzlich mit einem *Kardinalitäts-Intervall* versehen, welches die zugelassene Multiplizität eines Merkmals in einem Produkt definiert. Ist kein Intervall angegeben, so gilt $[0, 1]$ für optionale, $[1, 1]$ für obligatorische Features. Abbildung 3.2 bildet das Merkmal „Türen“ durch ein mehrfach instanziierbares Feature „Tür“ ab.

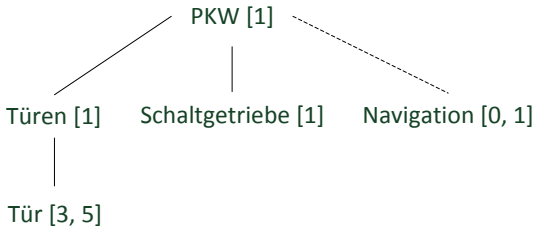

Abbildung 3.2: Das Featuremodell aus Abbildung 3.1 unter Berücksichtigung der **CBFM**.

Der von Antkiewicz und Czarnecki [3] ausgearbeitete Ansatz zur Featuremodellierung verallgemeinert die Konzepte der UND- bzw. ODER-Dekomposition durch die Möglichkeit der Angabe eines *Selektions-Intervalls*, welches die Anzahl der Unterfeatures einer Feature-Gruppe einschränkt, die in konkreten Produkten realisiert werden können[22]. Die UND-Dekomposition lässt sich folglich durch das Selektions-Intervall $[g, g]$ verallgemeinern, wobei g die Anzahl vorhandener Features einer Gruppe darstellt. Die ODER-Dekomposition einer Gruppe hingegen repräsentiert das Selektions-Intervall $[1, 1]$.

3.1.5 Feature-Konfigurationen: Elimination der Variabilität

Während Featuremodelle die Variabilität von Softwaremerkmalen der gesamten Produktlinie erfassen, beziehen sich *Featurekonfigurationen* (**FK**) auf die Ausprägung der erfassten Merkmale für ein spezielles Produkt. Der Prozess des Übergangs von einem Featuremodell zur beschreibenden Featurekonfiguration eines Produkts wird ebenfalls als *Konfiguration* bezeichnet.

Eine Featurekonfiguration umfasst die Menge der Entscheidungen, welche zur vollständigen Elimination der durch das entsprechende Featuremodell definierten Variabilität führen. Diese Elimination findet, bezogen auf die zuvor definierten Arten der Variablität, wie folgt statt:

Gemeinsamkeit Das Merkmal wird in die Featurekonfiguration des zu beschreibenden Produkts aufgenommen.

Optionalität Das Merkmal kann entweder in der FK enthalten sein oder nicht.

Alternativität Die das Produkt beschreibende FK muss genau eines der sich gegenseitig ausschließenden Merkmale enthalten.

Multiplizität Das Merkmal muss in einer festgelegten Anzahl von Instanzen, welche innerhalb des Kardinalitäts-Intervalls liegt, vorkommen. Zusätzlich müssen alle Selektions-Intervalle eingehalten werden.

3.1.6 Zusätzliche Parametrisierung durch Attribute

Zur Modellierung der *Alternativität* von Merkmalen eines Produkts existiert in vielen Ansätzen zusätzlich die Möglichkeit, Features mit sog. *Feature-Attributen* zu parametrisieren. Im Gegensatz zur Alternativen-Darstellung über eine ODER-Dekomposition ist der Wertebereich von Attributen nicht beschränkt, sondern auf endlichen oder nicht-endlichen Mengen definiert (z.B. die Menge aller natürlichen Zahlen $\mathbb{N}$). Ein Beispiel wäre die Modellierung eines Attributs „Höchstgeschwindigkeit“ des Features „PKW“ als reelle Zahl.

Bei der Erfassung der Variabilität im Featuremodell wird zunächst nur der Name des Attributs und dessen Wertebereich aufgenommen. Zur Elimination der Variabilität muss dem definierten Attribut in jeder Featurekonfiguration schließlich ein dem Wertebereich entsprechender Wert zugewiesen werden.

[22] Die Bezeichnungen beruhen auf der Arbeit von Lukas [39], die wiederum auf dem von Antkiewicz und Czarnecki [3] vorgestellten *FeaturePlugin* aufbaut. Die Autoren unterscheiden zwischen dem Kenngrößen *feature cardinality* und *group cardinality*, welche den hier verwendeten Begriffen Kardinalitäts- bzw. Selektions-Intervall entsprechen.

3.2 Manifestation der Variabilität im Domänenmodell

Im vorhergehenden Abschnitt wurden mit Featuremodellen und Featurekonfigurationen Formalismen zur Erfassung der Variabilität in Softwarefamilien beschrieben. Nun soll im Hinblick auf den praktischen Teil der Arbeit – die Entwicklung eines Mapping-gestützten Editors für modellgetriebene Softwareproduktlinien (**SPL**) – diskutiert werden, wie die in Featuremodell festgehaltene Variabilität auf ein Multivarianten-Domänenmodell (**DM**) abgebildet werden kann (vgl. den formulierten Aspekt *Manifestation der Variabilität* in Abschnitt 1.3.4). Dabei gilt die Annahme, dass jedes identifizierte Softwaremerkmal muss durch mindestens ein entsprechendes Element aus dem **DM** realisiert wird (vgl. Abbildung 3.3).

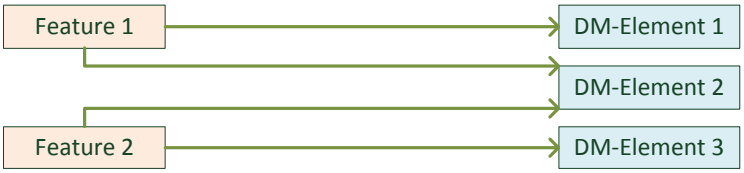

Abbildung 3.3: Die Abbildung zwischen Feature- und Domänenmodell erfolgt im Sinne einer *m*:*n*-Beziehung: Ein Feature kann mehrere **DM**-Elemente realisieren und **DM**-Element kann wiederum durch mehrere Features beschrieben werden.

3.2.1 Der Kernel: Die Menge aller gemeinsamen Elemente

Domänenmodell-Elemente, welche von *gemeinsamen* Softwaremerkmalen, also solchen, die in jeder Featurekonfiguration ausgewählt sein müssen, abgebildet werden, sind folglich in jedem Produkt enthalten. Das dadurch entstehende Gerüst nennt man auch den *Kernel* des Multivarianten-Domänenmodells. Umgekehrt werden gemeinsame Softwaremerkmale auch als *Kernel-Features* bezeichnet.

3.2.2 Optionale Elemente in Abhängigkeit von der Featurekonfiguration

Im Featuremodell als *optional* identifizierte Softwaremerkmale bilden DM-Elemente ab, die lediglich in bestimmten Produkten der Softwarefamilie enthalten sind. Die Abbildung muss hierbei sicherstellen, dass sich die existenzielle Abhängigkeit des optionalen Features von seinem unmittelbar beinhaltenden Feature im Domänenmodell wiederfindet.

Abbildung 3.4: Abbildung von optionalen Features auf existenzabhängige DM-Elemente. Das Entfallen eines solchen Elements verletzt nicht die Wohlgeformtheit des **DM**.

Abbildung 3.4 zeigt die Abbildung eines optionalen Softwaremerkmals (Feature 4) auf ein Domänenmodell-Element (DME 5). In diesem Beispiel ist DME 5 auch existenzabhängig von DME 4, das wiederum von dem Eltern-Feature 3 des optionalen Features abgebildet wird.

3.2.3 Variationspunkte durch sich gegenseitig ausschließende Alternativen

Das Variabilitätsmerkmal *Alternativität* kann ähnlich wie die zuvor beschriebene *Optionalität* auf das Domänenmodell abgebildet werden: Die möglichen Alternativen sind in nunmehr einer Gruppe zusammengefasst und bilden je ein DM-Element ab. Diese sind jeweils existenzabhängig von ihrem übergeordneten Element, welches wiederum vom Eltern-Feature abgebildet wird.

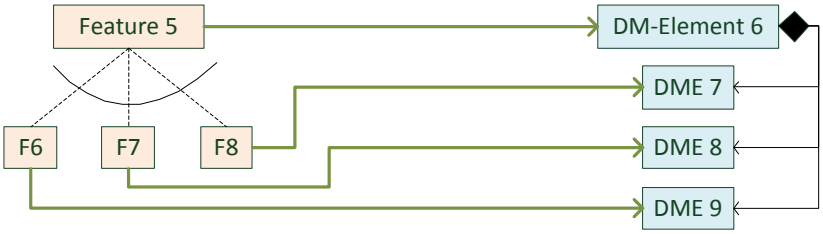

Abbildung 3.5: Abbildung von sich gegenseitig ausschließenden alternativen Features auf sich eventuell ebenfalls ausschließende existenzabhängige DM-Elemente. Auch hier darf das Entfallen der untergeordneten Elemente die Wohlgeformtheit des DM nicht verletzen.

Die Containment-Referenz, über die die Domänenmodell-Elemente 7 bis 9 von DME 6 abhängen, kann unter diesen Umständen sogar einwertig sein: Dadurch, dass sich die Features F6 bis F8 gegenseitig ausschließen, wird sichergestellt, dass in jedem Produkt nur eines der untergeordneten DM-Elemente realisiert wird. In diesem Fall stellt DME 6 einen *Variationspunkt* für die untergeordneten Elemente dar.

Insbesondere Alternativen für einwertige strukturelle Eigenschaften lassen sich nicht innerhalb eines Multivarianten-Domänenmodells abbilden: In einem wohlgeformten Ecore-basierten Modell dürfen von einer einwertigen Referenz nicht mehrere Elemente assoziiert werden. Dies erfordert neue Konzepte zur Modellierung von Variationspunkten, etwa den in Abschnitt 4.6 vorgestellten Mechanismus der *Alternativen-Mappings*.

3.2.4 Manifestation der Multiplizität von Features

Als zusätzliches Variabilitätsmerkmal wurde im Kontext dieser Arbeit die *Multiplizität* von Features definiert. Sie erlaubt die mehrfache Instanziierung eines in einem Featuremodell festgehaltenen Softwaremerkmals in einer entsprechenden Featurekonfiguration und kann sich auf zwei verschiedene Weisen im Domänenmodell abbilden. Es wird zwischen *objekt-* und *subjektbezogener* Manifestation der Multiplizität unterschieden.

Objektbezogene Manifestation der Multiplizität Ist die maximale Kardinalität eines mehrfach instanziierbaren Features bereits bei der Abbildung des Featuremodells auf das Domänenmodell bekannt, so können einzelne Instanzen auf entsprechende unterschiedliche Elemente des Domänenmodells abgebildet werden. Unterschreitet die tatsächliche die maximale Kardinalität, können fehlende Features so behandelt werden, als wären sie vorhanden aber nicht selektiert.

Abbildung 3.6 zeigt ein Beispiel für die objektbezogene (auch *instanzbezogene*) Manifestation eines mehrfach (in diesem Falle dreifach) instanziierbaren Features (dargestellt durch Feature 10a bis 10c). Jede Instanz des Features wird auf ein unterschiedliches Element des

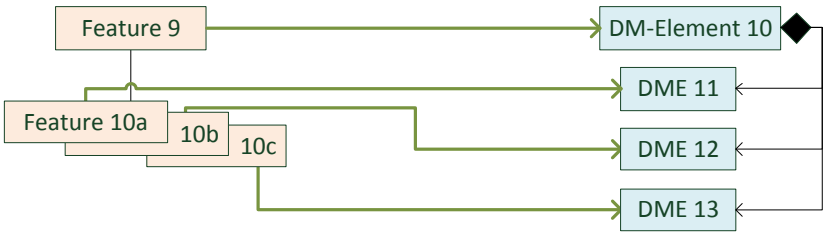

Abbildung 3.6: *Objekt-* bzw. *instanzbezogene* Manifestation des Variabilitätsmerkmals *Multiplizität*. Im Gegensatz zur *Alternativität* schließen sich die untergeordneten DM-Elemente (11 bis 13) nicht wechselseitig aus.

DM abgebildet. Umgekehrt drücken die DM-Elemente 11 bis 13 zwar dasselbe Merkmal aus, beziehen sich aber auf unterschiedliche *Objekte der Realität*.

Subjektbezogene Manifestation der Multiplizität Die subjektbezogene Manifestation der Multiplizität bezieht sich im Sinne des zuvor definierten Begriffs *Subjekt der Variabilität* auf den variierbaren – in diesem Falle multiplizierbaren – Gegenstand der realen Welt und somit auf den Variationspunkt selbst anstatt auf einzelne Objekte. Für eine Abbildung auf das Domänenmodell kommen demnach nicht einzelne Instanzen, sondern vielmehr die Eigenschaft, die sich durch die Menge aller Instanzen ergibt, nämlich die *Multiplizität*, in Frage. Diese kann sich als Attribut, z.B. wiederum als *Kardinalität* einer bestimmten strukturellen Eigenschaft, im Domänenmodell ausdrücken.

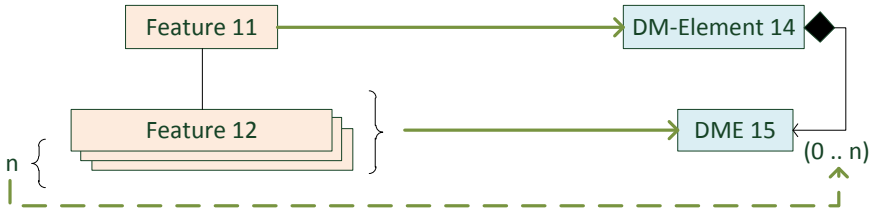

Abbildung 3.7: *Subjektbezogene* Manifestation des Variabilitätsmerkmals *Multiplizität*.

Ein Beispiel für diesen Zusammenhang ist in Abbildung 3.7 gegeben: Die tatsächliche Kardinalität n von Feature 12 steht erst nach der Erzeugung einer Featurekonfiguration fest und variiert zwischen mehreren FKs. Sie bildet im Domänenmodell eine Kardinalität, nämlich die der Containment-Referenz zwischen den Domänenmodell-Elementen 14 und 15, ab. **F2DMM** sieht hierfür die in Abschnitt 4.4.3 vorgestellten *Attribut-Ausdrücke* vor.

3.2.5 Manifestation von Feature-Attributen

Feature-Attribute dienen der weiteren Parametrisierung von Softwaremerkmalen, um das Variabilitätsmerkmal der *Alternativität* ohne Einschränkung des Wertebereichs zu realisieren. Mögliche Attributwerte können nicht, wie bei der Alternativen-Darstellung von Features als gruppierte ODER-Dekomposition, *prototypisch* abgebildet werden, sondern nehmen durch die explizite Angabe eines Wertes je Featurekonfiguration Einfluss auf das durch sie beschriebene Produkt. Im Folgenden werden zwei Möglichkeiten dargestellt, wie sich die Werte von Feature-Attributen auf das Domänenmodell abbilden lassen.

Abbildung als Domänenmodell-Attribut Elementare Artefakte des Multivarianten-Domänenmodells werden durch Attributwerte dargestellt: Beispielsweise lassen sich in **UML** die Namen von Klassen als Wert des Datentyps String angeben. Unter der Voraussetzung der Typverträglichkeit lässt sich die Abbildung eines Feature-Attributs auf ein Attribut aus dem Domänenmodell realisieren.

Abbildung 3.8: Manifestation von Feature-Attributen in elementaren Daten des Domänenmodells am Beispiel einer UML2-Klasse.

Abbildung 3.8 zeigt ein einfaches Beispiel für die beschriebene Art der Manifestation: Das Domänenmodell ist als **MOF**-Objektdiagramm dargestellt und definiert eine **UML2**-Klasse. Der Name dieser Klasse ist im Multivarianten-Domänenmodell noch nicht festgelegt. Er wird durch die Abbildung des Featuremodells durch den Wert von Attribut 1 bestimmt. Im Werkzeug **F2DMM** können derartige Zusammenhänge ebenfalls mit Hilfe von *Attribut-Ausdrücken* formuliert werden.

Optionalität durch Formulierung von Attribut-Constraints Feature-Attribute können auch das binäre Variabilitätsmerkmal *Optionalität* ausdrücken. Hierzu wird die Formulierung sog. *Attribut-Constraints*, booleschen Bedingungen auf den Werten von Attributen, erlaubt. Durch die Auswertung der Bedingung in Abhängigkeit einer Featurekonfiguration kann das durch den Constraint abgebildete Domänenmodell-Element selektiert oder deselektiert werden.

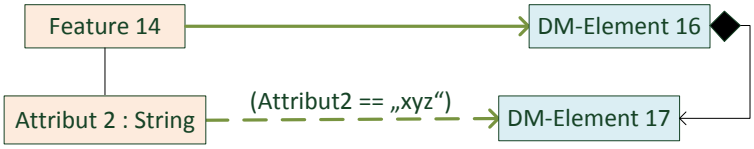

Abbildung 3.9: Attributwerte können im Sinne der *Optionalität* durch Constraints das Vorhandensein eines abgebildeten DM-Elements bedingen.

Im Beispiel in Abbildung 3.9 bedingt die Formulierung eines Constraints auf Attribut 2 das Vorhandensein von DM-Element 17. Dieser Ansatz zur Beschreibung von Variabilität ist im Mapping-Werkzeug **F2DMM** realisiert (s. Abschnitt 4.4.3).

3.3 Konsistenz und strukturelle Abhängigkeiten

Das Mapping von Features auf Elemente des Domänenmodells erfolgt durch eine *Abbildungsvorschrift*. Der Prozess der *Produktableitung* sieht vor, dass die in der Featurekonfiguration ausgedrückten Entscheidungen zur Elimination der Variabilität auf das Domänenmodell übertragen werden. In diesem Abschnitt sollen Bedingungen aufgestellt werden, unter denen abgeleitete Produkte *wohlgeformt* bezüglich des Domänen-Metamodells sind. Hierzu bedarf es der Identifikation und Auflösung sog. *struktureller Abhängigkeiten*. Hierfür werden im Folgenden einige Konzepte eingeführt, die den Beitrag dieser Arbeit unter dem in Abschnitt 1.3.1 formulierten Aspekt der *Konsistenz* rechtfertigen.

3.3.1 Annahmen

Während die im vorhergehenden Abschnitt dargestellten Konzepte zur Manifestation der Variabilität noch sehr generell formuliert waren, werden in diesem Abschnitt zur Untersuchung von Konsistenzkriterien einige Annahmen zugrunde gelegt.

MOF-Konformität des Domänenmodells Das Multivarianten-**DM** ist auf Basis eines Metamodells definiert, welches sich durch **MOF** beschreiben lässt. Insbesondere gilt die Annahme, dass jedes im DM enthaltenes Objekt einen eindeutigen *Container* hat und von diesem durch eine eindeutige, im Metamodell definierte Containment-Referenz existenziell abhängt (vgl. Abschnitt 2.2.1).

Validität des Domänenmodells Das Multivarianten-Domänenmodell entspricht seinem Metamodell bezüglich *Typkonformität* und *Multiplizität*.

Tupelweise Abbildung Das Mapping besteht aus einer Menge MM von Tupeln $m_{i,j}(f_i, d_j)$, welche Features f_i mit DM-Elementen d_j verknüpfen. Jedes Feature und jedes Element des Multivarianten-Domänenmodells muss in mindestens einem Tupel vorkommen.

Konformität der Featurekonfiguration Eine Featurekonfiguration ist konform mit ihrem Featuremodell, wenn sie genau die im **FM** beschriebenen Features f_i mit je einem eindeutigen *Selektionszustand* $s(f_i) \in \{\top, \bot\}$ auszeichnet.

Übertragung des Selektionszustandes Durch die Kombination einer Featurekonfiguration mit dem Multivarianten-DM, das durch das mit ihr konforme FM abgebildet wird, wird durch die tupelweise Abbildung die Übertragung der Selektionszustände von Features auf Domänenmodell-Elemente ermöglicht: Wird ein DM-Element d_j durch genau ein Feature f_i abgebildet, erhält es dessen Selektionszustand. Erfolgt die Abbildung durch mehrere Features, so gilt die Annahme, dass sich der Selektionszustand von d_j durch eine UND-Konjunktion derer Zustände ergibt.

In der in Abschnitt 4 vorgestellten **F2DMM**-Implementierung wird letztere Annahme sogar aufgeweicht: Features lassen sich über eine beliebige Kombination boolescher Ausdrücke auf Domänenmodell-Elemente abbilden. Die **MOF**-Konformität des Domänenmodells wird durch die Einschränkung auf Ecore-basierte Domänenmodelle erzielt.

3.3.2 Strukturelle Abhängigkeit von Domänenmodell-Elementen

In Abschnitt 3.2 wurde an einigen Stellen bereits impliziert, dass Elemente des Multivarianten-DM voneinander abhängen. Stehen durch das DM ausgedrückte *strukturelle Abhängigkeiten* im Widerspruch mit ihren durch eine Featurekonfiguration übertragenen Selektionszuständen, kann es zu *Abhängigkeitskonflikten* kommen (s. nächster Unterabschnitt). Zunächst bedarf es einer weiteren Formalisierung des Begriffs *strukturelle Abhängigkeit*.

Zwei Domänenmodell-Elemente d_i und d_j sind unter den im Folgenden genannten Voraussetzungen voneinander strukturell abhängig. Hierfür sei folgende Notation vereinbart:

$$d_i \Leftarrow d_j \qquad \text{„}d_i \text{ hängt von } d_j \text{ ab.“}$$

Diese Abhängigkeit kann sich entweder durch die Implikationen, **MOF** mit sich bringt, oder durch zusätzliche, vom Metamodell des DM abhängige, Konsistenzbedingungen ergeben.

Containment-Abhängigkeit Der MOF-Standard definiert ein Objekt als existenziell abhängig von seinem unmittelbar übergeordneten Container [42, Abschnitt 9.1]. Entsprechend gilt ein Element als *strukturell abhängig* von seinem – in *Ecore* über die Referenz `eContainer` abgebildeten – unmittelbar übergeordneten Element, wie im Folgenden formell dargestellt:

$$d_i \overset{Cont}{\Leftarrow} d_j \qquad \text{falls } d_j \text{ Container von } d_i$$

Abbildung 3.10 stellt dies an einem Beispiel dar: d_2 hängt über die Containment-Abhängigkeit von d_1 ab.

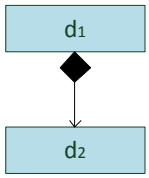

Abbildung 3.10: Ein MOF-Objekt hängt von seinem Container ab.

Metamodellspezifische Abhängigkeiten Die Wohlgeformtheit eines Modells ist im Allgemeinen nicht nur an dessen MOF-Konformität, sondern auch an weitere *Abhängigkeitsbedingungen* geknüpft, die nicht durch das Metamodell selbst ausgedrückt werden können; dies betrifft etwa die Existenz von bestimmten, im Metamodell definierten, Referenzzielen. Auch diese Bedingungen stellen *strukturelle Abhängigkeiten* dar, da ihre Nichtexistenz die Wohlgeformtheit von Modellen verletzen würde. Formell lässt sich eine Abhängigkeitsbedingung C zur Identifikation von Konsistenzverletzungen definieren:

$$d_i \overset{C}{\Leftarrow} d_j \qquad \text{falls eine auf } d_i \text{ und } d_j \text{ formulierte Abhängigkeitsbedingung } C \text{ zutrifft.}$$

Abbildung 3.11: Metamodellspezifische Nicht-Containment-Abhängigkeiten müssen separat definiert werden.

Abbildung 3.11 liegt eine explizit formulierte Abhängigkeitsbedingung C (d_3 hängt über r von d_4 ab) zugrunde. Weniger abstrakte Beispiele für spezifische, auf UML2-Klassendiagramme bezogene, Nicht-Containment-Abhängigkeiten sind im Folgenden aufgelistet (vgl. auch [10, Abschnitt 6.3.4]):

- Eine Klasse ist strukturell abhängig von ihren Oberklassen.
- Eine `Property` ist strukturell abhängig von ihrem Typ.
- Eine `Association` ist strukturell abhängig von den Properties, durch die sie in den verbundenen Klassen repräsentiert wird.

- Jede gerichtete Beziehung (`DirectedRelationship`) ist abhängig von deren Referenzziel.

An dieser Stelle wird noch keine Annahme darüber gemacht, in welcher Sprache diese Abhängigkeitsbedingungen formuliert werden. Im Rahmen des praktischen Teils dieser Arbeit wurde die Sprache **SDIRL** eingeführt, welche die Formulierung solcher Bedingungen auf Basis von **OCL**-Ausdrücken zulässt (s. Abschnitt 4.5).

3.3.3 Abhängigkeitskonflikte und mögliche Lösungen

Der Formalismus der strukturellen Abhängigkeiten wurde eingeführt, um die Konsistenz von abgeleiteten Produkten unter den zugrundeliegenden Annahmen sicherzustellen. Wie bereits erwähnt, kann die Konsistenz von Produkten nicht mehr sichergestellt werden, sobald die aus der Featurekonfiguration abgleiteten Selektionszustände von Elementen des **DM** deren strukturellen Abhängigkeiten widersprechen. Ein solcher Widerspruch wird im Folgenden als *Abhängigkeitskonflikt* bezeichnet. Er tritt genau dann auf, wenn folgende Bedingungen erfüllt sind:

1. Ein DM-Element d_i ist strukturell abhängig von d_j.
2. Der Selektionszustand $s(d_i)$ ist positiv ($\top$).
3. Der Selektionszustand $s(d_j)$ ist negativ ($\bot$).

Im Falle einer Containment-Abhängkeit erschließt sich dieser Zusammenhang intuitiv: Ein Nicht-Wurzel-Element kann nur dann existieren, wenn sein übergeordneter Container existiert. Zieht man das oben genannte Beispiel der strukturellen Nicht-Containment-Abhängigkeit bei einer Vererbung in Betracht, kann eine spezialisierte Klasse nur dann vorhanden sein, wenn deren Oberklasse im Produkt existiert. In beiden Fallen könnte die Wohlgeformtheit eines abgeleiteten Produktes verletzt werden.

Um die Wohlgeformtheit hingegen zu garantieren, müssen Abhängigkeitskonflikte eliminiert werden. Ein in dieser Arbeit ausgearbeitetes Verfahren zur Auflösung von Konflikten ist die *Propagation des Selektionszustandes*: Dabei wird der eigentliche, durch die Featurekonfiguration bestimmte, Zustand von betroffenen **DM**-Elementen künstlich überschrieben, so dass der Konflikt nicht mehr auftritt. Die Propagation kann entweder in die Richtung der Abhängigkeit (in Pfeilrichtung) oder in die Gegenrichtung stattfinden. Entsprechend werden folgende *Propagationsstrategien* (**PS**) unterschieden:

Vorwärts-Propagation (VP) Der Selektionszustand des positiven, strukturell abhängigen DM-Elements d_i wird negativiert.

Rückwärts-Propagation (RP) Der Selektionszustand des negativen, strukturell übergeordneten Elements d_j wird positiviert.

In Abschnitt 4.7.4 wird die Anwendung von Propagationsstrategien zur Auflösung von Abhängigkeitskonflikten im Kontext des **F2DMM**-Projekts aus Entwurfssicht behandelt.

3.3.4 Surrogate: Wiederherstellung der Konsistenz ohne Informationsverlust

Die Auswahl der Propagationsstrategie bringt einige Nebeneffekte mit sich: Entscheidet man sich für die Rückwärtspropagation, werden unter Umständen **DM**-Artefakte in Produkte integriert, in denen sie aufgrund bestimmter Restriktionen[23] gar nicht integriert werden dürften. Die Vorwärts-Propagation garantiert hingegen, dass nur Elemente, die aufgrund der Abbildung einen positiven Selektionszustand haben, Teil von abgeleiteten Produkten sein können. Hierbei kommt es zwar nicht zur Verletzung von Restriktionen, jedoch ist ein *Informationsverlust* möglich: DM-Elemente, die eigentlich in bestimmten Produkten integriert sein sollten, können durch Anwendung der Vorwärtspropagation entfallen.

Um dem entgegenzuwirken ohne dabei die Wohlgeformtheit von Produkten zu verletzen, können weiterführende Strategien, etwa die in Abschnitt 4.5 vorgestellten *Surrogate* angewendet werden. Ein durch Anwendung einer Propagationsstrategie nachträglich negativ selektiertes DM-Element könnte in einem bestimmten Kontext durch einen adäquaten Stellvertreter ersetzt werden. Auch dies sei anhand einiger **UML2**-bezogener Beispiele erläutert:

- Fehlt der Rückgabetyp einer Operation, kann er durch einen kompatiblen Typ, etwa einer Oberklasse, ersetzt werden.
- Fehlt in einer mehrstufigen Vererbungshierarchie eine Klasse, kann sie durch die nächsthöher gelegene generalisierte Klasse ersetzt werden.
- In Zustandsdiagrammen kann beim Fehlen eines Transitionsendes auf den entsprechenden Endzustand verwiesen werden.

Um geeignete Surrogate zu finden, ist Kontextwissen über das Modell, in dem der Informationsverlust aufgetreten ist, notwendig. Surrogat-Kandidaten wie in der obigen Auflistung können beispielsweise wiederum mit Hilfe der Anfragesprache **OCL** ermittelt werden. Dieser Ansatz wird in **F2DMM** verfolgt: Das Konzept der Surrogate ist hier in die Sprache **SDIRL** integriert (vgl. Abschnitt 4.5.5). Surrogat-Ausdrücke sind hierbei fest an Abhängigkeitsbedingungen geknüpft; falls eine Negativierung stattfindet, werden verfügbare Surrogat-Kandidaten durch Auswertung beigefügter OCL-Ausdrücke ermittelt.

3.4 MDPLE als Softwareentwicklungsprozess

Wie bei der Herstellung gewöhnlicher Software sind auch am Entwicklungsprozess von Softwareproduktlinien verschiedene Personen mit zugewiesenen *Rollen* beteiligt, die in aufeinanderfolgenden *Phasen* definierte *Funktionen* durchführen. Softwareentwicklungsprozesse [6] beinhalten die Definition von *Arbeitspaketen*, die Rollen, Phasen und Funktionen einander zuordnen. Während in der Softwareentwicklung Prozesse wie der *Rational Unified Process* (**RUP**) [38] oder das *V-Modell* [46] sehr konkrete Arbeitsschritte definieren, setzen Ansätze, welche auf die Entwicklung von Softwareproduktlinien oder die modellgetriebene Softwareentwicklung zugeschnitten sind, häufig auf einfacheren Prizipien wie dem *Wasserfall-* oder *Spiralmodell* auf. Nach der Vorstellung einiger aus der Literatur bekannten Prozessmodelle für **MDPLE** wird in diesem Abschnitt ein auf den beschriebenen **SPL**-Modellierungsansatz zugeschnittener Prozess identifiziert.

[23] Diese Restriktionen können ökonomischer, rechtlicher oder unternehmenspolitischer Natur sein: Man will beim Maßschneidern eines Produkts nur die nötigen Elemente preisgeben.

3.4.1 Das Doppelspiralmodell von Gomaa

Gomaa [23] setzt sich mit der **UML**-gestützten Entwicklung von Softwareproduktlinien (**SPL**) auseinander und definiert in einem Überblickskapitel verschiedene Prozesse zur kontinuierlichen Entwicklung derselben, unter anderem das *Doppelspiralmodell* (s. Abbildung 3.12). Beide Spiralen stellen je einen unabhängigen, iterativen Prozess dar, der jeweils die aus dem Wasserfallmodell bekannten Phasen (Planung, Analyse, Entwurf und Implementierung) umfasst. Die Ausführung der beiden Spiralen kann alternierend erfolgen: Nach der initialen Definition einer Softwareproduktlinie werden die ersten Produkte abgeleitet, analysiert und die Ergebnisse wiederum zur Weiterentwicklung der gesamten Produktlinie verwertet.

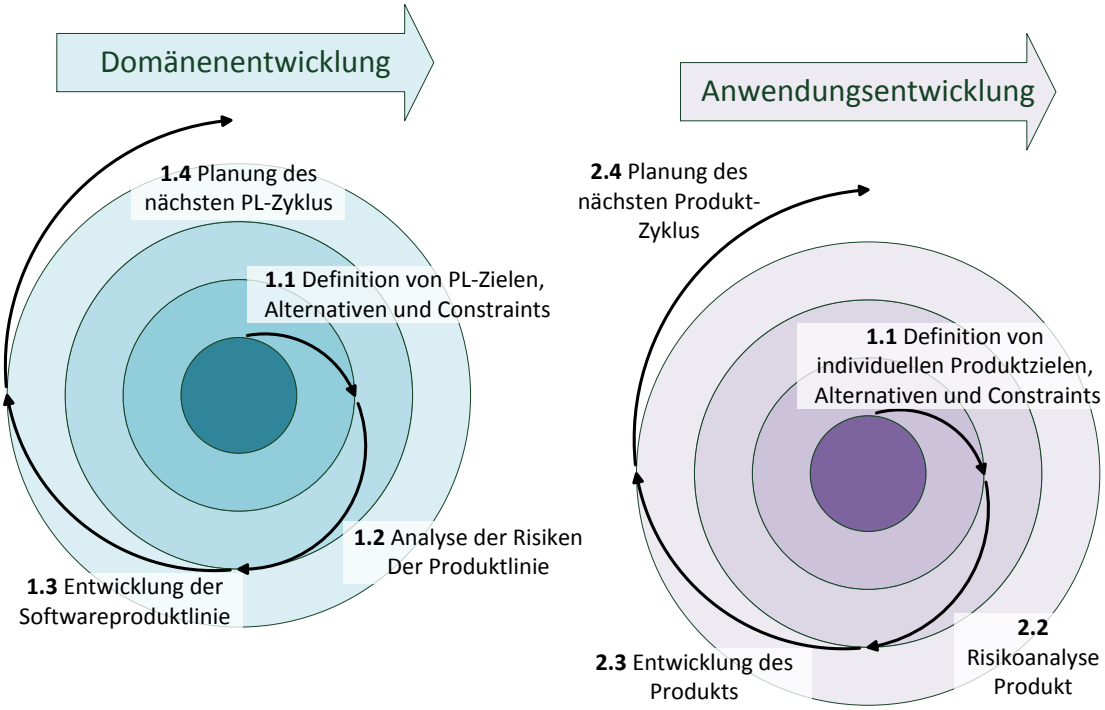

Abbildung 3.12: Das Doppelspiralmodell von Gomaa [23, Abschnitt 3.4.2] beschreibt die Entwicklung von Softwareproduktlinien durch zwei alternierende, inkrementelle Entwicklungsprozesse.

3.4.2 Model Driven Architecture

Frankel [20] beschreibt mit *Model Driven Architecture* (**MDA**) einen Ansatz, der die in der Einleitung dieser Arbeit beschriebene ansteigende Abstraktion bei der Spezifikation und Implementierung von Softwaresystemen berücksichtigt, um die Architektur eines Systems ebenfalls modellgetrieben zu spezifizieren. Ziel ist die vollautomatische Generierung aller Artefakte eines Softwaresystems, inklusive des ausführbaren Quellcodes, aus einer modellbasierten Beschreibung. Durch die schrittweise Übersetzung eines abstrakten Modells wird jeweils ein auf der nächstniedrigeren Abstraktionsebene gelegenes Artefakt erzeugt. Dadurch soll zum einen die Produktivität gesteigert, zum anderen der Wartungsaufwand nach der Auslieferung gesenkt werden.

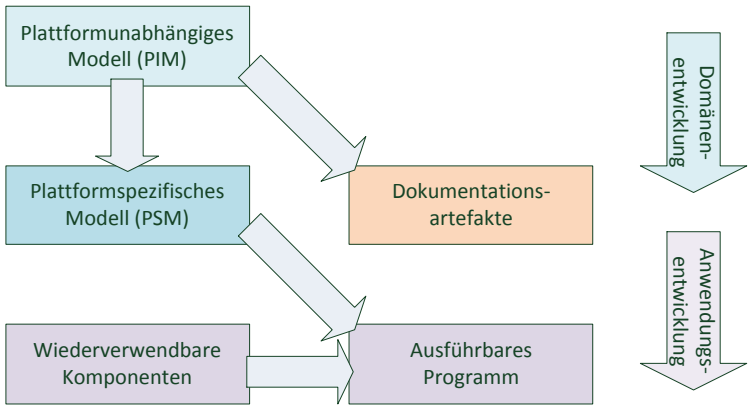

Abbildung 3.13: Der MDA-Ansatz von Frankel [20, Abschnitt 1] beschreibt die modellgetriebene Softwareentwicklung als idealerweise vollautomatisierten Prozess.

MDA beschreibt die Transformation vom beschreibenden Architekturmodell zum ausführbaren Programm in zwei Schritten: Zunächst erfolgt die Transformation eines *plattformunabhängigen Modells* (**PIM**), welches von den Spezifika von Zielsprache und -Plattform abstrahiert, in ein *plattformspezifisches Modell* (**PSM**), welches ebendiese Aspekte berücksichtigt. Das PIM wird häufig zur Generierung von Dokumentationsartefakten, etwa der Architekturbeschreibung eines Systems, herangezogen. Der Schritt vom PSM zum ausführbaren Programm erfolgt zunächst durch eine Transformation in die Zielsprache. Im nicht-idealisierten Prozess wird der generierte Quelltext manuell mit plattformspezifischen Fragmenten angereichert, um die Funktionalität des Zielsystems zu komplettieren.

3.4.3 Referenzprozess der Softwareproduktlinienentwicklung

Der im Rahmen des Vorgängerprojektes **MODPL** [10] definierte **MDPLE**-Prozess verfeinert den erstmals von Pohl u. a. [45] formalisierten Referenzprozess zur Softwareproduktlinienentwicklung hinsichtlich der Integration des modellgetriebenen Aspekts. Pohl unterscheidet zwischen der *Domänenentwicklung*, welche sich auf die komplette Produktfamilie bezieht, und der *Anwendungsentwicklung*, die die Implementierung einzelner Produkte beschreibt. Jedem der beiden – wiederum separat zu betrachtenden – Entwicklungsprozesse liegen die Prozessschritte Analyse und Entwurf, zugrunde (s. Abbildung 3.14).

Ergebnis der Domänenanalyse ist das *Featuremodell* (**FM**), in dem die Merkmale, welche innerhalb der Domäne identifiziert wurden, festgehalten werden (vgl. Abschnitt 3.1). Das Pendant auf der Seite der Anwendungsentwicklung sind die *Featurekonfigurationen* (**FK**s), die die Merkmale je Produkt aus der Softwarefamilie beschreiben. Die Erzeugung dieser beiden Artefakte wird im Folgenden unter dem Aspekt *Featuremodellierung* zusammengefasst. Auf der darunterliegenden Ebene werden jeweils Artefakte der *Entwicklung* beschrieben: Das *Multivarianten-Domänenmodell* beschreibt Software, welche in allen Produkten vorkommt, während das *konfigurierte Domänenmodell* ein Produkt der Softwarefamilie repräsentiert. Die Abbildung verdeutlicht das Hauptziel von **MDPLE**: Die automatisierte Erzeugung des konfigurierten Domänenmodells durch einen *Ableitungsschritt*, dem das Multivarianten-Domänenmodell und die das Produkt beschreibende Featurekonfiguration zugrunde liegen.

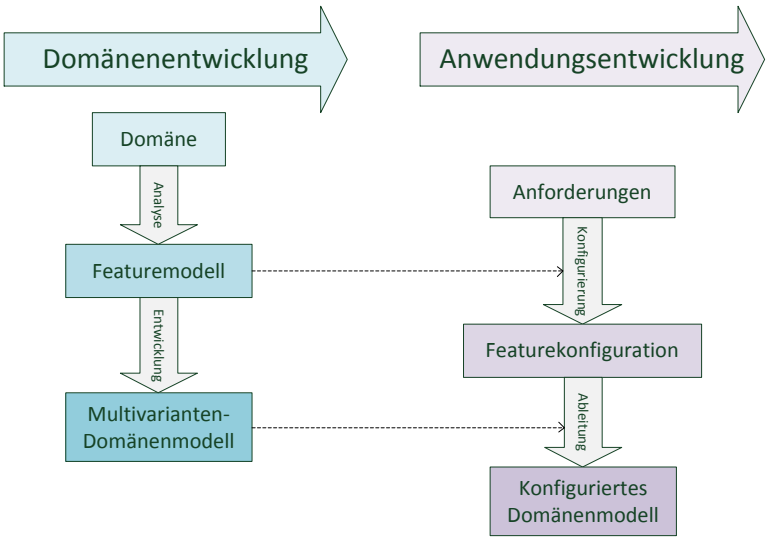

Abbildung 3.14: Von Buchmann [10] vorgeschlagener Prozess zur Entwicklung von modellgetriebenen Softwareproduktlinien.

3.4.4 Vorgeschlagener MDPLE-Prozess

Diese Arbeit befasst sich zum Großteil mit der *Abbildung* (vgl. engl. *mapping*) von Features auf ihre entsprechenden Elemente des Multivarianten-Domänenmodells. In den folgenden Abschnitten wird zu diesem Zweck das *Mapping-Modell* als zentrale Abbildungsstruktur eingeführt. Das Mapping-Modell stellt die Korrektheit der Abbildung bezüglich *Abhängigkeitsbedingungen* (vgl. Abschnitt 3.3) mit Hilfe eines sog. *Konsistenzmodells* sicher, welches sich wiederum auf die *Sprache* der Domäne bezieht: Verwandte MDPLE-Ansätze setzen hierbei häufig UML als Modellierungssprache voraus, sodass dieser Schritt entfällt (s.

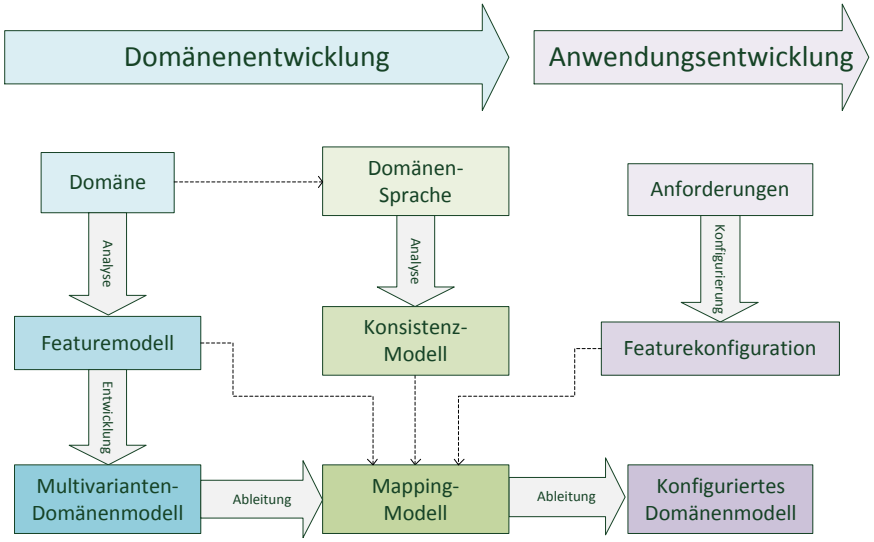

Abbildung 3.15: Ein im Kontext dieser Arbeit vorgeschlagenes Prozessmodell zur Entwicklung von modellgetriebenen Softwareproduktlinien.

Abschnitt 6). Das Konsistenzmodell entsteht durch Analyse dieser Modellsprache durch einen *DSL-Experten* (vgl. Abbildung 3.15).

Die Schritte *Analyse der Domänensprache*, *Konsistenzmodell* und *Mapping-Modell* werden der Domänenentwicklung zugeschrieben, da sie sich ebenfalls durch ihre Allgemeingültigkeit bezüglich der Softwarefamilie auszeichnen und nicht für jede **FK** neu erfolgen müssen. Die Ableitung von Produkten erfolgt schließlich im Rahmen der Anwendungsentwicklung.

3.4.5 Die mathematisch-formelle Sicht: Problem- und Lösungsraum

Unter den Begriff *Featuremodellierung* fällt, wie oben beschrieben, die Erzeugung von Featuremodell und -konfigurationen. Betrachtet man die Ableitung von Produkten als mathematisches Abbildungsproblem, definiert die Featuremodellierung einen *Problemraum* [15]: Hier werden lediglich abzubildende Softwaremerkmale definiert. Eine Featurekonfiguration FC_k ist genau dann konform zu einem Featuremodell FM_k, wenn sie jedem in ihm definierten Softwaremerkmal f_i einen *Selektionszustand* $s(f_i) \in \{\top, \bot\}$ zuweisen kann:

$$\forall f_i \in FC_k \Rightarrow s(f_i) \in \{\top, \bot\} \wedge f_i \in FM_k$$

Die Lösung des definierten Abbildungsproblems ergibt sich – um in der mathematisch-formellen Sichtweise zu bleiben – durch Ableitung des konfigurierten Domänenmodells aus der Featurekonfiguration. Das Multivarianten- sowie das konfigurierte Domänenmodell sind demnach Elemente des *Lösungsraums*. Im vorgeschlagenen **MDPLE**-Prozessmodell hat das Mapping-Modell die Rolle einer *Abbildungsvorschrift*:

$$MM(FM) : FC_k \Rightarrow P_k,\ FC_k \text{ konform zu } FM$$

Eine durch ein Mapping-Modell definierte Abbildungsvorschrift $MM(FM)$ kann folglich jede Featurekonfiguration FC_k, die konform zum mit dem Mapping-Modell verknüpften Featuremodell ist, auf ein Produkt P_k abbilden. Die Abbildung selbst beschreibt den Übergang zwischen Problem- und Lösungsraum [30] und ist Gegenstand des nachfolgenden Abschnitts.

4 F2DMM: Ein Mapping-basierter Editor für modellgetriebene Softwareproduktlinien

Dieser Abschnitt baut auf den zuvor vermittelten Grundlagen der modellgetriebenen Softwareentwicklung mit Eclipse (vgl. Abschnitt 2) sowie den Vorüberlegungen aus Abschnitt 3 auf und stellt mit **F2DMM** einen Editor zur modellgetriebenen Entwicklung von Softwareproduktlinien auf Basis eines Mapping-Modells vor. Die Beschreibung erfolgt alternierend aus Konzept- und aus Anwendersicht. Der nachfolgende Abschnitt 5 beschäftigt sich schließlich mit der Anwendung des Werkzeugs auf konstruierte Problemstellungen.

4.1 Übersicht: Modelle und Werkzeuge

Wie bereits erwähnt, liegt der praktische Beitrag dieser Arbeit in der Erstellung eines Werkzeuges, welches die Abbildung von Features auf Elemente eines Multivarianten-Domänenmodells unterstützt. Grundlage für den Editor ist ein Mapping-Metamodell, welches erlaubt, ein beliebiges *Ecore*-basiertes Domänenmodell mit zusätzlichen Mapping-spezifischen Informationen wie Feature-Ausdrücken, zu versehen. Die folgenden Ausführungen beziehen sich auf die in Abbildung 4.1 beschriebene Modell- und Werkzeuglandschaft.

Das entwickelte Rahmenwerk **F2DMM** (*Feature To Domain Mapping Model)* ist Bestandteil des lehrstuhlweiten Projekts **FAMILE** (*Features And Mappings In Lucid Evolution)*. In den nachfolgenden Abschnitten wird zunächst auf die Modelle und Werkzeuge zur *Featuremodellierung* eingegangen (s. Abschnitt 4.2), die im Rahmen eines kleinen Master-Projekts implementiert wurden und ausdrücklich kein Beitrag dieser Arbeit sind. Softwaremerkmale und deren Ausprägung werden in einem Featuremodell und mehreren Featurekonfigurationen definiert, die jeweils Instanz desselben Metamodells sind. Gegenstand der vorliegenden Arbeit war jedoch die Ausarbeitung von Mechanismen zum Erhalt der *Synchronität* von **FM** und **FK** in einem entsprechenden Modul.

Im darauffolgenden Abschnitt 4.3 werden die dem F2DMM-Metamodell zugrundeliegenden Entwurfsentscheidungen erläutert. Die wesentliche Komponente stellen *Mappings* dar, die Elemente des referenzierten Domänenmodells mit *Feature-Ausdrücken* (s. Abschnitt 4.4) versehen, welche in ihrer textuellen Repräsentation von der **FEL**-Grammatik (*Feature Expression Language*) definiert werden. Durch sie wird die *Manifestation* von Variabilität im Domänenmodell durch geeignete Sprachkonstrukte berücksichtigt.

Mappings können sich auch auf Elemente außerhalb der Domänenmodell-Ressource beziehen. In diesem Fall spricht man von Alternativen-Mappings (s. Abschnitt 4.6). Sie tragen zur Unterstützung der *Agilität* bei und erlauben die Definition von *Variationspunkten.*

Optional kann ein **SDIRL**-Dokument (*Structural Dependency Identification and Repair Language*, s. Abschnitt 4.5) das Domänen-Metamodell um *strukturelle Abhängigkeitsbedingungen* (vgl. Abschnitt 3.3.2) ergänzen. Für das Bearbeiten dieser Dokumente ist ein eigener textueller Editor vorgesehen, der Teile des vom *Eclipse Modeling Projekt* bereitgestellten **OCL**-Texteditors und des entsprechenden Metamodells wiederverwendet.

Aus einem konsistenten Mapping-Modell können unter Angabe jeweiliger Featurekonfigurationen *Produkte* (auch *konfigurierte Domänenmodelle*) abgeleitet werden (s. Abschnitt 4.8). Voraussetzung hierfür ist die *Synchronität* des Mappings mit dem zugrundeliegenden Multivarianten-**DM**, welche ebenfalls durch ein entsprechendes Modul sichergestellt wird (vgl. Abschnitt 4.9). Schließlich beleuchtet Abschnitt 4.10 einige zusätzliche Funktionen des F2DMM-Editors aus Anwendersicht.

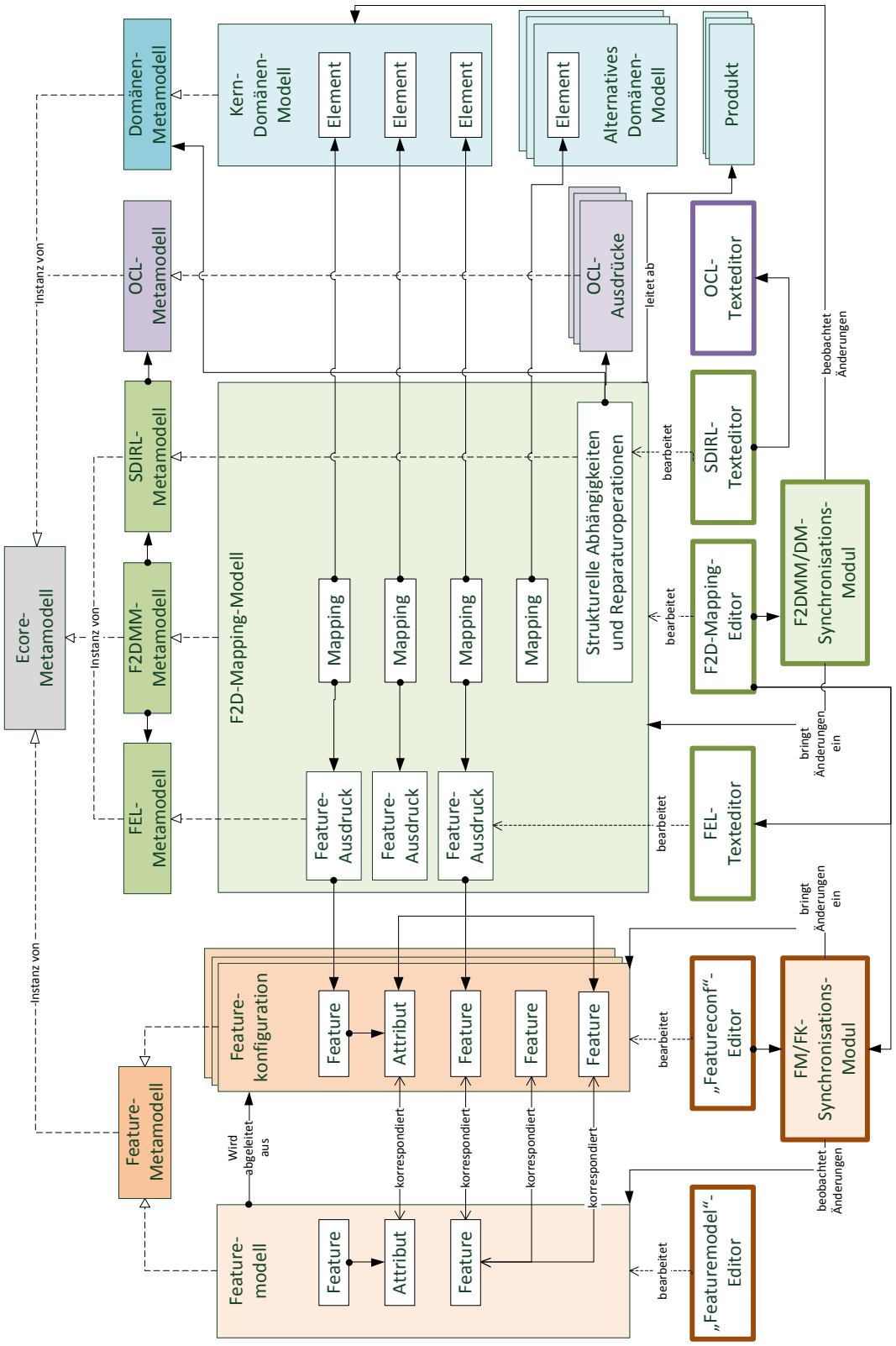

Abbildung 4.1: Überblick über die Modelle und Werkzeuge im Kontext des **FAMILE**-Projekts.

4.2 Featuremodellierung

Wie bereits in Abschnitt 3.1 erwähnt, haben sich Featuremodelle und -konfigurationen [35] zur formellen Beschreibung von Softwaremerkmalen etabliert. Im Kontext des **FAMILE**-Projekts wurde in einem kleinen Master-Projekt [39] ein Feature-Metamodell sowie Editoren für Featuremodell und Featurekonfiguration entwickelt. Im Rahmen der vorliegenden Masterarbeit wurden keine weiteren Änderungen am Metamodell und an den Editoren vorgenommen. Eine Ausnahme stellt die Integration des *Synchronisationsmoduls* in den Featurekonfigurations-Editor dar, welche in Abschnitt 4.2.4 behandelt wird. Aus Gründen der Dokumentation und zur besseren Nachvollziehbarkeit der nachfolgenden Ausführungen werden zunächst das Feature-Metamodell, die Editoren sowie der Validierungsmechanismus vorgestellt.

4.2.1 Das Feature-Metamodell

Featuremodelle und -konfigurationen sind Instanzen eines gemeinsamen, **EMF**-basierten Metamodells (vgl. Ecore-Klassendiagramm in Abbildung 4.2). Die Verknüpfung zwischen **FK** und **FM** entsteht durch das Setzen der Referenz `definingFeatureModel` in der Konfiguration.

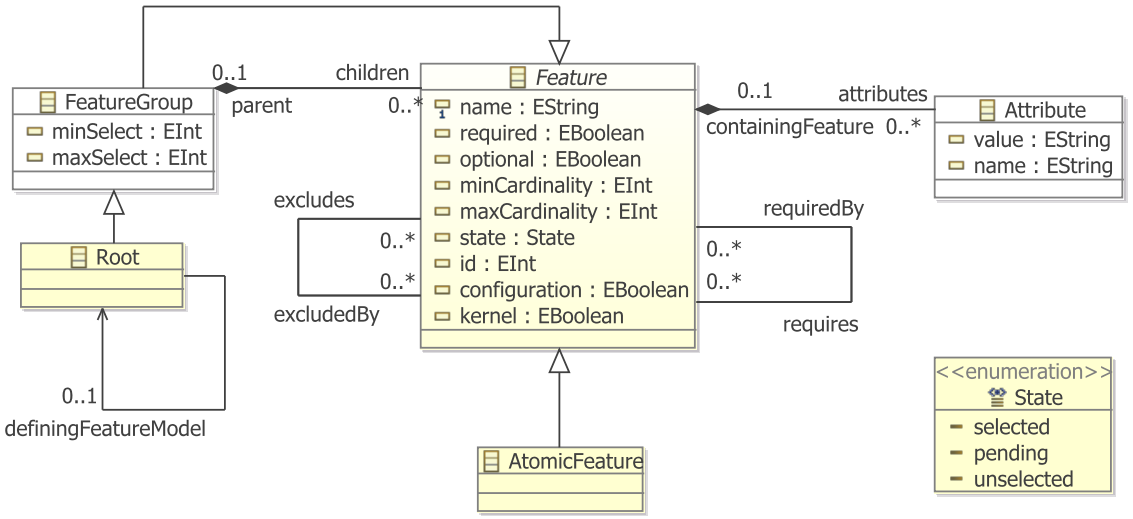

Abbildung 4.2: Das Feature-Metamodell unterscheidet zwischen atomaren Features (repräsentiert durch die Modellklasse `AtomicFeature`) und Feature-Gruppen (`FeatureGroup`), welche eine baumartige Verschachtelung nach dem Entwurfsmuster *Composite* [21, Kapitel 4.3] erlauben. Die Wurzel eines Featuremodells bzw. einer -konfiguration sind durch die Ableitung `Root` gekennzeichnet. Features können durch Feature-Attribute parametrisiert werden.

Jedes Feature bekommt vom Modellierer einen Selektionszustand (**SZ**) zugewiesen, repräsentiert durch den Enumerationstyp `State`. Während im Featuremodell noch keine Auswahl erfolgt (alle Features haben den **SZ** `pending`), muss im Sinne der in Abschnitt 3.1.5 geforderten *Elimination der Variabilität* innerhalb einer gültigen Featurekonfiguration jedes Feature entweder `selected` oder `unselected` sein.

Identität von Features Bei der Erzeugung eines Features wird eine eindeutige ID generiert und diesem zugewiesen. Während im Featuremodell eine ID genau einmal existieren darf, bekommen im Falle eines mehrfach instanziierbaren Features in einer entsprechenden Featurekonfiguration alle Instanzen dieselbe ID zugewiesen. Die Attribute `minCardinality` und `maxCardinality` legen Unter- bzw. Obergrenzen für die Anzahl der erlaubten Instanzen eines Features in einer Konfiguration fest. Im Gegensatz zu den IDs müssen die Namen von Features (Attribut `name`) nicht global eindeutig sein.

Lokale und globale Optionalität Das boolesche Attribut `kernel` gibt an, ob ein modelliertes Feature Bestandteil des Kernels, also der Menge der Gemeinsamkeiten aller Produkte der Produktlinie ist. In einer gültigen Featurekonfiguration müssen folglich alle Instanzen von Features, für die die Bedingung `kernel == true` gilt, selektiert sein. Das Attribut `required` hingegen bezieht sich auf die lokale Abhängigkeit eines Features von seiner enthaltenden Feature-Gruppe (Variabilitätsmerkmal *Optionalität*, vgl. Abschnitt 3.1.2). Das abgeleitete Attribut `optional` verhält sich invers zu `required`.

Feature-Gruppen, Kardinalität und Selektion Der im vorliegenden Feature-Metamodell gewählte Ansatz unterstützt *Cardinality-Based Feature Modeling* (**CBFM**, Abschnitt 3.1.3): Die Attribute `minSelect` und `maxSelect` geben eine Unter- bzw. Obergrenze für die erlaubte Anzahl selektierter Unterfeatures einer Gruppe an, bezogen auf alle Instanzen innerhalb einer Featurekonfiguration. Als Obergrenze ist außerdem die Konstante `GROUP_SIZE` zulässig, welche die aktuelle Anzahl der Unterfeatures je Konfiguration repräsentiert.

Die tatsächliche *Kardinalität* einer Feature-Gruppe ergibt sich in einer Featurekonfiguration jeweils *lokal* durch die Anzahl der Instanzen pro Unterfeature. Eine Konfiguration ist gültig, solange sich diese Größe im durch `minCardinality` und `maxCardinality` definierten Intervall befindet.

Beziehungen zwischen Features: requires und excludes Bisher wurden Featuremodell und -konfiguration als verschachtelte, baumartige Struktur verstanden: Features stehen über die `parent`- bzw. `children`-Referenz miteinander in Beziehung. Das vorliegende Feature-Metamodell erlaubt die Angabe weiterer, nicht existenzabhängiger Beziehungen: Features können sich entweder gegenseitig bedingen (modelliert über die `requires`-Kante) oder ausschließen (`excludes`) [28]. Die umgekehrte Beziehungsrichtung (`eOpposite`) wird durch die Referenzen `requiredBy` bzw. `excludedBy` ausgedrückt.

4.2.2 Validierung in Featuremodell und -konfiguration

Um die Wohlgeformtheit von Featuremodell und -konfiguration sicherzustellen, wurde, ebenfalls im Vorfeld des **FAMILE**-Projekts, das **EMF** Validation Framework (s. Abschnitt 2.4) unter Zuhilfenahme geeigneter Constraints in die Editoren eingebunden. Auch Heidenreich [28] beschreibt im Kontext des *FeatureMapper*-Projekts Bedingungen für die Wohlgeformtheit verschiedener Komponenten von Softwareproduktlinien (s. Abschnitt 6.4.1), u.a. Featuremodellen und -konfigurationen. Tabelle 4.1 ordnet die Constraint-Klassen für das vorliegende Metamodell den von Heidenreich definierten Bedingungen zu. Letztere sind den Bereichen Featuremodell (FM) und Featurekonfiguration (VM, vgl. engl. *variant model*) zugeordnet.

Heidenreich	**FAMILE**	**Beschreibung**
FM-Mandatory-Root	(implizit)	Das Wurzelelement muss in allen Featurekonfigurationen gewählt sein.
FM-Cardinality-Match	`MinCardinality-Constraint`, `Max-Cardinality-Constraint`	Kardinalitäten von Features müssen mit denen ihrer Elternfeatures übereinstimmen.
FM-Sound-Reference	(implizit)	`requires`- und `excludes`-Beziehungen dürfen sich nicht widersprechen.
FM-Existing-Reference	(implizit)	Referenzierte Features müssen im Featuremodell definiert worden sein.
(implizit)	`UniqueID-Constraint`	Jedes Feature muss identifizierbar sein.
VM-Mandatory-Parent	`ParentSelected-Constraint`	Features können nur selektiert werden, wenn die beinhaltende Feature-Gruppe selektiert ist.
VM-Mandatory-Child	`Obligatory-Constraint`	Als `required` gekennzeichnete Features müssen selektiert sein, wenn die beinhaltende Feature-Gruppe selektiert ist.
(`kernel` nicht unterstützt)	`KernelConstraint`	`kernel`-Features müssen in jeder Konfiguration selektiert sein.
(implizit)	`NonPending-Constraint`	In einer Featurekonfiguration muss jedes Feature einen eindeutigen Selektionszustand haben.
VM-Alternative	`MaxSelect-Constraint`	Die Anzahl selektierter Unterfeatures darf `maxSelect` nicht überschreiten.
VM-Or	`MinSelect-Constraint`	Die Anzahl selektierter Unterfeatures darf `minSelect` nicht unterschreiten.
VM-Requires	`Requires-Constraint`	Ein Feature darf nur selektiert werden, wenn alle in `requires`-Beziehung stehenden Features selektiert sind.
VM-Conflicts	`Excludes-Constraint`	Ein Feature darf nicht selektiert werden, eines der in `excludes`-Beziehung stehenden Features selektiert ist.

Tabelle 4.1: Constraints für Featuremodell und -konfiguration nach Heidenreich [28], verglichen mit der vorliegenden Implementierung. Der Hinweis „implizit“ bedeutet, dass eine Verletzung der Bedingung bereits durch Entwurfsentscheidungen im jeweiligen Metamodell ausgeschlossen werden konnte.

4.2.3 Werkzeugunterstützung

Die Benutzerschnittstelle zu Featuremodell und -konfiguration stellen jeweils generierte und modifizierte EMF-Baumeditoren dar (vgl. Abschnitt 2.3). Der **FK**-Editor (s. Screenshot in Abbildung 4.3) hebt selektierte Features in cyan, nicht selektierte Features in orange, hervor. Feature-Gruppen sind durch doppelte Kreise, atomare Features durch einzelne Kreise repräsentiert. Obligatorische (`required`) Features sind mit ausgefüllten Kreisen markiert. Zusätzlich zum Feature-Namen werden die gültigen Intervalle für die Kardinalität in eckigen Klammern, gefolgt von denen für die Selektion in runden Klammern, notiert. Attribut-Werte sind nur innerhalb einer Konfiguration relevant und bleiben in der Featuremodell-Ressource ungesetzt.

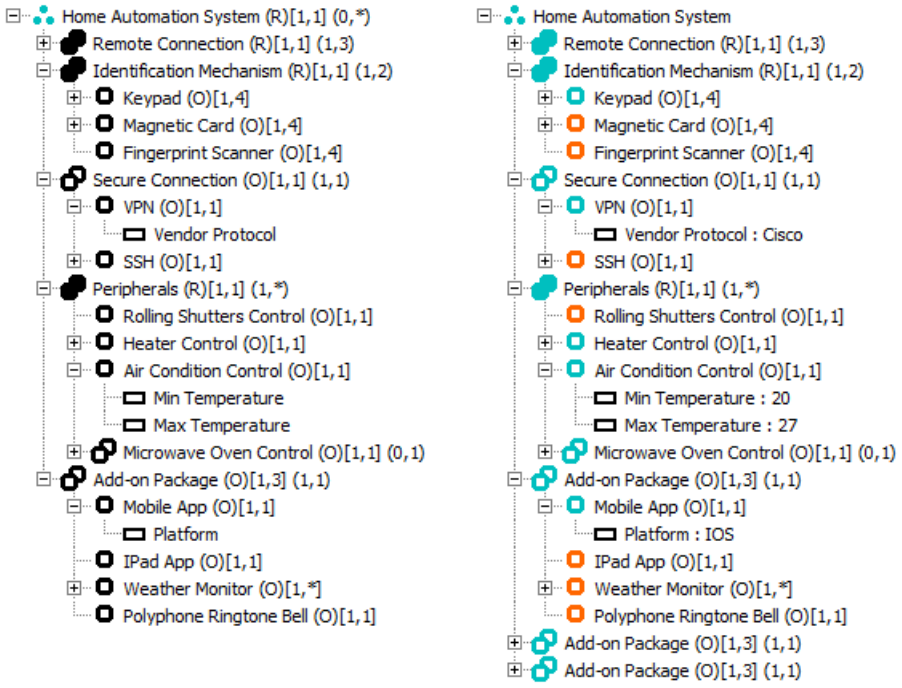

Abbildung 4.3: Baumrepräsentation für Featuremodell (links) und -konfiguration (rechts).

Im Featuremodell-Editor wird das Ableiten einer FK über einen entsprechenden Kontextmenü-Eintrag ermöglicht. Im Featurekonfigurations-Editor kann schließlich die Selektion bzw. Deselektion von Features per Doppelklick erfolgen. Zudem kann das Klonen eines Teilbaums mit Kardinalität größer eins veranlasst werden. Weitere Kontextmenü-Einträge erleichtern das rekursive Selektieren und Deselektieren von Teilbäumen.

4.2.4 Synchronisation von Featuremodell und -konfiguration

Wie bereits erwähnt, zählt die Implementierung der Werkzeuge nicht zum Beitrag dieser Arbeit. Jedoch wurde die Modellschnittstelle um ein Synchronisierungsmodul ergänzt, welches sicherstellt, dass erzeugte FKs mit ihrem ursprünglichen FM konsistent bleiben, auch wenn dieses nachträglich manipuliert wird. Ziel der Synchronisation ist, dass nach dem Laden einer FK sämtliche Features konsistent mit dem entsprechenden Feature des FM sind, aus dem sie ursprünglich abgeleitet wurde. Korrespondierende Features werden grundsätzlich durch

ihre übereinstimmende `ID` erkannt. Gegenstand der Konsistenzprüfung ist unter anderem die Wertgleichheit folgender Attribute (vgl. auch Abbildung 4.2.1):

- des Feature-Namens (`name`),
- der booleschen Attribute `required` und `kernel`,
- des durch `minCardinality` und `maxCardinality` ausgedrückten Kardinalitäts-Intervalls,
- der Referenzziele von `requires`- und `excludes`- Kanten,
- bei Feature-Gruppen zusätzlich der das Selektions-Intervall definierenden Attribute `minSelect` und `maxSelect`,
- der Namen (`name`) der Feature-Attribute, welche einem Feature untergeordnet sind.

Zusätzlich besteht die Synchronisation in der Propagation von Einfüge- und Löschoperationen, bezogen auf Features und Feature-Attribute. Die Synchronisation erfolgt in zwei Schritten: Zunächst werden FM und geöffnete FK miteinander verglichen, um *Konsistenzverletzungen* zu identifizieren. Diese werden dem Anwender präsentiert und im anschließenden *Propagationsschritt* zur Reparatur der Featurekonfiguration herangezogen.

Identifikation von Konsistenzverletzungen Das **FM**/**FK**-*Synchronisationsmodul* wurde nachträglich in den generierten Featurekonfigurations-Editor als reine Java-Implementierung integriert. Abbildung 4.4 zeigt ein **UML**-Diagramm mit den an der Synchronisation beteiligten Klassen: `FMConsistencyAnalyzer` führt die Identifikation von Konsistenzverletzungen in der `analyze`-Methode durch, in der die strukturellen Eigenschaften von Features mit übereinstimmender ID verglichen werden. Der Rückgabewert ist eine Menge von Instanzen der Schnittstelle `FMConsistencyViolation`, deren Unterklassen wiederum verschiedene Arten von Konsistenzverletzungen repräsentieren.

Jede Konsistenzverletzung muss eine Beschreibung (deklariert durch die Methode `getDescription()`) zurückgeben und das Feature, das die jeweilige Bedingung verletzt, referenzieren (`getViolatingConfiguredFeature()`). Zusätzlich wird von implementierenden Klassen die Erzeugung einer Reparaturaktion als Instanz von `Command` (aus dem **EMF** *Command Framework*, vgl. Abschnitt 2.3) verlangt, welche die Konsistenz mit dem Featuremodell wiederherstellen kann (s. nächster Unterabschnitt). Nachfolgende Auflistung nennt Unterklassen von `FmConsistencyViolation` und erklärt die jeweilige Ursache für deren Erzeugung:

- `FeatureNameViolation`: Der Name eines Features aus der FK stimmt mit dem Namen des Features mit derselben ID aus dem FM nicht überein.
- `IsKernelViolation`: Der Wahrheitswert von `kernel` eines konfigurierten Features stimmt nicht mit dem seines modellierten Features überein.
- `IsRequiredViolation`: Der Wahrheitswert von `required` stimmt nicht überein.
- `Min-` bzw. `MaxCardinalityViolation`: Das Kardinalitäts-Intervall eines FK-Features stimmt nicht mit dem des entsprechenden FM-Features überein.

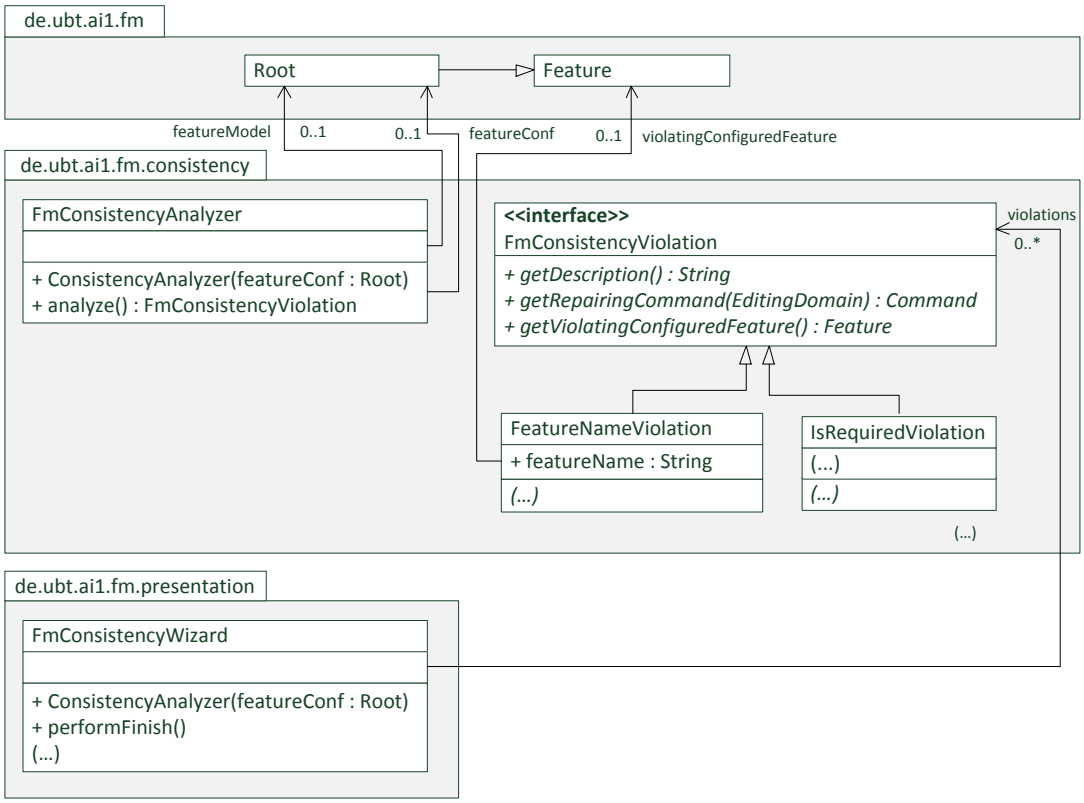

Abbildung 4.4: FM/FK-Synchronisationsmodul in UML-Klassendiagramm-Notation.

- `Min-` bzw. `MaxSelectViolation`: Das durch `min-` und `maxSelect` definierte Selektions-Intervall stimmt nicht überein.
- `MissingAttributeViolation`: Das FM-Feature enthält ein Attribut, das im entsprechenden FK-Feature nicht vorkommt.
- `DanglingAttributeViolation`: Das FK-Feature enthält ein überflüssiges Attribut, welches aus dem FM-Feature entfernt wurde.
- `MissingFeatureViolation`: Ein Unterfeature einer FM-Feature-Gruppe fehlt in der entsprechenden FK-Feature-Gruppe.
- `DanglingFeatureViolation`: Die FK-Feature-Gruppe enthält ein Feature, das aus der entsprechenden FM-Feature-Gruppe entfernt wurde.
- `MissingRequirement-` bzw. `MissingExclusionViolation`: Ein FM-Feature referenziert ein weiteres Feature über `requires` oder `excludes`. Diese Beziehung fehlt im entsprechenden Feature aus der FK.
- `DanglingRequirement-` bzw. `DanglingExclusionViolation`: Ein FK-Feature enthält eine im FM entfernte `requires-` oder `excludes`-Referenz.
- `CloneFeatureViolation`: Ein Feature aus der FK muss geklont werden, um seiner im FM neu definierten `minCardinality` zu genügen.

- `UncloneFeatureViolation`: Eine Instanz eines Features muss aus der FK entfernt werden, um die neu gesetzte Obergrenze `maxCardinality` nicht zu überschreiten.

Propagation der Änderungen zur Featurekonfiguration Die oben beschriebene Identifikation von Konsistenzverletzungen wird in der angepassten Version des FK-Editors bei jedem Öffnen sowie beim Refokussieren des Editor-Fensters durchgeführt. Wie bereits angedeutet, werden die Konsistenzverletzungen dem Anwender in einem *Wizard* mitgeteilt: Ist die Menge der von `analyze()` ermittelten Instanzen von `FmConsistencyViolation` nicht leer, öffnet sich der `FmConsistencyWizard` (s. Abbildung 4.5), der die Beschreibungen über Aufrufe von `getDescription()` tabellarisch ausgibt. Wird der Wizard über OK beendet, erfolgt im Aufruf der Methode `performFinish()` die Ausführung aller über `getRepairingCommand()` ermittelten Reparaturaktionen auf der `EditingDomain` des FK-Editors.

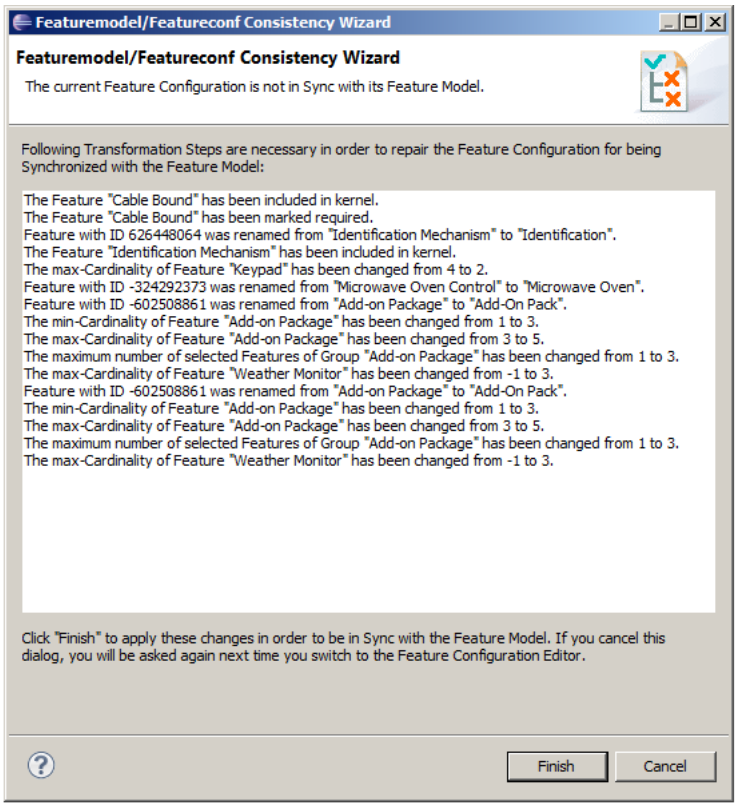

Abbildung 4.5: Der FM/FK-Synchronisations-Wizard zeigt die Beschreibungen identifizierter Konsistenzverletzungen an. Bei Bestätigung durch den Anwender werden die generierten Reparaturaktionen durchgeführt.

Das in diesem Abschnitt vorgestellte Synchronisationsmodul ist nicht nur in den Featurekonfigurations-Editor, sondern auch in den **F2DMM**-Mapping-Editor integriert, welcher in den folgenden Abschnitten vorgestellt wird. Die Integration selbst ist Gegenstand von Abschnitt 4.9.2.

4.3 Das F2DMM-Metamodell

4.3.1 Referenzierte Modelle

Die Bezeichnung **F2DMM** (vgl. engl. *Feature To Domain Mapping Model*) verrät bereits, dass in einem Mapping-Modell Artefakte eines Domänen-Modells auf Elemente eines Featuremodells und einer entsprechenden Featurekonfiguration abgebildet werden. Beim vorgestellten Ansatz handelt es sich um ein *explizites* Mapping: Sämtliche Informationen zur Abbildung werden in einer eigenen, vom Domänenmodell unabhängigen, Ressource persistiert. In Abbildung 4.6 werden die grundlegenden Entwurfsentscheidungen für das F2DMM-Metamodell dargestellt. Bei der Abbildung wird zwischen *Containment-Mappings* (**CM**s), *Referenz-Mappings* (**RM**s) und *Attribut-Mappings* (**AM**s) unterschieden.

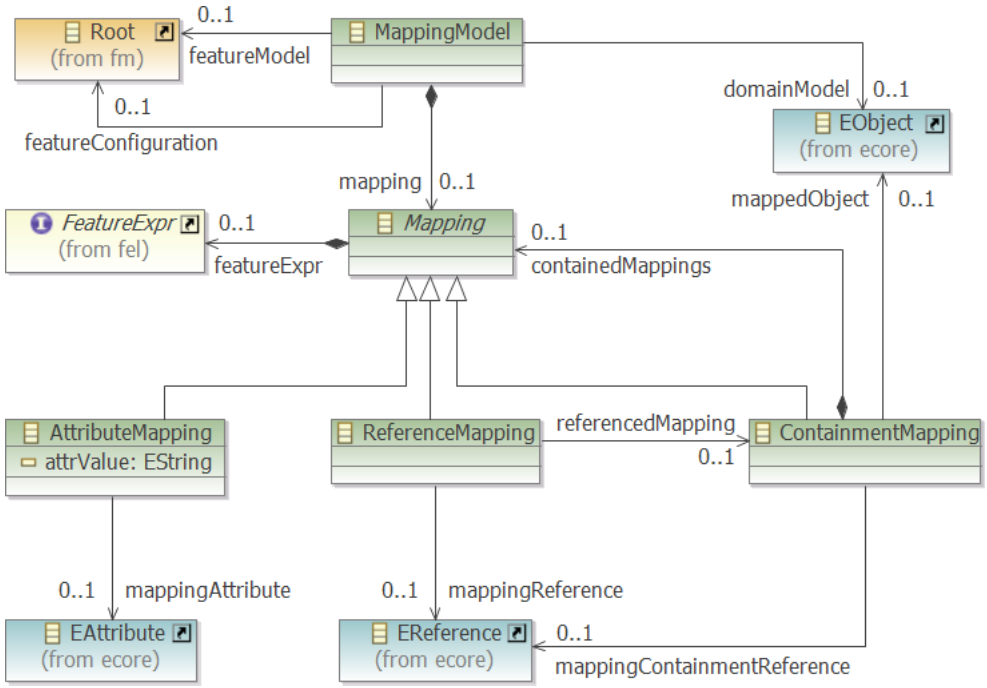

Abbildung 4.6: Vereinfachte Darstellung des **F2DMM**-Metamodells. Ein Mapping-Modell beinhaltet eine Instanz der abstrakten Klasse `Mapping`, welche wiederum über die konkrete Klasse `ContainmentMapping` baumartig verschachtelt sein kann.

Aus Gründen der Generizität enthält das F2DMM-Metamodell keinerlei Annahmen über das Metamodell des verwendeten Multivarianten-Domänenmodells, abgesehen davon, dass es auf dem Ecore-Metamodell basieren muss. Der Bezug zum Domänen-Metamodell wird über die Referenzen zu den Ecore-Metaklassen `EAttribute` und `EReference` der entsprechenden Mappings hergestellt. Containment-Mappings nehmen über `mappedObject` auf konkrete Domänenmodell-Elemente Bezug, die Instanzen von Unterklassen von `EObject` repräsentiert werden. Die Wurzel des Domänenmodells wird vom Mapping-Modell selbst referenziert.

Die Verbindung zu **FM** und **FK** wird über die Referenzen `featureModel` und `featureConfiguration`, welche beide auf die Modellklasse `Root` des Feature-Metamodells (vgl. Abschnitt 4.2.1) veweisen, hergestellt. Mappings enthalten Feature-Ausdrücke als Instanzen von `FeatureExpr`, welche sich wiederum auf einzelne Features beziehen (nicht im Diagramm dargestellt). Auf Feature-Ausdrücke wird in Abschnitt 4.4 ausführlich eingegan-

gen.

4.3.2 Der Mapping-Baum: Strukturelle Rekonstruktion des Domänenmodells

Die vereinfachte Illustration in Abbildung 4.6 definiert bereits die grundlegende Struktur eines Mapping-Modells: Der Containment-Baum des Domänenmodells wird rekonstruiert. Während ein Containment-Mapping sich stets auf eine Instanz von `EObject`, nämlich das abgebildete Domänenmodell-Element, bezieht (s. Referenz `mappedObject`), wird bei Attribut-Mappings die String-Serialisierung des entsprechenden elementaren Datentypen hinterlegt. Die gestrichelten, von Attribut-Mappings ausgehenden Kanten in Abbildung 4.8 stellen also keine tatsächliche Referenz dar.

Ähnlich verhalten sich Referenz-Mappings, welche sich auf Nicht-Containment-Referenzen aus dem Domänenmodell beziehen und somit das *angewandte Auftreten* von an anderer Stelle deklarierten Objekten realisieren. Anstatt auf das referenzierte Domänenmodell-Element zu verweisen, bezieht sich ein Referenz-Mapping auf ein Containment-Mapping, welches das deklarierte Objekt abbildet (`referencedMapping`). Im Beispiel in Abbildung 4.8 ist nicht etwa das Referenzziel `c1` der Referenz `type` mit dem Referenz-Mapping `rm1` assoziiert, sondern das Containment-Mapping `cm2`, welches sich auf `c1` bezieht. Diese zusätzliche Abstraktion ist zur Vorberechnung möglicher Konsistenzverletzungen (s. Abschnitt 4.7.1) vonnöten.

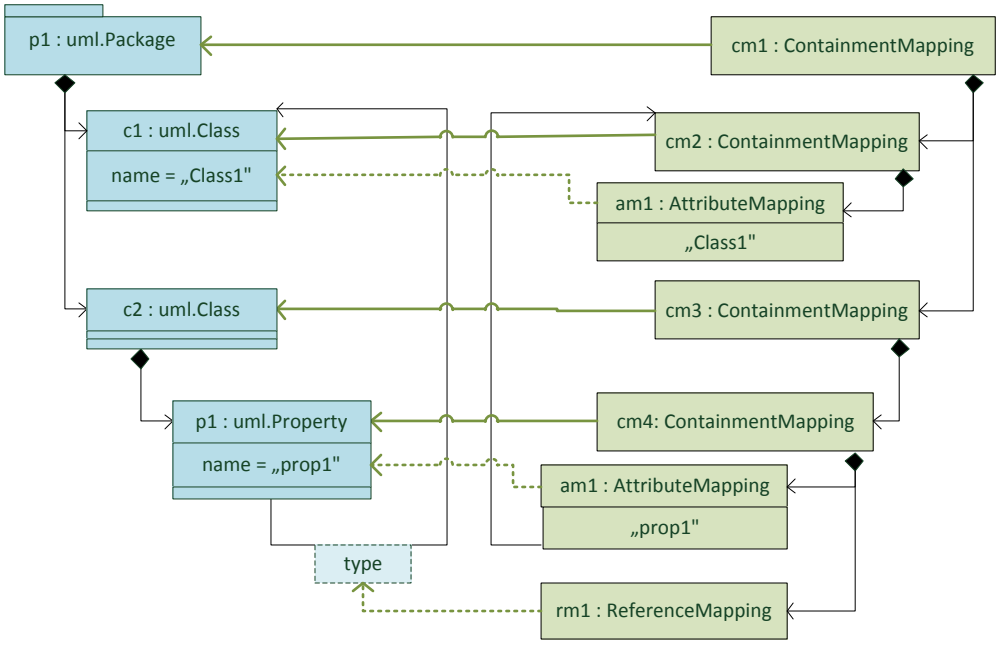

Abbildung 4.7: Beispiel-Mapping (rechts) für ein vorhandenes Domänenmodell (links). Die Struktur des Mapping-Modells ist durch den aufspannenden Containment-Baum des Domänenmodells definiert. Im Gegensatz zu Attribut- und Referenz-Mappings erlauben Containment-Mappings eine beliebige Verschachtelung (s. Referenz `containedMappings` in Abbildung 4.6).

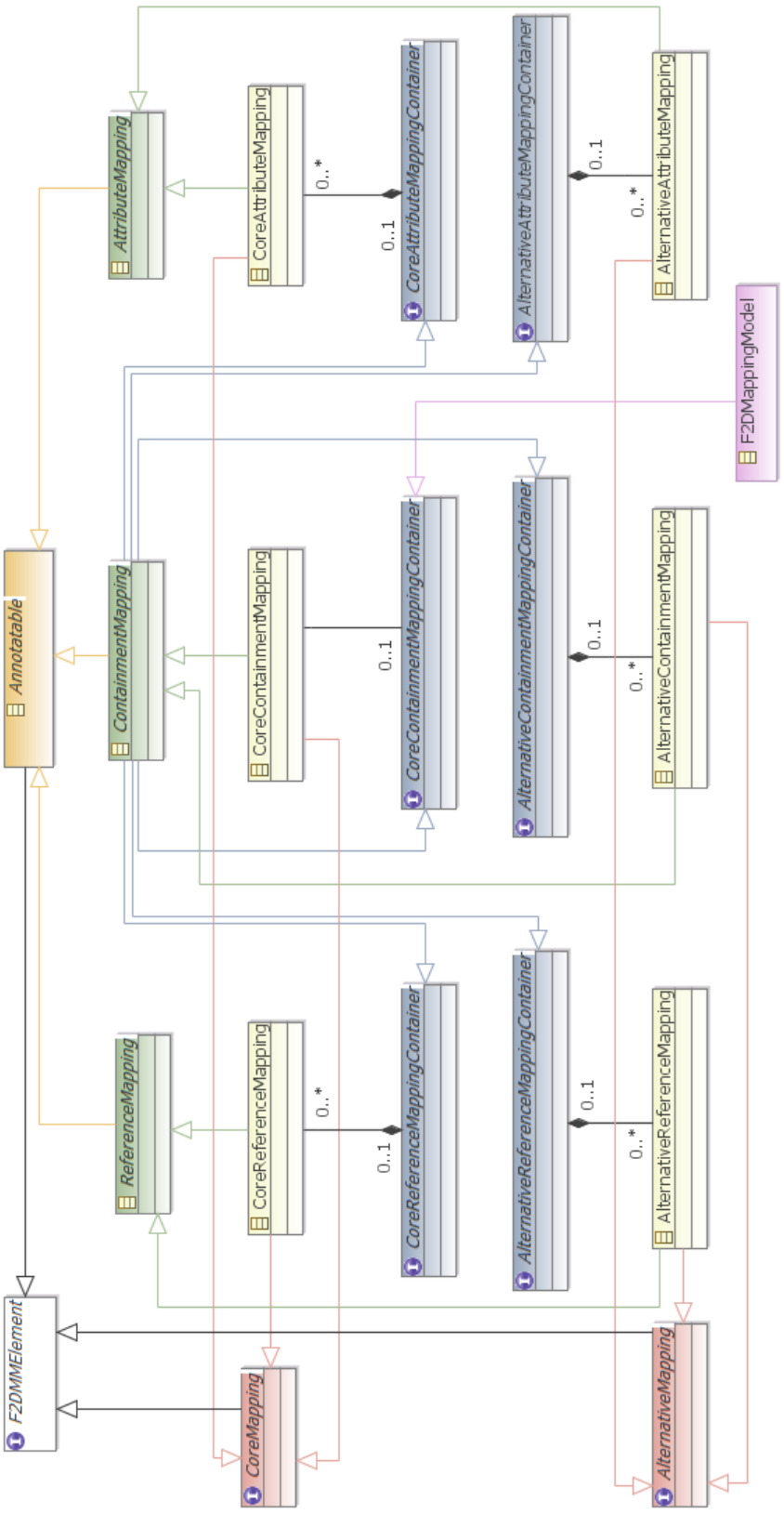

Abbildung 4.8: Ausschnitt des tatsächlichen **F2DMM**-Metamodells, der dem in der vereinfachten Darstellung in Abbildung 4.6 angedeuteten, nach dem Composite-Muster [21, Kapitel 4.3] entworfenen Teilmodell entspricht.

4.3.3 Kern- und Alternativen-Mappings

Das tatsächliche **F2DMM**-Metamodell unterscheidet neben der strukturellen Dimension (Containment-, Attribut- bzw. Referenz-Mappings) zusätzlich nach *Kern-* und *Alternativen*-Mappings. Kern-Mappings beziehen sich auf das referenzierte Multivarianten-Domänenmodell, indem dessen Struktur wie in Abbildung 4.7 angedeutet nachgebildet wird. Die Wahrung der strukturellen *Synchronität* mit dem Domänenmodell ist Aufgabe des **DM/MM**-*Synchronisationsmoduls*, welches in Abschnitt 4.9.1 vorgestellt wird.

Alternativen-Mappings erlauben die Erweiterung des Domänenmodells, etwa um Abbildungen zu realisieren, die nicht durch Kern-Mappings dargestellt werden können (s. Abschnitt 4.6.2). Im Gegensatz zu Kern-Mappings unterliegen Alternativen-Mappings keiner automatischen Synchronisierung und werden stattdessen vom Modellierer manuell gepflegt. Durch Kombination der beiden genannten Dimensionen ergeben sich schließlich sechs konkrete Mapping-Klassen:

- **Kern-Containment-Mapping**, implementiert durch `CoreContainmentMapping`,
- **Kern-Attribut-Mapping** (`CoreAttributeMapping`),
- **Kern-Referenz-Mapping** (`CoreReferenceMapping`),
- **Alternativen-Containment-Mapping** (`AlternativeContainmentMapping`),
- **Alternativen-Attribut-Mapping** (`AlternativeAttributeMapping`),
- **Alternativen-Referenz-Mapping** (`AlternativeReferenceMapping`).

Abbildung 4.8 stellt einen Ausschnitt des tatsächlichen **F2DMM**-Metamodells dar. Mappings sind in dieser Darstellung tabellarisch nach den Dimensionen *Struktur* (Containment, Referenz, Attribut; in grün) bzw. *Zugehörigkeit* (Kern, Alternative; in rot) angeordnet. Zusätzlich existiert für jede der sechs Mapping-Arten ein Container-Interface (in grau dargestellt) als weiterer Abstraktionsschritt. Die abstrakte Klasse `ContainmentMapping` implementiert alle sechs Container-Interfaces, um eine rekursive Verschachtelung zu ermöglichen.

Das Mapping-Modell selbst (`F2DMappingModel`) ist von `CoreContainmentMappingContainer` abgeleitet und repräsentiert die Wurzel des Mapping-Baumes. Alle Mapping-Klassen erben indirekt von der abstrakten Basisklasse `Annotatable`, die Gemeinsamkeiten von Mappings, welche mit Feature-Ausdrücken versehen werden können, berücksichtigt. Sie wird in Abschnitt 4.7.2 detailliert behandelt. Es sei jedoch vorweggenommen, dass sich jede Instanz von `Annotatable` zu jedem Zeitpunkt in einem von acht möglichen *Selektionszuständen* (**SZ**) befindet, welche Gegenstand des nächsten Unterabschnitts sind.

4.3.4 Selektionszustände

Instanzen von `Annotatable` wird nach Auswertung annotierter Feature-Ausdrücke (s. Abschnitt 4.4) ein SZ zugewiesen, welcher jeweils als Literal des Aufzählungstypen `SelectionState` (vgl. Abbildung 4.9) modelliert ist.

Nach der Auswertung von Feature-Ausdrücken steht der `initialSelectionState` fest. Er kann lediglich die Werte *active*, *inactive*, *incomplete* oder *corrupted* annehmen (nicht annotierte Elemente erhalten zunächst den SZ *incomplete*; fehlerhafte Ausdrücke werden als *corrupted* markiert). Um die Konsistenz der Annotationen sicherzustellen, wird im Mapping-Modell

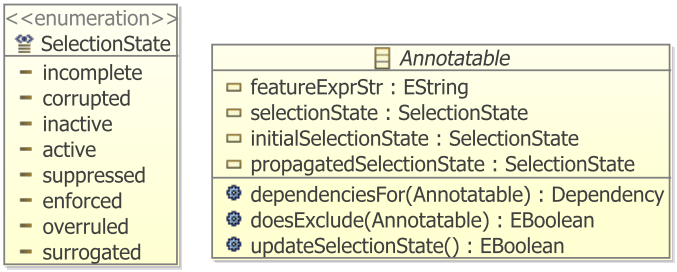

Abbildung 4.9: Darstellung der relevanten Elemente der Metaklasse `Annotatable` sowie des Enumerationstypen `SelectionState` als Ecore-Klassendiagramm.

das Konzept der *Propagation von Selektionszuständen* implementiert (vgl. Abschnitt 3.3.3). Hierdurch können die `propagatedSelectionStates` weitere Werte, nämlich *suppressed* oder *enforced*, annehmen. Durch *Ausschlusskonflikte* bzw. *Surrogate* können diese erneut revidiert werden und die endgültigen Zustände *surrogated* bzw. *overruled* annehmen. Die Mechanismen zur Bestimmung dieser Zustände werden in Abschnitt 4.7.5 erläutert. Abbildung 4.10 fasst mögliche **SZ** und deren grafische Notation, die auch im **F2DMM**-Editor verwendet wird, zusammen.

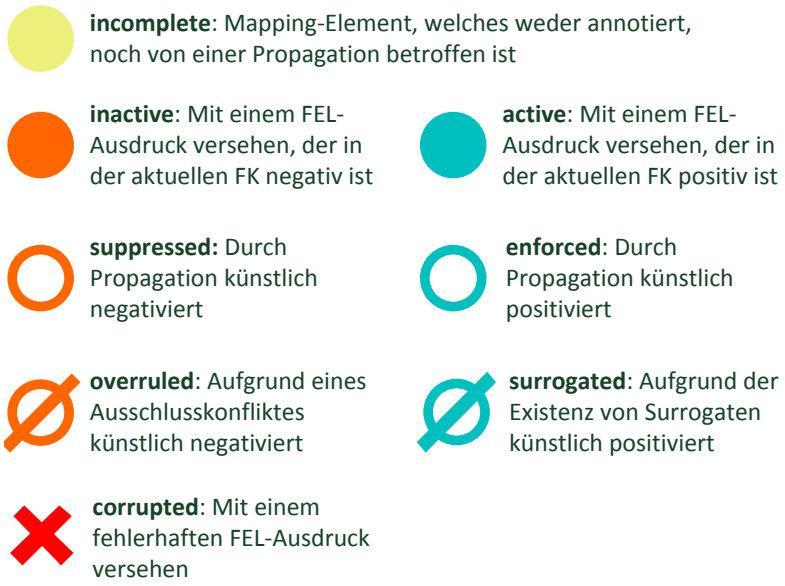

Abbildung 4.10: Mögliche Selektionszustände eines Mappings im F2DMM-Metamodell und deren Bedeutung.

4.4 FEL: Feature-Ausdrücke und deren Auswertung

Die textuelle Anfragesprache *Feature Expression Language* (**FEL**) dient innerhalb des **F2DMM**-Werkzeugs der Formulierung von *Feature-Ausdrücken*, welche als Annotationen von Mappings die Verbindung zum Featuremodell sowie zu einer eventuell geladenen Featurekonfiguration herstellen. Neben Feature-Ausdrücken definiert die Sprache auch *Attribut-Ausdrücke*, die Anfragen auf die Werte von Feature-Attributen einer Featurekonfiguration erlauben.

4.4.1 Elementare Feature-Ausdrücke: Qualifizierende Namen, Index-Bezug und boolesche Verknüpfungen

Feature-Ausdrücke beziehen sich auf die Selektionszustände von Features unter Berücksichtigung der aktuellen Featurekonfiguration und bekommen nach ihrer Auswertung selbst einen **SZ** zugewiesen. Mögliche Selektionszustände für Feature-Ausdrücke sind *incomplete*, *inactive*, *active* und *corrupted* (vgl. Abbildung 4.10). Syntaktisch ungültige Feature-Ausdrücke haben stets den Selektionszustand *corrupted*. In den folgenden Absätzen wird die Syntax von Feature-Ausdrücken anhand von Beispielen beschrieben.

Feature-Referenzen Eine Möglichkeit, Bezug zu Selektionszuständen von Features herzustellen, ist das Referenzieren des Features selbst. Eine *Feature-Referenz* ist ein Feature-Ausdruck, der den Selektionszustand des durch ihn referenzierten Features in der aktuell geladenen FK annimmt. Ist keine FK geladen, hat der Ausdruck den Wert *incomplete*.

Eine Feature-Referenz besteht in ihrer konkreten textuellen Syntax aus dem Feature-Namen selbst. Enthält der Feature-Name Sonder- oder Leerzeichen, muss er in doppelte Hochkommata (") eingeschlossen werden. Nachfolgendes Quelltextfragment enthält drei Feature-Referenzen, die sich auf das Modell aus Abbildung 4.3 auf Seite 59 beziehen:

```
1 VPN
2 "Rolling Shutters Control"
3 Cloud
```

In der geladenen Feature-Konfiguration würde der erste Ausdruck den Selektionszustand *active* zurückgeben, da das gleichnamige Feature selektiert ist. Der zweite Ausdruck evauliert dagegen zu *inactive*. Da im Featuremodell kein Merkmal mit dem Namen „Cloud“ vorhanden ist, wäre der dritte Ausdruck ungültig (*corrupted*).

Qualifizierende Namen In Abschnitt 4.2.1 wurde bereits erwähnt, dass Feature-Namen nicht global eindeutig sind. Falls ein Feature mit demselben Namen in einem Modell mehrfach vorkommen würde, könnte die Referenz nicht mehr eindeutig aufgelöst werden. Features über deren eindeutig vergebene IDs zu referenzieren, könnte das Problem lösen, ist aber aus Anwendersicht wenig praktikabel. Stattdessen erlaubt **FEL** das Auflösen von Mehrdeutigkeiten bei Feature-Namen über den aus Programmiersprachen bekannten Mechanismus des *qualifizierenden Namens* [51, Kapitel 3.6.1]: Einem Feature wird der Name der beinhaltenden Feature-Gruppe vorangestellt, getrennt durch einen Punkt. Dies lässt sich bis zur Wurzel des Featuremodells fortsetzen. Beispiele für qualifizierende Feature-Namen sind:

```
1 "Secure Connection".VPN
2 Peripherals."Air Condition Control"
3 "Home Automation System"."Identification Mechanism"
```

Index- und Wildcard-Zugriff Features mit Multiplizität größer eins können in einer Featurekonfiguration in mehreren Instanzen vorkommen. Ähnlich wie beim vorhergehenden Problem kann es dadurch zu Mehrdeutigkeiten kommen: Der Ausdruck `"Add-on Package"."Mobile App"` könnte sich im Beispiel auf eine der maximal drei Instanzen des entsprechenden Features beziehen[24]. Um diesem Problem zuvorzukommen, verlangt **FEL** nach einer Referenz auf ein mehrwertiges Feature die Angabe eines Index in eckigen Klammern. Der Index bezieht sich auf die Position des Features innerhalb seiner beinhaltenden Feature-Gruppe in der FK; die Zählung wird bei null begonnen.

Die Anzahl der Instanzen eines Features ist möglicherweise je nach Featurekonfiguration unterschiedlich. Außerdem mag eine boolesche Verknüpfung unterschiedlicher Instanzen, die sich auf dasselbe Feature aus dem Featuremodell beziehen, umständlich sein. Aus diesen Gründen ist zur Angabe eines Index zusätzlich die *Wildcard* (`*`) erlaubt: Sie verknüpft alle in der Featurekonfiguration vorkommenden Instanzen des referenzierten Features über eine ODER-Konjunktion und bezieht sich somit auf das *Subjekt der Variabilität* (vgl. Abschnitt 3.1.2). Folgende Feature-Ausdrücke stellen Beispiele für einen Index- oder Wildcardzugriff dar:

```
1 Keypad[3]
2 "Add-on Package"[0]."Mobile App"
3 "Add-on Package"[*]."Weather Monitor"
4 "Magnetic Card"[*]
```

In der vorliegenden Featurekonfiguration existiert nur eine Instanz des Features „Keypad“. Für den Ausdruck in der ersten Zeile würde sich demnach der Selektionszustand *unselected* ergeben: Zugriffe über die durch die tatsächliche Kardinalität definierte Indexgrenze hinaus sind (unter Berücksichtigung der `maxCardinality`) erlaubt. In diesen Fällen verhalten sich nicht vorhandene Instanzen wie deselektierte Features.

Boolesche Konstanten Weitere erlaubte Feature-Ausdrücke sind die Konstanten `true` und `false`, welche jeweils die Selektionszustände *selected* bzw. *unselected* haben. Sie erlauben das Mapping von Domänenmodell-Elementen ohne Bezug zu FM oder FK.

Boolesche Verknüpfungen: And, Or, Xor, Not Feature-Referenzen und boolesche Konstanten können über *boolesche Operatoren* miteinander verknüpft werden. **FEL** stellt hierzu die binären Operatoren `and`, `or` und `xor` sowie das unäre `not` zur Verfügung. Für die binären Operatoren ist keine Präzedenzregel definiert; sie verhalten sich linksassoziativ. Ist eine abweichende Assoziativität gewünscht, können beliebig verschachtelte runde Klammern gesetzt werden. Die Semantik der Operatoren wird im folgenden erläutert:

not Der Selektionszustand des folgenden Ausdrucks wird invertiert (*selected* wird zu *unselected*, und umgekehrt).

and Gibt den Selektionszustand *selected* zurück, falls beide verknüpften Feature-Ausdrücke *selected* sind.

or *selected*, falls mindestens einer der Operanden *selected* ist.

xor *selected*, wenn genau einer der Operanden *selected* ist, sonst *unselected*.

[24] oder aber auch auf die Gesamtheit aller Instanzen (vgl. Abschnitt 3.2.4, Stichworte *Subjekt-* bzw. *Objektbezogene* Manifestation der Multplizität).

Folgende Auflistung enthält einige gültige Feature-Ausdrücke, die boolesche Verknüpfungen enthalten und sich wiederum auf das Featuremodell und die Featurekonfiguration aus Abbildung 4.3 (s. S. 59) beziehen:

```
1 not Keypad[*]
2 true and not false
3 "Add-on Package"[0]."Mobile App" or "Add-on Package"[0]."IPad App"
4 "Secure Connection" and (VPN xor SSH)
```

Zugrundeliegende Xtext-Grammatik Quellcode 4.1 enthält den für Feature-Ausdrücke relevanten Auszug aus der der *Xtext*-Grammatik (vgl. Abschnitt 2.5), die die Sprache **FEL** definiert. Die Regeln in den Zeilen 9 bis 13 sind rekursiv formuliert, um eine beliebige Klammerung der Ausdrücke zu ermöglichen. Die Regeln `FeatureReference`, `FeaturePath` und `FeaturePathSegment` beschreiben den Aufbau einer Feature-Referenz und werden unter dem nachfolgenden Absatz als Ecore-Modellklassen näher betrachtet.

```
1  import "http://www.eclipse.org/emf/2002/Ecore" as ecore
2
3  generate fel "http://fel.f2dmm.ai1.ubt.de/1.0"
4
5  FELExpr : FeatureExpr | AttrExpr ;
6
7  FeatureExpr : OrExpr ;
8
9  OrExpr returns FeatureExpr : XorExpr ({OrExpr.left=current} 'or'
       right=FeatureExpr)* ;
10
11 XorExpr returns FeatureExpr : AndExpr ({XorExpr.left=current} 'xor'
       right=FeatureExpr)* ;
12
13 AndExpr returns FeatureExpr : PrimaryExpr ({AndExpr.left=current} 'and'
       right=FeatureExpr)* ;
14
15 PrimaryExpr returns FeatureExpr : NegativeExpr | BooleanExpr | FeatureReference |
       '(' OrExpr ')' ;
16
17 NegativeExpr : 'not' expr=FeatureExpr ;
18
19 BooleanExpr : value=Boolean ;
20
21 FeatureReference : featurePath=FeaturePath ('('
       attributeConstraint=AttributeConstraint ')')? ;
22
23 FeaturePath : (segments+=FeaturePathSegment) ('.' segments+=FeaturePathSegment)* ;
24
25 FeaturePathSegment : featureName=(ID|STRING) ('[' (index=INT | wildcard?='*') ']')?
       ;
26 Boolean returns ecore::EBoolean : 'true' | 'false' ;
```

Quelltext 4.1: Xtext-Grammatik für Feature-Ausdrücke. Die Regel `AttributeConstraint` wird im nächsten Unterabschnitt erläutert, `AttrExpr` in Abschnitt 4.4.3. Terminalsymbole wurden außer Acht gelassen.

Ecore-Modell Abbildung 4.11 zeigt ein Ecore-Klassendiagramm des Ausschnitts des FEL-Metamodells, welches aus der obigen Xtext-Grammatik generiert wurde. Die Xtext-Laufzeitumgebung erzeugt beim Parsen eines FEL-Dokuments jeweils eine Instanz von `FELExpr`, deren Subtyp `FeatureExpr` Feature-Ausdrücke repräsentiert. Jede Unterklasse von `FeatureExpr` muss das abgeleitete Attribut `state` vom Typ `FELState` implementieren, welches den Selektionszustand des jeweiligen Ausdrucks bestimmt. Die Unterklassen von `CompositeExpr` verschachteln weitere Feature-Ausdrücke paarweise rekursiv über die Containment-Beziehungen `left` bzw. `right`; auf entsprechende Weise verweist `NegativeExpr` über `expr` auf den zu negierenden Ausdruck.

Eine Feature-Referenz setzt sich aus einem qualifizierenden Pfad (`FeaturePath`) und einem optionalen Attribut-Constraint (s. nächster Abschnitt) zusammen. Jedes Segment des Pfades enthält neben dem Namen des Features einen eventuellen Index. Ist kein Index notiert, wird der Standardwert -1 angenommen. Das boolesche Attribut `wildcard` hat den Wert `true`, falls anstatt eines Index die Wildcard gesetzt wurde.

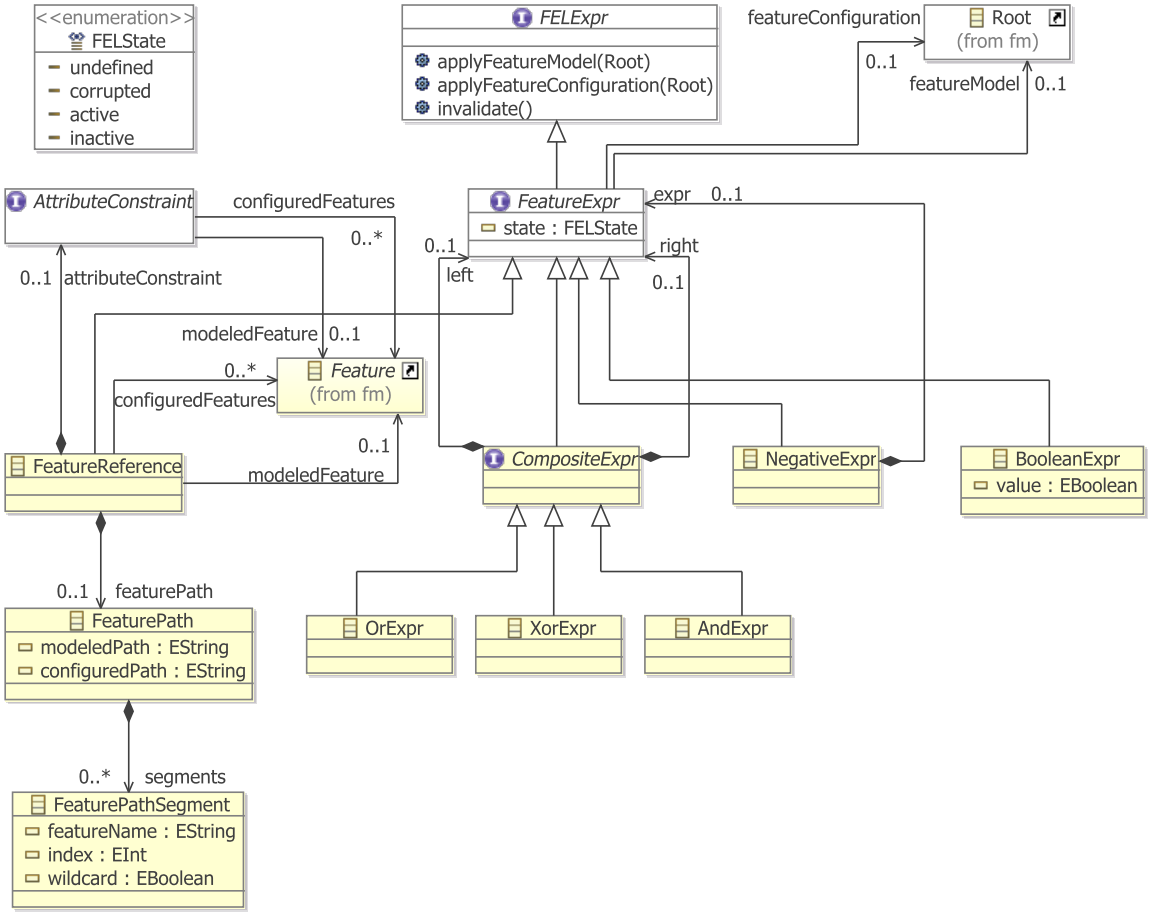

Abbildung 4.11: Von Xtext generiertes und manuell ergänztes Ecore-Modell, das der oben beschriebenen Grammatik zugrunde liegt. Die Vorgehensweise von der Grammatik zum anpassbaren Metamodell wurde in Abschnitt 2.5.3 beschrieben.

Verbindung zu Featuremodell und -konfiguration Die Verbindung zwischen Feature-Ausdruck und entsprechendem FM bzw. FK wird temporär nach dem Parsen eines Feature-Ausdrucks hergestellt: Die nachträglich hinzugefügten Methoden `applyFeatureModel(Root)` bzw. `applyFeatureConfiguration(Root)` der Schnittstelle `FELExpr` bewirken in allen nicht-abstrakten Unterklassen das Setzen der Referenzen `featureModel` bzw. `featureConfiguration` auf das übergebene Element. Bei verschachtelten Ausdrücken erfolgt der Funktionsaufruf rekursiv, so dass jedes Element eines geparsten Feature-Ausdrucks, insbesondere die Feature-Referenzen, auf das FM sowie die aktuelle FK zugreifen können.

Sobald ein Feature-Ausdruck mit einem Featuremodell sowie einer -konfiguration assoziiert wurde, wird in Instanzen von `FeatureReference` nach Features gesucht, die diesem Ausdruck entsprechen. Während eine Feature-Referenz nach Auflösung eventueller Mehrdeutigkeiten durch die Verwendung von qualifizierenden Namen genau ein Feature aus dem Featuremodell repräsentiert, können beim Gebrauch von Wildcards mehrere Features aus der geladenen Featurekonfiguration assoziiert sein. Dies spiegelt sich in den Wertigkeiten der abgeleiteten, nicht-flüchtigen Referenzen `modeledFeature` (`0..1`) und `configuredFeatures` (`0..*`) wider, die auf die entsprechenden Elemente aus FM und FK verweisen.

Die Methode `invalidate()` aus dem Interface `FELExpr` wird immer dann aufgerufen, wenn sich Änderungen im Featuremodell oder der -konfiguration ergeben haben, sowie nach dem erstmaligen Setzen letzterer. Sie implementiert das Zurücksetzen der abgeleiteten Attribute sowie des Selektionszustands bei Bedarf (vgl. auch Abschnitt 4.7.6).

4.4.2 Feature-Ausdrücke mit Constraints auf Attributwerten

In Abschnitt 3.2.5 wurde diskutiert, inwiefern sich das Variabilitätsmerkmal *Optionalität* in Feature-Attributen äußern kann. Eine Möglichkeit bestand in der Abhängigkeit einer Abbildung von sog. *Attribut-Constraints*: Ein Domänenmodell-Element realisiert ein Feature nur dann, wenn bestimmte Bedingungen auf dem Wert eines Attributs des Features gelten. In **FEL** integrierte Attribut-Constraints erlauben den Vergleich eines tatsächlichen Attribut-Werts mit einem Soll-Wert und weisen einem Feature-Ausdruck nur dann den Selektionszustand *selected* zu, falls die Bedingung zusätzlich von der aktuell geladenen FK erfüllt wird.

Elementare Constraints: Arten des Vergleichs Die Notation eines Attribut-Constraints erfolgt nach einer einzuschränkenden Feature-Referenz in runden Klammern. Sog. *elementare Constraints* vergleichen den Wert eines Attributs mit einem Soll-Wert unter Verwendung eines *Vergleichs-Operators*. Die Bedingungen werden nur ausgewertet, wenn die Feature-Referenz für die aktuelle Featurekonfiguration den Selektionszustand *active* hat. Trifft ein Constraint nicht zu, so wird der Selektionszustand zu *inactive*. In **FEL** sind folgende sechs Vergleichs-Operatoren definiert:

= **(Gleichheit)** Bezieht sich auf die lexikalische Gleichheit des aktuellen mit dem Soll-Wert.

<> **(Ungleichheit)** Ebenfalls bezogen auf lexikalische Gleichheit, jedoch mit umgekehrtem Wahrheitswert.

< **(Kleiner)** Stellen tatsächlicher Wert und Soll-Wert Zahlen dar, so wird der numerische Kleiner-Vergleich herangezogen, ansonsten der lexikalische.

<= **(Kleiner oder gleich)** Trifft zu im Falle einer lexikalischen Gleichheit oder eines numerischen Kleiner-Vergleichs.

> **(Größer)** Stellen tatsächlicher Wert und Soll-Wert Zahlen dar, so wird der numerische Größer-Vergleich herangezogen, ansonsten der lexikalische.

>= **(Größer oder gleich)** Trifft zu im Falle einer lexikalischen Gleichheit oder eines numerischen Größer-Vergleichs.

Attribute, die zum Vergleich herangezogen werden sollen, werden ebenfalls über ihren Namen referenziert. Wie auch bei Feature-Referenzen werden doppelte Hochkommata verlangt, sobald diese Leer- oder Sonderzeichen enthalten. Folgende Auflistung enthält einige Beispiele für Attribut-Constraints, wiederum bezogen auf Featuremodell und -konfiguration aus Abbildung 4.3 (s. S. 59):

```
1 not "Air Condition Control"("Min Temperature" >= 20)
2 "Secore Connection".VPN("Vendor Protocol" = "Cisco")
3 "Add-on-Package"[0]."Mobile App"(Platform = "Android")
```

Ohne Attribut-Constraint würde der Feature-Ausdruck aus Zeile 3 den Selektionszustand *active* ergeben. Jedoch trifft der Vergleich `Platform = "Android"` in der geladenen Featurekonfiguration nicht zu und der gesamte Ausdruck evaluiert zu *inactive*.

Boolesche Verknüpfungen von Constraints Wie auch Feature-Ausdrücke können Attribut-Constraints über boolesche Verknüpfungen in beliebiger Schachtelungstiefe miteinander kombiniert werden. Die Syntax und die Semantik der Operatoren sind identisch mit der für Feature-Ausdrücke. Für die Auswertung der Ausdrücke gilt das selbe wie für elementare Attribut-Constraints: Sie werden nur in Betracht gezogen, falls der übergeordnete Feature-Ausdruck *active* ergibt. Falls dann der Attribut-Constraint *inactive* ergibt, wird dieser Selektionszustand an den Feature-Ausdruck weiterpropagiert. Folgende Auflistung enthält einen weiteren, komplexen Feature-Ausdruck mit verschachteltem Attribut-Constraint:

```
1 "Add-on-Package"[0]."Mobile App"(Platform = "Android 3" or Platform = "Android 4")
    and not "Add-on-Package[0]."IPad App"
```

Zugrundeliegende Xtext-Grammatik Die Xtext-Grammatik in Quellcode 4.2 ist ergänzend zur Basisgrammatik (s. Quellcode 4.1) zu betrachten. Die Verschachtelung boolescher Ausdrücke wurde analog realisiert; pro Vergleichsoperator existiert jeweils eine eigene Regel.

```
1 AttributeConstraint : OrAttributeConstraint ;
2
3 OrAttributeConstraint returns AttributeConstraint :
    XorAttributeConstraint({OrAttributeConstraint.left=current} 'or'
    right=AttributeConstraint)* ;
4
5 XorAttributeConstraint returns AttributeConstraint :
    AndAttributeConstraint({XorAttributeConstraint.left=current} 'xor'
    right=AttributeConstraint)* ;
6
7 AndAttributeConstraint returns AttributeConstraint :
    PrimaryAttributeConstraint({AndAttributeConstraint.left=current} 'and'
    right=AttributeConstraint)* ;
8
9 PrimaryAttributeConstraint returns AttributeConstraint :
    ElementaryAttributeConstraint
```

```
10      | NegativeAttributeConstraint
11      | '(' OrAttributeConstraint ')' ;
12
13  ElementaryAttributeConstraint : EqualsAttributeConstraint
14      | InequalsAttributeConstraint
15      | GreaterAttributeConstraint
16      | LessAttributeConstraint
17      | GeqAttributeConstraint
18      | LeqAttributeConstraint ;
19  NegativeAttributeConstraint : 'not' constraint=AttributeConstraint ;
20
21  EqualsAttributeConstraint : attributeName=(ID|STRING) ('=') (intVal=INT |
        doubleVal=DOUBLE | strVal=(ID|STRING)) ;
22
23  InequalsAttributeConstraint : attributeName=(ID|STRING) ('<>') (intVal=INT |
        doubleVal=DOUBLE | strVal=(ID|STRING)) ;
24
25  GreaterAttributeConstraint : attributeName=(ID|STRING) ('>') (intVal=INT |
        doubleVal=DOUBLE | strVal=(ID|STRING)) ;
26
27  LessAttributeConstraint : attributeName=(ID|STRING) ('<') (intVal=INT |
        doubleVal=DOUBLE | strVal=(ID|STRING)) ;
28
29  GeqAttributeConstraint : attributeName=(ID|STRING) ('>=') (intVal=INT |
        doubleVal=DOUBLE | strVal=(ID|STRING)) ;
30
31  LeqAttributeConstraint : attributeName=(ID|STRING) ('<=') (intVal=INT |
        doubleVal=DOUBLE | strVal=(ID|STRING)) ;
```

Quelltext 4.2: Für Attribut-Constraints relevanter Auszug aus der **FEL**-Grammatik. Die Linksassoziativität der booleschen Verknüpfungen wird wieder durch rekursive Regelanwendung sichergestellt.

Ecore-Modell Das Ecore-Klassendiagramm in Abbildung 4.12 zeigt die Vererbungshierarchie ab `AttributeConstraint`. Alle Unterklassen beinhalten jeweils eine Referenz auf das FM und die geladene FK, sobald diese dem übergeordneten Feature-Ausdruck zugeordnet wurden. Letzterer stellt durch Aufruf der Methoden `applyModeledFeature(Feature)` und `applyConfiguredFeatures(Feature)` sicher, dass die Referenzen `modeledFeature` sowie `configuredFeature` denen des übergeordneten Feature-Ausdrucks entsprechen. Bei negativen und zusammengesetzten Attribut-Constraints wird rekursiv weiterpropagiert. Dies findet ebenfalls bei einer *Invalidierung* des Feature-Ausdrucks statt (s. Abschnitt 4.7.6).

4.4.3 Attribut-Ausdrücke: Abfrage von Attributwerten der aktuellen Featurekonfiguration

Neben Feature-Ausdrücken erlaubt **FEL** auch sog. *Attribut-Ausdrücke*, die den Wert eines Feature-Attributs der geladenen FK ermitteln. Attribut-Ausdrücke werden innerhalb sog. *Pattern-Ausdrücke* in *Alternativen-Mappings*, welche in Abschnitt 4.6.2 vorgestellt werden, verwendet. Da sie Bestandteil der FEL-Sprache sind, trifft auch für sie der in den vorhergehenden Absätzen beschriebene Invalidierungsmechanismus zu.

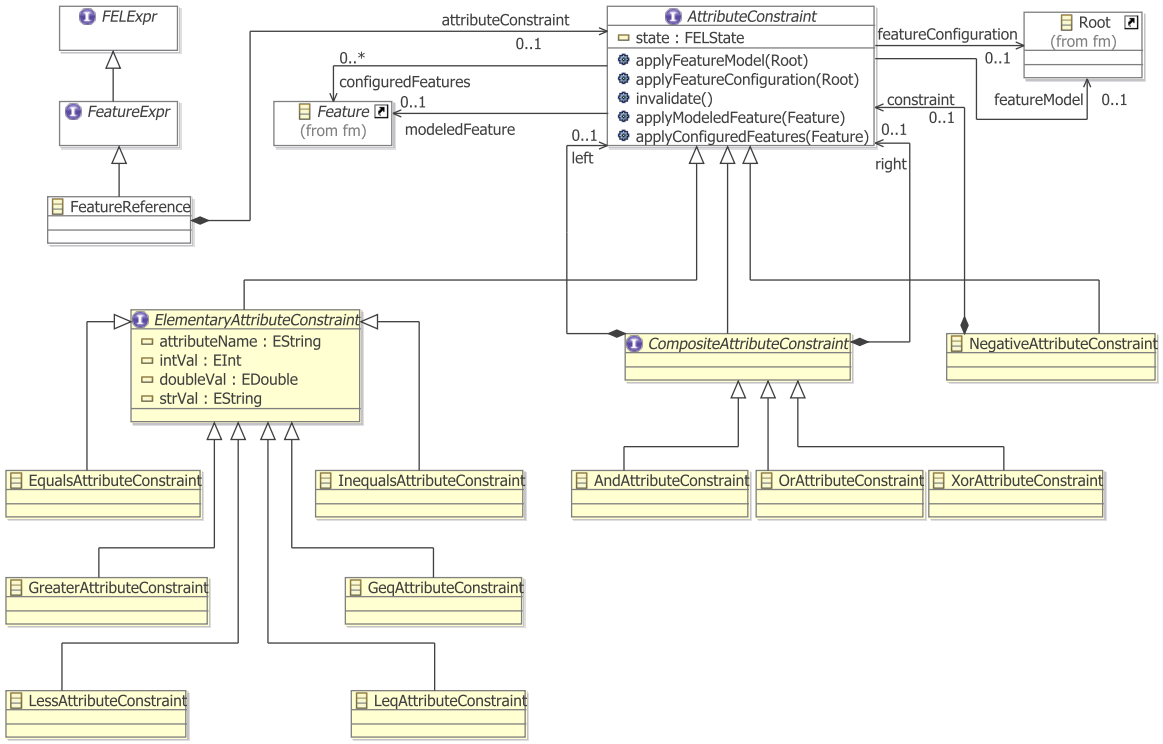

Abbildung 4.12: Ecore-Klassendiagramm für das Interface `AttributeConstraint` und seine Unterklassen.

Syntax Attribut-Ausdrücke sind grundsätzlich innerhalb geschweifter Klammern mit dem Präfix „`#`“ notiert. Sie setzen sich aus einer Feature-Referenz und der Angabe eines Attribut-Namens, in der textuellen Notation durch einen Doppelpunkt getrennt, zusammen. Nachfolgende Auflistung zeigt einige Beispiele für Attribut-Ausdrücke, die sich explizit auf den Wert von Attributen einer Featurekonfiguration beziehen:

```
#{"Air Condition Control":"Max Temperature"}
#{"Add-on-Package"[0]."Mobile App":Platform}
```

Zieht man einmal mehr Featuremodell und -konfiguration aus Abbildung 4.3 in Betracht, würden obige Attribut-Ausdrücke die Werte „`27`“ bzw. „`IOS`“ ergeben. Die Angabe ungültiger Attribut-Ausdrücke wird von der Modellvalidierung (s. Abschnitt 4.9.3) behandelt.

Implizite abgeleitete Attribute Zusätzlich zu im Featuremodell festgelegten Attributen ermöglicht FEL auch die Anfrage zweier *impliziter abgeleiteter Attribute* durch spezielle Sprachkonstrukte:

cardinality Gibt einen Ganzzahlwert zurück, der die tatsächlich vorhandene Anzahl der Instanzen des referenzierten Features in der geladenen Featurekonfiguration angibt.

select Gibt im Gegensatz zu `cardinality` nur die Anzahl derjenigen Instanzen zurück, die in der aktuellen Featurekonfiguration selektiert sind.

Die Neuauswertung dieser Attribute wird bei Änderungen innerhalb der Featurekonfiguration automatisch durch den Invalidierungsmechanismus erzwungen. Die textuelle Notation unterscheidet sich von expliziten Attribut-Angaben durch die Trennung mittels doppeltem Kolon („::“, z.B. `"Add-on-Package"::select`).

Zugrundeliegende Xtext-Grammatik Die in den beiden vorhergehenden Absätzen dokumentierte Xtext-Grammatik für **FEL** wird durch den in Quellcode 4.3 dargestellten Abschnitt komplettiert. Die impliziten Attribute `cardinality` und `select` sind jeweils durch eigene Produktionsregeln realisiert.

```
1 AttrExpr : '#{' featureRef=FeatureReference attrDesc=AttrDesc '}' ;
2
3 AttrDesc : ExplicitAttrDesc | CardinalityAttrDesc | SelectAttrDesc ;
4
5 CardinalityAttrDesc : {CardinalityAttrDesc} '::' 'cardinality' ;
6
7 SelectAttrDesc : {SelectAttrDesc} '::' 'select' ;
8
9 ExplicitAttrDesc : ':' attrName=(ID|STRING) ;
```

Quelltext 4.3: Ausschnitt der **FEL**-Grammatik mit den entsprechenden Regeln für Attribut-Ausdrücke.

Ecore-Modell Der in Abbildung 4.13 gezeigte Ausschnitt des **FEL**-Metamodells bedurfte keiner weiteren Anpassung nach der Generierung aus obiger Grammatik. Wie auch bei Feature-Ausdrücken ist eine Auswertung erst möglich, wenn ein FM und eine entsprechende FK zugewiesen und der Invalidierungsmechanismus angestoßen wurde.

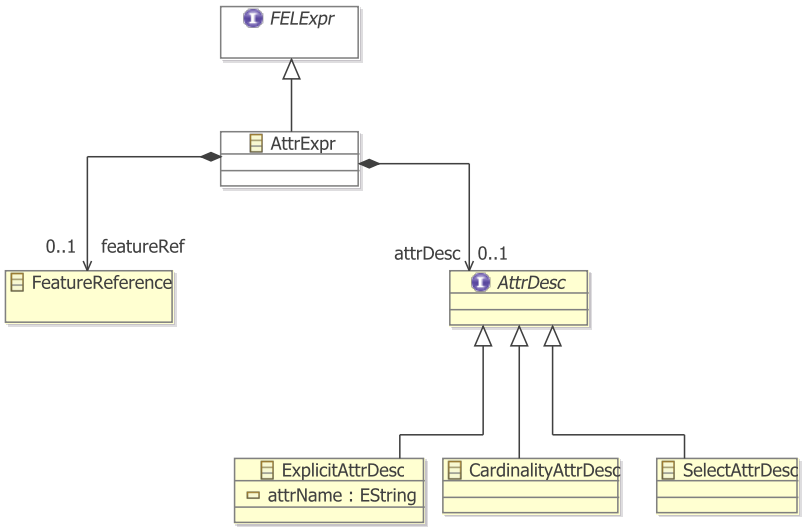

Abbildung 4.13: Für Attribut-Ausdrücke relevanter Ausschnitt des Ecore-Klassendiagramms für das **FEL**-Metamodell.

4.4.4 Serialisierung von Feature-Ausdrücken

Die Sprache **FEL** dient, wie bereits angedeutet, als textuelle Ergänzungssprache für **F2DMM**-basierte Mapping-Modelle. Feature-Ausdrücke werden nicht in einer eigenen Ressource, sondern innerhalb der Mapping-Ressource persistiert. Die Ausdrücke werden jedoch nicht als Teilbaum in der gängigen **XMI**-Serialisierung, sondern in ihrer textuellen Repräsentation hinterlegt. Diese Entscheidung wurde aus folgenden Gründen getroffen:

- Um die *Robustheit* gegenüber Änderungen im FM gewährleisten, werden in Mapping-Modellen keine Referenzen zu Features, mit Ausnahme von `Root`-Elementen, hinterlegt. Das Binden an die jeweiligen Feature-Instanzen erfolgt, wie oben beschrieben, während der Auswertung zur Laufzeit. Ungültige Referenzen werden weniger restriktiv gehandhabt als es die **EMF**-Deserialisierung vorsieht. Die Verwendung qualifizierender Ausdrücke garantiert die *Eindeutigkeit* der von Mappings assoziierten Feature-Ausdrücke.
- Außerdem war die *Kompaktheit* von FEL-Ausdrücken ein ausschlaggebender Grund: Jede Produktionsregel der FEL-Grammatik entspräche einem XML-Tag in der **XMI**-Serialisierung des Ausdrucks. Insbesondere bei verschachtelten FEL-Ausdrücken hätte dies einen erhöhten Speicherbedarf zur Folge.

4.5 SIDRL: Eine Sprache zur Formulierung modellspezifischer Abhängigkeitsbedingungen

Neben der im vorhergehenden Abschnitt vorgestellten Sprache **FEL** stellt die **F2DMM**-Werkzeugfamilie eine weitere textuelle Sprache zur Verfügung, die sich jedoch nicht auf Featuremodelle bzw. -konfigurationen, sondern auf das Metamodell des für das Mapping verwendeten Multivarianten-Domänenmodells (**DM**) bezieht. Der Anwendungsbereich der Sprache **SDIRL** ist die Definition von *strukturellen Abhängigkeiten* (vgl. Abschnitt 3.3) auf dem Domänen-Metamodell. Mit Hilfe dieser Abhängigkeiten lassen sich Inkonsistenzen innerhalb des Mapping-Modells erkennen und schließlich *Reparaturaktionen* ableiten, welche die Konsistenz von Produkten wiederherstellen. Das Akronym **SDIRL** setzt sich abermals aus einem englischsprachigen Begriff zusammen: *Structural Dependency Identification and Repair Language* (Sprache zur Identifikation und Reparatur struktureller Abhängigkeiten). Zunächst wird auf Aufbau und Grammatik der Sprache eingegangen, bevor einige Implementierungsspezifika wie die **OCL**-Integration im Vordergrund stehen.

4.5.1 Aufbau und Semantik eines SDIRL-Dokuments

Zu Beginn eines **SDIRL**-Dokuments werden ein oder mehrere referenzierte Ecore-Modelle über deren *Namespace-URI*s angegeben. Vorangestellt wird das Schlüsselwort `import`. Klassen eines importierten Namensraums können unter Angabe des voll qualifizierenden Namens (Paket-Hierarchie, durch Punkte getrennt vor dem Klassennamen notiert) innerhalb des gesamten Dokuments referenziert werden.

Ein **SDIRL**-Dokument beschreibt durch eine Menge von sog. *Abhängigkeitsdefinitionen* strukturelle, nicht-Containment-Abhängkeiten zwischen Elementen des Domänen-Metamodells (vgl. Abschnitt 3.3.2). Eine SDIRL-Abhängigkeitsdefinition wird durch einen eindeutigen Bezeichner identifiziert und besteht aus den folgenden vier Komponenten:

Kontext-Element Gekennzeichnet durch das Schlüsselwort `element`. Es definiert eine *freie Variable* für ein Element aus dem Domänenmodell, für das Abhängigkeiten berechnet werden sollen.

Abhängiges Element Gekennzeichnet durch `requires`. Es wird an eine weitere freie Variable aus dem Domänenmodell gebunden, um die Abhängigkeit des Kontext-Elements von ihm festzustellen.

Abhängigkeitsbedingung Repräsentiert durch einen booleschen **OCL**-Ausdruck nach dem Schlüsselwort `when`. Innerhalb des OCL-Ausdrucks können die in den `element`- bzw. `requires`-Komponenten definierten Variablen verwendet werden. Die Abhängigkeit trifft zu, falls der OCL-Ausdruck den Wahrheitswert `true` ergibt.

Surrogate (optional) Wiederum durch einen OCL-Ausdruck definiert. Beliebig viele `surrogate`-Ausdrücke können definiert werden, um mögliche *Stellvertreter* für das an die in `requires` deklarierte Variable gebundene Element zu berechnen und Informationsverlust durch Propagation zu verhindern (vgl. Abschnitt 3.3.4). Der Typ des OCL-Ausdrucks muss kompatibel mit dem Typ des abhängigen Elements und darf mengenwertig sein.

Beim Laden eines Mapping-Modells in den **F2DMM**-Editor werden zunächst strukturelle Abhängigkeiten auf Basis des referenzierten **SDIRL**-Dokuments berechnet (s. Abschnitt 4.7.3). Hierzu wird für jedes Paar von Elementen des Multivarianten-**DM** zunächst die Typverträglichkeit mit dem Kontext- bzw. abhängigen Element jeder Abhängigkeitsdefinition geprüft. Ist diese gegeben, erfolgt die Auswertung des `when`-Ausdrucks nach dem Binden der Variablen (s. Unterabschnitt 4.5.4) an den OCL-Ausdruck. Liefert dieser den Wahrheitswert `true` zurück, wird eine Abhängigkeit festgehalten.

```
1  import "http://www.eclipse.org/uml2/3.0.0/UML"
2
3  dependency SuperClassifier {
4    element spec : uml.Classifier
5    requires gen : uml.Classifier
6    when {
7      spec.general->includes(gen)
8    }
9    surrogate {
10     gen.general.allParents()
11   }
12 }
```

Quelltext 4.4: Beispiel für eine SDIRL-Abhängigkeitsdefinition mit Surrogat. Die `when`-Bedingung definiert eine Abhängigkeit zwischen einer UML-Klasse und einer ihrer Oberklassen. Deren Oberklassen können sie als Referenzziel ersetzen.

4.5.2 Xtext-Grammatik für SDIRL

Die Sprache **SDIRL** basiert ebenfalls auf dem textuellen Modellierungsrahmenwerk *Xtext* (s. Abschnitt 2.5). Die entsprechende Grammatik ist in Quelltext 4.5 abgebildet. Aufgrund der restriktiv definierten Syntax sind nur vier Produktionsregeln notwendig: Ein SDIRL-Modell (`StructuralDependencyModel`) setzt sich aus Import-Deklarationen und

Abhängigkeitsdefinitionen (`PackageImport` bzw. `Dependency`) zusammen. Durch Anwendung der Produktionsregel `Dependency` werden der generierten Ecore-Klasse die Attribute `name` (Bezeichner für die definierte Abhängigkeit), `elementName` (Bezeichner für die Variable, die das Kontext-Element repräsentiert) und `requiredName` (Variablenbezeichner für das abhängige Element) ermittelt. In `when`- und `surrogate`-Blöcken sind OCL-Ausdrücke (`OCLModel`) enthalten, auf die im nachfolgenden Absatz eingegangen wird. Das angewandte Auftreten referenzierter Ecore-Datentypen in `elementType` und `requiredType` wird im übernächsten Absatz erläutert.

```
1  grammar de.ubt.ai1.f2dmm.sdirl.Sdirl
2  hidden(WS, ML_COMMENT, SL_COMMENT)
3
4  import "http://www.eclipse.org/emf/2002/Ecore" as ecore
5
6  generate sdirl "http://sdirl.f2dmm.ai1.ubt.de/1.0"
7
8  StructuralDependencyModel:
9         (packageImports+=PackageImport)*
10        (dependencies+=Dependency)*;
11
12 PackageImport: 'import' importURI=STRING;
13
14 Dependency:
15     'dependency' name=SIMPLE_ID '{'
16         'element' elementName=SIMPLE_ID ':' elementType=[ecore::EClass|FQN]
17         'requires' requiredName=SIMPLE_ID ':' requiredType=[ecore::EClass|FQN]
18         'when' '{'
19             when=OCLModel
20         '}'
21         ('surrogate' '{'
22             surrogates+=OCLModel
23         '}')*
24     '}';
25
26 FQN: ID ("." ID)*;
```

Quelltext 4.5: Xtext-Grammatik für die Sprache **SDIRL**. Nicht enthalten sind Produktionsregeln von Essential **OCL** (vgl. Abschnitt 2.6.1, Startsymbol `OCLModel`), welches durch SDIRL erweitert wird.

Generiertes Ecore-Modell Das von obiger Xtext-Grammatik generierte Ecore-Modell (s. Abbildung 4.14) bedurfte keiner weiteren manuellen Anpassung. Validierung und Auswertung der Ausdrücke sind Aufgabe des **SDIRL**-Editors (s. Abschnitt 4.5.3) bzw. der in den **F2DMM**-Editor integrierten Invalidierungsmechanismen (4.7.6).

Integration von OCL Eine besondere Herausforderung bei der Formulierung der Xtext-Grammatik für **SDIRL** war die Integration der textuellen Anfragesprache *Essential* **OCL** (vgl. Abschnitt 2.6.1). Xtext unterstützt zwar einen Vererbungsmechanismus auf Grammatik-Ebene (vgl. Abschnitt 2.5.1), jedoch hätte die Erweiterung der Essential-OCL-Grammatik verschiedene Nachteile mit sich gebracht:

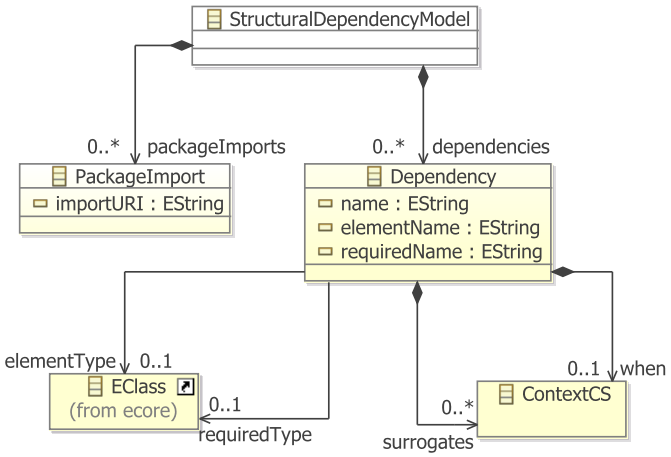

Abbildung 4.14: Metamodell für die abstrakte Repräsentation von SDIRL-Dokumenten. Die Klasse `ContextCS` repräsentiert die Regel `OCLModel` aus der Grammatik. Deren Unterklassen sind ebenfalls aus Gründen der Übersichtlichkeit nicht dargestellt.

- Zum einen handelt es sich bei der Xtext-Grammatik für Essential OCL um ein Projekt mit experimentellem Status: Der Build-Prozess ist nicht dokumentiert und der Übergang von der konkreten zur abstrakten Syntax noch nicht vollständig formal spezifiziert.
- Zum anderen hätte die Integration der Xtext-Grammatik im Sinne einer Vererbung die Notwendigkeit der Anpassung einiger OCL-Laufzeitmodule nach sich gezogen[25].

Aus den genannten Gründen wurde bei der Integration von OCL-Ausdrücken in `when`- bzw. `surrogate`-Blöcke der SDIRL-Grammatik auf den von Xtext bereitgestellten Vererbungsmechanismus verzichtet. Stattdessen wurden die Produktionsregeln der Essential-OCL-Grammatik in die SDIRL-Grammatik übernommen und die Referenzen auf das OCL-Metamodell eliminiert, um die Ecore-Klassen von Xtext in das SDIRL-Paket generieren zu lassen. Die syntaktische Korrektheit von OCL-Ausdrücken wird auf diese Weise im Texteditor sichergestellt. Jedoch entfällt beim gewählten Ansatz mangels wiederverwender OCL-Essential-Laufzeitmodule die Möglichkeit für die Erkennung semantischer Fehler und Editierunterstützung wie automatische Quelltextvervollständigung (*Code Completion*) in OCL-Blöcken.

Die gewählte Vorgehensweise erfordert außerdem ein zweimaliges Parsen der enthaltenen OCL-Blöcke: Zunächst erzeugt der Xtext-generierte SDIRL-Parser die konkrete Syntax – also die Modellrepräsentation – des Ausdrucks. Da diese aus den oben beschriebenen Gründen nicht mit dem Essential-OCL-Modell kompatibel ist, werden innerhalb des SDIRL-Editors die OCL-Ausdrücke ein weiteres mal vom entsprechenden Essential-OCL-Laufzeitmodul geparst: Dies geschieht im SDIRL-Validierungsmodul, um die semantische Korrektheit sowie die Kompatibilität der Rückgabewerte der eingebetteten Ausdrücke[26] sicherzustellen. Dem Anwender werden im angepassten SDIRL-Editor entsprechende Fehler angezeigt (s. Unterabschnitt 4.5.3).

[25] Etwa, um das durch das *Linking*-Modul beschriebene *angewandte Auftreten* von Ecore-Klassen hinsichtlich importierter Metamodelle anzupassen.

[26] Typ *Boolean* im `when`-Teil, kompatibel mit der `requires` angegebenen Klasse in `surrogate`-Blöcken.

Referenzieren von Ecore-Klassen Innerhalb der `element`- bzw. `requires`-Teile einer SDIRL-Regel können Ecore-Datentypen aus importierten Namensräumen angegeben werden. Das Referenzieren von Ecore-Klassen durch deren qualifizierenden Namen wird durch die Einbindung der im Rahmen des *Xtext-Ecore-Projekts*[27] entwickelten Ecore-Schnittstelle ermöglicht. Die Produktionsregel `FQN` definiert lediglich ein Symbol für qualifizierende Namen. Das Binden an ein *angewandtes Auftreten* erfolgt durch das Konstrukt `ecore::EClass|FQN`. Die entsprechende *Linking*-**API** (vgl. Abschnitt 2.5.2) wird durch referenzieren des Ecore-Metamodells und des Generator-Modells sowie das Registrieren der **MWE**-Komponente `org.eclipse.ecore.xtext.EcoreSupport` eingebunden:

```
bean = StandaloneSetup {
    (...)
    registerGeneratedEPackage = "de.ubt.ai1.f2dmm.sdirl.SdirlPackage"
    registerGeneratedEPackage = "org.eclipse.emf.ecore.EcorePackage"
    registerGenModelFile =
        "platform:/resource/org.eclipse.emf.ecore/model/Ecore.genmodel"
}

component = org.eclipse.xtext.ecore.EcoreSupport {}
```

Um den Import von Ecore-Namensräumen zu ermöglichen, genügt es, die von Xtext definierten Konventionen für explizite Importe zu beachten [19, Abschnitt 5]: Lautet ein Attribut vom Datentyp `EString` auf den Bezeichner `importURI`, sucht die Xtext-Laufzeitumgebung bei der Anwendung der definierten Produktionsregel nach dem im entsprechenden Dokument spezifizierten Namensraum. Folgende Anpassung im generierten Laufzeitmodul sorgt dafür, dass auch Importe, die nicht auf „`.ecore`“ enden, mit dem Ecore-*Linking*-Laufzeitmodul assoziiert werden:

```
public class SdirlRuntimeModule extends
    de.ubt.ai1.f2dmm.sdirl.AbstractSdirlRuntimeModule {

    @Override
    public Registry bindIResourceServiceProvider$Registry() {
        Registry registry = super.bindIResourceServiceProvider$Registry();
        registry.getExtensionToFactoryMap().put("*",
            registry.getExtensionToFactoryMap().get("ecore"));
        return registry;
    }
}
```

4.5.3 Editor-Unterstützung

Der von Xtext generierte Editor wurde bis auf die im vorhergehenden Unterabschnitten dargestellten Änderungen bezüglich der Auswertung von OCL-Ausdrücken und der Referenzierung von Ecore-Klassen nicht weiter manuell angepasst. Wie bereits beschrieben, wurde die kontextsensitive Validierung von SDIRL-Modellen um die Überprüfung der eingebetteten OCL-Ausdrücke ergänzt. Der Screenshot in Abbildung 4.15 zeigt die Validierung aus Anwendersicht.

[27] `http://download.eclipse.org/modeling/tmf/xtext/javadoc/2.0.1/org/eclipse/xtext/ecore/EcoreSupport.html`

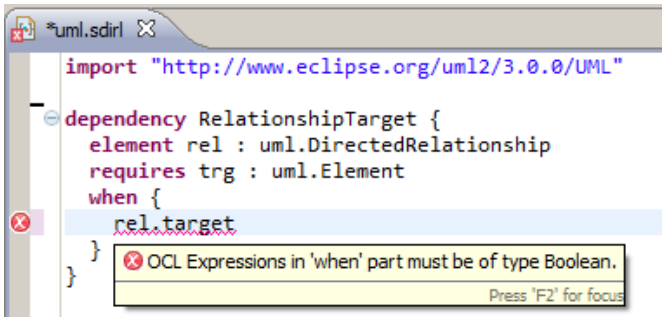

Abbildung 4.15: Bildschirm-Ausschnitt, der die Benutzung des textuellen SDIRL-Editors demonstriert: Der Anwender wird darauf hingewiesen, dass ein OCL-Ausdruck im `when`-Block keinen Wahrheitswert zurückliefert.

4.5.4 Auswertung der eingebetteten OCL-Ausdrücke

Während im **SDIRL**-Editor noch keine Objekte an die in `element` und `requires` deklarierten Variablen gebunden werden, findet innerhalb der **F2DMM**-Laufzeitumgebung eine solches Binden für jedes Paar von gemappten **DM**-Elementen – kompatible Datentypen vorausgesetzt – statt. Die Auswertung der Ausdrücke erfolgt wiederum durch einen zusätzlichen Parse-Schritt, der von dem vom *Eclipse-OCL-Projekt*[28] bereitgestellten Essential-OCL-Interpreter durchgeführt wird. Auch das Binden der Variablen erfolgt mit Hilfe der bereitgestellten **API**. Quelltext 4.6 dokumentiert den Aufruf der Eclipse-OCL-API zur Auswertung von OCL-Ausdrücken zur Laufzeit.

```
1  public boolean isWhenConditionTrue(String elementName, EClass elementType, EObject
       elementValue, String requiredName, EClass requiredType, EObject requiredValue,
       String whenConditionExpr) throws ParserException {
2
3      OCL ocl = OCL.newInstance(EcoreEnvironmentFactory.INSTANCE);
4      OCLHelper<EClassifier, EOperation, EStructuralFeature, Constraint> helper =
           ocl.createOCLHelper();
5
6      Variable<EClassifier, EParameter> elementVar =
           ExpressionsFactory.eINSTANCE.createVariable();
7      elementVar.setName(elementName);
8      elementVar.setType(elementType);
9      ocl.getEnvironment().addElement(elementName, elementVar, true);
10
11     Variable<EClassifier, EParameter> requiredVar =
           ExpressionsFactory.eINSTANCE.createVariable();
12     requiredVar.setName(requiredName);
13     requiredVar.setType(requiredType);
14     ocl.getEnvironment().addElement(requiredName, requiredVar, true);
15
16     OCLExpression<EClassifier> oclExpression =
           helper.createQuery(whenConditionExpr);
17     Query query = ocl.createQuery(oclExpression);
18
19     oclQuery.getQuery().getEvaluationEnvironment().add(elementName, elementValue);
```

[28] `http://www.eclipse.org/modeling/mdt/?project=ocl`

```
20      oclQuery.getQuery().getEvaluationEnvironment().add(requiredName, requiredValue);
21
22      return oclQuery.getQuery().check((EObject) null);
23  }
```

Quelltext 4.6: Java-Methode, die die Überprüfung des `when`-Teils einer SDIRL-Abhängigkeitsdefinition zur Laufzeit implementiert (vereinfachte Darstellung). Zunächst wird eine OCL-Ausführungsumgebung initialisiert (Zeilen 3 und 4) und freie Variablen für `element` bzw. `requires` definiert (Z. 6 bis 14). Anschließend wird der OCL-Ausdruck geparst (Z. 16 und 17) und die zur Laufzeit bekannten Werte an die Variablen gebunden (Z. 19 und 20). Schließlich wird der Ausdruck ausgewertet und der dabei ermittelte Wert zurückgegeben (Z. 22).

In der Implementierung des SDIRL-Moduls wurde die Instanziierung und Ausführung von OCL-Anfragen dieser Art verallgemeinert und hinsichtlich Wiederverwendbarkeit optimiert. Die Klasse `SdirlOCLEvaluator` kapselt alle Anfragen an die Eclipse-OCL-API und stellt Mechanismen wie Ausnahmebehandlung zur Verfügung. Auf die bereitgestellte Funktionalität wird vom SDIRL-Editor sowie von der **F2DMM**-Werkzeugumgebung zugegriffen (s. Abschnitt 4.7.3).

4.5.5 Auswertung von Surrogat-Ausdrücken

Die Auswertung von in `surrogate`-Blöcken definierten OCL-Anfragen erfolgt ähnlich wie bei den im vorhergehenden Unterabschnitt beschriebenen `when`-Bedingungen. Die Verbindung zur Eclipse-OCL-API wird über eine Instanz von `SdirlOCLEvaluator` hergestellt. Innerhalb des **F2DMM**-Editors geschieht dies im Rahmen des *großen Invalidierungszyklus* (vgl. Abschnitt 4.7.6) zur Surrogat-Vor- bzw. Neuberechnung.

Bei erfolgreicher Auswertung eines Surrogat-Ausdrucks wird eine Instanz von `EObject` zurückgegeben, welche ein Element aus dem Kern-Domänenmodell repräsentiert. Das Mapping-Modell ist wiederum in der Lage, das Mapping, welches das Element abbildet, zu ermitteln, um *Reparaturaktionen* abzuleiten, welche Gegenstand von Abschnitt 4.8.5 sind. Bei mengenwertigen Rückgabewerten wird iterativ vorgegangen.

4.6 Mapping-Beschreibungen und Alternativen-Mappings

In den vorhergehenden Abschnitten wurde darauf eingegangen, wie die Verknüpfung eines Mappings zum Featuremodell und der Featurekonfiguration erfolgt. Dieser Abschnitt legt den Fokus auf die Verknüpfung zum Domänenmodell. Abbildung 4.6 stellt diese Verknüpfung vereinfacht als strukturelle Eigenschaften der `Mapping`-Unterklassen dar, nämlich über die Referenzen `mappedObject` bzw. `referencedMapping` und das Attribut `attrValue`. Zusätzlich wird auf das abgebildete Attribut bzw. die abgebildete Referenz aus dem Metamodell verwiesen. In der tatsächlichen Implementierung des **F2DMM**-Metamodells ist an dieser Stelle ein weiterer Abstraktionsschritt vorgesehen, die sog. *Mapping-Beschreibungen*. Zunächst werden deren zugrundeliegende Konzepte erläutert, bevor auf die Benutzersicht, die Erzeugung von *Alternativen-Mappings*, eingegangen wird.

4.6.1 Die Konzeptsicht: Mapping-Beschreibungen als Verknüpfung zum Domänenmodell

Mappings beschreiben nicht selbst abgebildete Artefakte des Domänenmodells; vielmehr übernehmen Mapping-Beschreibungen als Unterklassen von der F2DMM-Modellklasse `MappingDescription` diese Aufgabe. Ziel dieser weiteren Abstraktion ist die Unterstützung verschiedener Arten der Abbildung: Es wurde bereits darauf eingegangen, dass Attribut-Mappings beispielsweise sog. *Patterns* enthalten können, über die mit Hilfe von Attribut-Ausdrücken (s. Abschnitt 4.4.3) Bezug zu Attributwerten der aktuellen Featurekonfiguration Bezug genommen werden kann. Ähnliche Mechanismen greifen bei Containment- oder Referenz-Mappings. In den nachfolgenden Unterabschnitten werden die Entwurfsentscheidungen für die unterschiedlichen Arten von Mapping-Beschreibungen erläutert.

Containment-Mapping-Beschreibungen Containment-Mappings beinhalten als Implementierungsklasse von `ContainmentMappingDescriptionContainer` jeweils eine `ContainmentMappingDescription` (vgl. Abbildung 4.16). Die Schnittstelle verweist auf eine `EReference`: Eine Containment-Referenz aus dem Domänen-Metamodell, welche die Beziehung des abgebildeten **DM**-Elements zu seinem `eContainer` modelliert. Sie deklariert außerdem die abgeleitete Referenz `mappedObject` vom Typ `EObject`, welche von den Unterklassen implementiert wird, um das abgebildete Objekt zu repräsentieren. Es existieren zwei Arten von Mapping-Beschreibungen für Containment-Mappings, für die folgendes Verhalten definiert ist:

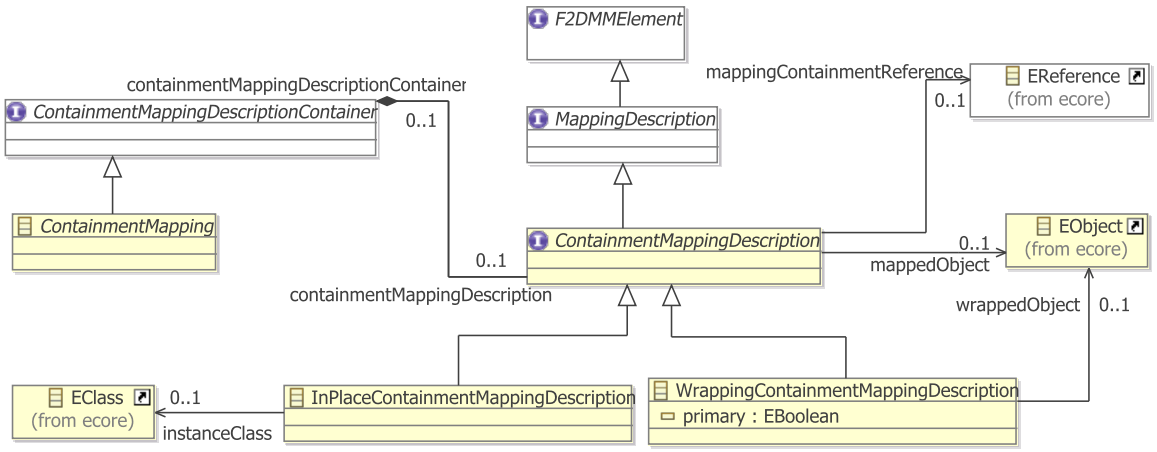

Abbildung 4.16: Beschreibungen für Containment-Mappings werden durch Instanzen von `ContainmentMappingDescription` repräsentiert. Die Referenz `mappedObject` ist abgeleitet (`derived`).

- `WrappingContainmentMappingDescription`: Bezieht sich auf ein vorhandenes **DM**-Element als Instanz von `EObject`. Dieses muss sich innerhalb einer existierenden EMF-Ressource, aber nicht notwendigerweise in der Ressource des Multivarianten-**DM**, befinden. Die abgeleitete Referenz `mappedObject` der implementierten Schnittstelle wird durch den Wert von `wrappedObject` definiert. Weiterhin wird zwischen *primären* und *sekundären* Mapping-Beschreibungen unterschieden: Sekundäre Mapping-Beschreibungen entstehen durch Kopie einer primären Beschreibung. Sie werden ausschließlich von Alternativen-Mappings verwendet.

- `InPlaceContainmentMappingDescription`: Mapping-Beschreibungen dieser Art entstehen ausschließlich benutzergesteuert als Beschreibungen für Alternativen-Mappings. Anstatt auf eine konkrete Instanz zu verweisen, wird hierbei auf eine Klasse aus dem Domänen-Metamodell verwiesen (`instanceClass`), die erst zum Zeitpunkt der Produktableitung als Referenzziel von `mappedObject` instanziiert wird. Verschachtelte Objekte und Attribute können rekursiv unter dem entsprechenden Mapping eingefügt werden.

Attribut-Mapping-Beschreibungen Attribut-Mappings implementieren das Interface `AttributeMappingDescriptionContainer`, welches eine Containment-Referenz zu `AttributeMappingDescription` definiert (vgl. Abbildung 4.17). Dieses Interface implementierende Klassen enthalten eine Referenz `mappingAttribute` vom Typ `EAttribute`, die auf dasjenige Attribut aus dem Domänen-Metamodell verweist, welches die Beziehung des unmittelbar übergeordneten Containment-Mappings zu dem Attributwert beschreibt, der in seiner String-serialisierten Form vom abgeleiteten Attribut `finalMappedValue` bestimmt wird. Für Attribut-Mappings existieren die folgenden zwei konkreten Arten von Mapping-Beschreibungen:

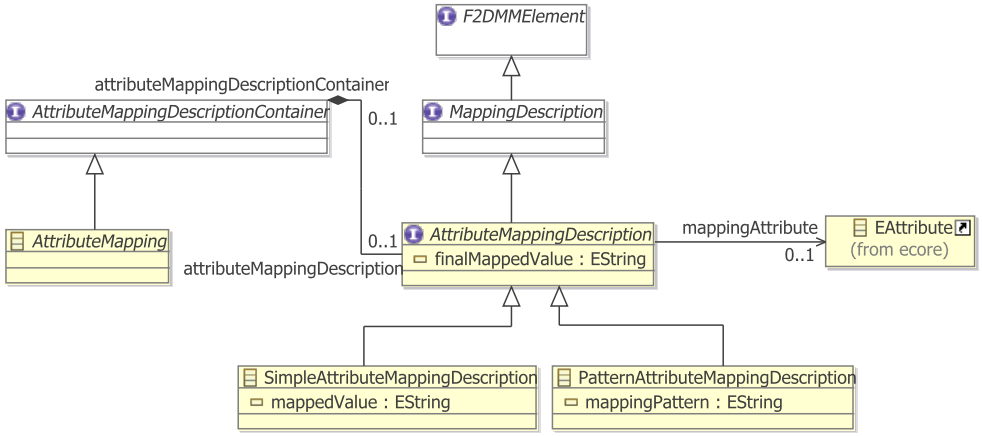

Abbildung 4.17: Das Interface `AttributeMappingDescription` definiert ein abgeleitetes Attribut `finalMappedValue`, welches die String-Repräsentation des abgebildeten Attributwerts enthält.

- `SimpleAttributeMappingDescription`: Beinhaltet direkt den String-serialisierten Wert `mappedValue`, der unverändert an `finalMappedValue` weitergegeben wird.

- `PatternAttributeMappingDescription`: Definiert ein `mappingPattern`, das eine beliebige Anzahl von Attribut-Ausdrücken (vgl. Abschnitt 4.4.3) beinhalten kann. Diese werden aus dem String extrahiert und zur Laufzeit vom **FEL**-Parser in ihre abstrakte Syntax überführt und ausgewertet. Der resultierende Attributwert, ebenfalls ein String, tritt an Stelle des definierenden Patterns. Diese Art des Mappings ist nur für Alternativen-Mappings relevant und ermöglicht dem Anwender, Attribute des abgeleiteten Produkts in Abhängigkeit von der geladenen Featurekonfiguration zu festzulegen.

Referenz-Mapping-Beschreibungen Das vereinfachte F2DMM-Metamodell in Abbildung Abbildung 4.6 deutet bereits an, dass sich Referenz-Mappings nicht direkt auf das Ziel einer Nicht-Containment-Referenz des Domänenmodells, sondern vielmehr auf dessen abbildendes Containment-Mapping beziehen. Dies ist jedoch nur dann möglich, wenn das referenzierte Objekt bereits durch ein Containment-Mapping repräsentiert wird. Deshalb existieren auch für Referenz-Mapping-Beschreibungen zwei Möglichkeiten der Abbildungen, um den Wert der abgeleiteten Referenz `referencedObject` zu bestimmen:

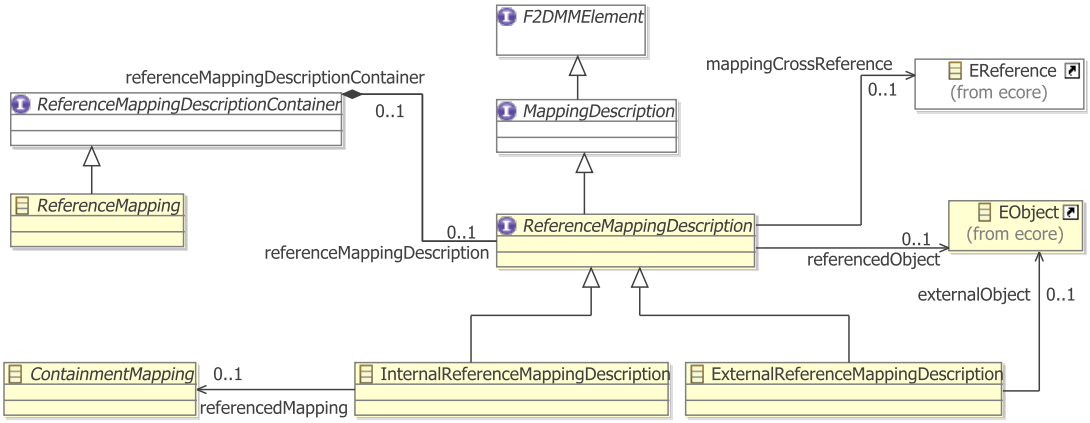

Abbildung 4.18: Ecore-Klassendiagramm für Referenz-Mapping-Beschreibungen.

- `InternalReferenceMappingDescription`: Wird verwendet, wenn das abzubildende Referenzziel von einem primären Containment-Mapping abgebildet wird. Der Wert für `referencedObject` ergibt sich als das von letzterem abgebildete Element.
- `ExternalReferenceMappingDescription`: Befindet sich das Referenzziel in keinem Containment-Mapping, sondern in einer externen EMF-Ressource, bestimmt es direkt als `externalObject` den Wert von `referencedObject`. Ein auf diese Weise abgebildetes Referenzziel verhält sich während der Produkt-Ableitung so, als befände es sich innerhalb eines Containment-Mappings mit dem annotierten Feature-Ausdruck `true`.

4.6.2 Die Anwendersicht: Erzeugung von Alternativen-Mappings

In Abschnitt 3.2.3 wurde diskutiert, inwieweit die Notwendigkeit zur Definition von *Variationspunkten* auf Anwenderseite besteht: Einwertige strukturelle Eigenschaften wie etwa der Name einer UML-Klasse dürfen in einem Multivarianten-Domänenmodell nur einmal vorkommen. Bestandteil der Vorüberlegungen war auch, inwieweit Attribute des Featuremodells die Werte von Attributen abgeleiteter Produkte beeinflussen können (vgl. Abschnitt 3.2.5).

Bei der Beschreibung des **F2DMM**-Metamodells in Abschnitt 4.3 wurden Alternativen-Mappings bereits als benutzerdefinierte Ergänzungen für Kern-Mappings, welche mit dem Domänenmodell synchron gehalten werden, erwähnt. Sie enthalten jeweils eine Mapping-Beschreibung, durch die die Verbindung zum Kern- oder zu einem alternativen Teildomänenmodell hergestellt wird. In diesem Abschnitt wird die Erweiterung des durch Kern-Domänenmodell-Elemente definierten Mapping-Modells um Alternativen-Mappings aus Anwendersicht erläutert.

Der Alternativen-Wizard Das Anlegen von Alternativen-Mappings in einem geöffneten Mapping-Modell erfolgt im Editor durch Auswahl des entsprechenden Kontextmenü-Eintrags New Alternative unter einem existierenden Containment-Mapping. Es erscheint der in Abbildung 4.19 gezeigte Wizard, der den Anwender die strukturelle Eigenschaft – also eine kompatible Referenz bzw. ein kompatibles Attribut aus dem Domänen-Metamodell – wählen lässt, für die ein Alternativen-Mapping erzeugt werden soll. Abhängig davon, ob eine Containment-, eine Nicht-Containment-Referenz oder ein Attribut gewählt wurde, stehen anschließend die in den folgenden Absätzen beschriebenen Auswahlmöglichkeiten zur Verfügung.

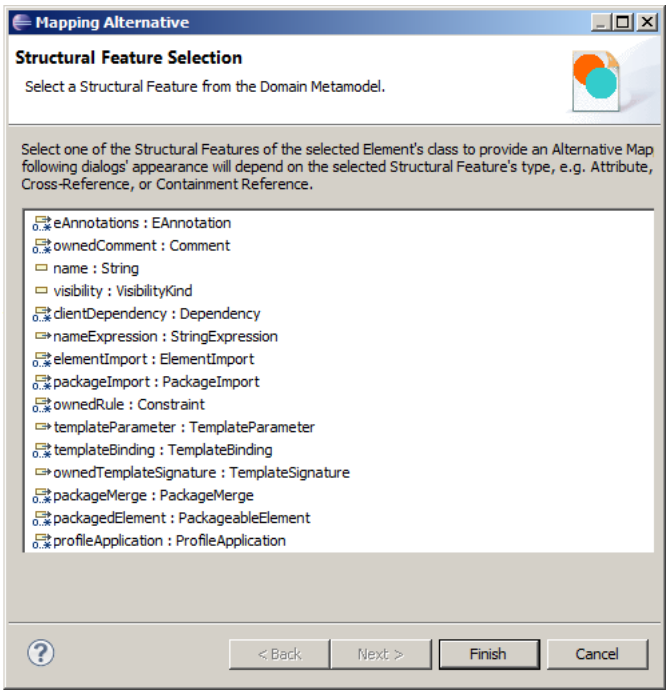

Abbildung 4.19: Die erste Seite des Wizards zur Erstellung von Alternativen Mappings verlangt nach der Auswahl eines Attributs oder einer Referenz aus dem Domänen-Metamodell.

Alternativen für Containment-Mappings In Abschnitt 4.6.1 wurden die beiden Containment-Mapping-Beschreibungen *Wrapping* bzw. *In-Place* vorgestellt. Ersterer referenziert ein tatsächlich vorhandenes Element als Instanz von `EObject`, während die In-Place-Variante die Angabe einer zu instanziierenden Klasse verlangt. Abbildung 4.20 zeigt einen Screenshot der Wizard-Seite, die den Anwender eine der folgenden drei Möglichkeiten zur Erzeugung eines Alternativen-Mappings inklusive einer entsprechenden Mapping-Beschreibung wählen lässt:

- Primäres *Wrapping*-Containment-Mapping durch Angabe eines Domänenmodell-Elements aus einer externen Ressource: Dem Anwender wird im Anschluss ein Auswahldialog angezeigt, in dem er ein Objekt, dessen Typ mit der in der ersten Wizard-Seite gewählten Containment-Referenz übereinstimmen muss, aus einer externen Ressource wählen soll. Mappings für in dem Objekt verschachtelte Elemente werden entsprechend seines aufspannenden Containment-Baums rekursiv erzeugt.

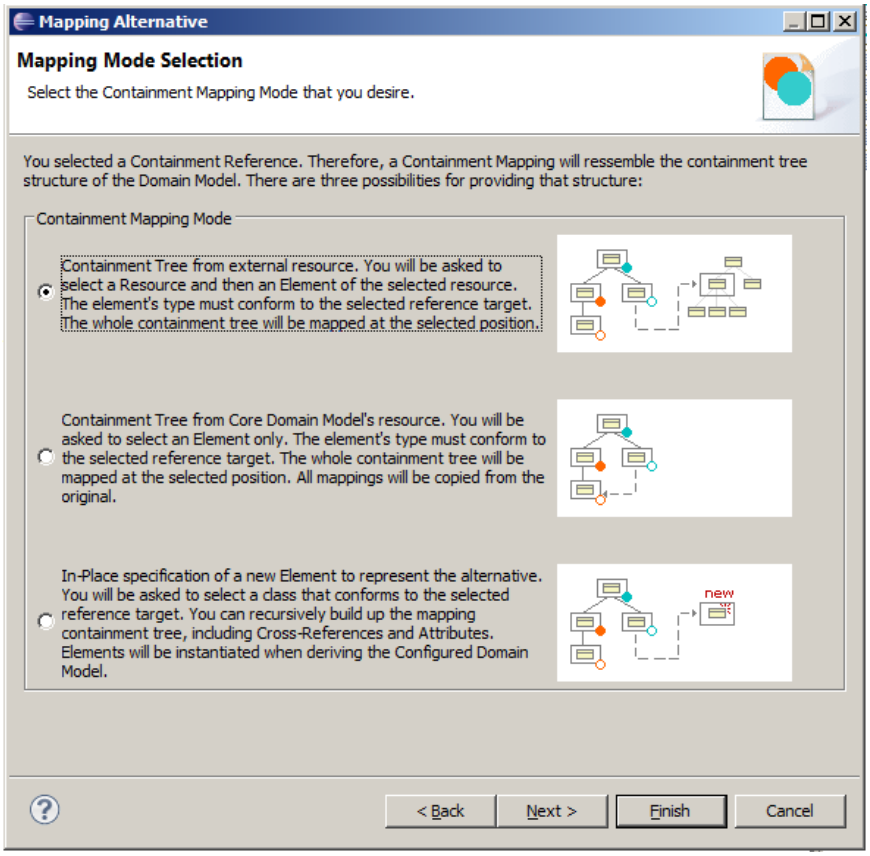

Abbildung 4.20: Für die Erstellung von Alternativen-Containment-Mappings stehen drei Möglichkeiten zur Auswahl.

- Sekundäres *Wrapping*-Containment-Mapping als Kopie eines vorhandenen, primären Kern-Mappings. Das gewählte Mapping und sein beinhalteter Teilbaum werden kopiert und als sekundär (`primary = false`) markiert. Die Kopie wird nach ihrer Erzeugung nicht durch Änderungen im Original-Domänenmodell beeinflusst.
- *In-Place*-Definition eines Containment-Mappings durch Angabe einer nicht-abstrakten, dem Typ der Containment-Referenz entsprechenden Ecore-Klasse aus dem Domänen-Metamodell. Es wird eine `InPlaceContainmentMappingDescription` mit Referenz auf die gewählte Klasse erzeugt. Enthaltene Mappings müssen manuell vom Anwender angelegt werden.

Alternativen für Attribut-Mappings Abbildung 4.21 zeigt einen Bildschirm-Ausschnitt der zweiten Wizard-Seite unter der Voraussetzung, dass die zuvor gewählte strukturelle Eigenschaft ein Attribut ist. Sie stellt dem Anwender zwei verschiedene Möglichkeiten zur Verfügung, um den Wert des gewählten Attributs zu spezifizieren:

- *Direkte* Angabe der String-Repräsentation des Attributwerts über ein Eingabefeld. Es wird eine entsprechende `SimpleAttributeMappingDescription` erzeugt. Wurde ein Aufzählungstyp ausgewählt, wird auf der folgenden Seite anstatt einer Eingabemaske ein Auswahlfeld mit allen verfügbaren Literalen angezeigt.

- *Pattern*-basierte Erzeugung eines Mappings auf Basis einer `PatternAttributeMappingDescription`, die einen in einem Eingabefeld anzugebenden Text nach Attribut-Ausdrücken (vgl. Abschnitt 4.4.3) durchsucht. Diese werden unter Berücksichtigung der aktuell geladenen Featurekonfiguration ausgewertet und durch ihr Ergebnis ersetzt. Der Wert des Attributs ändert sich entsprechend, wenn die geladene Featurekonfiguration manipuliert oder eine andere geladen wird.

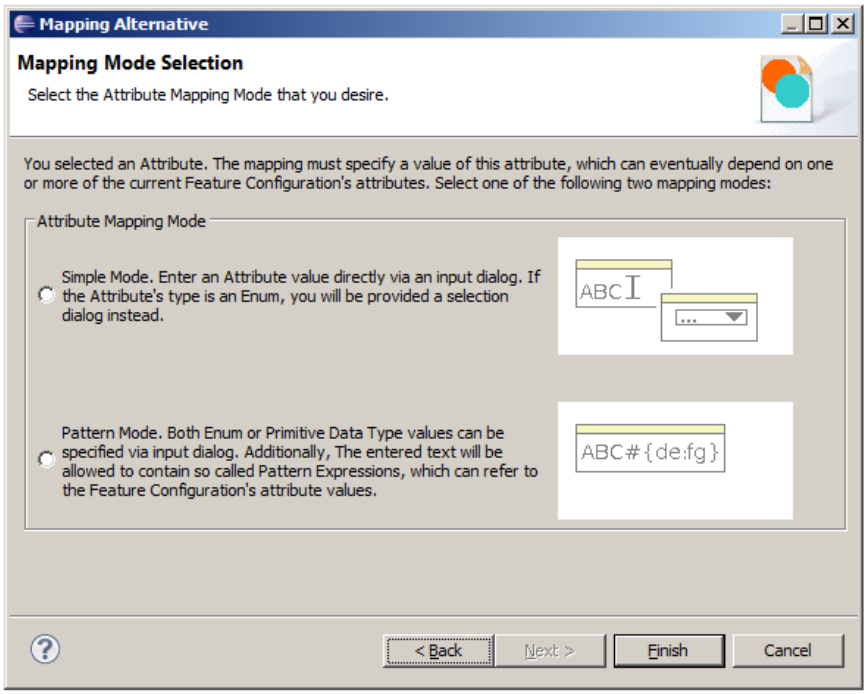

Abbildung 4.21: Die auf Alternativen-Attribut-Mappings bezogene Wizard-Seite erlaubt die Auswahl eines direkten oder eines Pattern-basierten Modus zur Angabe von Attributwerten.

Beide Modi verlangen die Angabe der String-Repräsentation eines Attributwerts für einen primitiven Datentypen. Dieser wird durch den Ecore-*Adapter-Factory*-Mechanismus (vgl. Abschnitt 2.3.1) interpretiert und zur Laufzeit in den entsprechenden Datentyp umgewandelt. Hierbei auftretende Fehler, wie etwa die Angabe eines vom Attribut-Datentyp nicht interpretierbaren Strings werden von der Modellvalidierung (s. Abschnitt 4.9.3) behandelt.

Alternativen für Referenz-Mappings Auch bei der Angabe von alternativen Referenzzielen für Nicht-Containment-Referenzen wird vom Anwender die Wahl eines von zwei Mapping-Modi verlangt (vgl. Abbildung 4.22):

- Angabe eines *externen* Referenzziels zur Erzeugung einer neuen Mapping-Beschreibung auf Basis der Klasse `ExternalReferenceMappingDescription`. Auf der folgenden Wizard-Seite wird in diesem Fall die Angabe einer externen Ressource und anschließend eines kompatiblen Referenzziels aus dieser Ressource verlangt. Das erzeugte `AlternativeReferenceMapping` referenziert das angegebene Objekt in seiner Mapping-Beschreibung.

- Angabe eines kompatiblen Referenzziels aus der Ressource des Kern-Domänenmodells. Hier wird, im Gegensatz zum vorhergehenden Modus, die Angabe eines Mappings anstatt eines Domänenmodell-Elements verlangt. Stimmt der Typ des gemappten Elements mit dem Typ der auf der ersten Seite gewählten Referenz überein, wird ein alternatives Referenz-Mapping mit einer `InternalReferenceMappingDescription` erzeugt, die sich über `referencedMapping` auf das gewählte Containment-Mapping bezieht.

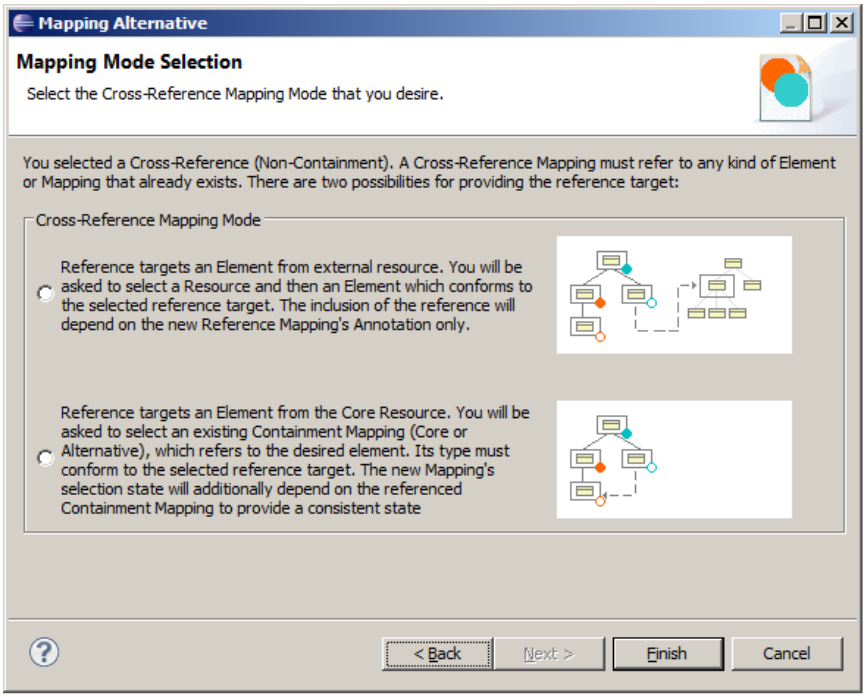

Abbildung 4.22: Die Wizard-Seite zur Auswahl des Abbildungsmodus für Alternativen-Referenz-Mappings. Sowohl der Rückbezug auf ein vorhandenes Containment-Mapping als auch die Angabe eines Elements aus einer externen Ressource sind möglich.

4.7 Konflikterkennung, Propagation und Invalidierung

Ein wesentlicher in der Einleitung formulierter Beitrag dieser Arbeit zur modellgetriebenen Softwareentwicklung von Produktlinien sind Mechanismen zur automatischen Wiederherstellung der *Konsistenz* (vgl. Abschnitt 1.3.1). In Abschnitt 3.3 wurden Überlegungen angestellt, unter welchen Voraussetzungen ein Mapping-Modell konsistent ist: Die wesentliche Eigenschaft war die Möglichkeit der Erzeugung eines *wohlgeformten Produkts* aus jeder erlaubten Featurekonfiguration. Unter den in Abschnitt 3.3.2 definierten Annahmen können Beziehungen zwischen Mappings *vorberechnet* werden, um mögliche *Konflikte* bereits vor der Produktableitung zu erkennen (vgl. Unterabschnitt 4.7.3 dieses Abschnitts). Das Wiederherstellen der Konsistenz im Falle eines Konflikts ist hingegen die Aufgabe sog. *Propagationsstrategien*, welche in Unterabschnitt 4.7.4 behandelt werden. **F2DMM** unterscheidet zwei Arten von Konflikten, die zu Inkonsistenzen innerhalb eines Mapping-Modells führen:

Abhängigkeitskonflikt Tritt gemäß Definition in Abschnitt 3.3.3 genau dann auf, wenn ein im Produkt enthaltenes (positiv annotiertes) Domänenmodell-Element A von einem nicht enthaltenem (negativ annotiertem, B) abhängt.

Ausschlusskonflikt Zugehörigkeit zweier oder mehrerer sich einander ausschließender Domänenmodell-Elemente E und F zu einem Produkt. Dies kann beispielsweise ein Paar von Kern- und Alternativen-Mappings betreffen, welche dieselbe einwertige strukturelle Eigenschaft – etwa den Namen einer Klasse – abbilden.

4.7.1 Phasen bei der Ermittlung von Selektionszuständen

Bevor im Einzelnen auf Konzepte der Konflikterkennung und Propagation eingegangen wird, sollen diese in einen gemeinsamen Kontext gestellt werden. Die Ermittlung des Selektionszustands eines Mappings erfolgt in **F2DMM** in mehreren Phasen: Im Rahmen des Vorberechnungsschritts (Phase 0 in Abbildung 4.23) wird jedes Paar von im Mapping-Modell abgebildeten **DM**-Elementen miteinander verglichen und auf Abhängigkeit bezüglich Containment-Hierarchie und definierten **SDIRL**-Bedingungen (vgl. Abschnitt 4.5) geprüft. Trifft die Abhängigkeit zu, wird sie in einer geeigneten Datenstruktur vorgemerkt. Auf ähnliche Weise werden Beziehungen zwischen sich gegenseitig ausschließenden Mappings vorgemerkt. Phase 0 wird etwa beim Laden eines Mapping-Modells durchgeführt.

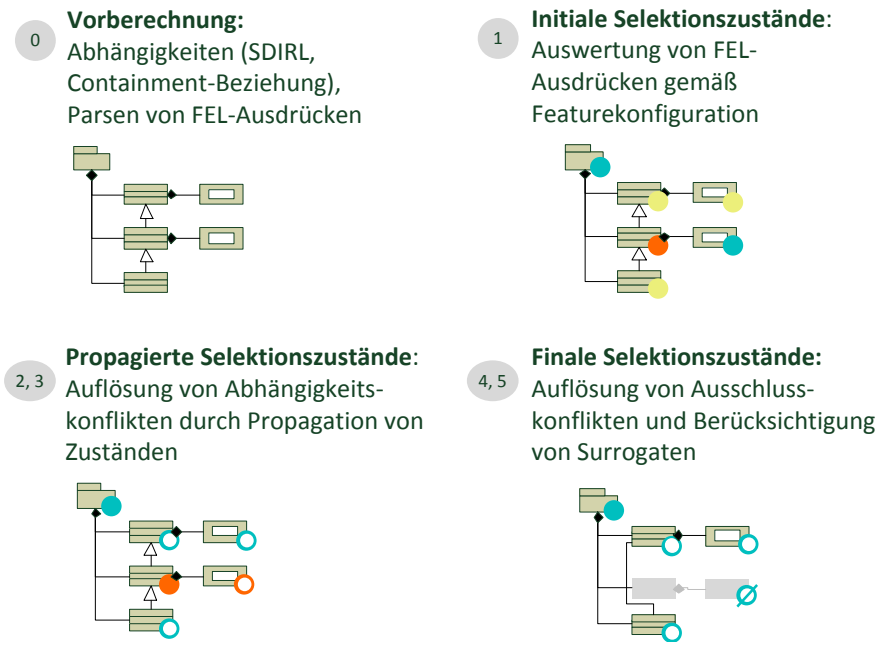

Abbildung 4.23: Überblick über die Phasen zur Bestimmung von Selektionszuständen für Mappings am Beispiel konkreter **UML2**-Syntax.

Selektionszustände wurden bereits in Abschnitt 4.3.4 behandelt: Sobald eine Featurekonfiguration geladen wurde, kann für jedes Mapping ein *initialer Selektionszustand* bestimmt werden (Phase 1). Dieser bezieht sich ausschließlich auf den assoziierten Feature-Ausdruck und kann zu Abhängigkeitskonflikten innerhalb des Mapping-Modells führen. Die Auflösung solcher Konflikte ist die Aufgabe von *Propagationsstrategien*. Sie werden in den Phasen 2 und 3 verwendet, um ein Mapping mit *propagierten* Selektionszuständen zu gewinnen.

Nach Abhängigkeitskonflikten werden Ausschlusskonflikte behandelt. Hierfür ist kein den Propagationsstrategien entsprechendes Konzept vorgesehen; stattdessen werden in Phase 4 *Präzedenzregeln* involviert, um Variationspunkte auf Produktebene aufzulösen. Bereits in Phase 0 werden mögliche *Surrogate* (vgl. Abschnitt 4.5.5) vorberechnet. Bekommt ein Referenz-Mapping in Phase 5 den Zustand *suppressed* zugewiesen, können Surrogate dessen abgebildetes Referenzziel im Kontext der definierten Abhängigkeit ersetzen und sein Zustand wird *surrogated*. Die Selektionszustände sind nun *final*.

4.7.2 Beziehungen zwischen Mappings: Die abstrakte Klasse Annotatable

Die abstrakte Modellklasse `Annotatable` wurde bereits als gemeinsame Basisklasse von Mappings eingeführt. Sie stellt unter anderem über Feature-Ausdrücke (s. Abschnitt 4.4) die Verknüpfung zu Featuremodell und -konfiguration her und speichert den aktuellen Selektionszustand. Diese strukturellen Eigenschaften werden jedoch nicht persistiert[29]; lediglich die konkrete textuelle Repräsentation des annotierten Feature-Ausdrucks wird als Wert des Attributs `featureExprStr` in der Ressource festgehalten (vgl. Abschnitt 4.4.4).

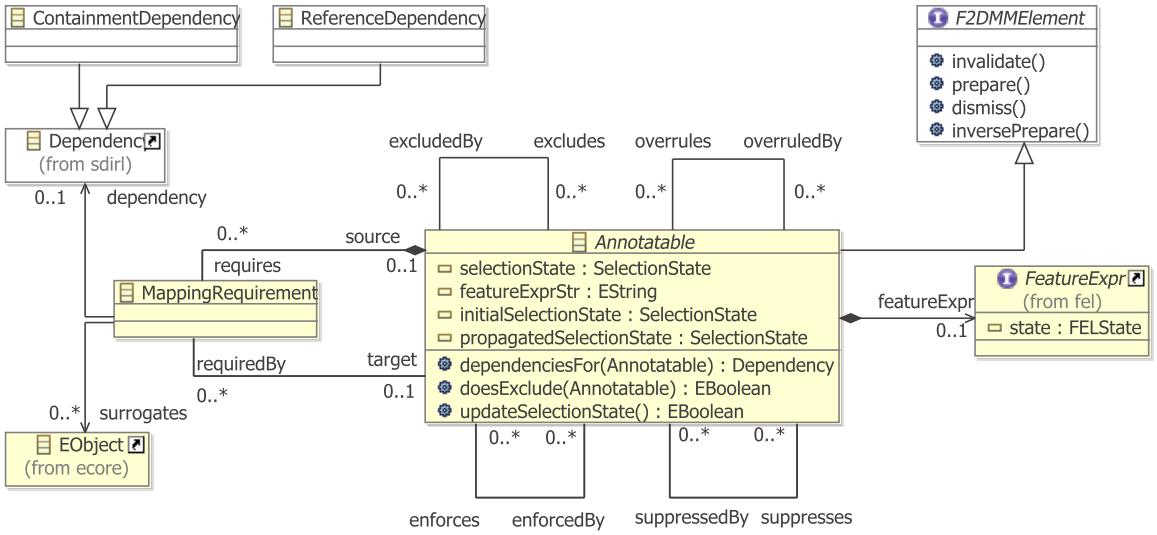

Abbildung 4.24: Die abstrakte Klasse `Annotatable` beschreibt mögliche Beziehungen zwischen Mappings im F2DMM-Metamodell.

Ebenfalls nicht-persistent sind die im Ecore-Klassendiagramm in Abbildung 4.24 dargestellten Beziehungen zwischen Instanzen von `Annotatable`. Das Referenzpaar `excludes` – `excludedBy` bezieht sich auf Mappings, die sich konkurrierend ausschließen. Für die Repräsentation von ebenfalls in Schritt 0 vorberechneter struktureller Abhängigkeiten und Surrogaten ist die Assoziationsklasse `MappingRequirement` vorgesehen. Die Referenzen `enforces` und ihr Gegenteil `enforcedBy` sowie `suppresses` bzw. `suppressedBy` modellieren die durch Anwendung von Propagationsstrategien bedingte künstliche Positivierung bzw. Negativierung eines Mappings durch ein oder mehrere weitere. Ähnlich erlaubt das Referenzpaar `overrules`/`overruledBy` das Nachvollziehen der Auflösung von Ausschlusskonflikten.

[29] Die Attribute `transient` und `derived` sind im Metamodell auf `true` gesetzt.

4.7.3 Phase 0: Vorberechnung von Abhängigkeits- und Ausschlussbeziehungen

`Annotatable` erbt die von der implementierten Schnittstelle `F2DMMElement` deklarierte Methode `prepare()`, welche beim Laden des Mapping-Modells sowie im Falle einer Änderung innerhalb der Ressourcen referenzierter Feature- oder Domänenmodelle im Rahmen des *großen Invalidierungszyklus* (s. Abschnitt 4.7.6) aufgerufen wird. Sie implementiert mit Schritt 0 aus der Darstellung in Abbildung 4.23 die Vorberechnung von Abhängigkeiten als Instanzen der Assoziationsklasse `MappingRequirement`. Hierzu erfolgt für jedes Paar (A, B) von im Mapping-Modell existierenden `Annotatable`-Instanzen der Methodenaufruf `A.dependenciesFor(B)`. Der Rückgabewert beinhaltet eine Menge von `Dependency`-Instanzen, welche die jeweils zutreffende (Containment- oder in **SDIRL** definierte metamodellspezifische) Abhängigkeit referenzieren. Die ermittelten Abhängigkeiten werden durch jeweils ein `MappingRequirement` repräsentiert, welches strukturell dem `source`-Mapping (in diesem Fall A) untergeordnet und vom `target`-Mapping (B) über die inverse `requiredBy`-Referenz verfügbar gemacht wird.

Containment-Abhängigkeiten Jede Instanz von `EObject` ist existenzabhängig von seinem `eContainer` (vgl. Abschnitt 3.3.3). Dies muss in der Vorberechnung von Abhängigkeiten zum Konsistenzerhalt berücksichtigt werden. Aufgrund der strukturellen Nachbildung des Mapping-Modells lässt sich ein Mapping ebenfalls als existenzabhängig von seinem `eContainer` definieren. Das **F2DMM**-Metamodell sieht zur Darstellung der Containment-Abhängigkeit eine eigene, nach dem *Singleton*-Entwurfsmuster [21, Abschnitt 3.5] implementierte Klasse `ContainmentDependency` vor. Quelltext 4.7 dokumentiert das Vormerken einer Containment-Abhängigkeit für Mappings.

```
1  public EList<Dependency> dependenciesFor(Annotatable other) {
2      EList<Dependency> dependencies = new BasicEList<Dependency>();
3      if (other.equals(eContainer())) {
4          dependencies.add(ContainmentDependency.INSTANCE);
5      }
6      return dependencies;
7  }
```

Quelltext 4.7: Die Implementierung der Methode `dependenciesFor()` in der Basisklasse `AnnotatableImpl` berücksichtigt Containment-Abhängigkeiten.

Referenz-Abhängigkeiten Eine Referenz zwischen zwei Instanzen von `EObject` ist nur dann gültig, wenn sowohl Quell- als auch Ziel-Objekt vorhanden sind. Für Mappings hat dies zur Folge, dass ein Referenz-Mapping strukturell von seinem übergeordneten Containment-Mapping abhängig ist, was bereits durch die Implementierung in der Basisklasse `Annotatable` sichergestellt wird. Zusätzlich muss jedoch die Abhängigkeit vom Referenzziel beachtet werden, welche von der Klasse `ReferenceDependency` (ebenfalls nach dem *Singleton*-Entwurfsmuster, s. Abbildung 4.24) modelliert wird.

```
1  @Override
2  public EList<Dependency> dependenciesFor(Annotatable other) {
3      EList<Dependency> dependencies = super.dependenciesFor(other);
4      if (getReferenceMappingDescription() instanceof
           InternalReferenceMappingDescription) {
5          InternalReferenceMappingDescription internalRmd =
               (InternalReferenceMappingDescription)
               getReferenceMappingDescription();
```

```
6             if (other.equals(internalRmd.getReferencedMapping())) {
7                 dependencies.add(ReferenceDependency.INSTANCE);
8             }
9         }
10        return dependencies;
11    }
```

Quelltext 4.8: Überschreiben der Methode `dependenciesFor()` in der Unterklasse `ReferenceMappingImpl` von `Annotatable`, um die Abhängigkeit einer Referenz von ihrem Ziel zu berücksichtigen.

Die Repräsentation des Ziels einer Referenz hängt vom Typ der dessen Mapping zugeordneten Mapping-Beschreibung ab: Handelt es sich um eine `ExternalReferenceMappingDescription`, verhält diese sich so, als wäre das Ziel ein Containment-Mapping mit positiver Annotation (vgl. Abschnitt 4.6.1). Die Möglichkeit eines Abhängigkeitskonflikts entfällt hierbei. Eine `InternalReferenceMappingDescription` verweist hingegen auf ein weiteres Containment-Mapping, welches durchaus eine negative Annotation beinhalten kann, wodurch wiederum Abhängigkeitskonflikte entstehen können. In Quelltext 4.8 wird folglich eine `ReferenceDependency` genau dann vorgemerkt, wenn per `InternalReferenceMappingDescription` das im Formalparameter `other` angegebene Mapping referenziert wird.

SDIRL-Abhängigkeiten Mapping-Elemente können zusätzlich durch die Auswertung modellspezifischer Konsistenzbedingungen als abhängig von weiteren Mappings gelten. Diese Constraints werden in einem **SDIRL**-Modell, welches vom Mapping-Modell referenziert wird, formuliert (vgl. Abschnitt 4.5). Ein SDIRL-Modell enthält eine Menge von Abhängigkeitsdefinitionen als Instanzen der Klasse `Dependency`. Diese wiederum enthalten jeweils einen **OCL**-Ausdruck, welcher die Konsistenzbedingung für zwei durch ihren Typen und einen frei wählbaren Bezeichner definierten Variablen formuliert.

SDIRL-Ausdrücke werden in der vorliegenden Implementierung nur für Containment-Mappings ausgewertet: Für Attribut-Mappings wurde keine Möglichkeit zur Formulierung von Konsistenzbedingungen vorgesehen; Referenz-Mappings werden durch das jeweils von ihnen referenzierte Containment-Mapping durch die transitive Abhängigkeit `ReferenceDependency` abgedeckt. Als abhängige Elemente (`target`) kommen hingegen Containment- oder Referenz-Mappings in Frage.

```
1  @Override
2  public EList<Dependency> dependenciesFor(Annotatable other) {
3      EList<Dependency> dependencies = super.dependenciesFor(other);
4
5      if (getMappingModel().getSdirlModel() != null) {
6          StructuralDependencyModel sdirl = getMappingModel().getSdirlModel();
7          EObject sourceElement = getContainmentMappingDescription().getMappedObject();
8          EObject targetElement = ... // extrahiere von other abgebildetes Element
9
10         for (Dependency dep : sdirl.getDependencies()) {
11             if (dep.getElementType().isInstance(sourceElement) &&
12                     dep.getRequiredType().isInstance(targetElement)) {
13                 String whenExpr = EmbeddedOCLExprUnparser.unparse(dep.getWhen());
14                 boolean whenConditionHolds =
                       getMappingModel().getOCLEvaluator().isWhenConditionTrue(dep.getName(),
```

```
                    dep.getElementName(), sourceElement, dep.getRequiredName(),
                    targetElement, whenExpr);
15              if (whenConditionHolds) {
16                  dependencies.add(dep);
17              }
18          }
19      }
20  }
21  return dependencies;
22 }
```

Quelltext 4.9: Die Implementierung der Methode `dependenciesFor()` in der Unterklasse `ContainmentMappingImpl` berücksichtigt SDIRL-formulierte Abhängigkeitsbedingungen (vereinfachte Darstellung).

In Quelltext 4.9 ist folgende Vorgehensweise zur Ermittlung SDIRL-definierter Abhängigkeiten implementiert:

1. Falls kein SDIRL-Modell vorhanden ist, breche ab (Z. 5).
2. Extrahiere Quell- und Ziel-**DM**-Elemente aus den Parametern von `dependenciesFor()` (Z. 7 und 8).
3. Iteriere über alle vom SDIRL-Modell definierten Abhängigkeiten (Z. 10).
4. Falls die Typen von Quell- und Zielelement nicht mit den in der Abhängigkeit definierten `element`- und `requires`-Variablen übereinstimmen, breche ab (Z. 11 und 12).
5. Binde die Elemente an ihre entsprechenden Variablen und Werte die `when`-Bedingung aus (Z. 14).
6. Wenn die Bedingung zutrifft, merke die Abhängigkeit vor (Z. 15 und 16).

Die Auswertung der OCL-Bedingung erfolgt über eine Instanz von `OCLEvaluator`, welche in Abschnitt 4.5.4 zur Kapselung der Auswertung von OCL-Ausdrücken eingeführt wurde. Die Klasse `EmbeddedOCLExprUnparser` implementiert die Überführung eines im SDIRL-Modell in deserialisierter Form vorliegenden OCL-Ausdrucks in dessen textuelle Repräsentation. Die Notwendigkeit für ein zweimaliges Parsen des OCL-Ausdrucks wurde ebenfalls im Abschnitt über **SDIRL** (4.5.2) begründet.

Vormerken von Surrogaten Noch nicht diskutiert wurde die von `MappingRequirement` ausgehende Referenz `surrogates` (s. Abbildung 4.24). Sie verweist auf Surrogat-Objekte als diejenigen Instanzen von `EObject` aus dem Domänenmodell, die das `target`-Mapping bei Verletzung der als `dependency` referenzierten Konsistenzbedingung ersetzen können.

```
1 public EList<EObject> getSurrogates() {
2     if (surrogates == null) {
3         getSurrogatesGen(); // Aufruf der Ecore-generierten Methode
4         EObject sourceElement = ... // extrahiere source-Element
5         EObject targetElement = ... // extrahiere target-Element
6         for (ContextCS surrogate : getDependency().getSurrogates()) {
7             String surrogateExpr = EmbeddedOCLExprUnparser.unparse(surrogate);
```

```
8            EList<EObject> depSurrogates =
                 getMappingModel().getOCLEvaluator().getSurrogates(
9                    getDependency().getName(),
10                   getDependency().getElementName(), sourceElement,
11                   getDependency().getRequiredName(), targetElement,
12                   surrogateExpr);
13           surrogates.addAll(depSurrogates);
14       }
15   }
16   return getSurrogatesGen();
17 }
```

Quelltext 4.10: Berechnung von Surrogat-Objekten durch Auswertung entsprechender OCL-Ausdrücke in der Methode `getSurrogates()` der Klasse `MappingRequirementImpl`.

Die Auswertung von Surrogat-Ausdrücken findet bei erstmaligem Aufruf der abgeleiteten, transienten Referenz `surrogates` statt (falls die `surrogates`-Liste noch nicht initialisiert ist, vgl. Quelltext 4.10, Z. 2). Auch hier wird von der Abstraktionsklasse `OCLEvaluator` Gebrauch gemacht, in der die tatsächliche Auswertung aller Surrogat-Ausdrücke erfolgt. Die Rückgabewerte der OCL-Anfragen werden entsprechend in `surrogates` aufgenommen.

Sich gegenseitig ausschließende Mappings Zusätzlich zu der über die Assoziationsklasse `MappingRequirement` abgebildete Beziehung `requires` existiert die Referenz `excludes` zwischen Instanzen von `Annotatable`, welche in Phase 0 populiert wird. Vom Anwender erzeugte Alternativen-Mappings konkurrieren möglicherweise mit vom Synchronisierungsmodul automatisch erzeugten Kern-Mappings oder mit weiteren Alternativen-Mappings. Dies ist der Fall, sobald unterhalb eines Containment-Mappings mehrere Alternativen-Mappings existieren, die sich auf dieselbe einwertige strukturelle Eigenschaft (`isMany() == false`) beziehen. Zur Lösung solcher durch *konkurrierende* Mappings entstehender *Ausschlusskonflikte* (s. Phase 4) wird a priori eine *Rangfolge* definiert, die festlegt, welches positiv annotierte Mapping den Wert eines einwertigen Attributs oder einer einwertigen Referenz bestimmen wird:

1. *Kern-Mappings* entstehen durch Abbildung eines wohlgeformten Multivarianten-Domänenmodells. Folglich kann es nur je ein Kern-Mapping unterhalb eines Containment-Mappings geben, welches sich auf eine einwertige strukturelle Eigenschaft bezieht. Ein solches Kern-Mapping hat prinzipiell den Vorrang vor konkurrierenden Alternativen-Mappings.

2. *Alternativen-Mappings* können auch für einwertige strukturelle Eigenschaften in beliebiger Anzahl vom Anwender erzeugt werden. Eventuell konkurrierende Alternativen-Mappings bekommen in der Rangfolge den Stellenwert ihrer Position im Mapping-Modell zugewiesen: Ausschlusskonflikte werden anhand der Einfügereihenfolge aufgelöst.

Innerhalb der `prepare()`-Methode eines `Annotatable` wird in Phase 0 für jedes Paar (E, F) die Methode `E.doesExclude(F)` aufgerufen, welche nach den oben definierten Prioritätsregeln prüft, ob die Annotation von E mit einem positiven Feature-Ausdruck das Vorhandensein von F ausschließt. Ist dies der Fall, wird F in die *excludes*-Menge von E aufgenommen und die inverse Referenz `excludedBy` entsprechend gesetzt.

4.7.4 Die Propagationsstrategien „Vorwärts“ und „Rückwärts“

In den obigen Ausführungen wurden bereits die in Abschnitt 3.3.3 vorgestellten *Propagationsstrategien* (**PS**) aufgegriffen. Werden sie zur Auflösung von Abhängigkeitskonflikten verwendet, spricht man von *primären* **PS**. Sie finden zusätzlich Anwendung in der Berechnung von Selektionszuständen nicht annotierter Mappings als *sekundäre* **PS**. In den Phasen 2 und 3 (s. Abschnitt 4.7.1) der **SZ**-Berechnung werden primäre und sekundäre **PS** eingesetzt, um die Konsistenz eines Mapping-Modells wiederherzustellen.

Auflösung von Abhängigkeitskonflikten Ein Abhängigkeitskonflikt tritt auf, wenn ein Mapping A mit positivem Selektionszustand von einem Mapping mit negativem Selektionszustand B abhängt ($A \Leftarrow B$ in der in Abschnitt 3.3 vereinbarten Notation). Um diese Konsistenzverletzung aufzulösen, wurden die folgenden zwei *primären* Propagationsstrategien (**PS**) für Phase 2 ausgearbeitet (vgl. auch Abschnitt 3.3.3 und Abbildung 4.25):

- **Vorwärtspropagation**: Die Konsistenz wird wiederhergestellt, indem das positiv annotierte Mapping A, welches von B abhängt, mit einem künstlichen, negativen Selektionszustand (*suppressed*) versehen wird. Die Propagation des Selektionszustands erfolgt also in Richtung der Abhängigkeit und löst Abhängigkeitskonflikte ausschließlich durch *Negativierung* von Selektionszuständen.
- **Rückwärtspropagation**: Der positive Selektionszustand des Mappings A wird entgegen der Richtung der Abhängigkeit zum Mapping B propagiert. B wird also mit einem künstlichen, positiven Selektionszustand (*enforced*) versehen. Die Anwendung dieser Strategie hat die ausschließliche *Positivierung* betroffener Selektionszustände zur Folge.

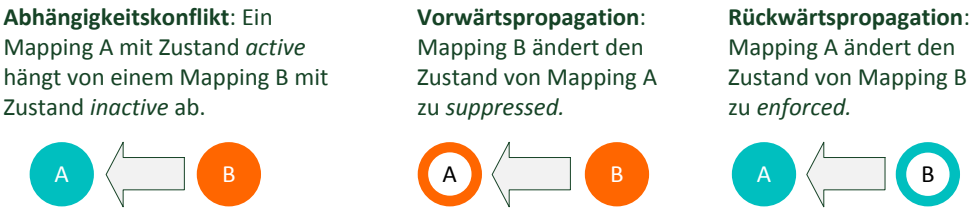

Abbildung 4.25: Primäre Propagationstrategien werden angewendet, um Abhängigkeitskonflikte innerhalb des Mapping-Modells aufzulösen.

Selektionszustand nicht annotierter Mappings *Sekundäre* **PS** kommen in Phase 3 zum Einsatz: Zur Wahrung der Konsistenz sowie zur Vermeidung redundanter Annotationen sollen die Selektionszustände *nicht annotierter* Mappings bestimmt werden, ohne die formulierten Konsistenzbedingungen zu verletzen. Abbildung 4.26 zeigt, dass wiederum die **VP** und **RP**, diesmal als *sekundäre* Propagationsstrategien, angewendet werden können:

- **Vorwärtspropagation**: Der Selektionszustand eines nicht annotierten Elements B kann durch den Selektionszustand eines Mappings C bestimmt werden, falls $B \Leftarrow C$ gilt. B bekommt den Selektionszustand *suppressed* oder *enforced*, je nach dem ob C sich in einem negativen oder positiven Selektionszustand befindet, zugewiesen. Es kann also sowohl eine Positivierung, als auch eine Negativierung erfolgen; ist beides möglich, wird die Negativierung bevorzugt.

- **Rückwärtspropagation**: Wiederum sei B ein nicht annotiertes Mapping. Falls es ein Mapping A gibt, welches von B abhängt ($A \Leftarrow B$), so wird A zur Bestimmung des Selektionszustands von B herangezogen, was wiederum sowohl zu einer Positivierung als auch einer Negativierung von Selektionszuständen führen kann. B wird *suppressed*, falls A negativ, bzw. *enforced*, falls A positiv annotiert ist. Die Positivierung von Zuständen wird gegenüber der Negativierung bevorzugt.

Die sekundäre **PS** kann unabhängig von der primären **PS** festgelegt werden. Desweiteren ist bei nicht annotierten Elementen eine Kombination durch Priorisierung der beiden Strategien denkbar: Beispielsweise kann zunächst die Vorwärtspropagation angewendet werden; ist kein Element verfügbar, von dem das nicht annotierte Mapping abhängt, kann schließlich die Rückwärtspropagation herangezogen werden.

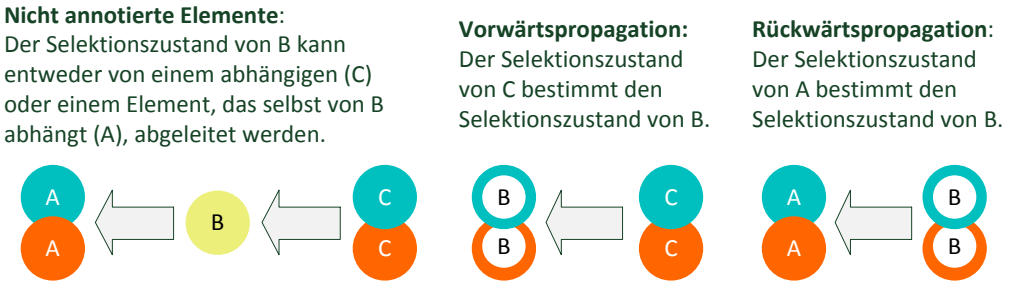

Abbildung 4.26: Anwendung sekundärer Propagationsstrategien, um die Selektionszustände nicht annotierter Mappings unter Wahrung der Konsistenz abzuleiten.

Implementierung im F2DMM-Metamodell Die Unabhängigkeit von primärer und sekundärer Propagationsstrategie begründet die Trennung dieser beiden Konzepte im **F2DMM**-Metamodell. Für jede der beiden Strategien ist ein Ecore-Aufzählungstyp definiert: Während `DependencyConflictStrategy` die primären Strategien `forward` und `reverse`, sowie die Option `ignore` enthält, beinhaltet `MissingAnnotationStrategy` die kombinierten Strategien `forwardThenReverse` bzw. `reverseThenForward` für sekundäre **PS** (s. obige Ausführungen). Jedem Mapping-Modell ist eine Instanz von `PropagationStrategy` zugeordnet, die jeweils einen der beiden Enumerationstypen als Attribut enthält (s. Abbildung 4.27).

Zusätzliche Attribute und Operationen `PropagationStrategy` enthält zudem zwei boolesche Attribute, die wie folgt definiert sind, um das Verhalten der **PS** zu beeinflussen:

- `transitiveConflictResolution`: Ist der Wahrheitswert `true` angegeben, so verhalten sich bei der Anwendung von Propagationsstrategien die Selektionszustände *suppressed* und *enforced* wie *inactive* bzw. *active*. Insbesondere gilt für die Voraussetzungen $A \Leftarrow B$ und $B \Leftarrow C$ der Schluss $A \Leftarrow C$.
- `includeIncomplete`: Gibt an, ob Mappings, die sich im *finalen* Selektionszustand *incomplete* befinden, in Produkten enthalten sein sollen oder nicht.

Zusätzlich enthält die Klasse `PropagationStrategy` drei Methoden, die zur Identifikation und Elimination von Abhängigkeits- bzw. Ausschlusskonflikten verwendet werden:

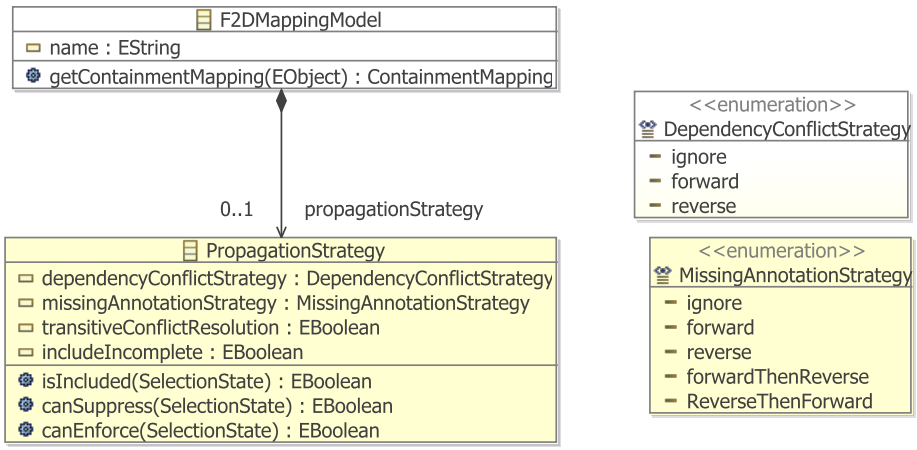

Abbildung 4.27: Ecore-Klassendiagramm, das den Zusammenhang zwischen den Propagationsstrategien auf Metamodell-Ebene darstellt.

- `isIncluded(SelectionState)` entscheidet, ob ein Mapping mit übergebenem Selektionszustand in einem abgeleiteten Produkt enthalten wäre. Der Rückgabewert ist `true` für die Selektionszustände *active*, *enforced* und *surrogated* und negativ für *inactive*, *suppressed* und *corrupted*. Für den **SZ** *incomplete* entspricht der Rückgabewert der Option `includeIncomplete`.
- `canSuppress(SelectionState)` bestimmt, ob ein Mapping mit übergebenem Selektionszustand bei einem anderen Mapping den Selektionszustand *suppressed* bewirken kann. Dies ist der Fall, wenn der gegebene **SZ** entweder *inactive* ist, bzw. *suppressed* oder *overruled*, falls die Option `transitiveConflictResolution` gesetzt ist.
- `canEnforce(SelectionState)` prüft hingegen, ob ein gegebener Selektionszustand die Negativierung eines anderen bewirken kann. Der Rückgabewert ist `true`, wenn der Selektionszustand *active*, bei gesetzter Option `transitiveConflictResolution` auch *enforced* oder *surrogated*, ist.

4.7.5 Phasen 1 bis 5: Identifikation und Auflösung von Konflikten

In Phase 0 werden Abhängigkeits- und Ausschluss*beziehungen* zwischen Mappings vorberechnet. Diese dienen der Identifikation von Abhängigkeits- bzw. Ausschluss*konflikten*, um diese anschließend durch Anwendung einer Propagationsstrategie auflösen zu können. Im *kleinen Invalidierungszyklus* (s. Abschnitt 4.7.6) wird innerhalb für jedes Mapping ein konsistenter Selektionszustand in mehreren Phasen berechnet. Abbildung 4.28 zeigt mögliche Selektionszustände eines `Annotatable` und weist den möglichen Übergängen zwischen ihnen Phasen zu, die im Folgenden genauer beschrieben werden.

Phase 1: Bestimmung des initialen Selektionszustands Der initiale Selektionszustand (`initialSelectionState`) eines durch `Annotatable` repräsentierten Mappings (vgl. Abbildung 4.24) ergibt sich unmittelbar durch Auswertung seines assoziierten Feature-Ausdrucks.

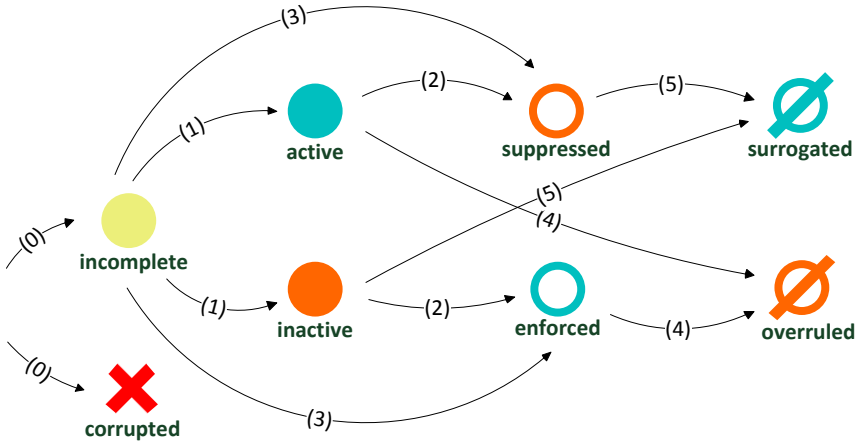

Abbildung 4.28: Mögliche Übergänge zwischen Selektionszuständen eines Mappings, den jeweiligen Phasen zugeordnet. Selbstübergänge werden nicht dargestellt.

Phase 2: Auflösung von Abhängigkeitskonflikten Befindet sich ein Mapping (im Folgenden Kontext-Mapping) in einem der initialen Selektionszustände *active* oder *inactive*, muss auf Verstöße gegen vorberechnete Abhängigkeiten geprüft werden. Hierbei wird je nach Selektionszustand (**SZ**) und primärer Propagationsstrategie (**PS**) unterschiedlich vorgegangen:

- SZ *active* und primäre PS *forward*: Die Selektionszustände der über `requires` assoziierten Mappings werden ermittelt. Ist eines dieser Mappings in der Lage, das Kontext-Mapping zu negativieren (`canSuppress()`), so wird der Selektionszustand des Kontext-Elements zu *suppressed* geändert. Zusätzlich wird das referenzierte Mapping in die Menge `suppressedBy` aufgenommen (vgl. auch Quelltext 4.11).

```
1  if (state == SelectionState.ACTIVE
2          && dcs == DependencyConflictStrategy.FORWARD) {
3      for (MappingRequirement required : getRequires()) {
4          Annotatable target = required.getTarget();
5          if (ps.canSuppress(target.getSelectionState())) {
6              state = SelectionState.SUPPRESSED;
7              getSuppressedBy().add(target);
8          }
9      }
10 }
```

Quelltext 4.11: Auszug aus der Methode `updateSelectionState()` der abstrakten Klasse `AnnotatableImpl`, in dem die primäre Propagationsstrategie *forward* durchgesetzt wird.

- SZ *inactive* und primäre PS *reverse*: Iteriert wird über die von `requiredBy` erreichten Mappings. Ist ein referenziertes Mapping in der Lage, das Kontext-Mapping zu positivieren (`canEnforce()`), wird der Selektionszustand des Kontext-Mappings auf *enforced* gesetzt und das referenzierte Mapping in `enforcedBy` aufgenommen.

Phase 3: Nicht annotierte Mappings Befindet sich ein Mapping im initialen Selektionszustand *incomplete*, ist die sekundäre PS von Bedeutung:

- *forward*: Es wird über alle über `requires` assoziierten Mappings iteriert. Befindet sich eines der referenzierten Mappings nicht im Zustand *incomplete*, wird dieser je nach Selektionszustand `suppressedBy` bzw. `enforcedBy` angefügt. Ist die Menge `suppressedBy` nach der letzten Iteration nicht leer, so wird der Selektionszustand des Kontext-Mappings entsprechend auf `suppressed` geändert. Ansonsten kann die Positivierung im Falle einer nicht leeren `enforcedBy`-Menge erfolgen.
- *reverse*: Wie *forward*, mit dem Unterschied, dass über die über `requiredBy` referenzierten Mappings iteriert wird. Außerdem wird die Menge `enforcedBy` gegenüber `suppressedBy` bevorzugt.
- *forwardThenReverse*: Zunächst wie *forward*. Befindet sich das Kontext-Mapping danach immer noch im Selektionszustand *incomplete*, so wird wie bei *reverse* fortgefahren.
- *reverseThenForward*: Nur wenn sich das Kontext-Mapping nach Anwendung der *reverse*-Strategie immer noch im Zustand *incomplete* befindet, wird *forward* angewendet.

Phase 4: Ausschlusskonflikte Sich gegenseitig ausschließende Mappings wurden bei der Vorberechnung möglicher Konflikte in Phase 0 über die Referenzen `excludes` bzw. `excludedBy` vorgemerkt. Befindet sich das Kontext-Mapping nach Anwendung der PS in einem positiven Selektionszustand, muss noch geprüft werden, ob dieser nicht durch ein eventuell ebenfalls positiv annotiertes, in `excludedBy` enthaltenes Mapping, negativiert werden muss. Ist dies der Fall, wird der SZ des Kontext-Mappings auf *overruled* gesetzt sowie das ausschließende Mapping in `overruledBy` aufgenommen (vgl. Quelltext 4.12).

```
1 if (ps.isIncluded(state)) {
2     for (Annotatable excluding : getExcludedBy()) {
3         if (ps.isIncluded(excluding.getSelectionState())) {
4             state = SelectionState.OVERRULED;
5             getOverruledBy().add(excluding);
6         }
7     }
8 }
```

Quelltext 4.12: Auszug aus der Methode `updateSelectionState()` von `AnnotatableImpl`, wodurch Ausschlusskonflikte berücksichtigt werden.

Phase 5: Surrogate Während für Attribut- und Containment-Mappings der *finale* Selektionszustand an dieser Stelle feststeht, muss bei Referenz-Mappings noch geprüft werden, inwiefern bei der Vorberechnung ermittelte *Surrogate* den Zustand des Kontext-Mappings weiter beeinflussen können: Der Selektionszustand *surrogated* ist für Mappings reserviert, die durch Anwendung von Propagationsstrategien negativiert wurden (sich also im **SZ** *suppressed* befinden), jedoch nach der Auswertung ihrer Surrogat-Ausdrücke mindestens ein Mapping mit positiven Selektionszustand vorweisen können, welches das abhängige Element als Referenzziel ersetzen kann. Quelltext 4.13 zeigt den entsprechenden Quelltextausschnitt.

```
1  if (state == SelectionState.SUPPRESSED && getReferenceMappingDescription()
       instanceof InternalReferenceMappingDescription) {
2      InternalReferenceMappingDescription internalRmd =
           (InternalReferenceMappingDescription) getReferenceMappingDescription();
3      for (MappingRequirement required :
           internalRmd.getReferencedMapping().getRequires()) {
4          for (EObject surrObj : required.getSurrogates()) {
5              ContainmentMapping cm = getMappingModel().getContainmentMapping(surrObj);
6              if (cm != null && ps.isIncluded(cm.getSelectionState())) {
7                  getSurrogateCandidates().add(cm);
8                  state = SelectionState.SURROGATED;
9              }
10         }
11     }
12 }
```

Quelltext 4.13: Auszug aus der überschriebenen Methode `updateSelectionState()` der Klasse `ReferenceMappingImpl`, in dem auf verfügbare Surrogate geprüft wird.

Zur Identifikation von Surrogaten erfolgt eine Iteration über die vom untergeordneten `MappingRequirement` bereitgestellten `surrogates`, welche auf je ein **DM**-Element verweisen. Kann im Mapping-Modell ein Containment-Mapping identifiziert werden, welches dieses abbildet und einen positiven SZ hat (Z. 5 und 6), wird es als Surrogat-Kandidat aufgenommen und der Selektionszustand des Kontext-Mappings auf *surrogated* gesetzt (Z. 7 und 8).

Besonderheit: Surrogate und Ausschlusskonflikte Die beiden zuletzt erwähnten Konzepte *Ausschlusskonflikte* und *Surrogate* ergänzen sich in der bisher vorgestellten Implementierung noch nicht gewinnbringend: An einigen Stellen ist es erforderlich, dass Alternativen-Mappings, die die Rolle eines aufgrund wechselseitigen Ausschlusses als *overruled* gekennzeichneten Kern-Mappings übernehmen, als Surrogat-Kandidaten vorgeschlagen werden können. Um dies zu realisieren, wurden in der tatsächlichen Implementierung von `ReferenceMapping.updateSelectionState()` folgende Erzänzungen durchgeführt:

1. Iteriere über alle über `excludes` erreichbaren, konkurrierenden Referenz-Mappings.
2. Prüfe, ob es sich um ein implizites Referenz-Mapping handelt, und bestimme in diesem Fall dessen referenziertes Containment-Mapping. Ansonsten breche ab. Der Selektionszustand des konkurrierenden Mappings spielt hierbei keine Rolle.
3. Füge das referenzierte Containment-Mapping der Menge `surrogateCandidates` des Kontext-Mappings hinzu, falls sein Selektionszustand positiv ist.

Ein Beispiel, in dem das verbesserte Zusammenspiel zwischen Surrogaten und Ausschlusskonflikten greift, wird in Abschnitt 5.3.6 gegeben.

4.7.6 Invalidierungszyklen

Ein Mapping-Modell ist ständigen Manipulationen unterworfen: Feature-Ausdrücke können vom Anwender erzeugt, geändert oder gelöscht werden, was eine Änderung von SZ einzelner Mappings zur Folge haben kann. Die Struktur des Mapping-Modells kann sich ändern, sei es durch Änderungen am Kern-**DM** oder durch das Hinzufügen von Alternativen-Mappings.

Weitere mögliche Änderungen betreffen Featuremodell und -konfiguration. All diese Manipulationen haben Auswirkungen auf den globalen Zustand des Mapping-Modells, dessen Mappings sich wechselseitig beeinflussen. Erforderlich ist also ein Konzept zur effektiven Neuberechnung des Modellzustands, insbesondere der nicht-persistenten Referenzen zwischen Mappings sowie deren Selektionszustände. Im **F2DMM**-Editor wird hierzu zwischen einem *großen* und einem *kleinen* Invalidierungszyklus unterschieden:

Großer Invalidierungszyklus Tritt bei Änderungen innerhalb der referenzierten Modelle (**SDIRL**, **DM**, **FM** oder **FK**), beim Einfügen von Alternativen, sowie beim erstmaligen Laden eines Mappings ein. Er beinhaltet folgende Aktionen:

- Aufruf der `dismiss()`-Methode auf dem Mapping-Modell, welche die Ergebnisse aller Vorberechnungen verwirft. Dies betrifft neben den transienten Referenzen auch sämtliche vorberechnete Surrogate.
- Synchronisation von **FM** und **FK** (vgl. Abschnitt 4.2.4).
- Synchronisation von **MM** und **DM**, um die strukturelle Nachbildung des ersteren durch das letztere sicherzustellen (s. Abschnitt 4.9.1).
- Aufruf der `prepare()`-Methode auf dem Mapping-Modell, welche zur rekursiven Vorberechnung von Abhängkeiten und Surrogaten (Phase 0) führt.

Der große Invalidierungszyklus unterstützt neben seiner globalen Anwendung auch die Invalidierung einzelner Teilbäume des Mapping-Modells, wodurch der Vorgang merkbar beschleunigt werden kann. Die *inkrementelle* Invalidierung wird etwa beim Einfügen von Alternativen-Mappings eingesetzt: Der große Invalidierungszyklus wird dabei nicht auf dem gesamten Mapping-Modell, sondern auf dem Teilbaum, der das eingefügte Alternativen-Mapping unmittelbar beinhaltet, durchgeführt. Zusätzlich erfolgt ein Aufruf von `inversePrepare`: Abhängigkeiten werden hierbei partiell in entgegengesetzte Richtung neu berechnet. Effektiv entfallen hier sämtliche Neuberechnungen von Abhängigkeiten zwischen Elementen, die bereits vor dem Einfügen der Alternative im Mapping-Modell vorhanden waren.

Kleiner Invalidierungszyklus Tritt ein, wenn sich ein Feature-Ausdruck ändert, eine neue Featurekonfiguration geladen oder die Propagationsstrategie angepasst wird, sowie beim erstmaligen oder erneuten Laden nach dem großen Invalidierungszyklus.

- Aufruf von `invalidate()` auf dem Mapping-Modell, was zum rekursiven Aufruf derselben Methode auf allen Mappings führt. Hierbei werden zunächst die Mengen `enforces`, `suppresses` und `overrules` geleert.
- Neuberechnung der Selektionszustände aller Mappings. Hierzu implementiert die Klasse `F2DMappingModel` die Methode `updateAll()`, die wiederum für jedes Mapping `updateSelectionState()` aufruft, in der die in Phase 1 bis 5 beschriebenen Berechnungen durchgeführt werden. Aufgrund transitiver Abhängigkeiten erfolgt die Neuberechnung iterativ, bis sich keine Änderungen der **SZ** mehr ergeben[30].
- Update der Baum-Ansicht des Editors, um die Änderungen sichtbar zu machen.

[30] Zyklische Abhängigkeiten könnten sich an dieser Stelle in einer Endlosschleife niederschlagen. In der aktuellen Implementierung werden Zyklen weder erkannt noch vermieden; stattdessen ist in `updateAll` eine Abbruchbedingung formuliert, die zutrifft, sobald sich die Menge derjenigen Mappings, deren Selektionszustand in einer Iteration geändert wurde, mehrmals nicht verkleinert.

4.8 Ableitung von Produkten

Der in Abschnitt 3.4.4 vorgeschlagene **MDPLE**-Prozess sieht die Ableitung eines *konfigurierten Domänenmodells* bzw. eines *Produkts* aus einer gewählten Featurekonfiguration vor. Zieht man ausschließlich *negative Variabilität* in Betracht, besteht das Ableiten eines Produkts im Entfernen aller Artefakte aus dem Multivarianten-DM, deren annotierte Feature-Ausdrücke als negativ evaluieren. Im **F2DMM**-Editor ist die Produktableitung als Modelltransformation auf Java-Basis (vgl. Abschnitt 2.7.1) realisiert. In diesem Abschnitt werden außerdem die Konzepte der *Alternativen*, *Surrogate* und abgeleiteter *Reparaturaktionen* während der Produktgenerierung berücksichtigt.

4.8.1 Notwendige Voraussetzungen für die Ableitung von wohlgeformten Produkten

Bevor auf die eigentliche Transformation eingegangen wird, soll diskutiert werden, welche Mindestvoraussetzungen erfüllt sein müssen, damit ein wohlgeformtes Produkt entsteht. Die Annahme hierbei ist, dass alle beteiligten Domänenmodelle, das Kern- (bzw. Multivarianten-)**DM**, sowie die für externe Alternativen-Mappings herangezogenen Teildomänenmodelle, selbst syntaktisch wie semantisch wohlgeformt sind[31].

- Es muss eine Featurekonfiguration (**FK**) geladen sein, die konform zum Featuremodell ist, das die Produktlinie beschreibt.
- Die geladene FK muss valide sein (vgl. Abschnitt 4.2.2) und darf insbesondere keine Features beinhalten, die sich im Selektionszustand *pending* befinden.
- Die primäre **PS** darf nicht *ignore* sein, und die Option `transitiveConflictResolution` muss gewählt sein. Dies stellt die konsistente Anwendung der Propagationsstrategien bei der Produktableitung sicher.
- Feature-Ausdrücke dürfen keine Feature-Referenzen enthalten, welche sich auf nicht existierende Features beziehen. Ansonsten kann der Selektionszustand von betroffenen Features nicht ermittelt werden. Heidenreich [28] definiert eine entsprechende Wohlgeformtheitsbedingung MM-Existing-Feature.
- Das Mapping-Modell muss in seinen Kern-Mappings strukturell *synchron* mit dem Kern-Domänenmodell sein.
- Alternativen-Mappings, die sich auf externe Ressourcen beziehen, müssen auf vorhandene Modell-Elemente verweisen. Diese und die vorhergehende Voraussetzung entsprechen der Wohlgeformtheitsbedingung MM-Existing-ModelElement von Heidenreich [28].

Die Prüfung dieser Voraussetzungen ist zunächst die Aufgabe der **F2DMM**-Modellvalidierung, welche auf dem **EMF** *Validation Framework* aufbaut (vgl. Abschnitt 4.9.3). Die Erfüllung von Voraussetzungen, welche nicht von dieser abgedeckt werden, wird vor dem Ableiten des Produkts im Editor geprüft. Die strukturelle Synchronität von Kern-Mappings mit dem Kern-Domänenmodell wird durch das **DM**/**MM**-Synchronisationsmodul (s. Abschnitt 4.9.1) sichergestellt.

[31] Heidenreich [28] formuliert auch hierfür Wohlgeformtheitsbedingungen (SM-Multiplicity, SM-Typing, SM-Semantics). Hierauf wird im vorliegenden Ansatz verzichtet, da stattdessen die Annahme eines wohlgeformten, Ecore-basierten *solution space model* zur Voraussetzung erhoben wird.

Im F2DMM-Editor ist für die Ableitung eines konfigurierten Domänenmodells der Kontextmenü-Eintrag Derive Product vorgesehen. Wird dieser ausgewählt, erfolgt zunächst ein großer und ein kleiner Invalidierungsschritt (vgl. vorhergehender Abschnitt) sowie die Prüfung der eben definierten Voraussetzungen. Sind diese erfüllt, wird vom Anwender die Auswahl einer Ressource für das abgeleitete Produkt gefordert und die Transformation durchgeführt.

4.8.2 Anpassung des EMF-Copiers für die Basistransformation

Die Modelltransformation vom allgemeinen zum konfigurierten Domänenmodell soll in diesem Unterabschnitt rein *destruktiv* verstanden werden: Es erfolgt eine Kopie des Multivarianten-Domänenmodells, wobei alle **DM**-Elemente, für die ein Mapping mit negativem Selektionszustand vorliegt, entfallen. Die Transformation ist als Ableitung von der Klasse `EcoreUtil.Copier` implementiert, die über die Methode `EObject copy(EObject)` die Erzeugung einer exakten Kopie eines `EObject` unter Berücksichtigung der *referenziellen Integrität*[32] realisiert. `Copier` erbt zusätzlich von `HashMap<EObject, EObject>`: Nach dem Kopieren eines Elements wird ein Eintrag <Original, Kopie> in die `Map` eingefügt.

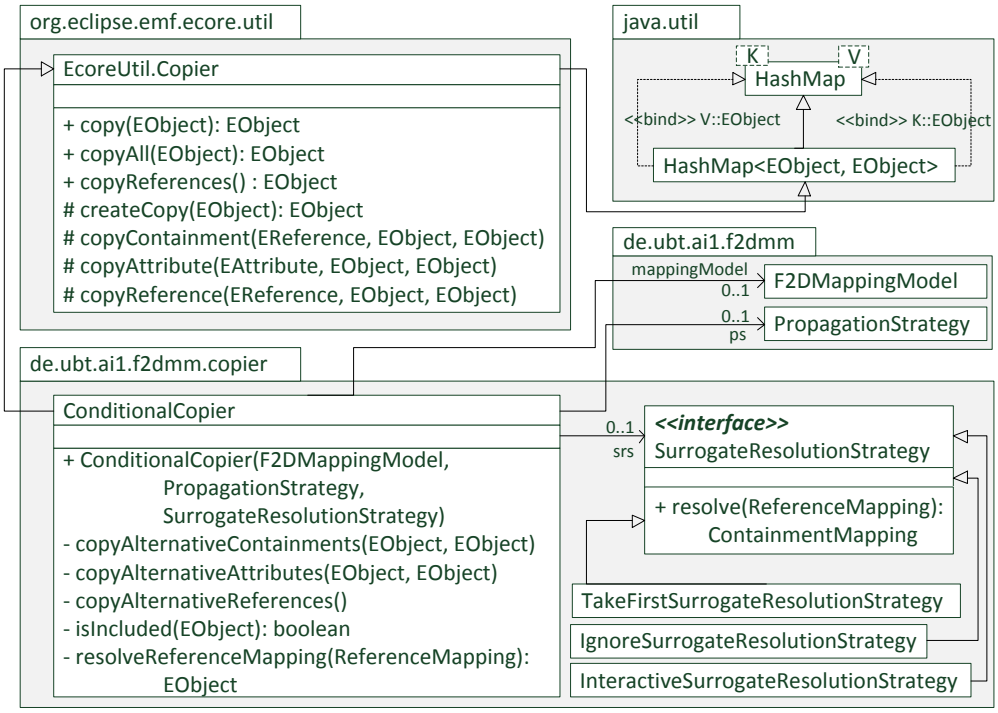

Abbildung 4.29: **UML**-Klassendiagramm, welches den von `EcoreUtil.Copier` erbenden `ConditionalCopier` als Transformationsklasse darstellt. Alle von `Copier` definierten Objektmethoden werden von `ConditionalCopier` überschrieben (nicht abgebildet). Das Mapping-Modell, die entsprechende Propagationsstrategie, sowie eine Stategie zur Auflösung von Surrogat-Mappings (`SurrogateResolutionStrategy`, s. Abschnitt 4.8.4) werden von ihm referenziert.

Abbildung 4.29 führt die Klasse `ConditionalCopier` als Spezialisierung des EMF-Copiers ein. Sie beinhaltet die Methode `isIncluded(EObject)`, welche prüft, ob ein beliebiges

[32] Alle Referenzen der Kopie verweisen nicht auf das Original, sondern auf das entsprechende Element der Kopie [48, Kapitel 16.4].

übergebenes Objekt aus dem Kern-Domänenmodell unter Berücksichtigung der aktuellen Featurekonfiguration von einem positiven Mapping abgebildet wird. Die als `protected` deklarierten Objektmethoden `createCopy`, `copyContainment`, `copyAttribute` sowie `copyReference` wurden so überschrieben, dass sie die entsprechende Methode der Oberklasse `Copier` für das als zweites Argument übergebene Element `e` aus dem Multivarianten-Domänenmodell nur dann aufrufen, wenn `isIncluded(e)` zutrifft.

4.8.3 Behandlung von Alternativen

Bisher wurde die Transformation zur Produktableitung als rein destruktive Kopie im Sinne negativer Variabilität dargestellt. **F2DMM** unterstützt jedoch, wie in Abschnitt 4.6 erläutert, in eingeschränktem Umfang die *positive Variabilität* durch Alternativen-Mappings. Diese müssen ebenfalls während der Transformation berücksichtigt werden, um ein vollständiges Produkt zu erzeugen.

Die Berücksichtigung von Alternativen-Mappings wird durch die Objektmethoden `copyAlternativeContainments` und `copyAlternativeAttributes`, denen jeweils ein Element aus dem allgemeinen und dem konfigurierten Domänenmodell übergeben wird, erzielt. Diese werden innerhalb der überschriebenen `copy()`-Methode pro Domänenmodell-Element rekursiv aufgerufen. *In-Place*-definierte Alternativen-Mappings müssen hier gesondert behandelt werden: Bei ihnen entspricht das Original der Kopie und folglich wird ein Paar identischer Objekte in die `Map` eingefügt.

Der EMF-Copier sieht für das Kopieren von Nicht-Containment-Referenzen eine eigene Methode `copyReferences()` vor, welche nach dem `copy()`-Vorgang aufgerufen werden soll, um die zuvor erwähnte referenzielle Integrität wiederherzustellen: Alle Referenzen werden rekursiv kopiert; das für die Kopie gültige Referenzziel `c` wird über den in der `Map` unter `o` abgelegten Wert über `c = get(o)` ermittelt. Die entsprechende von `ConditionalCopier` überschriebene Methode ruft zusätzlich `copyAlternativeReferences()` auf: Hier werden im beteiligten Mapping-Modell rekursiv `AlternativeReferenceMapping`s bestimmt und deren Referenzziele über ihr referenziertes Domänenmodell-Element, ebenfalls über einen entsprechenden `get`-Aufruf, ermittelt.

4.8.4 Auswahl von Surrogat-Mappings

`ConditionalCopier` referenziert eine Instanz von `SurrogateResolutionStrategy`. Diese Schnittstelle verlangt die Implementierung der Methode `resolve`, welche zu einem übergebenen, sich im Selektionszustand *surrogated* befindlichen[33], `ReferenceMapping` ein entsprechendes `ContainmentMapping` als Surrogat ermittelt. Die Methode kann `null` zurückgeben, um anzudeuten, dass die Strategie keines der von dem übergebenen `ReferenceMapping` vorgeschlagenen `surrogateCandidates` akzeptiert.

In der vorliegenden Implementierung existieren drei Strategien zur Auflösung von Surrogat-Mappings mit folgender Semantik:

[33] Es existiert also mindestens ein Surrogat-Kandidat mit positivem Feature-Ausdruck.

Take First Wählt stets das erste Element aus der Liste von Surrogat-Kandidaten, welche vom übergebenen `ReferenceMapping` ermittelt wurden.

Interactive Lässt den Anwender während der Transformation in einem Benutzerdialog einen der vorgeschlagenen Surrogat-Kandidaten wählen. Dem Benutzer steht frei, den Dialog abzubrechen; in diesem Fall wird `null` zurückgegeben.

Ignore Gibt immer `null` zurück und verhindert so die Durchführung von Reparaturaktionen.

4.8.5 Durchführung von Reparaturaktionen

Die übergebene `SurrogateResolutionStrategy` findet ihre Anwendung in der Methode `resolveReferenceMapping`, welche zur Auflösung von Nicht-Containment-Referenzzielen aufgerufen wird. Sie gibt ein `EObject` zurück, dessen Wert abhängig vom Selektionszustand des übergebenen `ReferenceMapping`s ist[34]:

- Ist der Selektionszustand *inactive*, *suppressed*, *overruled* oder falls `includeIncomplete` auf `false` gesetzt ist auch *incomplete*, so wird `null` zurückgegeben und das Referenzziel nicht gesetzt.
- Ist der Selektionszustand *active*, *enforced* oder im Falle der gesetzten Option `includeIncomplete` auch *incomplete*, wird das Referenzziel des Multivarianten-Modells über `referencedObject` ermittelt. Das entsprechende Objekt des abgeleiteten Produkts wird wiederum über `get()` bestimmt.
- Ist der Selektionszustand *surrogated*, wird das entsprechende Referenzziel des Multivarianten-Modells über die Methode `resolve()` der entsprechenden `SurrogateResolutionStrategy` bestimmt. Ist der Rückgabewert `null`, verhält sich die Transformation wie im Falle eines negativen SZ. Ansonsten wird mittels `get()` die Entsprechung des Surrogats im Zielmodell ermittelt.

Abhängig vom Rückgabewert eines eventuellen `resolve()`-Aufrufs für Mappings mit dem SZ *surrogated* kommt es während der Ableitung eines Produkts folglich zu unterschiedlichen *Reparaturaktionen* für ungültige Referenzziele:

Nachträgliches Entfallen des Referenzziels Gibt `resolve()` den Wert `null` zurück, wird das Ziel der betroffenen Referenz nicht gesetzt. Infolgedessen verhält sich das Mapping so, als hätte es sich vor der Transformation im Zustand *suppressed* befunden.

Ersetzen des Referenzziels durch ein Surrogat Ist der Rückgabewert ein Objekt, für welches in der `Map` über den Aufruf von `get()` eine Kopie – also ein im Produkt enthaltenes Element – ermittelt werden kann, so wird das ursprüngliche Ziel der betroffenen Referenz durch dieses ersetzt.

[34] Zu diesem Zeitpunkt können Mappings mit Selektionszustand *corrupted* bereits ausgechlossen werden.

4.9 Synchronisations- und Validierungsmechanismen

In diesem Abschnitt werden einige Mechanismen erläutert, die die *Synchronität* des Mapping-Modells im **F2DMM**-Editor garantieren (vgl. auch die gleichnamige Anforderung in Abschnitt 1.3.2). Diese besteht zum einen in der strukturellen Rekonstruktion des Mapping-Modells mit dem Kern-**DM**. Zum anderen wurde das in Abschnitt 4.2.4 vorgestellte **FM**/**FK**-Synchronisationsmodul ergänzt, um **FEL**-Annotationen weitestgehend konsistent mit dem Featuremodell zu halten.

Zusätzlich ermöglichen auf dem **EMF** *Validation Framework* basierende Constraints die Erkennung und Korrektur fehlerhafter Feature- oder Attribut-Ausdrücke im entwickelten Mapping-Editor. Hierauf wird zum Ende dieses Abschnitts eingegangen.

4.9.1 Synchronisation von Kern-Domänen- und Mapping-Modell

In Abbildung 4.7 wurde bereits angedeutet, dass die Baumstruktur der Kern-Mappings des **MM** von der Containment-Struktur des Multivarianten-**DM** vorgegeben wird. Ebenfalls fest vorgegeben ist die Struktur von Referenz- und Attribut-Mappings, die sich über vorhandene Containment-Mappings bzw. implizit über die String-Repräsentation von Datentypen auf das Domänenmodell beziehen. Das **DM**/**MM**-*Synchronisationsmodul* stellt die strukturelle Synchronität mit dem Multivarianten-Domänenmodell her, sobald ein Mapping-Modell in den **F2DMM**-Editor geladen wird. Wie bereits in Abschnitt 4.7.6 beschrieben, ist es Bestandteil des *großen Invalidierungszyklus*, der etwa zur Anwendung kommt, wenn das Kern-Domänenmodell manipuliert wurde.

Ähnlich wie im **FM**/**FK**-Synchronisationsmodul ist der Synchronisationsprozess in zwei Stufen realisiert: Zunächst werden *Konsistenzverletzungen* identifiziert, um anschließend *Reparaturaktionen* abzuleiten und diese durchzuführen. Auf eine ausführliche technische Dokumentation der Implementierung wird an dieser Stelle aufgrund der konzeptionellen Ähnlichkeit zum FM/FK-Synchronisationsmodul verzichtet. Vielmehr wird auf die Vorgehensweise bei der Ermittlung von Konsistenzverletzungen eingegangen.

Das implementierte Verfahren betrachtet Containment-, Attribut- und Referenz-Mappings unabhängig voneinander und sucht hierin zuerst nach fehlenden und anschließend nach überflüssigen Mappings. Die Reparaturaktionen müssen nach jedem Einzelschritt durchgeführt werden, um auch ganze Teilbäume rekursiv synchronisieren zu können. Im Folgenden werden die Verfahren zur Identifikation von Konsistenzverletzungen bei den unterschiedlichen Mapping-Typen in algorithmischer Notation erläutert.

Identifikation fehlender und überflüssiger Containment-Mappings Gegeben sei ein Domänenmodell-Element e sowie ein Containment-Mapping cm, das dieses abbildet. Nachfolgender Algorithmus beschreibt, wie Domänenmodell-Elemente unterhalb von e, welche durch kein Unter-Containment-Mapping von cm referenziert werden, sowie Containment-Mappings, welche auf ein nicht (mehr) unter e existierendes Element verweisen, identifiziert werden:

fehlt $\leftarrow \top$
for *ce* $\in$ eContents von *e* **do**
  for *ccm* $\in$ Containment-Mappings von *cm* **do**
    if (Von *ccm* gemapptes Element = *ce*) $\wedge$ (von *ccm* abgebildete Referenz = eContainingReference von *ce*) **then**
      fehlt $\leftarrow \bot$

```
    end if
  end for
end for
if fehlt then
  Merke Einfügen eines neuen Mappings für ce vor
end if
for ccm ∈ Containment-Mappings von cm do
  ueberfluessig ← ⊤
  r ← von ccm abgebildete Containment-Referenz
  for ce ∈ von e über r enthaltene Elemente do
    if Von ccm gemapptes Element = ce then
      ueberfluessig = ⊥
    end if
  end for
  if ueberfluessig then
    Merke Löschen von ccm vor
  end if
end for
```

Identifikation fehlender und überflüssiger Attribut-Mappings Während sich Containment-Mappings immer auf eine Instanz von `EObject` beziehen, beinhalten Attribut-Mappings den String-serialisierten Wert eines Attributs. Bis auf entsprechende Änderungen gleicht der Algorithmus zur Identifikation fehlender bzw. überflüssiger Attribut-Mappings dem oben beschriebenen Algorithmus für Containment-Mappings:

```
fehlt ← ⊤
for a ∈ eAttributes der Klasse von e do
  for v ∈ alle in e über a abgebildete elementare Werte do
    for cam ∈ Attribut-Mappings von cm do
      if (String-Repräsentation von v = Attributwert von cam) ∧ (von cam abgebildetes
      Attribut = a then
        fehlt ← ⊥
      end if
    end for
  end for
end for
if fehlt then
  Merke Einfügen eines neuen Mappings für v vor
end if
for cam ∈ Attribut-Mappings von cm do
  ueberfluessig ← ⊤
  a ← von cam abgebildetes Attribut
  for v ∈ von e über a enthaltene elementare Werte do
    if String-Repräsentation von v = Attributwert von cam then
      ueberfluessig = ⊥
    end if
  end for
```

```
  if ueberfluessig then
    Merke Löschen von ccm vor
  end if
end for
```

Identifikation fehlender und überflüssiger Referenz-Mappings Auch bei der Synchronisation gelten für Referenz-Mappings, die sich auf Nicht-Containment-Referenzen beziehen, besondere Regeln: Bildet das Referenzziel ein Element, welches innerhalb des Kern-Domänenmodells liegt, ab, so wird dessen Mapping in einer `InternalReferenceMappingDescription` referenziert. Eine `ExternalReferenceMappingDescription` verweist dagegen direkt auf ein Element aus einer zusätzlichen Ressource. Aus Gründen der *referenziellen Integrität* findet die Synchronisation von Nicht-Containment-Referenzen erst statt, nachdem die Containment-Struktur entsprechend wiederhergestellt wurde. Die Identifikation von Konsistenzverletzungen erfolgt nach folgender Vorschrift:

```
fehlt ← ⊤
for r ∈ eCrossReferences der Klasse von e do
  for ce ∈ alle in e über r abgebildeten Referenzziele do
    for crm ∈ Referenz-Mappings von cm do
      if ce = von crm referenziertes Element then
        fehlt ← ⊥
      end if
    end for
  end for
end for
if fehlt then
  if ce ist in Ressource von e enthalten then
    rcm ← Containment-Mapping, das ce repräsentiert
    Merke Einfügen eines neuen, internen Mappings für rcm vor
  else
    Merke Einfügen eines neuen, externen Mappings für ce vor
  end if
end if
for crm ∈ Referenz-Mappings von cm do
  ueberfluessig ← ⊤
  r ← von crm abgebildete Referenz
  for ce ∈ von e über r abgebildeten Referenzziele do
    if ce = von crm referenziertes Element then
      ueberfluessig = ⊥
    end if
  end for
  if ueberfluessig then
    Merke Löschen von crm vor
  end if
end for
```

4.9.2 Einbettung der Synchronisation von Featuremodell und -konfiguration

In Abschnitt 4.2.4 wurde ein Wizard vorgestellt, welcher den Benutzer über die Wiederherstellung der *Synchronität* zwischen einer geöffneten Featurekonfiguration und deren assoziiertem Featuremodell informiert (vgl. Abbildung 4.5). Dabei wurde bereits erwähnt, dass dieser Wizard auch in den **F2DMM**-Editor integriert und um eine weitere Dialogseite ergänzt wurde. Abbildung 4.30 zeigt diese: Der Anwender kann aus einer Liste automatisch erzeugter *Umbenennungsvorschläge* wählen, um die Konsistenz von **FEL**-Ausdrücken wiederherzustellen.

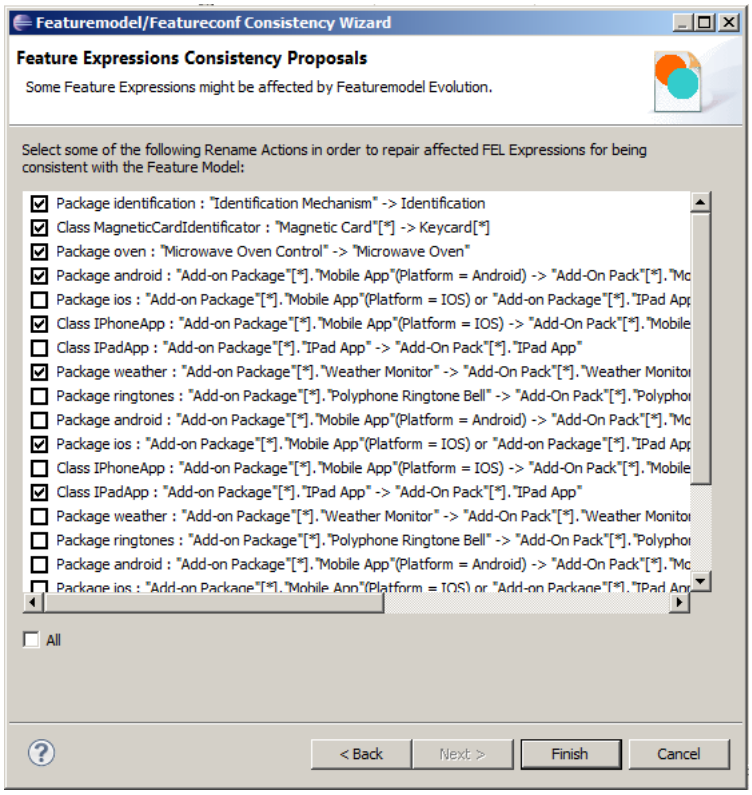

Abbildung 4.30: Automatisch erzeugte Umbenennungsvorschläge für **FEL**-Ausdrücke im in den F2DMM-Editor integrierten FM/FK-Synchronisations-Wizard.

Zur Erzeugung von Umbenennungsvorschlägen werden zunächst die im ersten Schritt der **FM**/**FK**-Synchronisation (vgl. Abschnitt 4.2.4) identifizierten Konsistenzverletzungen analysiert: Für die Generierung von Umbenennungsvorschlägen sind Instanzen von `FeatureNameViolation` von Bedeutung. Sie lassen auf den alten und neuen Namen eines im Featuremodell umbenannten Features schließen.

Feature-Ausdrücke des Mapping-Modells werden dahingehend überprüft, ob sie eines oder mehrere der von einer Umbenennung betroffenen Features referenzieren. Ist dies der Fall, wird ein entsprechender Umbenennungsvorschlag für den gesamten Ausdruck generiert und vorgemerkt. Dabei wird unter anderem sichergestellt, dass auch nach der Umbenennung die Feature-Referenzen eindeutig qualifizierend sind (vgl. Abschnitt 4.4.1). Wie schon in der Abbildung ersichtlich, bleibt es dem Benutzer freigestellt, die Umbenennungsvorschläge zu akzeptieren.

An dieser Stelle sei darauf hingewiesen, dass die Konsistenzprüfung nicht immer vollständig ist: Beispielsweise wird die Kardinalität von Features in der vorliegenden Implementierung nicht überprüft: Eine Feature-Referenz könnte sich über einen Index auf ein Feature beziehen, das im aktualisierten **FM** aufgrund neuer Einschränkungen bezüglich der Kardinalität nicht mehr existieren darf. Die Wiederherstellung der Synchronität ist in solchen Fällen dem Anwender selbst überlassen; er wird jedoch bei der Identifikation von Konsistenzverletzungen von der *Modellvalidierung* unterstützt.

4.9.3 Zusätzliche Konsistenzprüfung mittels EMF-Validierung

Aus den zuvor genannten Gründen, sowie um bei der Ableitung von Produkten einen konsistenten Zustand zu garantieren, wurde in den **F2DMM**-Editor eine auf dem **EMF** Validation Framework (vgl. Abschnitt 2.4) basierende Validierung integriert. Der Anwender kann optional die *Live-Validierung* aktivieren, welche die Laufzeit unter Umständen verschlechtert. Die *Batch-Validierung* findet beim Laden eines Mapping-Modells, durch Auswahl des entsprechenden Kontextmenü-Eintrags, sowie vor der Produkt-Ableitung statt. Zunächst werden exemplarisch drei Modell-Constraints erläutert, bevor eine vollständige Auflistung aller mit Hilfe des Frameworks formulierten Konsistenzbedingungen erfolgt.

Validität von Feature-Referenzen Feature-Referenzen wurden im Abschnitt über **FEL** (4.4) als die Verknüpfung vom Mapping- zum Featuremodell und evtl. einer geladenen Featurekonfiguration ausgemacht. Um gültig zu sein, muss sich eine Feature-Referenz auf ein existierendes Feature aus dem **FM** beziehen. Ist zusätzlich eine **FK** geladen, darf die Menge `configuredFeatures` einer `FeatureReference` ebenfalls nicht leer sein. Aus diesen beiden Bedingungen lässt sich ein Constraint formulieren. `ExistingFeatureRefConstraint` bezieht sich auf Instanzen von `Annotatable` und prüft rekursiv die Gültigkeit von Feature-Ausdrücken:

- Eine Feature-Referenz ist konsistent, wenn die Referenz `modeledFeature` auf ein existierendes Feature aus dem Featuremodell zeigt. Ist eine Featurekonfiguration geladen, darf `configuredFeatures` zusätzlich nicht die leere Menge ergeben.
- Boolesche Konstanten sind immer valide gegenüber dieser Bedingung.
- Boolesche Verknüpfungen sind nur dann valide, wenn ihre Operanden valide sind.

Der Constraint hat die Fehlerstufe *Error*. Wie bei allen auf Feature-Ausdrücken bezogenen Constraints besteht die Möglichkeit der erneuten Annotation des entsprechenden Mappings durch einen Doppelklick auf den vom **EMF** Validation Framework in der Problems-Ansicht generierten *Marker*.

Validität von Attribut-Ausdrücken Die Constraints `IllegalAttributePatternConstraint` und `IncompatibleAttributeValueConstraint`, beide mit der Fehlerstufe *Error*, beziehen sich auf Attribut-Ausdrücke. Letztere kommen im Mapping-Modell innerhalb von `PatternAttributeMappingDescription`s zum Einsatz, um den Wert von Attributen abgeleiteter Produkte als abhängig von der Ausprägung eines Attributs der Featurekonfiguration zu definieren. Unter folgenden Bedingungen wird bei der Validierung durch die genannten Constraints ein Fehler ausgegeben:

- *Nicht wohlgeformtes Attribut-Pattern*: Der innerhalb eines durch „#{“ und „}“ begrenzten Attribut-Constraints stehende Ausdruck enthält entweder Syntaxfehler oder verweist auf ein nicht vorhandenes Feature oder Attribut.
- *Nichtkompatibler Attributwert*: Der nach dem Auswerten einer Pattern-basierten Attribut-Mapping-Beschreibung entstehende String lässt sich nicht über den Ecore-*Adapter-Factory*-Mechanismus[35] in eine Instanz des dem Attribut entsprechenden Datentyps konvertieren.

Die beiden auf Attribut-Ausdrücke bezogenen Constraints unterscheiden sich von allen anderen Validierungsconstraints dahingehend, dass durch einen Doppelklick auf diese sich nicht das Eingabefenster für Feature-Ausdrücke (s. nächster Abschnitt), sondern die entsprechende Seite des Alternativen-Wizards öffnet (vgl. Abschnitt 4.6.2), in der das betroffene Mapping-Pattern zur Korrektur eingegeben werden kann.

Erfüllbarkeit von Feature-Ausdrücken Ebenfalls von der Modellvalidierung überprüft wird die *Erfüllbarkeit* von Feature-Ausdrücken. Ein Feature-Ausdruck gilt genau dann als erfüllbar, wenn mindestens eine mögliche Featurekonfiguration existiert, für den dieser den Wert *active* annimmt[36]. Der Modellconstraint `SatisfiableExprConstraint` setzt diese Idee folgendermaßen für jeden Feature-Ausdruck um (Fehlerstufe *Error*, vgl. auch Quelltext 4.14):

1. Zunächst werden *freie Variablen* identifiziert: Dies sind alle durch den gesamten Ausdruck referenzierten Features der aktuellen **FK**. Jede freie Variable bekommt einen Index zugewiesen.
2. Zum Test auf Erfüllbarkeit wird eine *Kopie* der aktuellen Featurekonfiguration angelegt und dem Feature-Ausdruck als Konfiguration mitgeteilt.
3. Ein *Bitvektor* mit einer Länge n, die sich durch die Anzahl freier Variablen ergibt, wird erzeugt und jeder Eintrag mit 0 initialisiert.
4. Für jede der 2^n möglichen Permutationen des Bitvektors erfolgt nun die Auswertung:
   a) Zunächst wird die Variablenbelegung der kopierten **FK** entsprechend deren Indices im Bitvektor angepasst (*selected*, falls indiziertes Bit 1, sonst *unselected*).
   b) Es wird geprüft, ob die aktuelle Belegung gültig ist: Als *kernel* markierte Features müssen *selected* sein, *requires*- und *excludes*-Kanten dürfen sich nicht widersprechen.
   c) Ist die Belegung gültig, wird der Feature-Ausdruck ausgewertet und der resultierende Selektionszustand festgehalten.
5. Ergibt die Auswertung des gesamten Feature-Ausdrucks zumindest für eine Belegung den Wert *active*, so ist der Ausdruck erfüllbar.

[35] Der Aufruf `EcoreUtil.createFromString()` schlägt nach Übergabe der String-Repräsentation und des Attribut-Datentypen fehl.

[36] In verwandten Ansätzen werden kommen hierfür häufig sog. *SAT-Solver* zum Einsatz, die die Erfüllbarkeit (vgl. engl. *satsifiability*) boolescher Ausdrücke auf Basis eines Logik-Kalküls unterstützen [40].

```
1     private boolean isSatisfiable(Annotatable annotatable) {
2         FeatureExpr expr = EcoreUtil.copy(annotatable.getFeatureExpr());
3         Root featureConf = EcoreUtil.copy(annotatable.getMappingModel()
              .getCurrentFeatureConfiguration());
4         expr.applyFeatureModel(annotatable.getMappingModel().getFeatureModel());
5         expr.applyFeatureConfiguration(featureConf);
6
7         List<Feature> freeVariables = new ArrayList<Feature>(getFreeVariables(expr));
8         long nFree = freeVariables.size();
9         long permSize = 1L << nFree;
10        boolean satisfiable = false;
11
12        for (long permutation = 0; permutation < permSize; permutation++) {
13            applyValues(freeVariables, permutation);
14            expr.invalidate();
15            if (isConfigurationValid(freeVariables)) {
16                if (expr.getState() == FELState.ACTIVE) {
17                    satisfiable = true;
18                    break;
19                }
20            }
21        }
22
23        return satisfiable;
24    }
```

Quelltext 4.14: Die Methode `isSatisfiable` von `SatisfiableExprConstraint` prüft die Erfüllbarkeit von Feature-Ausdrücken durch Permutation freier Variablen.

Folgende weitere Constraints, jeweils auf der Kontextklasse `Annotatable` definiert, stellen die Wohlgeformtheit des Mapping-Modells sicher, indem sie die beschriebenen Konsistenzverletzungen identifizieren:

ExistingAttributeRefConstraint Bezieht sich auf Attribut-Constraints innerhalb von Feature-Referenzen und stellt sicher, dass diese sich auf im Featuremodell existierende Attribute beziehen (*Error*).

DiscouragedWildcardConstraint Das Wildcard-Symbol (`*`) wird als Index für ein Feature verwendet, welches nur in einer Instanz vorkommen darf (*Warning*).

EncouragedWildcardConstraint Kein Index wird an einer Stelle verwendet, wo die Benutzung ein Wildcard-Symbol möglich wäre (*Warning*).

NonTautologicalExprConstraint Ein Feature-Ausdruck ist eine *Tautologie*: Er liefert für jede mögliche Variablenbelegung den Selektionszustand *active* (*Info*).

NoSyntaxErrorsConstraint Aufgrund eines Syntaxfehlers in einem annotierten Feature-Ausdruck eines Mappings existiert die String-serialisierte Form `featureExprStr`, jedoch nicht die deserialisierte `featureExpr` (*Error*). Zusätzlich wird dies durch den Selektionszustand *corrupted* hervorgehoben.

4.10 Weitere Bedienkonzepte des Mapping-Editors

Zum Abschluss des Implementierungsabschnitts werden einige typische Vorgänge während des Editierens von Mapping-Modellen aus Benutzersicht beleuchtet. Erweiterte Einstellungen betreffen zumeist die grafische Repräsentation des Mappings.

4.10.1 Die F2DMM-Perspektive

Der **F2DMM**-Editor definiert eine eigene *Eclipse*-Perspektive, die die für das Bearbeiten eines Mapping-Modells benötigten visuellen Elemente anordnet. Abbildung 4.31 zeigt einen Screenshot und markiert die nachfolgend beschriebenen **UI**-Elemente:

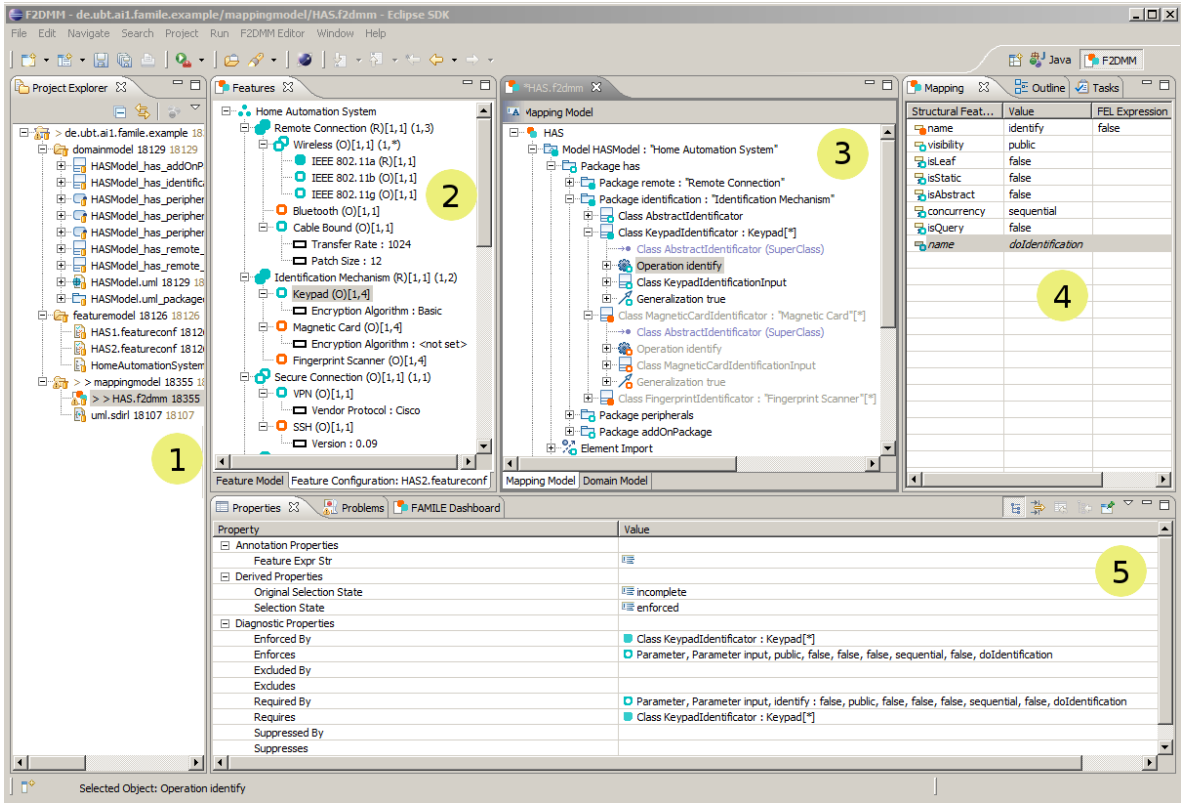

Abbildung 4.31: Die F2DMM-Perspektive bei geöffnetem Mapping-Modell.

1. Der Project Explorer stellt wie in der *Java*-Perspektive die Projektstruktur des *Workspace* dar. Im Beispiel befindet sich das geöffnete Mapping-Modell zusammen mit einem **SDIRL**-Dokument im Verzeichnis `mappingmodel`.

2. Die manuell hinzugefügte Features-Ansicht stellt das Featuremodell und eine eventuell geladene Featurekonfiguration in zwei Reitern dar. Die Darstellung erfolgt jeweils wie im **FM**- bzw. **FK**-Editor als Baum. Die Ansicht dient zur Überprüfung des Selektionszustands von Features. Außerdem bietet sie *Drag-and-Drop*-Unterstützung zur Annotation von Mappings (s. nächster Abschnitt).

3. Die Editor-Ansicht stellt die Hierarchie der Containment-Mappings als Baum dar. Die Symbole eines Mappings entsprechen denen des gemappten Elements, versehen mit einem *Overlay*, welches den aktuellen Selektionszustand (vgl. Abbildung 4.10) abbildet. Die Beschriftung ergibt sich ebenfalls durch die des gemappten Elements und wird ergänzt durch die textuelle Repräsentation des annotierten Feature-Ausdrucks, falls vorhanden. Die Wiederverwendung des Symbols und der Beschriftung des gemappten Elements wird durch Verwendung des Ecore-*Adapter-Factory*-Mechanismus [48, Abschnitt 5.6] realisiert (vgl. auch Abschnitt 2.3.1). Ein weiterer Reiter erlaubt die direkte Manipulation des Domänenmodells, ebenfalls in dessen Baumrepräsentation.

4. Die ebenfalls manuell implementierte Ansicht Mapping Properties stellt Attribut- und Referenz-Mappings dar, welche dem im Editor selektierten Containment-Mapping untergeordnet sind. Die Darstellung erfolgt tabellarisch: In der ersten Spalte ist der Name der gemappten Referenz bzw. des gemappten Attributs sowie sein Symbol – ebenfalls mit einem durch den Selektionszustand bestimmten *Overlay* – abgebildet. Die Value-Spalte beinhaltet die String-Repräsentation des jeweiligen Attributwerts oder die Beschriftung des abgebildeten Referenzziels. Die letzte Spalte ist dem annotierenden Feature-Ausdruck vorbehalten.

5. Die Properties-Ansicht wurde von der **EMF**-Codegenerierung als Teil des Editors erzeugt. Sie stellt Eigenschaften von Elementen dar, die in der Editor-, der Features- oder der Mapping-Properties-Sicht ausgewählt wurden[37]. Unterabschnitt 4.10.3 beschäftigt sich mit dargestellten Eigenschaften von Mappings. Ebenfalls im unteren Bereich befindet sich der Reiter Problems, der von der Modellvalidierung identifizierte Fehler darstellt. Das FAMILE Dashboard ist Gegenstand von Unterabschnitt 4.10.5.

4.10.2 Annotation von Mappings

Bisher wurde die Annotation von Mappings aus Anwendersicht noch nicht genauer beschrieben. Hierfür stehen drei Mechanismen zur Verfügung, die zunächst die textuelle Repräsentation eines Feature-Ausdrucks erzeugen, um diesen anschließend vom generierten Parser in seine Modellrepräsentation übersetzen zu lassen. Nach dem Erzeugen oder Ändern einer Annotation wird grundsätzlich der kleine Invalidierungszyklus durchgeführt (vgl. Abschnitt 4.7.6). Kommt es zu Syntaxfehlern, werden diese dem Anwender sofort mitgeteilt und von der Modellvalidierung als Marker in der Problems-Ansicht eingefügt.

Textuelle Eingabe Durch einen Doppelklick auf ein Mapping in der Editor- oder Mapping-Properties-Ansicht, sowie durch Auswahl des Kontextmenü-Eintrags Annotate Mapping öffnet sich ein Eingabefeld (s. Abbildung 4.32). Ist das Mapping bereits mit einem Feature-Ausdruck versehen, wird dessen bisherige textuelle Repräsentation in den Dialog aufgenommen.

Drag-and-Drop Elemente aus der Features-Ansicht können per Drag-and-Drop (**DnD**) auf ein in der Editor- oder Mapping-Properties-Ansicht dargestelltes Mapping gezogen werden. Dabei wird automatisch eine Feature-Referenz erzeugt, die das gewählte Feature eindeutig qualifiziert. Im Falle eines mehrfach instanziierbaren Features wird der Index aus der **FK**, bzw.

[37] Zur Darstellung der Eigenschaften der beiden manuell definierten Ansichten bedurfte es einer Integration in den von der Eclipse-Plattform bereitgestellten *Selection Service* [48].

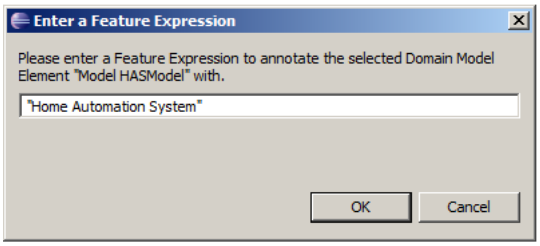

Abbildung 4.32: Textuelle Eingabe eines Feature-Ausdrucks.

das Wildcard-Symbol beim Ziehen aus dem Feature-Model-Reiter, eingefügt. In der F2DMM-Preference-Page (s. Abschnitt 4.10.4) kann eine der folgenden Optionen für das Verhalten bei einer bereits vorhandenen Annotation gewählt werden:

Replace Ein vorhandener Ausdruck wird von der erzeugten Feature-Referenz ersetzt.

And Der bisherige wird mit dem erzeugten Ausdruck *and*-verknüpft.

Or Es erfolgt eine *or*-Verknüpfung.

Xor Es erfolgt eine *xor*-Verknüpfung.

Kopieren per Zwischenablage Ähnlich wie das Erzeugen einer Feature-Referenz per **DnD** funktionert der integrierte Copy-And-Paste-Mechanismus für Features. Durch einen Rechtsklick auf das gewählte Element in der Features-Ansicht und Auswahl des Eintrags Copy FEL Expression to Clipboard (Schritt 1 in Abbildung 4.33) wird eine eindeutig qualifizierende Feature-Referenz erzeugt und in der Zwischenablage des Editors gespeichert. Sie kann durch Auswahl eines entsprechenden Kontextmenü-Eintrags als Annotation eines Mappings eingefügt werden (Schritt 2). Hierbei steht dem Anwender die direkte Auswahl einer der vier im vorhergehenden Absatz definierten Einfügeoptionen zur Verfügung. Der Copy-And-Paste-Ansatz unterstützt die Auswahl mehrerer Features in der Features-Ansicht.

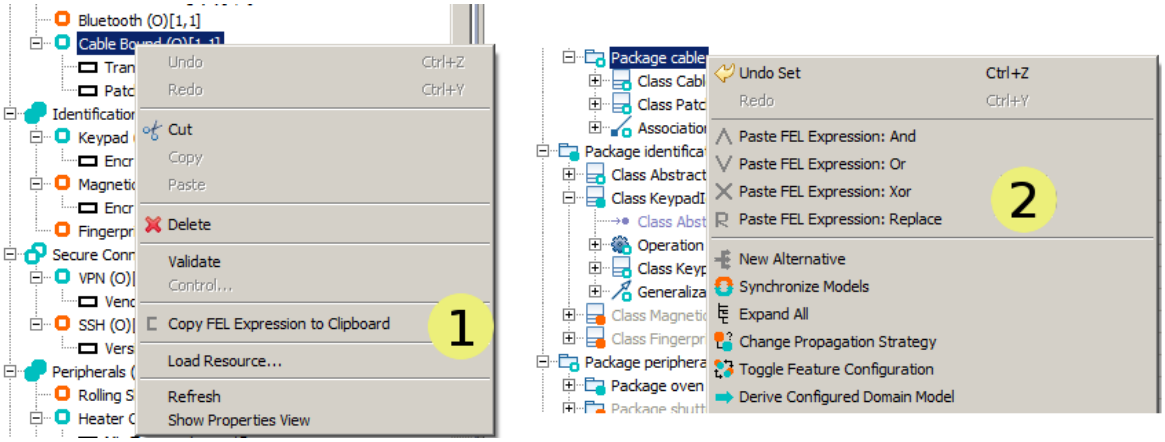

Abbildung 4.33: Eingabe eines Feature-Ausdrucks per Copy-And-Paste.

4.10.3 Das Property-Sheet als Diagnose-Ansicht

In Abbildung 4.31 ist die Properties-Ansicht bei einem geöffnetem Mapping-Modell zu sehen: Bei der Generierung des Editors wurde das Generator-Modell [48, Abschnitt 2.4] so angepasst, dass sie ausschließlich nicht-editierbare Felder enthält. Sie dient also nicht, wie häufig bei generierten **EMF**-Baumeditoren, zur Manipulation, sondern vielmehr zur Analyse von Modell-Eigenschaften. Nach Auswahl eines Mappings ist etwa dessen String-repräsentierte Annotation sowie der aktuelle Selektionszustand dargestellt. Zusätzlich ist unter Original Selection State der initiale Zustand eines Mappings vor der Anwendung von Propagationsstrategien zu sehen. Alle weiteren Eigenschaften sind als Diagnostic Properties zusammengefasst und bieten Hilfestellung, um die Auswirkungen von Propagations- und Surrogat-Mechanismen nachzuvollziehen. Sie stellen die Beziehungen zwischen `Annotatable` dar (vgl. Abbildung 4.24):

- Requires bzw. Required By: Vorberechnete, ein- und ausgehende Abhängigkeiten des gewählten Mappings: Entspricht den jeweils über Instanzen der Assoziationsklasse `MappingRequirement` abgebildeten Referenzzielen.
- Excludes bzw. Excluded By: Andere Mappings, die aufgrund vorberechneter Ausschlusskonflikte vom gewählten Mapping ausgeschlossen werden, bzw. dieses ausschließen.
- Enforces bzw. Enforced By: Andere Mappings, die aufgrund des Selektionszustands des gewählten Mappings den Zustand *enforced* zugewiesen bekommen, bzw. diejenigen Mappings die den Zustand *enforced* des gewählten Mappings bedingen.
- Suppresses bzw Suppressed By: Mappings, die aufgrund des gewählten Mappings in den Zustand *suppressed* fallen, bzw. Mappings, die den Zustand *suppressed* des gewählten Mappings bedingen.
- Overrules bzw Overruled By: Mappings, die aufgrund des gewählten Mappings in den SZ *overruled* fallen, bzw. Mappings, die diesen SZ für das gewählte Mapping bedingen.

4.10.4 Anzeigeoptionen auf der F2DMM-Preference-Page

Die Eclipse-Plattform [14] stellt für *Plugins* die Möglichkeit der Definition von *Workspace*-weiten Benutzereinstellungen (*Preferences*) zur Verfügung. Die Persistierung der Einstellungen findet in einem sog. *Preference Store* statt; In einer *Preference Page* können Anwender diese Einstellungen nach eigenen Bedürfnissen anpassen. Auch der **F2DMM**-Editor bietet über diesen Mechanismus Einstellungsmöglichkeiten an, die überwiegend die Darstellung des Mapping-Baums betreffen. Zusätzlich zum Eclipse-eigenen Menüeintrag Preferences ist die entsprechende Preference Page direkt über das Kontextmenü erreichbar. Abbildung 4.34 bildet die verfügbaren Einstellungen ab.

Die beiden unteren Optionsgruppen sowie die unmittelbar darüberliegende Option wurden bereits im Vorfeld diskutiert: Sie erlauben die Auswahl einer Strategie für Surrogat-Auflösung (vgl. Abschnitt 4.8.4) sowie einer Drag-and-Drop-Operation (vgl. vorhergehenden Unterabschnitt 4.10.2). Außerdem kann an dieser Stelle die Live-Validierung (vgl. Abschnitt 4.9.3) aktiviert werden. Die weiteren – die Baumansicht des Editors beeinflussenden – Einstellungen werden im Folgenden diskutiert.

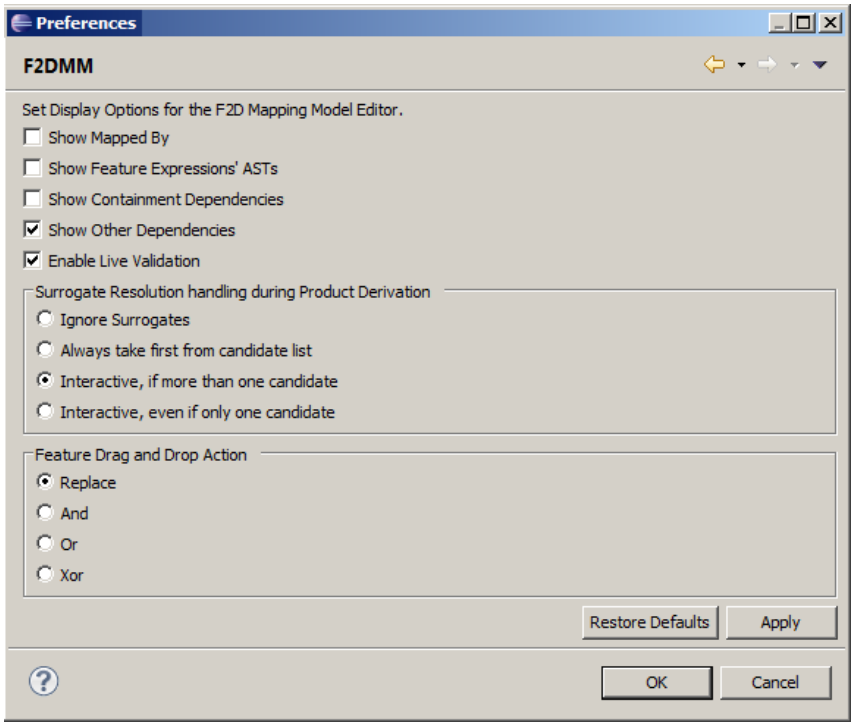

Abbildung 4.34: Preference Page für den F2DMM-Editor.

Abstrakter Syntaxbaum von FEL-Ausdrücken Die Option Show Feature Expressions' ASTs aktiviert die Darstellung *abstrakter Syntaxbäume* (**AST**s) unterhalb mit Feature-Ausdrücken versehener Mappings. Abbildung 4.35 stellt die **EMF**-Baumrepräsentation für einen komplexen Feature-Ausdruck dar: Teilausdrücke werden zusätzlich in ihrer textuellen Repräsentation angezeigt. Feature-Referenzen und boolesche Verknüpfungen sowie Attribut-Constraints sind mit einem *Overlay* versehen, welches den Selektionszustand für den entsprechenden Teilausdruck visualisiert. Dies erleichtert dem Anwender die Diagnose von Feature-Ausdrücken anhand einer geladenen Featurekonfiguation.

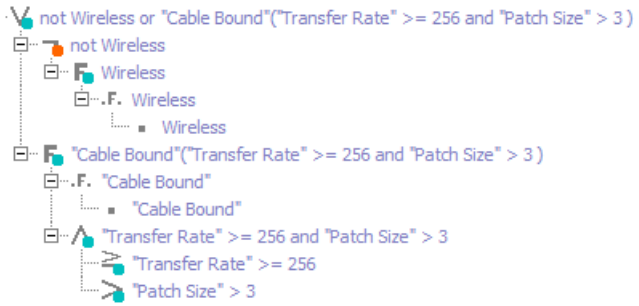

Abbildung 4.35: Darstellung des abstrakten Syntaxbaums eines komplexeren Feature-Ausdrucks.

Mapping-Beschreibungen Mapping-Beschreibungen wurden in Abschnitt 4.6.1 als Verknüpfung zwischen Mapping-Modell und Kern-Domänenmodell eingeführt. Sie dienen nicht nur der logischen Abstraktion, sondern auch zur optionalen Darstellung von Eigenschaften, die sich auf die Abbildung von Domänenmodell-Elementen beziehen. Abbildung 4.36 stellt die Eigenschaften einer selektierten Mapping-Beschreibung für ein Attribut-Mapping in der Properties-Ansicht dar: Neben dem String-serialisierten Attributwert ist das gemappte Attribut aus dem Domänen-Metamodell aufgeführt. Die Darstellung wird über die Option Show Mapped By aktiviert, welche sich auch auf die Mapping-Properties-Ansicht auswirkt.

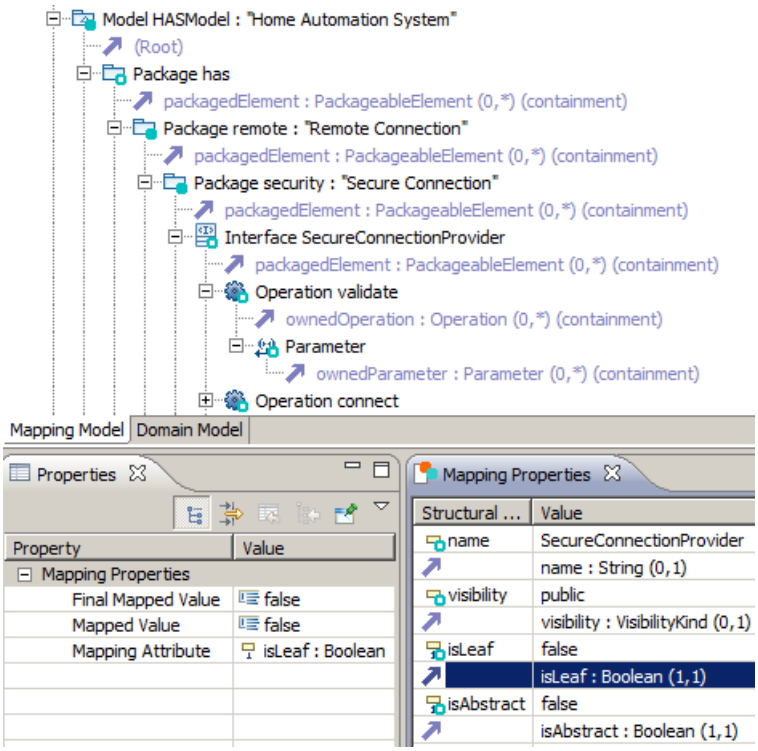

Abbildung 4.36: Mapping-Beschreibung eines Attribut-Mappings im Property Sheet.

Visualisierung von Abhängigkeiten Bei der Vorberechnung von möglichen Konflikten (vgl. Abschnitt 4.7.3) werden identifizierte Abhängigkeiten als Instanzen von `MappingRequirement` im Modell festgehalten. Auch diese werden nach Setzen entsprechender Optionen in der *Preference Page* angezeigt: Die Optionen Show Containment Dependencies und Show Other Dependencies aktivieren unabhängig voneinander die Darstellung von Containment- bzw. **SDIRL**-definierten Abhängigkeiten im Mapping-Baum sowie in der Mapping-Properties-Ansicht (vgl. Abbildung 4.37). Die Darstellung umfasst Quelle und Ziel (Source bzw. Target) der unter Dependency festgehaltenen, zutreffenden Abhängigkeit. Als Surrogate vorberechnete Domänenmodell-Elemente werden unter Surrogates aufgeführt.

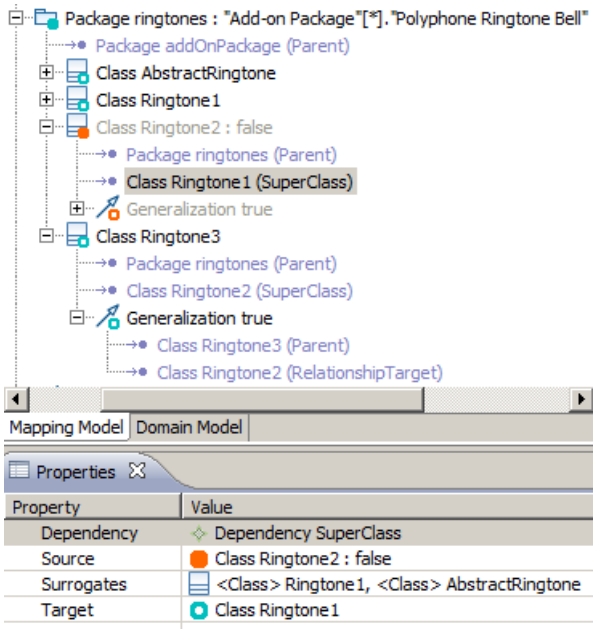

Abbildung 4.37: Repräsentation einer Abhängigkeit zwischen Mappings (`MappingRequirement`) im Property Sheet.

4.10.5 Der F2DMM-Wizard und das FAMILE-Dashboard

Die Initialisierung eines **F2DMM**-basierten Mapping-Modells verlangt die Angabe der beteiligten Modelle. Hierfür ist ein entsprechender Wizard (vgl. Abbildung 4.38) vorgesehen. Er wurde in seiner ursprünglichen Form als Teil des *Editor*-Plugins generiert (vgl. Abschnitt 2.3.2) und nachträglich manuell ergänzt. Bei der Erzeugung eines Mapping-Modells wird initial die Angabe eines Bezeichners sowie eines Multivarianten-Domänenmodells und eines Featuremodells verlangt. Optional können eine Featurekonfiguration sowie ein **SDIRL**-Dokument referenziert werden.

Nicht als Gegenstand dieser Masterarbeit, sondern im Rahmen des Lehrstuhlprojekts **FAMILE** wurde ein sog. *Dashboard* integriert, welches den Anwender bei der Erzeugung des Mapping-Modells und referenzierter Modelle begleitet. In der vorliegenden Version fehlt die Unterstützung für das automatische Ableiten des konfigurierten Domänenmodells aus dem Mapping-Modell (vgl. auch Abbildung 4.39).

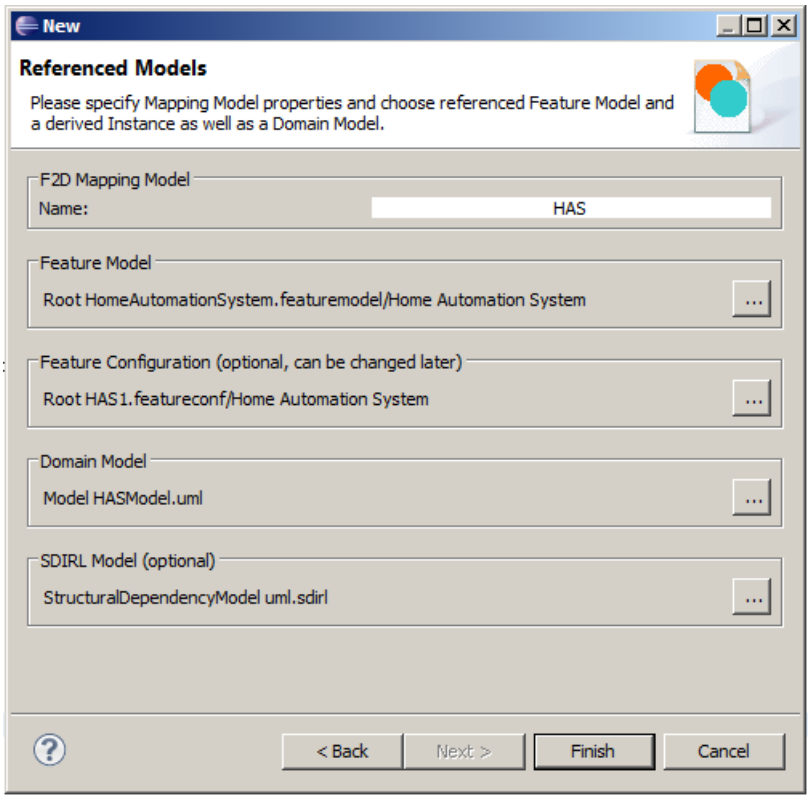

Abbildung 4.38: Angabe referenzierter Modelle im **F2DMM**-Modell-Wizard.

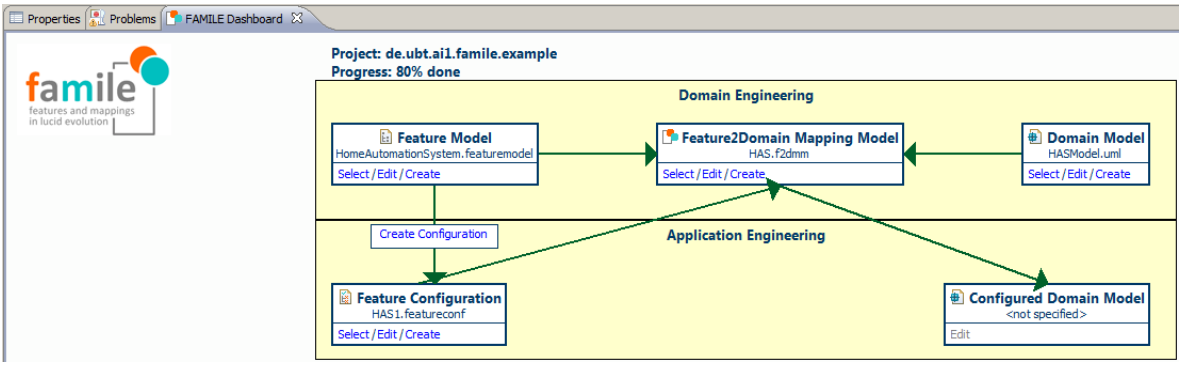

Abbildung 4.39: Zusammenführung der Werkzeuglandschaft im FAMILE-Dashboard.

5 Beispiel zur Evaluierung des Ansatzes

Im vorhergehenden Abschnitt wurde der im Kontext dieser Arbeit implementierte Mapping-Editor **F2DMM** für modellgetriebene Softwareproduktlinien vorgestellt. Gegenstand dieses Abschnitts ist die Evaluierung der Implementierung sowie der zugrundeliegenden Konzepte anhand eines konstruierten Beispiels. Dabei steht nicht die Anwendung des Werkzeugs im Sinne einer Fallstudie im Vordergrund; hierauf wird in Abschnitt 5.5 ein Ausblick gegeben. Stattdessen wurde das aus der **SPL**-Literatur [45] bekannte Beispiel der „Home-Automation"-Systeme (**HAS**) aufgegriffen und hinsichtlich der vorgestellten Mechanismen zur Wiederherstellung der Konsistenz sowie zur Vermeidung von Informationsverlust – etwa durch *Propagation* und *Surrogate* – angepasst.

Zunächst werden beteiligte Modelle (**FM**, zwei repräsentative **FK**s, sowie ein **UML2**-basiertes Domänenmodell) eingeführt (Abschnitt 5.1), bevor Constraints auf dem **UML2**-Metamodell mit Hilfe der Sprache **SDIRL** formuliert werden (Abschnitt 5.2). Sie beziehen sich lediglich auf Klassen- und Zustandsdiagramme. Anschließend werden in Abschnitt 5.3 *repräsentative Konstellationen* im Domänenmodell identifiziert, für die eine Abbildung zunächst Konsistenzverletzungen nach sich ziehen würde. Die Auflösung dieser Konflikte durch die dem **F2DMM**-Editor zugrundeliegenden Konzepte wird im selben Abschnitt beschrieben. Aus den beiden Featurekonfigurationen abgeleitete Produkte werden in Abschnitt 5.4 diskutiert.

5.1 Beteiligte Modelle

5.1.1 Featuremodell

Das vorliegende Beispiel beschreibt eine Produktlinie für sog. „Home-Automation"-Systeme (**HAS**), deren Ziel die Integration von Haushaltsgeräten zur automatischen Wohnraumsteuerung ist [45, Kapitel 3]. Komponenten wie Heizung, Klimaanlage, Sicherheit und Küchengeräte können über eine zentrale Einheit gesteuert werden und sind über einen Kommunikationskanal miteinander verbunden. Maßgeschneiderte *Produkte* einer **HAS**-Produktlinie enthalten diese Komponenten optional und in verschiedenen Ausführungen; diese Variabilität ist im Featuremodell in Abbildung 5.1 festgehalten.

Das Beispiel-**FM** enthält unter dem Wurzel-Feature `Home Automation System` (Kardinalits-Intervall $[1,1]$, Selektions-Intervall $[0,*]$) fünf voneinander unabhängige Feature-Gruppen, deren Aufbau und Funktion im Folgenden erklärt wird:

Remote Connection (obligatorisch): Enthält identifizierte Softwaremerkmale, die die Charakteristika der Datenübertragung zwischen den Komponenten eines **HAS** beschreiben: Diese kann entweder drahtlos per `Bluetooth` oder `Wireless LAN` erfolgen. Bei zweiterer Alternative wird zwischen den Übertragungsstandards `IEEE 802.11a` (obligatorisch, falls Eltern-Feature selektiert), `802.11b` und `802.11g` (beide optional) unterschieden. Die optionale kabelgebundene Übertragung (`Cable Bound`) deklariert die Attribute `Transfer Rate` (Übertragungsrate in Mbit) und `Patch Size` (Anzahl der Steckplätze im Patchfeld).

Identification Mechanism (obligatorisch): Definiert drei Mechanismen zur Benutzeridentifikation, von denen einer oder zwei gewählt werden dürfen (vgl. Selektions-Intervall $[1,2]$). Die Features `Keypad` und `Magnetic Card` enthalten jeweils ein Attribut `Encrpytion Algorithm`, mit dem der Verschlüsselungsalgorithmus als String-Konstante angegeben

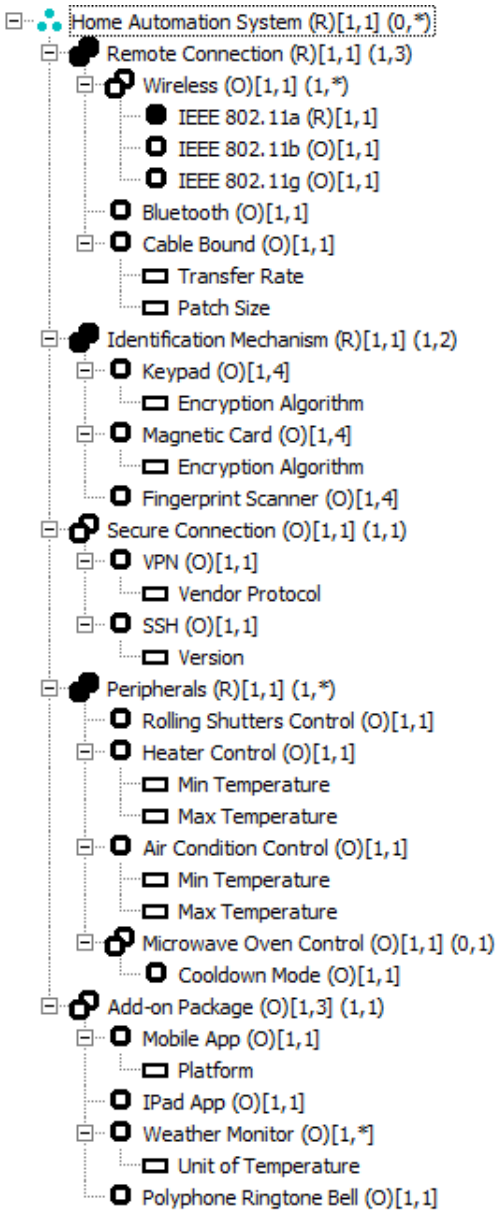

Abbildung 5.1: Beispiel-Featuremodell für „Home-Automation“-Systeme in der durch den Featuremodell-Editor definierten Baumsyntax.

werden soll. Zusätzlich existiert ein `Fingerprint-Scanner`-Mechanismus. Jedes der gewählten Identifikationsmodule kann jeweils in ein bis vier Instanzen vorkommen (Kardinalitäts-Intervall $[1, 4]$).

Secure Connection (optional): Erlaubt eine gesicherte (`SSH`) bzw. eine `VPN`-getunnelte Datenverbindung, unabhängig vom Übertragungsstandard. Die jeweils einwertigen Features sind durch die Attribute `Version` bzw. `Vendor Protocol` parametrisiert.

Peripherals (obligatorisch): In dieser Feature-Gruppe ist die Kommunikation mit den Endgeräten des Haushalts festgehalten. Sie enthält die folgenden Features: `Rolling Shutters Control` modelliert die Ansteuerung der elektronischen Rollläden. Die Features `Heater Control` und `Air Condition Control` steuern jeweils Heizung bzw. Klimaanlage an und sind mit jeweils mit den Attributen `Min Temperature` und `Max Temperature` parametrisiert, welche das gewünschte Temperaturintervall definieren. In der übergeordneten Feature-Gruppe ist zudem die Gruppe `Microwave Oven Control` (Mikrowellen-Ofen) vorhanden, die das optionale Feature `Cooldown Mode` enthält. Eine Selektion soll bewirken, dass nach dem Erhitzungsvorgang die Ofentür für eine gewisse Zeit verriegelt ist, um Verbrennungen zu vermeiden.

Add-on Package (optional): Durch Selektion dieses Features können bis zu drei Erweiterungspakete im konfigurierten **HAS** gewählt werden. In jeder Instanz kann wiederum eines von vier voneinander unabhängigen Zusatz-Features gewählt werden: `Mobile App` erlaubt die Fernsteuerung per Mobiltelefon. Die spezifische Mobilplattform ist unter dem Attribut `Platform` anzugeben. Das Feature `IPad App` modelliert die Fernsteuerung für den gleichnamigen Tablet-Computer. Das Merkmal `Weather Monitor` erlaubt die Installation beliebig vieler Wetterstationen, die sich in der anzuzeigenden Temperatureinheit (`Unit of Temperature`) unterscheiden können. `Polyphone Ringtone Bell` bietet zusätzliche Benachrichtigungstöne an.

5.1.2 Featurekonfigurationen

Aus dem eben beschriebenen Featuremodell wurden für das **HAS**-Beispiel zwei Featurekonfigurationen `HAS1` und `HAS2` abgeleitet, die in Abbildung 5.2 nebeneinander abgebildet sind. Sie repräsentieren zwei unterschiedlich konfigurierte Produkte aus der Produktlinie: Während `HAS1` beispielsweise zwei Erweiterungspakete (`Mobile App` mit Plattform `Android`, `Weather Monitor`) definiert, beansprucht `HAS2` alle drei möglichen Instanzen, um eine `Mobile App` mit Plattform `IOS`, eine `IPad App` sowie polyphone Benachrichtigungstöne zu selektieren. Weiterhin beinhaltet `HAS2` im Gegensatz zu `HAS1` den erwähnten `Cooldown Mode` für den Mikrowellen-Ofen.

5.1.3 Multivarianten-Domänenmodell

Das Domänenmodell, auf welches die Features abgebildet werden sollen, ist als Instanz des Eclipse-**UML2**-Metamodells speziell für dieses konstruierte Beispiel erstellt worden. Zur Modellierung sowie zur Visualisierung entsprechender Teile des Domänenmodells (Paket-, Klassen- und Zustandsdiagramme) wurde das am Lehrstuhl entwickelte UML2-Modellierungswerkzeug *Valkyrie* [11] eingesetzt. Im Folgenden werden relevante Teile des Multivarianten-**DM** anhand geeigneter Diagrammarten erläutert.

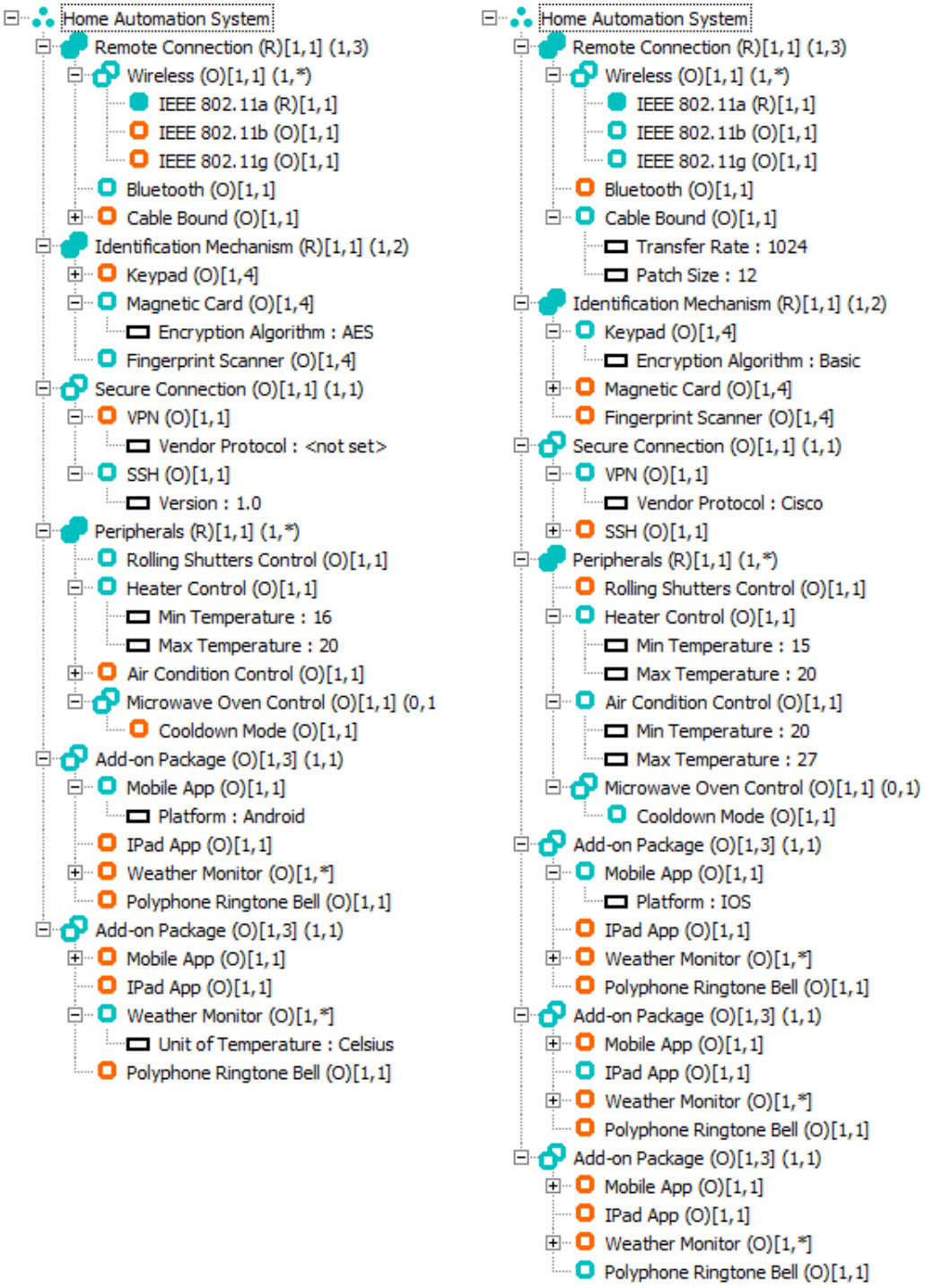

Abbildung 5.2: Die beiden Beispiel-Featurekonfigurationen `HAS1` (links) und `HAS2` (rechts) in der durch den Featurekonfigurations-Editor definierten Syntax.

Paketdiagramm Das Paketdiagramm in Abbildung 5.3 stellt die Hierarchie der ineinander verschachtelten Pakete des UML2-Domänenmodells dar. Obgleich sich die Struktur in einigen Details unterscheidet, wurde die Hierarchie der des Featuremodells zu großen Teilen nachempfunden. Zur Abbildung von Feature-Gruppen wurden häufig abstrakte Klassen vorgesehen (z.B. `AbstractIdentificator`, in der Diagrammdarstellung kursiv notiert). Enthaltene Features hingegen sollen auf nicht-abstrakte Unterklassen abgebildet werden.

Klassendiagramm für das `wifi`-Paket Abbildung 5.4 stellt das Klassendiagramm aus dem Paket `has.remote.wifi` des **DM** dar: Die Funktionalität der kabellosen Kommunikation ist in der abstrakten Klasse `AbstractWifiConnector` zusammengefasst. Während `BluetoothConnector` direkt von dieser erbt, ist für die Wireless-**LAN**-Kommunikationsklassen eine weitere abstrakte Klasse `AbstractIEEE802Connector` vorgesehen, von der je im Featuremodell festgehaltenem Übertragungsstandard eine konkrete Unterklasse existiert.

Klassendiagramm für das `addOnPackage`-Paket Das UML2-Klassendiagramm in Abbildung 5.5 stellt die existenzielle Abhängigkeit der Klasse `AddOnPackage` von seiner übergeordneten `AddOnRegistry` dar. Die Kardinalität der Assoziation soll jedoch in abgeleiteten Produkten durch deren Featurekonfiguration, genauer durch das Variabilitätsmerkmal *Multiplizität* im Bezug auf das Feature `Add-on Package` definiert werden können.

Zustandsdiagramm: Verhalten der Klasse `MicrowaveOven` Das Verhalten mehrerer Klassen aus den Paketen `peripherals` wird mit Hilfe von UML2-Zustandsdiagrammen beschrieben. Als laufendes Beispiel für diese Diagrammart dient das Zustandsdiagramm der Klasse `MicrowaveOven` (s. Abbildung 5.6). Neben dem Start- bzw. Endzustand existieren `Off_Closed`, `Off_Opened`, welche den ausgeschaltenen Mikrowellen-Ofen bei geschlossener bzw. geöffneter Tür repräsentieren, sowie `On_Heating`. Das im **FM** als optional identifizierte Merkmal `Cooldown Mode` wird im **DM** durch den Zustand `On_Cooldown` realisiert. Er fungiert als Verbindung zwischen `On_Heating` und `Off_Closed`; zusätzlich ist ein direkter Übergang `stopOven` zwischen diesen beiden Zuständen realisiert.

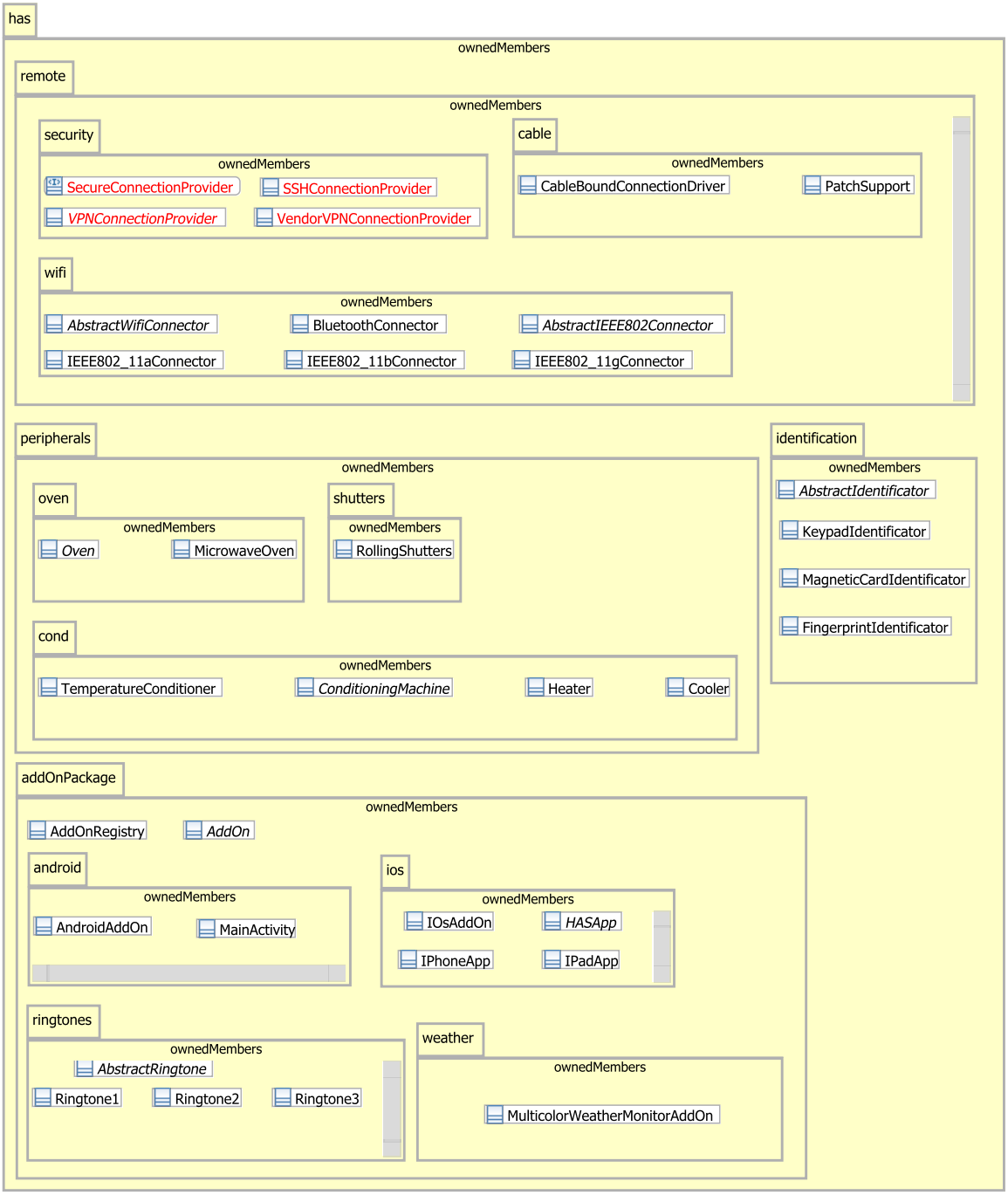

Abbildung 5.3: Paketdiagramm des Multivarianten-Domänenmodells.

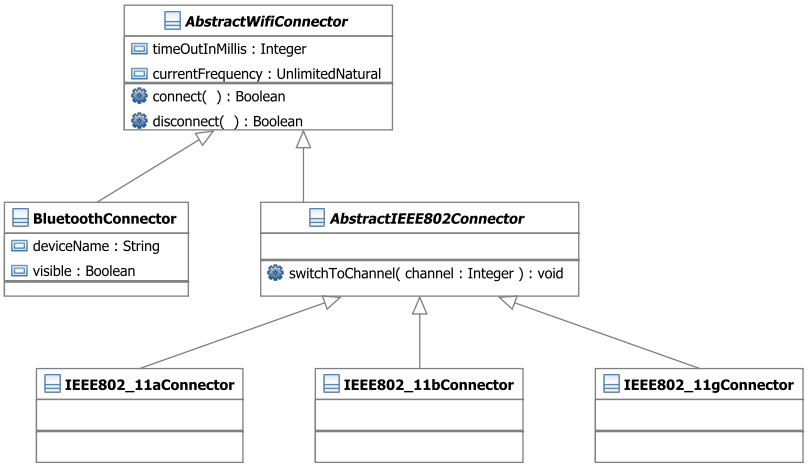

Abbildung 5.4: Klassendiagramm für das `wifi`-Paket des Beispiel-Domänenmodells.

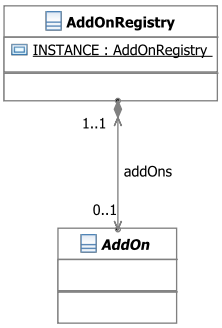

Abbildung 5.5: Klassendiagramm für das Paket `addOnPackage`.

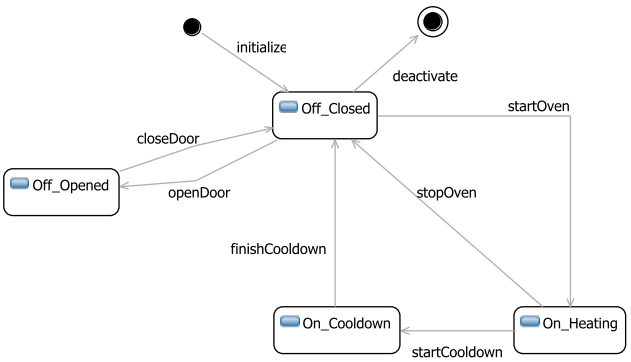

Abbildung 5.6: Zustandsdiagramm für die Klasse `MicrowaveOven` des Multivarianten-DM.

5.2 Strukturelle Abhängigkeiten für UML2-Klassen- und Zustandsdiagramme

Der im Rahmen dieser Arbeit vorgeschlagene Prozess für die modellgetriebene Entwicklung von Softwareproduktlinien (vgl. Abschnitt 3.4) definiert die Entwicklung eines *Konsistenzmodells* durch Analyse der Domänensprache als Voraussetzung für die Erzeugung eines Mapping-Modells. **F2DMM** sieht hierfür die textuelle Sprache **SDIRL** vor (vgl. Abschnitt 4.5). Für dieses Beispiel wurden SDIRL-basierte *Abhängigkeitsbedingungen* für UML2-Klassen- und Zustandsdiagramme formuliert (vgl. Quelltext 5.1). Diese Bedingungen werden im Folgenden erläutert, um die aus ihnen abgeleiteten *Reparaturaktionen* (s. Abschnitt 5.4) nachvollziehen zu können.

```
1  import "http://www.eclipse.org/uml2/3.0.0/UML"
2
3  /* CLASS DIAGRAMS */
4
5  dependency RelationshipTarget {
6    element rel : uml.DirectedRelationship
7    requires trg : uml.Element
8    when {
9      rel.target->includes(trg)
10   }
11 }
12
13 dependency MemberEndAssociation {
14   element ass : uml.Association
15   requires prop : uml.Property
16   when {
17     (ass.memberEnd->includes(prop)) or (ass.ownedEnd->includes(prop))
18   }
19 }
20
21 dependency PropertyType {
22   element prop : uml.Property
23   requires tp : uml.Type
24   when {
25     prop.type = tp
26   }
27 }
28
29 dependency SuperClassifier {
30   element spec : uml.Classifier
31   requires gen : uml.Classifier
32   when {
33     spec.general->includes(gen)
34   }
35   surrogate {
36     gen.general.allParents()
37   }
38 }
39
40 /* STATE CHARTS */
41
42 dependency TransitionState {
```

```
43      element trans : uml.Transition
44      requires state : uml.State
45      when {
46          trans.source = state or trans.target = state
47      }
48  }
```

Quelltext 5.1: **SDIRL**-Dokument, in dem einige Abhängigkeitsbedingungen für das **UML2**-Metamodell (Klassen- und Zustandsdiagramme) formuliert werden.

- `RelationshipTarget` Die Abhängigkeit einer durch `DirectedRelationship` repräsentierten gerichteten Beziehung von ihrem Quellelement ist bereits durch die Containment-Beziehung abgedeckt. Zusätzlich muss das Ziel der Beziehung vorhanden sein, was durch diese benutzerdefinierte Abhängigkeit sichergestellt wird.

- `MemberEndAssociation` Eine ungerichtete Assoziation, dargestellt durch die UML2-Klasse `Association`, ist auf die Existenz beider Assoziationsenden angewiesen. Diese können über die Referenzen `memberEnd` oder `ownedEnd` von der Assoziation selbst abhängen.

- `PropertyType` Eine `Property`, die strukturelle Eigenschaften in UML2-Klassendiagrammen modelliert, hängt von ihrem Typ ab, falls ein solcher existiert[38].

- `SuperClassifier` Eine UML2-Klasse hängt von ihren eventuell vorhandenen Oberklassen ab. An dieser Stelle wird ein *Surrogat* definiert, um Informationsverlust zu vermeiden: Wird eine Oberklasse aus dem Produkt aufgrund eines negativen Feature-Ausdrucks ihres Mappings entfernt, kann sie wiederum durch eine derer Oberklassen, möglicherweise auch transitiv, ersetzt werden. In einer mehrstufigen Vererbungshierarchie kann auf diese Weise eine mittlere Klasse entfallen (s. unten).

- `TransitionState` UML2-Zustandsdiagramme enthalten Zustände (repräsentiert durch die Klasse `State`) und Übergänge (`Transition`). Ein Übergang kann nur dann existieren, wenn sowohl sein Quell- (`source`) als auch sein Zielzustand (`target`) Bestandteil eines jeweiligen Produkts sind.

5.3 Auflösung von Inkonsistenzen im Mapping-Modell

Nachdem in den vorhergehenden Abschnitten die referenzierten Modelle sowie eine Menge von SDIRL-Abhängigkeitsdefinitionen vorgestellt wurden, kann die Erzeugung eines Mapping-Modells in Angriff genommen werden. Die Initialisierung erfolgt über den entsprechenden Wizard (vgl. Abschnitt 4.10.5).

Der Vorgang der Annotation einzelner Mappings wird an dieser Stelle nicht weiter aus Anwendersicht betrachtet. Stattdessen werden im Folgenden *repräsentative Konstellationen* innerhalb des MM diskutiert, die zunächst eine Konsistenzverletzung oder Informationsverlust implizieren. Diese Teile des Mapping-Modells werden schließlich aufgegriffen, um die Anwendung der erarbeiteten Konzepte zur Wiederherstellung der Konsistenz – etwa der *Propagationsstrategien* – zu untersuchen.

[38] Nicht existierende Typen sind in UML erlaubt und entsprechen dem Java-Datentyp `void`.

5.3.1 Unvollständig annotierte Pakethierarchie

In diesem Teil des Beispiels soll zunächst davon ausgegangen werden, dass keine Propagationsstrategie zum Einsatz kommt (primäre und sekundäre **PS** sind im Editor auf *ignore* gesetzt). `HAS1` sei die geladene Featurekonfiguration. Abbildung 5.7 zeigt eine Konsistenzverletzung durch einen Verstoß gegen die in Abschnitt 3.3.2 definierte Containment-Abhängigkeit. Eine Ableitung von wohlgeformten Produkten aus dem vorliegenden Mapping wäre nicht möglich, da der `eContainer` des Pakets `remote` nicht vorhanden ist.

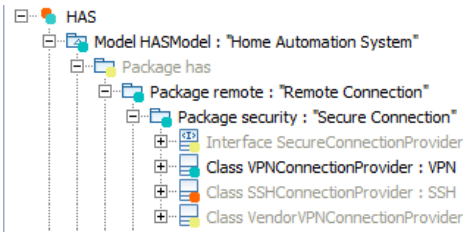

Abbildung 5.7: Konsistenzverletzung: Die unvollständige Annotation der Pakethierarchie führt zur Abhängigkeit des selektierten Pakets `remote` von dessen nicht selektierten Eltern-Paket `has`.

Ein Einsatz einer der in Abschnitt 4.7.4 vorgestellten sekundären Propagationsstrategien bewirkt die Wiederherstellung der Konsistenz, wie sie in Abbildung 5.8 dargestellt ist: Die Ermittlung des Selektionszustands des nicht annotierten Pakets `has` erfolgt im Fall der Vorwärtspropagation durch das übergeordnete, positiv annotierte Element `HASModel`, von dem `has` abhängt. Im Falle der Rückwärtspropagation wird der Zustand von `has` durch Elemente, die wiederum von ihm abhängen, bestimmt: Das unmittelbar untergeordnete Paket `remote`, ebenfalls positiv annotiert, führt zu einer künstlichen Positivierung des nicht annotierten Elements durch den Selektionszustand *enforced*. In beiden Fällen zeigt der dargestellte Ausschnitt nun eine konsistente Pakethierarchie.

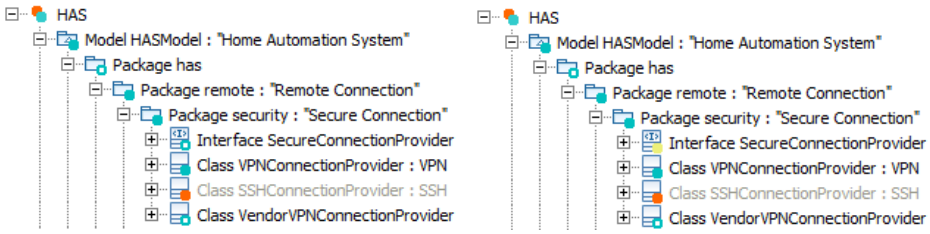

Abbildung 5.8: Wiederherstellung der Konsistenz durch die Anwendung der Propagationsstrategien *forward* (links) bzw. *reverse* (rechts).

5.3.2 Abbildung mehrwertiger Features

Feature-Ausdrücke erlauben das Referenzieren mehrfach instanziierbarer Features durch Angabe eines Index bzw. einer Wildcard, die einer ODER-Verknüpfung sämtlicher Instanzen des Features entspricht (vgl. Abschnitt 4.4.1). Abbildung 5.9 zeigt eine von der Modellvalidierung identifizierte Konsistenzverletzung, die durch das Nichtvorhandensein eines Index bei mehrfach instanziierbaren Features entsteht.

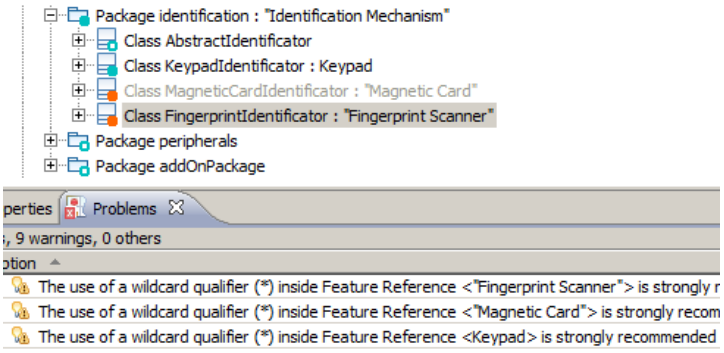

Abbildung 5.9: Konsistenzverletzung: Die mehrfach instanziierbaren Features `Keypad`, `Magnetic Card` und `Fingerprint Scanner` kommen jeweils als Feature-Referenz ohne Index auf entsprechenden DM-Elementen zum Einsatz. Die Validierung empfiehlt den Einsatz eines Wildcard-Symbols.

Die Auflösung dieses Konflikts obliegt dem Anwender: Der Feature-Ausdruck soll so berichtigt werden, dass er sich entweder im Sinne der *subjekt-* oder der *objektbezogenen Manifestation* des Variabilitätsmerkmals *Multiplizität* an das Featuremodell bzw. die geladene Featurekonfiguration richtet. Abbildung 5.10 beschreibt die Herstellung der Beziehung zum *Subjekt* – also dem im **FM** identifizierten variierbaren Merkmal – durch den Einsatz des *Wildcard*-Symbols.

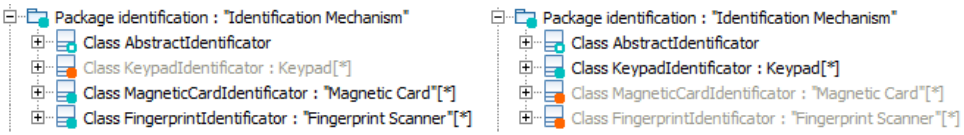

Abbildung 5.10: Wiederherstellung der Konsistenz: Durch Verwendung des vorgeschlagenen Wildcard-Symbols beziehen sich die betroffenen Feature-Ausdrücke nun auf alle Instanzen eines mehrwertigen Features. Der linke Ausschnitt bezieht sich auf die Featurekonfiguration `HAS1`, der rechte auf `HAS2`.

5.3.3 Abhängigkeiten von Zuständen und Transitionen

Bei der Einführung des Featuremodells (vgl. Abschnitt 5.1.1) sowie des Multivarianten-Domänenmodells (5.1.3) wurde bereits das Zustandsdiagramm für die Klasse `MicrowaveOven` sowie dessen zugehörige Features `Microwave Oven Control` bzw. `Cooldown Mode` hervorgehoben. Nun soll gezeigt werden, wie sich durch lediglich zwei Annotationen eine entsprechende Abbildung im Mapping-Modell unter Berücksichtigung der Forderung des Konsistenzerhalts realisieren lassen. Für die **FK** `HAS1` entfällt das Feature `Cooldown Mode`, während es in `HAS2` realisiert werden soll. Abbildung 5.11 zeigt die Baumrepräsentation des Zustandsdiagramms im Mapping-Modell, zunächst ohne Anwendung einer Propagationsstrategie: Der zusätzliche Zustand `On_Cooldown` ist mit dem entsprechenden Feature annotiert. Die direkte Transition `stopOven` zwischen den Zuständen `On_Heating` und `Off_Closed` soll bei der Integration dieses Features entfallen und ist entsprechend mit `not "Cooldown Mode"` annotiert.

Die im referenzierten **SDIRL**-Dokument (vgl. Abschnitt 5.2) formulierte Abhängigkeitsbedingung `TransitionState` bewirkt das Entfallen einer Kante, die einen nicht existierenden Zustand referenziert, sobald eine Propagationsstrategie im Mapping-Modell

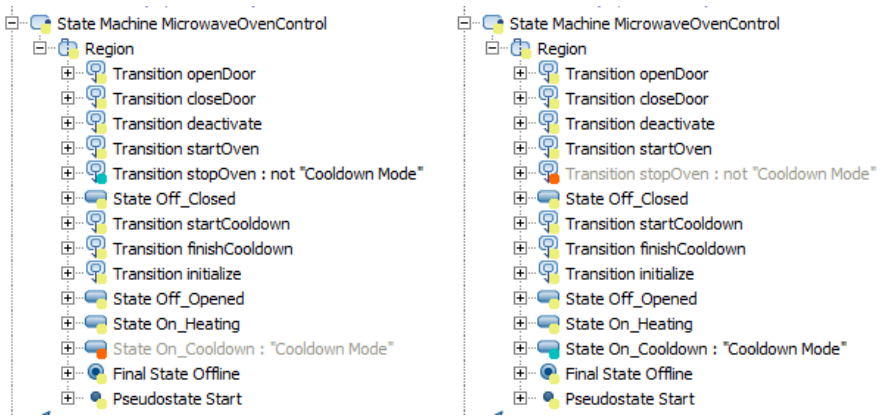

Abbildung 5.11: Konsistenzverletzung: Ein- bzw. ausgehende Transitionen des Zustands `On_Cooldown` hätten in einem durch `HAS1` (links) beschriebenen Produkt keinen Quell- bzw. Zielzustand. Die Annotation `HAS2` (rechts) wäre in diesem Fall auch ohne Anwendung einer Propagationsstrategie konsistent.

eingesetzt wird. Entsprechend wird beim Wiederherstellen der Konsistenz durch den Einsatz der Propagationsstrategie *forward* (s. Abbildung 5.12) der identifizierte Abhängigkeitskonflikt in `HAS1` durch automatische Negativierung des Selektionszustands der ein- bzw. ausgehenden Transitionen `startCooldown` und `finishCooldown` aufgelöst.

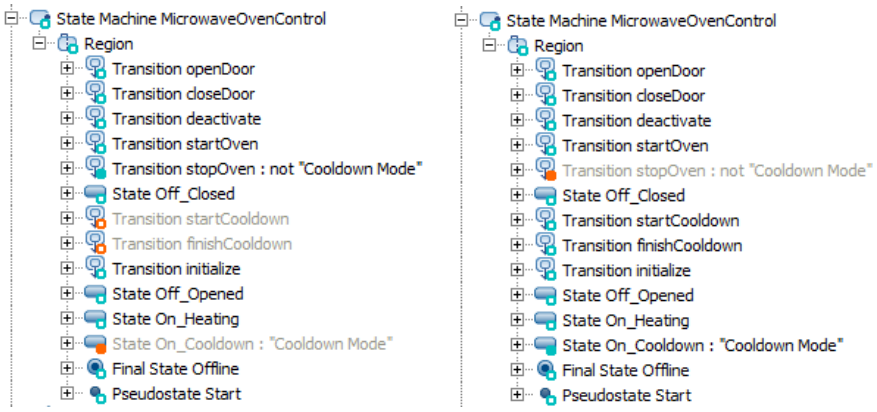

Abbildung 5.12: Wiederherstellung der Konsistenz: Die primäre Propagationsstrategie *forward* erzwingt jeweils den Selektionszustand *suppressed* für die Transitionen `startCooldown` und `finishCooldown` in `HAS1` (links). Als sekundäre Strategie wurde ebenfalls die Vorwärtspropagation eingesetzt, um nicht annotierte Elemente, die existenziell von ihrer übergeordneten `Region` abhängen, zu positivieren (*enforced*).

5.3.4 Attribut-abhängige Kardinalität einer Assoziation

Eine weitere für die Evaluierung des Ansatzes interessante, repräsentative Konstellation im Beispiel-Mapping-Modell ist die Klasse `addOnPackage`. In Abbildung 5.5 wurde die Kompositions-Assoziation `addOns` zwischen `AddOnRegistry` und `AddOn` eingeführt und bereits darauf hingewiesen, dass die Kardinalität des von `AddOnRegistry` ausgehenden Assoziationsendes abhängig von der Multiplizität des Features `Add-on Package` sein soll.

Um dies zu realisieren, wird im Beispiel das *implizite abgeleitete Attribut* `cardinality` in einen *Attribut-Ausdruck* (vgl. Abschnitt 4.4.3) eines Alternativen-Mappings (vgl. Abschnitt 4.6.2) für das Attribut `value` der die Obergrenze der Assoziation definierenden `ValueSpecification` eingebettet. Der Zusammenhang mit den um einwertige strukturelle Eigenschaften *konkurrierenden* Mappings (vgl. Abschnitt 4.7.5) wird in Abbildung 5.13 deutlich: Das Kern-Mapping für das Attribut `value` (s. Mapping-Properties-Ansicht) wurde negativ annotiert (`false`), um einem manuell erzeugten, mit dem entsprechenden Pattern-Ausdruck `#{"Add-on Package"::cardinality}` versehenen Alternativen-Mapping den Vorrang zu geben. Für die Konfiguration `HAS1` wertet der Attribut-Ausdruck zu 2 aus, da für sie zwei Instanzen des Merkmals `Add-on Package` existieren.

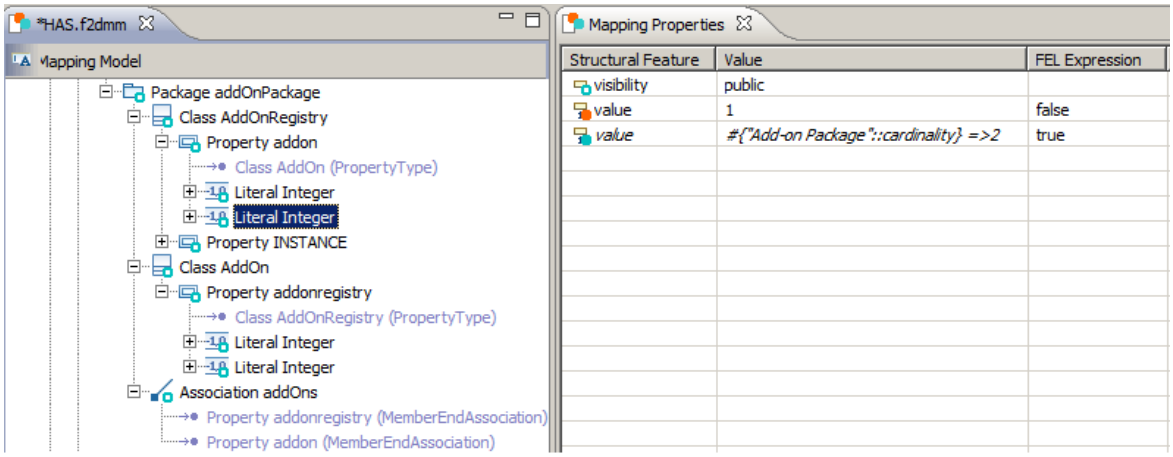

Abbildung 5.13: Einbettung des Attribut-Ausdrucks `"Add-on Package"::cardinality` in ein alternatives Attribut-Mapping, um die Kardinalität eines Assoziationsendes abhängig von der Featurekonfiguration `HAS1` zu bestimmen.

Ein Wechsel zur Featurekonfiguration `HAS2` bewirkt eine sofortige Neuauswertung des den Attributwert bestimmenden Pattern-Ausdrucks. Abbildung 5.14 zeigt das Ergebnis der entsprechenden Auswertung, wiederum in der Mapping-Properties-Ansicht: Da in `HAS2` drei Instanzen von `Add-on Package` existieren, evaluiert das Pattern zu 3.

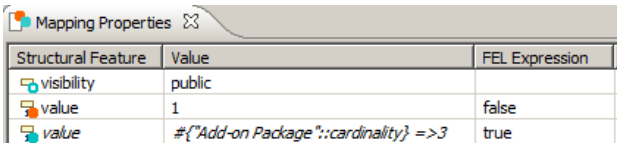

Abbildung 5.14: Abbildung des Feature-Attributs `"Add-on Package"::cardinality` in `HAS2`.

Auf ähnliche Weise wird in abgeleiteten Produkten auch der Name der Klasse `VendorVPNConnectionProvider` (s. Paketdiagramm in Abbildung 5.3) in Abhängigkeit von einem Feature-Attribut gesetzt: Das Präfix `Vendor` wird hierbei durch den im Feature-Attribut `Vendor Protocol` der jeweiligen **FK** gesetzten Wert ersetzt. Der hierfür vorgesehene, explizite Attribut-Ausdruck lautet:

```
1 #{VPN:"Vendor Protocol"}VPNConnectionProvider
```

5.3.5 Unterbrechung einer mehrstufigen Vererbungshierarchie

Das **UML2**-Klassendiagramm in Abbildung 5.4 stellt eine dreistufige Vererbungshierarchie innerhalb des Pakets `has.remote.wifi` des Multivarianten-**DM** dar. Diese soll nun im vorliegenden konstruierten Beispiel ihre Anwendung in aus *Surrogat*-Ausdrücken abgeleiteten *Reparaturaktionen* finden (vgl. Abschnitte 4.5.5 und 4.7.1). Hierbei wird geprüft, inwiefern `AbstractWifiConnector` deren Subklasse `AbstractIEEE802Connector` als Oberklasse der auf untester Ebene angesiedelten konkreten Klassen als Stellvertreter ersetzen kann.

Die Evaluierung soll in diesem Fall unabhängig von der geladenen **FK** (in diesem Beispiel `HAS2`) erfolgen; hierzu wird die mittlere Klasse mit der Konstanten `false` annotiert (vgl. Abbildung 5.15). Der der Abhängigkeitsdefinition `SuperClassifier` zugehörige Surrogat-Ausdruck wird zunächst aus der SDIRL-Deklaration entfernt, um die Auswirkungen seines Fehlens zu demonstrieren: Ohne ihn könnten die konkreten Klassen aufgrund der Abhängigkeitsbedingung `SuperClassifier` nicht existieren — es kommt zum *Informationsverlust* in abgeleiteten Produkten.

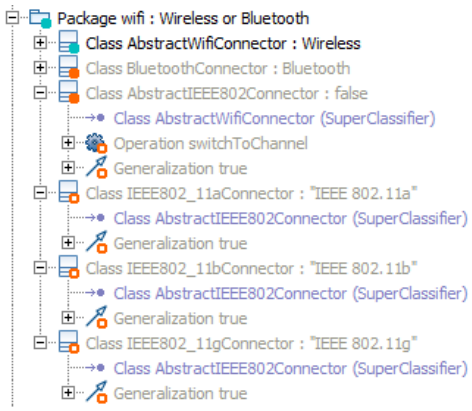

Abbildung 5.15: Informationsverlust durch Unterbrechung einer mehrstufigen Vererbungshierarchie ohne den Einsatz von Surrogaten. Die konkreten Klassen `IEEE802_11XConnector`, $X \in \{a, b, g\}$, werden aufgrund ihrer Abhängigkeit von der negativ annotierten Oberklasse als *suppressed* markiert.

Das gegebene Mapping verletzt zwar keine Konsistenzbedingungen, jedoch kann durch die Anwendung des im **SDIRL**-Dokument (vgl. Abschnitt 5.2) definierten Surrogat-Ausdrucks für die Abhängigkeit `SuperClassifier` die in der Vererbungshierarchie auf oberster Ebene befindliche Klasse `AbstractWifiConnector` als Surrogat fungieren. Abbildung 5.16 zeigt in der Mapping-Properties-Ansicht, dass die Mappings der zuvor aufgrund des identifizierten Abhängigkeitskonflikts negativierten konkreten Klassen in ihren ursprünglichen Selektionszustand *active* zurückversetzt wurden. Zusätzlich weist die Mapping-Properties-Ansicht, die sich auf die selektierte `Generalization`-Beziehung einer konkreten Klasse bezieht, auf die Verfügbarkeit von Surrogaten hin, die die entfallene Klasse `AbstractIEEE802Connector` als Oberklasse (`general`) ersetzen können (vgl. nächster Abschnitt).

5.3.6 Surrogate und Ausschlusskonflikte

Im letzten Absatz von Abschnitt 4.7.5 wurde darauf eingegangen, wie im **F2DMM**-Editor Surrogate und Ausschlusskonflikte ergänzend zusammenspielen können. Der in Abbildung 5.17 abgebildete Screenshot zeigt, wie die Klasse `FingerprintIdentificator`

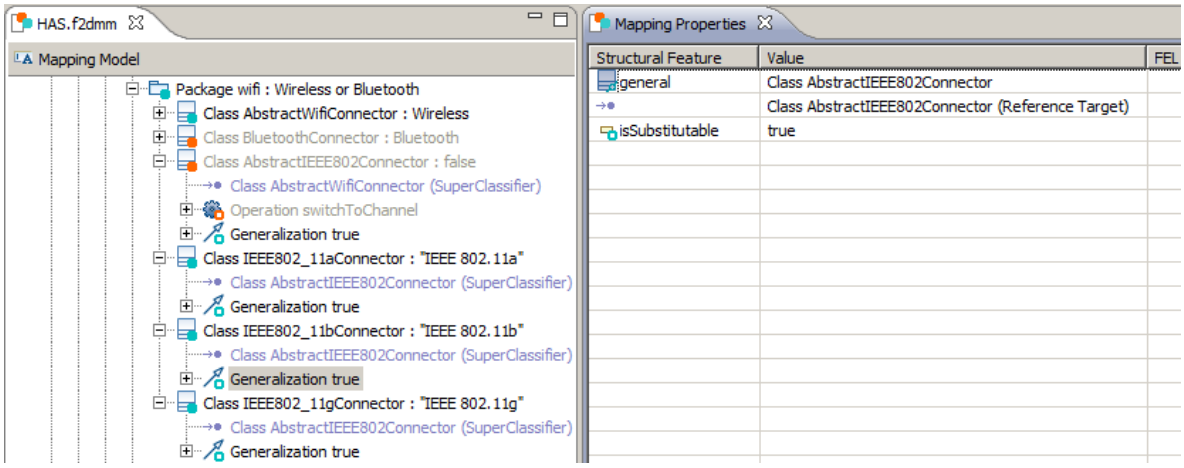

Abbildung 5.16: Reparatur der unterbrochenen Vererbungshierarchie durch die Definition geeigneter *Surrogate* im **SDIRL**-Dokument.

durch explizite Negativierung deren Referenz zur Oberklasse `AbstractIdentificator` (s. Annotation `false` in der Mapping-Properties-Ansicht) ebenfalls von einer Negativierung Vorwärtspropagation betroffen ist: Sie befindet sich zunächst im Selektionszustand *suppressed*.

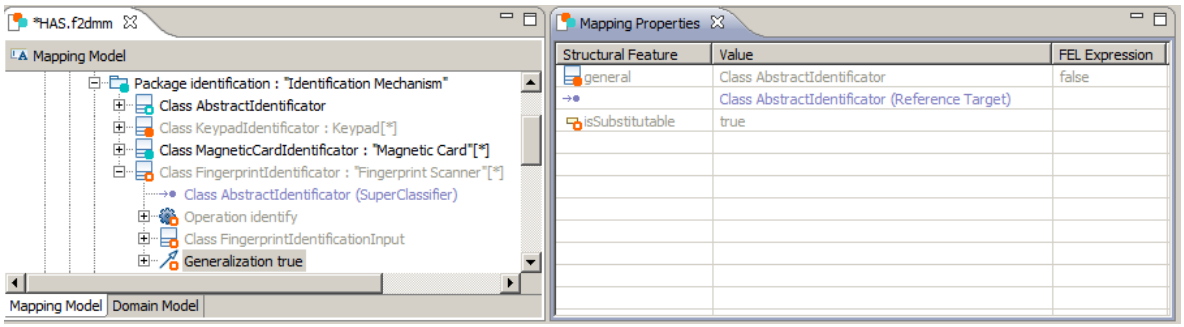

Abbildung 5.17: Informationsverlust: Surrogate und Ausschlusskonflikte im Zusammenspiel.

Um die vollständige Negativierung der Klasse `FingerprintIdentificator` zu verhindern, kann das erwähnte Zusammenspiel zwischen Surrogaten und Ausschlusskonflikten ausgenutzt werden: Abbildung 5.18 bildet das als Alternativen-Mapping manuell eingefügte Referenz-Mapping über `general` zu `MagneticCardIdentificator` ab. Ohne dass ein manuelles Eingreifen nötig wäre, ändert sich der Selektionszustand des ursprünglichen Referenzziels zu *surrogated*: In der Properties-Ansicht wird das durch das eingefügte Alternativen-Referenz-Mapping abgebildete Element `MagneticCardIdentificator` als Surrogat vorgeschlagen. Die `excludes`- und `overrules`-Beziehungen zu `MagneticCardIdentificator` haben sich zuvor durch die Erkennung und Auflösung von Ausschlusskonflikten in Phase 0 (vgl. Abschnitt 4.7.3) ergeben.

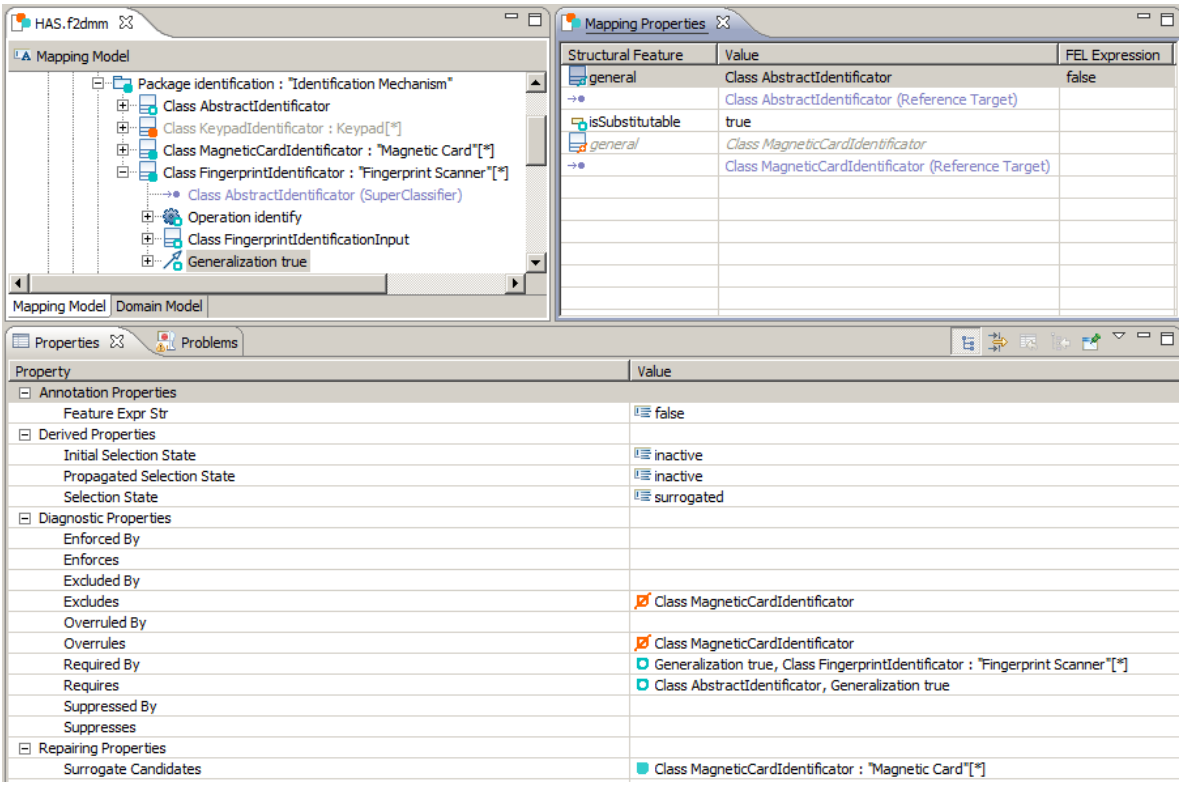

Abbildung 5.18: Reparatur: Surrogate und Ausschlusskonflikte im Zusammenspiel.

5.4 Abgeleitete Produkte

Im Folgenden liegt der Fokus nicht mehr auf dem Mapping-Modell selbst, sondern vielmehr auf den aus ihm durch das Laden unterschiedlicher **FK**s (`HAS1` und `HAS2`) abgeleiteten Produkten, den *konfigurierten Domänenmodellen* (vgl. Abschnitt 3.4). Hierbei soll die Anwendung der in dieser Arbeit vorgestellten Mechanismen zum Konsistenzerhalt an den im vorhergehenden Unterabschnitt identifizierten *repräsentativen Konstellationen* untersucht werden. Zur Visualisierung der unterschiedlichen Diagramme kam wiederum *Valkyrie* [11] zum Einsatz.

5.4.1 Auswahl von Surrogat-Kandidaten

Die Ableitung von Produkten wird in einem geöffneten Mapping-Modell durch den entsprechenden Kontextmenü-Eintrag ermöglicht. Voraussetzungen für die Ableitung konsistenter Produkte sowie die Transformation selbst wurden in Abschnitt 4.8 erläutert. Während der Produktableitung sollen im *interaktiven* Modus vom Anwender gewünschte Surrogate aus einer Kandidaten-Liste gewählt werden. Die gewählten Elemente stellen entsprechende Referenzziele im abgeleiteten Produkt dar. Abbildung 5.19 zeigt den Dialog, der den Benutzer vor die Wahl eines Surrogats für die im Beispiel aus Abschnitt 5.3.5 beschriebene mittlere Klasse `AbstractIEEE802Connector` stellt. In diesem Fall ist deren Oberklasse `AbstractWifiConnector` der einzige verfügbare Kandidat.

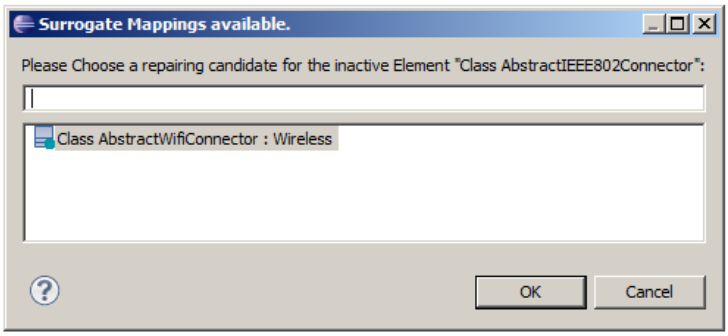

Abbildung 5.19: Auswahl eines Surrogat-Kandidaten während der Produktableitung.

5.4.2 Wiederherstellung der Vererbungshierarchie

Die Auswahl eines Surrogats für die entfallende Klasse hat zur Folge, dass dieses die Rolle des Referenzziels für die Vererbungsbeziehung der konkreten Klassen auf untester Ebene einnimmt. Abbildung 5.20 zeigt die Klassendiagramme des Pakets `has.remote.wifi` der beiden abgeleiteten Produkte. Im Vergleich zum Klassendiagramm des Multivarianten-Domänenmodells (Abbildung 5.4) entfällt in den durch die **FK**s `HAS1` (links) sowie `HAS2` (rechts) beschriebenen Produkte die mittlere Klasse `AbstractIEEE802Connector`. Deren konkrete Unterklassen erben – unter der Voraussetzung, nicht aufgrund eines negativen Selektionszustands selbst zu entfallen – direkt von `AbstractWifiConnector`.

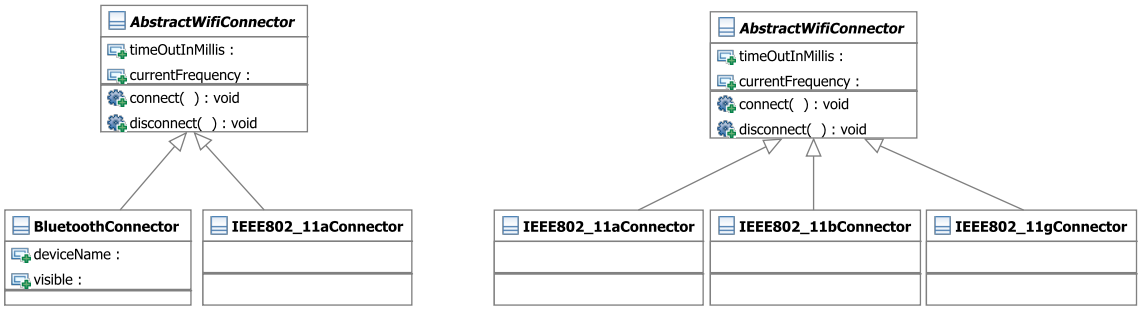

Abbildung 5.20: Klassendiagramm für das Paket `has.remote.wifi` in den durch `HAS1` (links) bzw. `HAS2` (rechts) beschriebenen abgeleiteten Produkten.

5.4.3 Zustandsdiagramme in unterschiedlichen Produkten

In Abschnitt 5.3.3 lag der Fokus auf der Abbildung der Softwaremerkmale `Microwave Oven Control` und `Cooldown Mode`. Letzteres Feature sollte für `HAS2` einen zusätzlichen Zustand `On_Cooldown` definieren, wohingegen die direkte Transition `stopOven` entfallen sollte. Umgekehrt wurde `HAS1` so konfiguriert, dass `On_Cooldown` inklusive seiner ein- und ausgehenden Transitionen entfällt. Abbildung 5.21 visualisiert die entsprechenden Zustandsdiagramme beider Produkte in UML2.

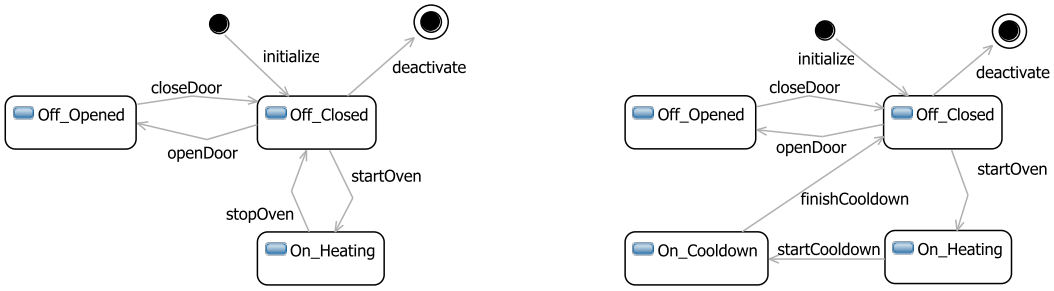

Abbildung 5.21: Zustandsdiagramm für die Klasse `MicrowaveOven` in den durch `HAS1` (links) bzw. `HAS2` (rechts) beschriebenen, abgeleiteten konfigurierten Domänenmodellen.

5.4.4 Abbildung von Attributen: Kardinalitäten und Klassennamen

Die Assoziation `addOns` zwischen den Klassen `AddOnRegistry` und `AddOn` wurde in Abschnitt 5.3.4 hinsichtlich der Abbildbarkeit auf die Multiplizität des Features `Add-on Package` untersucht. Die Anwendung eines Alternativen-Attribut-Mappings führt schließlich zu unterschiedlichen Kardinalitäten des Assoziationsendes in den abgeleiteten konfigurierten Domänenmodellen (s. Abbildung 5.22).

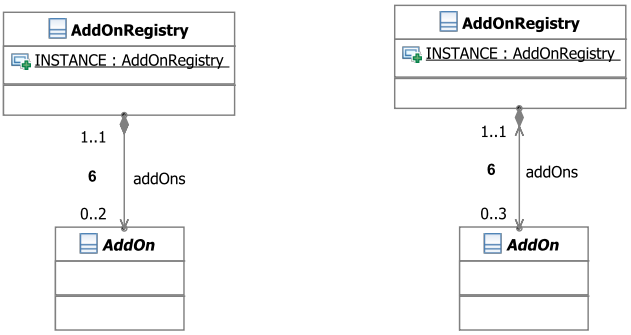

Abbildung 5.22: Klassendiagramm des Multivarianten-Domänenmodells (Abbildung 5.5) mit an die entsprechenden Featurekonfigurationen angepassten Kardinalitäten: Die obere Grenze des Assoziationsendes von `addOns` auf der Seite von `AddOnRegistry` wird durch die jeweilige Anzahl der Instanzen des Features (2 in `HAS1`, links, sowie 3 in `HAS2`, rechts) bestimmt.

Weiterhin wurde die Abbildung des im Featuremodell identifizierten Attributs `Vendor Protocol` des Merkmals `VPN` beschrieben. Das Feature entfällt in der Konfiguration `HAS1`, sodass lediglich das durch `HAS2` repräsentierte Produkt von der Manipulation dieses Attributs betroffen ist: Abbildung 5.23 zeigt, dass der Klassenname wie erwartet zu `CiscoVPNConnectionProvider` geändert wurde.

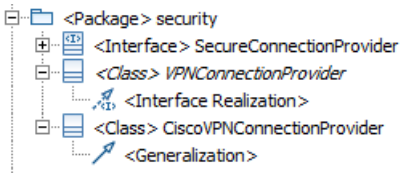

Abbildung 5.23: Das Paket `has.remote.security` des UML2-Modells des durch `HAS2` beschriebenen Produkts in Baumrepräsentation.

5.5 Ausblick: Evaluierung in einer Fallstudie mit der MOD2-SCM-Produktlinie

In diesem Abschnitt wurde ein konstruiertes Beispiel vorgestellt, in dem möglichst viele der in den Abschnitten 3 und 4 vorgestellten Konzepte zum Erhalt der Konsistenz von Softwareproduktlinien zum Einsatz kommen sollten, um deren Funktionsweise in der Praxis zu demonstrieren und aus Konzeptsicht zu evaluieren. Aus den erzeugten Produkten soll in diesem Fall kein Quellcode generiert werden. Auch die Größe des vorgestellten Beispiels steht wurde bewusst beschränkt, um nicht durch Spezifika des Domänenmodells von der Funktionalität des **F2DMM**-Mapping-Editors abzulenken. Jedoch ist zu weiteren Untersuchungen bezüglich Benutzerfreundlichkeit, Übersichtlichkeit und Performanz des **F2DMM**-Editors eine Fallstudie größeren Umfangs auf lange Sicht unumgänglich.

Dotor [17] stellt in seiner Dissertation den Entwurf einer Produktlinie für Softwarekonfigurations-Management-Systeme (**SKMS**) vor[39]. Zur Modellierung von Softwaremerkmalen kam hierzu im Rahmen einer Domänenanalyse das Werkzeug *Feature-Plugin* [3] zum Einsatz. Im Rahmen eines Praktikums wurde des Featuremodell bereits in das **FAMILE**-interne Format übersetzt [39]. Die Modellierung des **UML**-basierten Domänenmodells erfolgte im **MDSE**-Rahmenwerk *Fujaba*. Durch Modelltransformationen (s. Abschnitt 2.7) konnte eine auf dem Eclipse-**UML2**-Metamodell basierende Instanz des **MOD2-SCM**-Domänenmodells gewonnen werden. Die Abbildung des Feature- auf das Domänenmodell erfolgte durch Werkzeuge des durch Buchmann [10] beschriebenen, lehrstuhleigenen Projekts **MODPL**, welche in Abschnitt 6.3.1 zum Vergleich mit dem in dieser Arbeit beschriebenen Ansatz aufgegriffen werden.

Gegenstand einer weiteren Evaluierung könnte das Nachbilden der **MOD2-SCM**-Produktlinie mit den hier beschriebenen Werkzeugen sein. Für das Rahmenwerk könnte dadurch wertvolle Rückmeldung zur Verbesserung des Werkzeugs gewonnen werden. Aus entwurfstechnischer Sicht wäre interessant zu untersuchen, inwieweit sich Zusammenhänge zwischen Feature- und Domänenmodell durch die in dieser Thesis ausgearbeiteten Möglichkeiten der *Manifestation der Variabilität* (vgl. Abschnitt 3.2) auf geeignetere Weise beschreiben ließen.

Zur weiteren Evaluierung der *Synchronisationsmechanismen* (vgl. Abschnitte 4.2.4 und 4.9.1) müsste hingegen ein neues Szenario erarbeitet werden, in dem Feature- und Domänenmodell sowie die durch das Mapping-Modell beschriebene Abbildung miteinander wachsen und somit der *Koevolution* unterworfen wären.

[39] Die Bezeichnung der Produktlinie **MOD2-SCM** steht für „Modular and Model driven Software Configuration Management“.

6 Abgrenzung zu verwandten MDPLE-Ansätzen

In diesem Abschnitt werden verwandte Ansätze zur modellgetriebenen Entwicklung von Softwareproduktlinien mit entsprechenden Literaturverweisen vorgestellt. Zunächst erfolgt die Darstellung nach steigendem Abstraktionsgrad: Angefangen von Methoden des *bedingten Übersetzens*, wie sie von *Präprozessor-Makros* etwa aus der Programmiersprache *C* bekannt sind und keine Trennung zwischen Domänen- und Anwendungsmodellierung vorsehen (vgl. Abschnitt 6.1), werden deren modellgetriebene Entsprechungen in der Modellierungssprache **UML** vorgestellt (vgl. Abschnitt 6.2). Vertreter dieser Kategorie sind die Ansätze **PLUS** und **PLiBS**.

Anschließend wird das **F2DMM**-Vorgängerprojekt **MODPL** als Repräsentant *impliziter* Abbildung von Features auf das Domänenmodell vorgestellt (vgl. Abschnitt 6.3): Die Trennung zwischen Domänen- und Anwendungsmodellierung wird hier unterstützt, jedoch findet die Abbildung innerhalb des Domänenmodells statt. *Explizite* Mapping-Ansätze (s. Abschnitt 6.4 sehen hingegen ein dediziertes Mapping-Modell in einer eigenen Ressource vor und unterstützen zumeist beliebige Modelle, unabhängig von deren Metamodell — das in dieser Arbeit vorgestellte Werkzeug fällt in diese Kategorie.

Zum Abschluss des *Related-Work*-Abschnitts wird auf den Unterschied zwischen *positiver* und *negativer Variabilität* eingegangen. Alle bis dorthin vorgestellten Ansätze fallen in die zweite Kategorie; positive Variabilität (s. Abschnitt 6.5) lässt sich etwa mit Hilfe von *Modell-zu-Modell-Transformationen* (**M2M**) beschreiben. Als Repräsentanten werden die Ansätze **VML*** und **MATA** vorgestellt.

6.1 Vom bedingten Übersetzen zu programmiersprachenzentrierten Ansätzen

6.1.1 Präprozessor-Makros

Die Programmiersprache *C* unterstützt sog. *Präprozessor-Makros*[40], die vor dem eigentlichen Übersetzungsvorgang auf dem Quelltext bearbeitet werden. Die `#ifdef`-Anweisung wird kommt dabei häufig bei der Unterscheidung von Spezifika der Zielplattform zum Einsatz: Sie erlaubt, bestimmte Teile des Codes als abhängig von im Vorfeld definierten *Symbolen* zu deklarieren. Quelltext 6.1 listet ein Beispiel für die Verwendung solcher Makros.

```
1 int main(int argc, char** argv) {
2     f_all_platforms(); // plattformunabhängiger Code
3     #ifdef PLATFORM_A
4     f_platform_a() // spezifisch für Plattform A
5     #endif
6     #ifdef PLATFORM_B
7     f_platform_b() // spezifisch für Plattform B
8     #endif
9 }
```

Quelltext 6.1: Beispiel für den Einsatz von C-Präprozessor-Makros zum bedingten Übersetzen von Quelltext.

Die durch Kompilierung auf unterschiedliche Zielplattformen entstehenden binären Artefakte lassen sich als Mitglieder einer Produktfamilie, die durch den gemeinsamen Quelltext beschrieben wird, auffassen. Jedoch fehlt im Vergleich zu „echten“ Produktlinien-Ansätzen die

[40] `http://gcc.gnu.org/onlinedocs/cpp/Macros.html`

Unterstützung für den Erhalt der *syntaktischen Korrektheit*: Ein vorhandenes *C*-Programm darf nicht beliebig mit `#ifdef`-Anweisungen angereichert werden. Insbesondere bei der Verschachtelung dieser Konstrukte wird es zunehmend schwieriger, die Korrektheit aller Produkte und somit die Konsistenz der „Produktlinie“ sicherzustellen.

6.1.2 CIDE

Die von Kästner [36] als „Präprozessor 2.0“ vorgestellte Produktlinien-Umgebung **CIDE** (*Colored Integrated Development Environment*)[41] spezialisiert sich im Gegensatz zu modellzentrierten Ansätzen auf die Annotation von Quelltexten mit Softwaremerkmalen. **CIDE** beruht auf dem aus der *aspektorientierten Programmierung* [55] bekannten Prinzip der *Trennung von Interessen* bzw. *Belangen* (vgl. engl *separation of concerns*): Funktionalitäten wie Ablauflogik, Logging, Fehlerbehandlung oder Dateiverwaltung werden im Quelltext voneinander getrennt. **CIDE** hebt die Trennung von Belangen durch die Verwendung unterschiedlicher Hintergrundfarben im Texteditor hervor.

Die Annotation von Textabschnitten mit Features erlaubt die Erzeugung einer *programmiersprachenzentrierten* Produktlinie: Entfallen bestimmte Belange in einer Featurekonfiguration, werden entsprechende Quelltextfragmente entfernt. Das Werkzeug geht über das Konzept des *bedingten Übersetzens* hinaus und garantiert die Erzeugung *syntaktisch korrekter* Produkte unter der Voraussetzung eines korrekten Ausgangsquelltextes. Die Prüfung der Korrektheit erfolgt in drei Schritten [36, Abschnitt 5.1]:

1. In einer sprachspezifischen Grammatik als *optional* gekennzeichnete Elemente dürfen in durch konkrete Syntax repräsentierten Dokumenten entweder komplett auftreten oder müssen komplett entfallen; ist dies nicht der Fall, sind *Syntaxfehler* in einigen Produkten möglich.

2. Das *angewandte Auftreten* eines in einem annotierten Fragment definierten Typ ist in mit davon unabhängigen Belangen ausgezeichneten Regionen nicht erlaubt, es sei denn, die Belange hängen voneinander ab (*Typfehler*).

3. Die Erkennung *semantischer Fehler* setzt die Definition einer *formalen Spezifikation* voraus. Dies betrifft etwa das Schließen einer Datei, nach dem sie in einem Programm geöffnet wurde.

CIDE ist als Plugin für die Entwicklungsumgebung *Eclipse* verfügbar. Derzeit werden eine Vielzahl gängiger Programmier- bzw. Auszeichnungssprachen wie Java, C oder **XML** durch entsprechende Grammatiken unterstützt. Die XML-Unterstützung erlaubt schließlich die Erzeugung von Produktlinien auf Basis **XMI**-serialisierter Modelle. Jedoch beschränkt sich die Überprüfung der syntaktischen Korrektheit auf die Konformität zum entsprechenden **XML**-Schema; so kann es etwa zu ungültigen Nicht-Containment-Referenzen in einzelnen Produkten kommen. Zur Visualisierung von Features bzw. Belangen im Werkzeug dient eine zugehörige Feature-Diagramm-Ansicht (rechts in Abbildung 6.1). Die Integration der Feature-Modellierungsumgebung *pure::variants*[42] erlaubt eine feingranularere Definition von *Concerns* bzw. Abhängigkeiten zwischen ihnen.

[41] `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/cide/`

[42] Dieses Werkzeug bietet selbst kommerzielle Unterstützung für die Featuremodellierung sowie Abbildung auf Dokumente im Stile von Präprozessor-Anweisungen. Projekt-Website: `http://www.pure-systems.com/pure_variants.49.0.html`

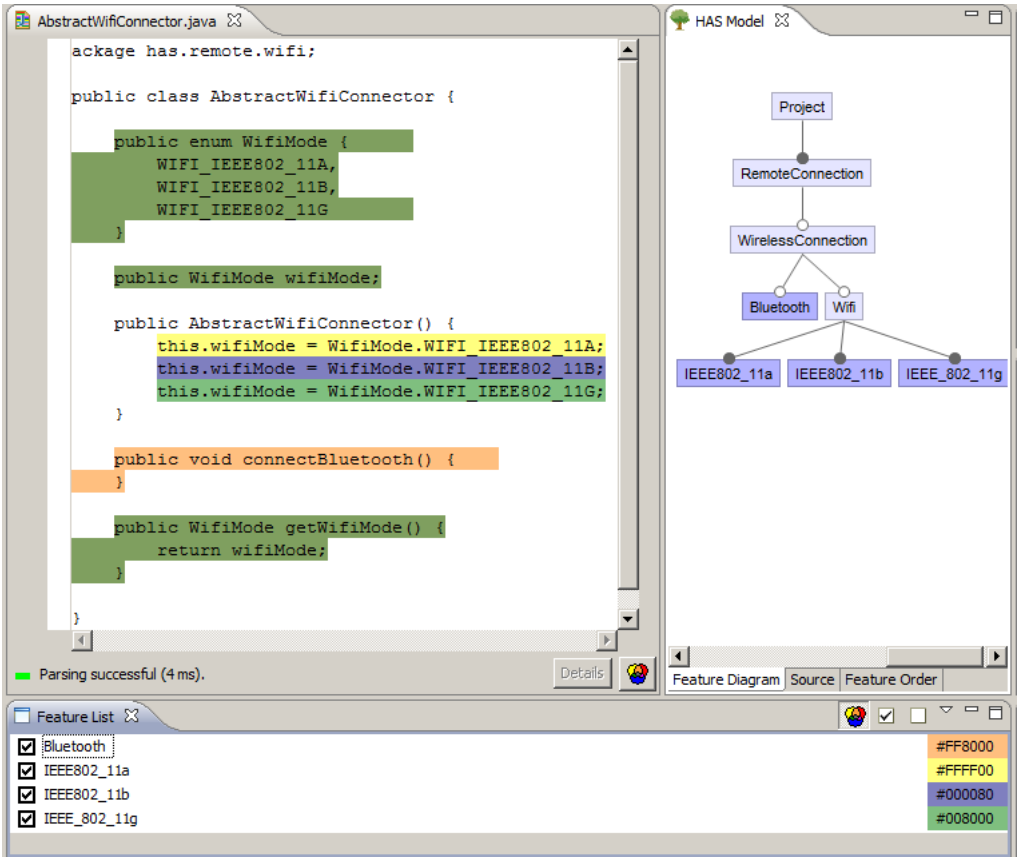

Abbildung 6.1: Beispiel für die Abbildung von Features in der programmiersprachenzentrierten Produktlinien-Entwicklungsumgebung **CIDE**.

Obige Abbildung zeigt den Nachbau eines Teils des im vorhergehenden Abschnitt vorgestellten **HAS**-Beispiels in **CIDE**. Die syntaktische Korrektheit aller ableitbaren Produkte ist sichergestellt.

Ein Vergleich zum in dieser Arbeit vorgestellten Werkzeug ist aufgrund des konzeptionellen Unterschieds zwischen den beiden Ansätzen nur bedingt möglich. Ähnlich wie bei **F2DMM** lassen sich innerhalb einer Grammatik *metamodellspezifische Konsistenzbedingungen* formulieren, die sich jedoch auf optionale Elemente innerhalb des *abstrakten Syntaxbaums* beschränken. **CIDE** fehlt darüber hinaus ein Mechanismus zur automatischen Ableitung von *Reparaturaktionen*; stattdessen wird die Auflösung von Konflikten dem Benutzer überlassen, der Rückmeldung vom Compiler oder Interpreter der jeweiligen Zielsprache erhält. Inwieweit sich **F2DMM** hingegen für die Produktlinien-Modellierung in textuellen Sprachen eignet, wird in Abschnitt 7.4.2 diskutiert.

6.2 Ansätze ohne Trennung der Feature- von der Domänenmodellierung

Alle im Folgenden vorgestellten Methoden und Werkzeuge beziehen sich wieder auf die *modellgetriebene* Entwicklung von Softwareproduktlinien (**MDPLE**). In der Modellierung von Softwaresystemen hat sich die Modellsprache **UML** bzw. **UML2** etabliert, so dass viele Ansätze die Verwendung des entsprechenden Metamodells zur Voraussetzung erheben. Go-

maa [23] und Ziadi und Jézéquel [56] stellen zwei Ansätze vor, um Produktlinien direkt in UML, ohne Verwendung eines beschreibenden Featuremodells, zu modellieren.

6.2.1 PLUS: UML-basierte SPL-Entwicklung

PLUS (*Product Line UML based Software engineering*) erweitert die aus der UML bekannten Modellierungsmethoden für Einzel-Softwaresysteme um Konzepte und Techniken zur Beschreibung von Softwareproduktlinien (**SPL**). Gemeinsamkeiten und Unterschiede werden hierbei im Domänenmodell selbst beschrieben [23, Kapitel 1.8]. Ein Produkt wird demnach durch eine definierte *Sicht* auf die Produktfamilie – repräsentiert durch ein Multivarianten-Domänenmodell – beschrieben. Zur Definition von Sichten werden aus dem objektorientierten Entwurf bekannte Methoden erweitert. Identifizierte Features spiegeln sich in unterschiedlicher Granularität im Domänenmodell wider: Der Schwerpunkt wird auf die *komponentenbasierte* Entwicklung gelegt.

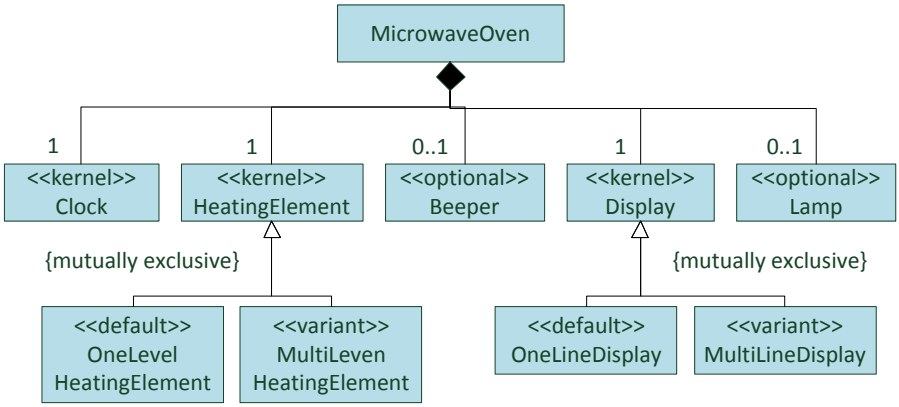

Abbildung 6.2: Beispiel für die Verwendung zusätzlicher, von **PLUS** definierter, Modellierungskonstrukte in einem UML-Klassendiagramm [23, Abbildung 6.3].

Abbildung 6.2 zeigt ein Beispiel eines durch PLUS-Modellierungskonstrukte angereicherten Klassendiagramms [23, Kapitel 6.3]. Für die Annotation von Modell-Elementen einer **PLUS**-basierten Produktlinie sind folgende *Stereotypen*[43] reserviert:

- `<<kernel>>`: Das Element muss in jedem Produkt vorkommen.
- `<<optional>>`: Das Element ist optional und kommt in manchen Produkten vor.
- `<<default>>`: Das Element stellt den Standard-Repräsentanten eines *Variationspunkts* dar.
- `<<variant>>`: Das Element stellt einen alternativen konkreten Repräsentanten eines Variationspunkts dar.

Dem PLUS-Ansatz fehlt die Trennung von Feature- und Domänenmodellierung: Konzepte wie Featuremodell oder Featurekonfigurationen sind nicht vorgesehen. Die Beschreibung eines Produkts ergibt sich stattdessen aus einer Menge von Entscheidungen zur Auflösung der

[43] Stereotypen sind „*sprachinhärente Erweiterungsmechanismen*“, die vorhandene UML-Sprachelemente durch Spezialisierung mit zusätzlicher Funktionalität anreichern [32, Kapitel 5.3].

o.g. Stereotypen. Gomaa [23] beschreibt keine Werkzeugunterstützung zum Festhalten dieser Entscheidungen bzw. zur Ableitung von Produkten aus diesen. Vielmehr stellt PLUS eine formelle Beschreibung dieses Prozesses dar: Für optionale Elemente muss bestimmt werden, ob sie in einem Produkt enthalten sind oder nicht; Variationspunkte müssen durch Angabe des Standard- oder eines der alternativen Repräsentanten eliminiert werden, um *Subjekt* und *Objekt der Variabilität* (vgl. Abschnitt 3.1) aneinander zu binden. Ein gängiges Modell ist die Darstellung des Variationspunktes als abstrakte Klasse und die sich gegenseitig ausschließenden Varianten bzw. Repräsentanten als konkrete Unterklassen.

Eine Gemeinsamkeit mit dem in dieser Arbeit vorgestellten Werkzeug **F2DMM** ist die Art und Weise der Darstellung von Variationspunkten: Diese beinhalten jeweils einen Standardrepräsentanten, der in F2DMM durch ein entsprechendes Kern-Mapping repräsentiert wäre. Alternativen-Mappings haben ihre Entsprechung im **PLUS**-Stereotypen `<<variant>>`.

6.2.2 Der Ansatz von Ziadi und Jézéquel für die statische Modellierung

Ziadi und Jézéquel [56] beschreiben einen Modellierungsansatz, der dem oben beschriebenen PLUS-Ansatz in weiten Teilen ähnelt: In einem **UML2**-Profil sind wiederum Stereotypen definiert, die Multivarianten-Modelle um Informationen zu deren Variabilität ergänzen. Die Autoren beschreiben die *werkzeuggestützte* Ableitung von Produkten aus UML2-Modellen, nämlich Klassendiagrammen (*statischer* Aspekt) und Sequenzdiagrammen (*verhaltensbezogener* Aspekt). Eine weitere Gemeinsamkeit zu PLUS ist das Fehlen einer Trennung zwischen Feature- und Domänenmodellierung.

Ähnlich wie **F2DMM** unterstützt der Ansatz von Ziadi und Jézéquel eine Menge von generischen oder modellspezifischen *Constraints* [56, Abschnitt 15.2.3], die den Prozess der Produktableitung hinsichtlich des Konsistenzerhalts beeinflussen. Ähnlich wie in Abschnitt 4.7 beschrieben erfolgt zunächst die Identifikation von *Abhängigkeitskonflikten*. Jedoch werden keine automatischen Mechanismen zur Auflösung solcher Inkonsistenzen vorgestellt: Die Produktableitung schlägt im Falle eines oder mehrerer Konflikte fehl.

Wie beim **PLUS**-Ansatz muss die im Multivarianten-Domänenmodell festgehaltene Variabilität durch eine Menge von Entscheidungen eliminiert werden. Im Ansatz von Ziadi und Jézéquel [56, Abschnitt 15.2.4] kommt hierzu ein sog. *Entscheidungsmodell* (vgl. engl. *decision model*) zum Einsatz. Es stellt das Pendant zu den in Abschnitt 3.1 eingeführten Featurekonfigurationen dar, ist jedoch ebenfalls als UML-Klassendiagramm realisiert. Das Entscheidungsmodell implementiert das *Abstract-Factory*-Entwurfsmuster [21, Abschnitt 3.1], indem die als Variationspunkt markierten abstrakten Klassen durch die jeweiligen konkreten Klassen, die die gewünschte Variante implementieren, instanziiert werden.

Die Ableitung *konfigurierter Domänenmodelle* erfolgt als Modelltransformation in der **M2M**-Sprache **ATL**[44] (vgl. Abschnitt 2.7.1). Die formulierten Constraints kommen hierbei als Vor- bzw. Nachbedingung zum Einsatz und haben lediglich validierenden Charakter.

[44] In [56] wird in diesem Kontext noch die ehemalige Bezeichnung **MTL** (*Model Transformation Language*) verwendet.

6.2.3 Der PLiBS-Ansatz für die dynamische Modellierung

Ziadi und Jézéquel [56, Abschnitt 15.3] beschreiben zusätzlich die Modellierung verhaltensbezogener Aspekte einer Produktfamilie in **UML2**-Sequenzdiagrammen sowie das Werkzeug **PLiBS** (*Product Line Behavior Synthesis*) [57] in einem weiteren Artikel.

Neben der Optionalität und der Alternativität wird im Kontext der dynamischen Modellierung mit der *Virtualität* ein zusätzliches Variabilitätsmerkmal eingeführt. Mit dem Stereotyp `<<virtual>>` markierte Elemente fungieren als *Platzhalter* und können nach der eigentlichen Transformation durch produktspezifische Artefakte ersetzt werden. Dieses Konzept ist dadurch begründet, dass sich Softwaremerkmale in Sequenzdiagrammen auf wesentlich feinerer Granularität als beispielsweise in Klassendiagrammen niederschlagen können. Das *Entscheidungsmodell* muss um die entsprechenden konkreten Realisierungen von als virtuell gekennzeichneten Regionen ergänzt werden (vgl. Abbildung 6.3). Während der Transformation erfolgt schließlich die *Synthese* der durch die Modelltransformation gewonnenen Produkte mit den die Virtualität eliminierenden Artefakten.

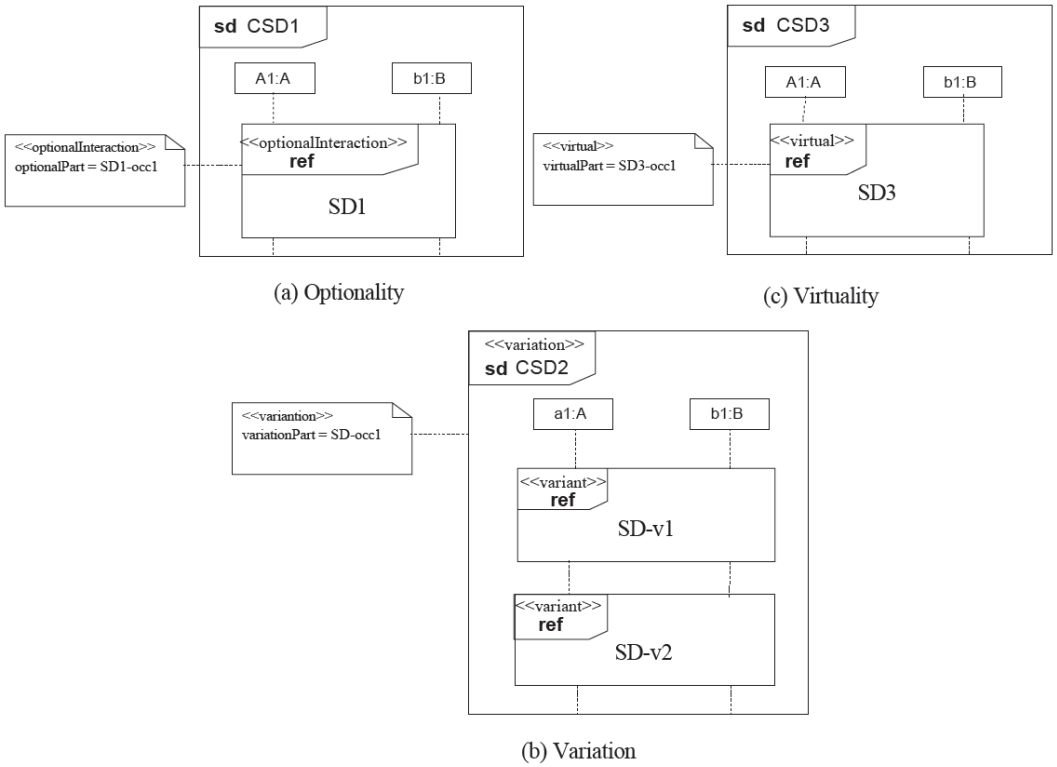

Abbildung 6.3: Die Variabilitätsmerkmale *Optionalität*, *Variation* und *Virtualität* in **PLiBS** [56, Abbildung 15.8].

Das Konzept der Virtualität bzw. der Synthese ähnelt in gewisser Weise den in **SDIRL** formulierbaren *Alternativen* (vgl. Abschnitt 4.6.2): Eine negative Annotation eines Kern-Mappings entspricht der Deklaration `<<virtual>>`. Durch Angabe eines auf die selbe Referenz oder das selbe Attribut passenden, positiv annotierten Alternativen-Mappings kann eine konkrete Realisierung eingebunden und die Virtualität schließlich aufgelöst werden. Während der Transformation erfolgt schließlich die Synthese.

6.3 Implizite Abbildung von Features

Die in diesem und im nachfolgenden Abschnitt vorgestellten Ansätze sehen im Gegensatz zu den soeben beschriebenen Methoden und Werkzeugen nicht nur eine konzeptionelle, sondern darüber hinaus eine physische Trennung zwischen Feature- und Domänenmodellierung gemäß der von Pohl u. a. [45] postulierten *orthogonalen Variabilität* vor: Die Variabilität einer Produktlinie wird in einem Featuremodell (**FM**) festgehalten; Die Ausprägung identifizierter Merkmale in einzelnen Produkten ist Inhalt jeweils einer Featurekonfiguration (**FK**).

6.3.1 MODPL

Buchmann [10, Kapitel 6] beschreibt das am Lehrstuhl im Rahmen des Projekts **MODPL** entwickelte Werkzeug *MODPLFeaturePlugin*. Es erlaubt die Abbildung von Elementen eines Domänenmodells auf durch *FeaturePlugin* [3] beschriebene Merkmalsmodelle. Im Unterschied zum in dieser Arbeit beschriebenen **F2DMM**-Ansatz erfolgt keine Trennung zwischen Domänen- und Mapping-Modell: Sämtliche zur Abbildung nötige Information ist in der Ressource des Domänenmodells enthalten.

Die Modellierungsumgebung *Fujaba* wurde zu diesem Zweck modifiziert: Das zugrundeliegende UML-Metamodell wurde durch den bereits im vorherigen Abschnitt erwähnten *Profil*-Mechanismus erweitert, um einen eigenen *Stereotypen* `VariableElement` zu unterstützen. Dieser erbt von der Basisklasse `Element`, die die Wurzel der Typhierarchie des Fujaba-UML-Metamodells darstellt. Der Stereotyp enthält ein mehrwertiges Attribut (auch: *Tagged Value*) `id` vom Typ `String` (vgl. Abbildung 6.4). Es referenziert die Bezeichner von Features, die auf das entsprechende **DM**-Element gemappt werden.

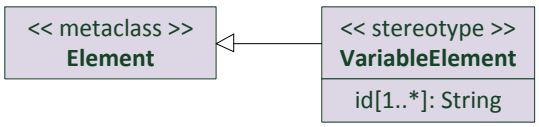

Abbildung 6.4: UML-Profil zur Abbildung von Features in **MODPL** (vgl. [10, Abbildung 6.6]).

Im Folgenden wird auf Gemeinsamkeiten und Unterschiede zwischen den Modellierungsansätzen und Bedienkonzepten von *MODPLFeaturePlugin* und **F2DMM** eingegangen:

Verknüpfungen von Features F2DMM stellt mit der Sprache **FEL** die Möglichkeit der Verknüpfung von Features mit beliebigen booleschen Operatoren zur Verfügung. In MODPL werden mit mehreren Features annotierte Elemente implizit mit einer UND-Konjunktion verknüpft. **FEL** erlaubt außerdem die Formulierung von *Attribut-Constraints*. Die Eindeutigkeit von Feature-Bezeichnern wird in beiden Ansätzen sichergestellt.

Identifikation von Konsistenzverletzungen Im F2DMM-Ansatz werden Konsistenzverletzungen als Verstöße gegen Abhängigkeitsbedingungen identifiziert. Metamodellspezifische Abhängigkeiten werden in der textuellen Sprache **SDIRL** formuliert. MODPL verlangt hingegen, dass die annotierten Features abhängiger Elemente eine echte Teilmenge des Kontext-Elements sind. Die Bedingungen hierzu werden als *Propagationsregeln* in **OCL** formuliert. In beiden Ansätzen werden Konsistenzverletzungen zunächst registriert; die Auflösung erfolgt in einem weiteren Schritt.

Auflösung von Konsistenzverletzungen F2DMM unterstützt die Propagationsstrategien *Vorwärts* bzw. *Rückwärts*, um je einen der Selektionszustände des von einem Konflikt betroffenen **DM**-Element-Paars zu überschreiben. In MODPL werden nicht Selektionszustände, sondern annotierte Features propagiert, um der o.g. Teilmengenforderung gerecht zu werden. Die Propagation selbst ist wiederum als Teil der jeweiligen Propagationsregel in Java formuliert. Die Strategie selbst ähnelt der F2DMM-Propagationsstrategie *Vorwärts*. Anstelle von *Surrogaten* treten im MODPL-Plugin Benutzernotifikationen, etwa um eine unvollständige Vererbungshierarchie wiederherzustellen.

Darstellung von abgeleiteten Produkten Beide Ansätze unterstützen das Ableiten von *konfigurierten Domänenmodellen* bzw. Produkten aus einer Featurekonfiguration und einer validen Abbildung. MODPL erlaubt darüber hinaus die Generierung von Quelltext aus UML-basierten Produkten. Außerdem können in Produkten vorhandene Elemente in konkreter Syntax im Multivarianten-**DM** hervorgehoben werden.

Annahmen für das Domänen-Metamodell **F2DMM** erlegt keinerlei Annahmen über das verwendete Domänen-Metamodell auf, außer dass es auf Ecore basieren muss. Domänenspezifische Konsistenzbedingungen werden in **SDIRL** formuliert. Die MODPL-Werkzeugunterstützung ist für UML2-Paket- sowie Klassendiagramme, basierend auf dem *Fujaba*-UML-Metamodell, vorgesehen. Zudem werden Fujaba-*Storydiagramme* [33] unterstützt, die das Verhalten von UML-Operationen spezifizieren können.

Integration in die Werkzeuge der Domänenmodellierung MODPL zeichnet sich durch enge Verzahnung der Werkzeuge aus: Die Domänenmodellierung selbst sowie die Annotation mit Features erfolgt in einem modifizierten Fujaba-Editor. F2DMM trennt hingegen strikt die Domänenmodellierung von der Abbildung; hierdurch verbietet sich letztendlich die Darstellung von Produktlinien in der konkreten Syntax des Domänenmodells. Derzeit wird lediglich eine Baumansicht unterstützt (vgl. auch Abschnitt 7.4.2).

6.4 Explizite Abbildung von Features: Mapping-Modelle

F2DMM serialisiert die zur Abbildung von Feature- auf Domänenmodell-Elemente notwendige Information innerhalb einer eigenen Ressource, dem *Mapping-Modell*. Dieses Prinzip des *expliziten* Mappings wird auch von anderen **MDPLE**-Ansätzen verfolgt, von denen im Folgenden *Feature Mapper* als konkreter Repräsentant untersucht werden soll.

6.4.1 Feature Mapper

Das von Heidenreich u. a. [30] beschriebene Eclipse-basierte Werkzeug *FeatureMapper*[45] implementiert einen *visuellen* und *interaktiven* Ansatz zur Abbildung von Softwaremerkmalen auf Artefakte des Domänenmodells. Es unterstützt die Abbildung von Features auf **DM**-Elemente in deren konkreter (grafischer oder textueller) Syntax.

Der visuelle Charakter des Ansatzes wird in [29] unter dem Schlagwort „*Controlled Visualization*“ hervorgehoben. Sog. *Mapping Views* erlauben die Repräsentation von Feature-Ausdrücken durch je eine benutzerdefinierte Farbe. Die farblichen Hervorhebungen von **DM**-

[45] `http://featuremapper.org`

Elementen sollen Unterstützung bei der Analyse der Abbildung bieten. Sie integrieren sich in jeden **GMF**-basierten Editor: In Abbildung 6.5 werden Features aus dem Beispiel-Modell aus Abschnitt 5 auf ein Klassendiagramm abgebildet, welches wiederum mit *Valkyrie* [11] modelliert wurde.

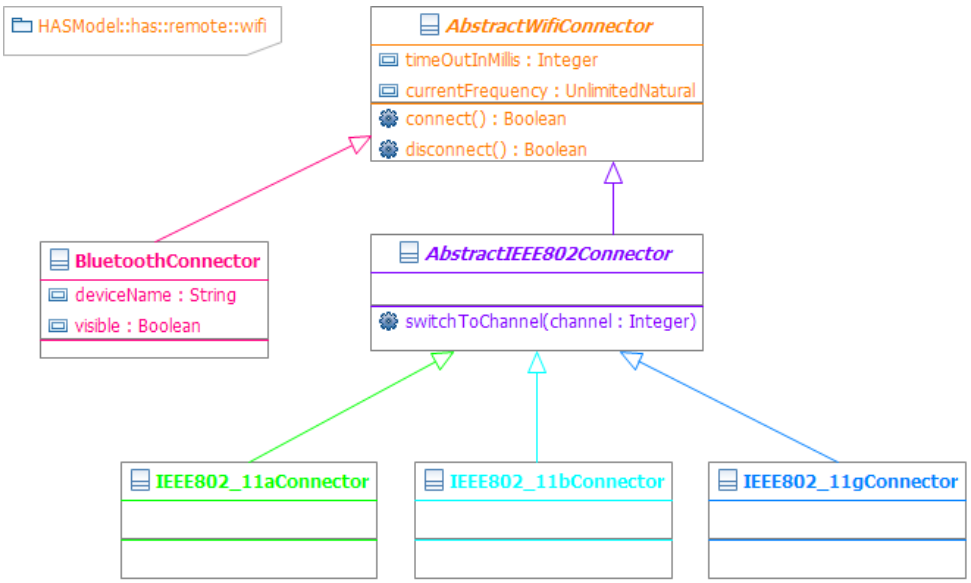

Abbildung 6.5: Farbliche Visualisierung einer Abbildung auf das Beispiel-Domänenmodell mit dem Werkzeug *FeatureMapper*.

Die Abbildung selbst erfolgt in zwei Schritten: Zunächst muss in der `MappingView`-Ansicht (s. Abbildung 6.6) ein *Feature-Ausdruck* erzeugt werden, der auf die anschließend ausgewählten Elemente des **DM** abgebildet wird. Die Auswahl der **DM**-Elemente kann durch Selektion dieser in ihrer grafischen oder textuellen Editor-Repräsentation geschehen (*manueller* Modus). Darüber hinaus erlaubt ein *automatischer* Modus das Mitschneiden von Manipulationen auf dem **DM** nach Betätigung der Record-Schaltfläche (rot ausgefüllter Kreis). Nach Beendigung der „Aufnahme“ werden alle neu erzeugten Elemente mit dem gewählten Feature-Ausdruck annotiert. Hier spiegelt sich der *interaktive* Charakter des Werkzeugs wider.

Im Folgenden sollen einige Aspekte untersucht werden, um das Werkzeug *FeatureMapper* von dem in dieser Arbeit vorgestellten **F2DMM**-Mapping-Editor abzugrenzen:

Featuremodellierung *FeatureMapper* definiert wie F2DMM ein eigenes Metamodell für **FM**- bzw. **FK**-Instanzen, welches jedoch nicht die in Abschnitt 3.1 eingeführte *kardinalitätsbasierte Featuremodellierung* (**CBFM**) unterstützt. Ein Import von mit *pure::variants* erstellten Merkmalsbeschreibungen wird jedoch unterstützt.

Domänenmodellierung Wie auch F2DMM unterstützt *FeatureMapper* sämtliche Ecore-basierten Domänenmodelle, mit der Einschränkung, dass diese **XMI**-serialisiert vorliegen müssen. Durch konkrete textuelle Syntax repräsentierte Modelle werden durch *EMFText*[46] unterstützt. F2DMM unterstützt das Mapping textuell repräsentierter Modelle bisher nur in deren abstrakter Syntax.

[46] Ein Rahmenwerk zur Entwicklung von konkreter Syntax für Ecore-basierte Modelle, ähnlich *Xtext* (vgl. Abschnitt 2.5). Projekt-Website: `http://www.emftext.org/index.php/EMFText`.

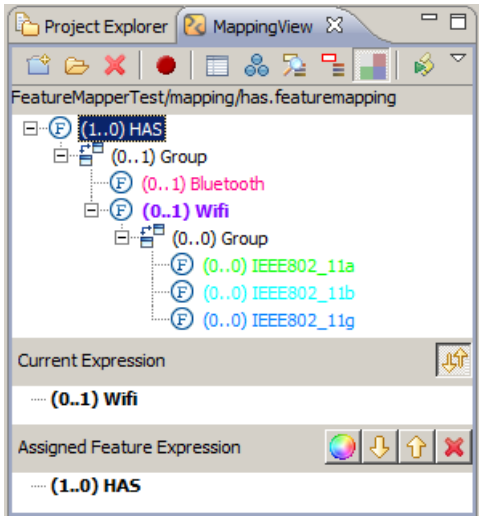

Abbildung 6.6: Die MappingView des Werkzeugs *FeatureMapper* erlaubt die Annotation von **DM**-Elementen mit Feature-Ausdrücken bereits bei deren Erzeugung in einem automatischen sowie nachträglich einem manuellen Modus.

Format und Serialisierung Während F2DMM auf den vom **EMF**-Rahmenwerk bereitgestellten Deserialisierungsmechanismus zurückgreift und lediglich auf der abstrakten Syntax von Domänenmodellen operiert, unterstützt *FeatureMapper* verschiedene *Composer*, die die Spezifika der Modellpersistenz berücksichtigen. Derzeit werden **GMF**-Editoren sowie *EMFText*-Editoren für **EMF**-Modelle von je einem *Composer* unterstützt.

Abbildung Ähnlich wie F2DMM erlaubt *FeatureMapper* die Abbildung durch beliebig verschachtelbare Feature-Ausdrücke. Deren abstrakte Repräsentation wird jeweils in der MappingView angezeigt; es existiert keine textuelle Syntax. Auch Konzepte wie Attribut-Constraints oder Attribut-Mappings fehlen, wodurch die Möglichkeiten der *Manifestation der Variabilität* (vgl. Abschnitt 3.2) im Gegensatz zu F2DMM eingeschränkt sind.

Konsistenzprüfung *FeatureMapper* prüft zwar die bereits erwähnten, von Heidenreich [28] formulierten *Wohlgeformtheitsbedingungen*; diese haben jedoch lediglich validierenden Charakter: Ein wie in dieser Arbeit beschriebenes Konzept zur Ableitung von *Reparaturaktionen*, etwa durch *Propagationsstrategien* oder *Surrogate*, ist nicht vorhanden.

Synchronisation Die Synchronität von Domänen- und Mappingmodell wird in *FeatureMapper* nur sichergestellt, solange der *automatische* Annotationsmodus Manipulationen des Anwenders am **DM** mitschneidet. Ist dies nicht der Fall, werden durch eventuelle Manipulationen ungültig gewordene Mappings erst beim erneuten Laden des Modells von der Modellvalidierung erkannt und dem Anwender als Verletzung der entsprechenden Wohlgeformtheitsbedingung[47] mitgeteilt. Die Auflösung dieser Inkonsistenzen obliegt dem Anwender.

[47] MM-Existing-ModelElement: Von einem Mapping repräsentierte Artefakte des Lösungsraummodells (hier: Domänenmodells) müssen existieren [28, Abschnitt 3.2].

6.5 Modellierung auf Basis positiver Variabilität

Alle bisher erwähnten Ansätze verfolgen die Modellierung von Produktlinien auf Basis *negativer Variabilität*: Es existiert stets eine gemeinsame Basis — sei es ein Quelltextdokument im Falle von programmiersprachenzentrierten Ansätzen (vgl. Abschnitt 6.1) oder ein Multivarianten-Domänenmodell im Falle modellgetriebener Produktlinien-Ansätze. Sie stellt die *Vereinigungsmenge* aller ableitbaren Produkte dar, welche sich durch die Menge *deselektierter* Domänenmodell-Elemente bzw. Textfragmente unterscheiden. Folglich enthält kein Produkt ein Artefakt, welches nicht auch Bestandteil der gemeinsamen Basis ist (vgl. Abbildung 6.7).

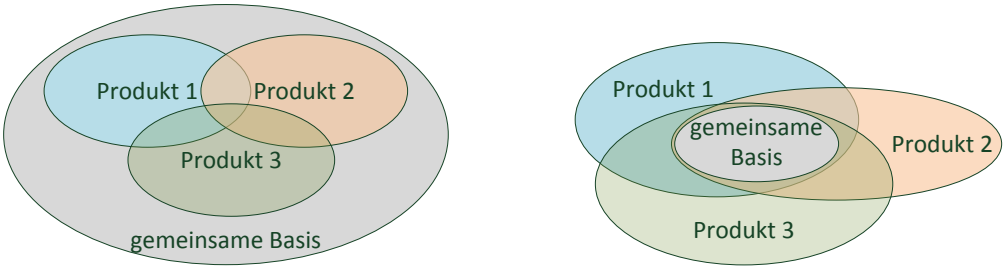

Abbildung 6.7: Negative (links) und positive Variabilität (rechts).

Eine weitere Klasse von Ansätzen setzt im Gegensatz dazu auf die *positive Variabilität* zur Beschreibung von Softwareproduktlinien [27]. Die *gemeinsame Basis* ist hierbei die kleinstmögliche *Schnittmenge* von in allen Mitgliedern der Produktfamilie vorkommenden Artefakten. Folglich enthält jedes Produkt mindestens die gemeinsame Basis und erweitert sie um spezifische Artefakte. Zur Beschreibung modellgetriebener **SPL** auf Basis positiver Variabilität eignen sich vor allem die in Abschnitt 2.7 vorgestellten *Modell-zu-Modell-Transformationen*: Die Erzeugung eines Produkts erfolgt durch Anwendung einer Menge von *erweiternden Operationen* auf der gemeinsamen Basis. Erweiternde Artefakte werden hierzu häufig in separaten Ressourcen, sog. *Slices* (vgl. engl. für „Scheibe“) persistiert und während der Transformation an die Basis gebunden.

6.5.1 MATA

Positive Variabilität wird häufig zur *aspektorientierten Modellierung* (**AOM**) herangezogen: Ähnlich wie in Abschnitt 6.1 beschrieben, werden in Produkten verschiedene Belange (vgl. engl. *concerns*) identifiziert. Der von Whittle u. a. [54] vorgestellte **SPL**-Ansatz **MATA** (*Modeling Aspects using a Transformational Approach*) sieht die Lösung des Problems der Produktableitung in der *Modellkomposition* bzw. *Modellfusion*: Verschiedene Aspekte können mittels Transformationen an ein *Referenzmodell*, welches die gemeinsame Basis repräsentiert, geknüpft werden.

Die aus der *aspektorientierten Programmierung* (**AOP**) [55] bekannten Ansätze der *Joinpoints* und *Advices* kommen auch bei der **AOM** zum Einsatz, um die Trennung von Belangen zu realisieren. Während ein *Joinpoint* in der **AOP** eine Region im Kontrollfluss eines Programms definiert, an welche zur Laufzeit *Advices* gebunden werden, die jeweils aspektspezifisches Verhalten (z.B. Logging, Debugging) implementieren, werden in **AOM** spezielle Modell-Elemente als *Joinpoints* ausgezeichnet. Bereits bei der Komposition von Modellen werden diese mit in einbezogenen Aspekten modellierten *Advices* angereichert.

Die Beschreibung der Advices – also der dem Referenzmodell hinzuzufügenden Artefakte – erfolgt im **MATA**-Ansatz *deklarativ* im Stile sog. *attribuierter Graph-Grammatiken* (**AGG**). Die zur Produktgenerierung verwendete gleichnamige Ausführungsumgebung wird von Taentzer [49] beschrieben. Ähnlich wie die in Abschnitt 2.7.2 vorgestellten Tripel-Graph-Grammatiken beschreiben **AGG**s die Manipulation von Modellen — jedoch als *In-Place*-Transformation: Letztere wird auf dem Referenzmodell selbst durchgeführt, folglich existiert nur eine einzige *Modelldomäne*.

Ähnlich wie **TGG**s beinhalten **AGG**s eine Menge von *Produktionsregeln*. Auf der linken Seite wird ein *Pattern* definiert. Kann dies auf eine Region des zu manipulierenden Modells angewendet werden, tritt die sog. *Kompositionsspezifikation* in Kraft, welche die Manipulation des Modells selbst beschreibt. Diese kann im **MATA**-Ansatz in der konkreten Syntax des Domänenmodells definiert werden: Die Modell-Stereotypen `<<create>>` bzw. `<<delete>>` geben zu erzeugende bzw. zu entfernende Modellartefakte an; die Menge mit `<<context>>` annotierter Elemente einer Regel definieren hingegen deren *Pattern*. Abbildung 6.8 zeigt die Anwendung einer MATA-Regel auf das in der Mitte abgebildete UML-Sequenzdiagramm. Sie beschreibt die Einführung eines neuen *Fragments* `par` sowie zwei neuer *Nachrichten* `q` und `b`, sobald eine mit dem Pattern übereinstimmende Nachricht `p` gefunden wird.

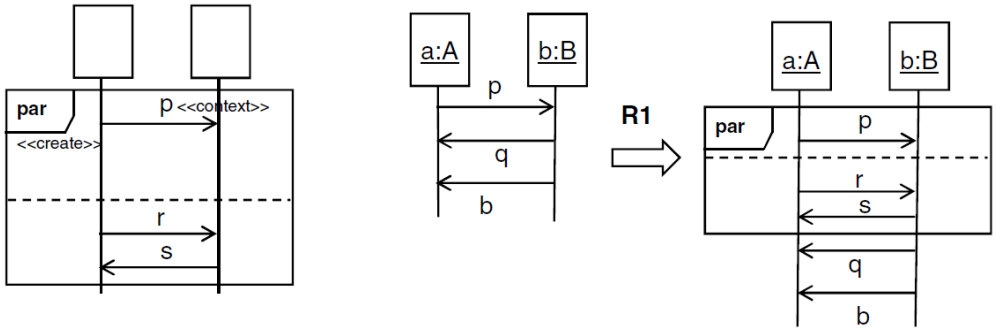

Abbildung 6.8: Eine **MATA**-Regel für UML-Zustandsdiagramme [54, Abbildung 11].

MATA-Regeln werden vor ihrer Anwendung in ausführbare **AGG**-Regeln übersetzt, die die beschriebene Manipulation letztlich durchführen. Auf positiver Variabilität basierende Ansätze kennen jedoch ein Problem: Die Reihenfolge, in der die beschriebenen Transformationen durchgeführt werden, ist entscheidend. Sobald zu einem Zeitpunkt mehrere Regeln anwendbar sind, verläuft die Produktableitung nicht mehr *deterministisch*. Als Lösung schlagen Jayaraman u. a. [34] die Identifikation von Abhängigkeiten und Konflikten zwischen AGG-Regeln durch *Analyse kritischer Paare* (**CPA**, vgl. engl. *critical pair analysis*) vor. Die Bestimmung der endgültigen Reihenfolge paarweise konfligierender Regeln bleibt jedoch dem Anwender überlassen und kann bei entsprechender Größe des Referenzmodells und Anzahl von *Advices* beliebig komplex werden.

MATA bietet bei der Formulierung von Advices bzw. Kompositionsspezifikationen Unterstützung für UML-Klassen-, Zustands- und Sequenzdiagramme. Die zugrundeliegende Ausführungsumgebumg erlaubt jedoch die Erweiterung auf sämtliche Ecore-basierten Metamodelle; die grafische Notation von Regeln muss jeweils entsprechend der konkreten Syntax neu definiert werden.

6.5.2 VML*

Zschaler u. a. [58] beschreiben in ihrem Artikel einen *Bootstrapping*-Ansatz, in dem sie mit **VML***[48] eine Familie von Sprachen zur Beschreibung von Softwareproduktlinien einführen. Die Autoren motivieren ihren Ansatz durch die schlechte Eignung von generischen **M2M**-Ansätzen, wie sie in Abschnitt 2.7 vorgestellt wurden, zur Beschreibung von **SPL** durch positive Variabilität. **VML*** erlaubt die Definition textueller domänenspezifischer Sprachen, durch die eine Produktlinie auf Basis der jeweiligen Domäne, z.B. UML-Klassendiagramme, beschrieben werden kann.

Für die Definition konkreter Sprachen für eine Zieldomäne stellt **VML*** wiederum eine textuelle Sprache zur Verfügung. Durch sie werden sog. *language instance descriptions* formuliert, die verfügbare *Actions* der zu erzeugenden domänenspezifischen Sprache syntaktisch sowie semantisch definieren. Konkrete *Anwendungs-Spezifikationen* sind wiederum Instanzen der generierten **DSL** und verwenden die definierten Actions, um die an ein Feature geknüpfte Manipulation eines Teils des Referenzmodells (in [58] als *Core* bezeichnet) zu beschreiben. In Quelltext 6.2 wird als Beispiel die Spezifikation eines Anwendungsfalldiagramms im **VML***-Dialekt *VML4RE* (VML for Requirements) aufgegriffen.

```
1 import features <"/SmartHome.fmp">;
2 import core <"/SmartHome.uml">;
3
4 variant Security {
5     insertPackage ("Security", "");
6     insertUseCase ("SecureTheHouse", "Security");
7     insertUseCase ("ActivateSecureMode", "Security");
8     createActorToUseCaseLink ("Inhabitant", "Security::.*");
9 }
```

Quelltext 6.2: Teil einer Anwendungs-Spezifikation der Produktlinie „Smart Home“ durch den VML*-Dialekt *VML4RE* [58, Tabelle 4] (gekürzt). Sie bezieht sich auf das Feature `Security`. Die verwendeten Operationen wurden in der entsprechenden *language instance description* definiert.

Im Vergleich zu **MATA** bedarf es bei **VML*** keines Konzepts der Auflösung von durch Abhängigkeiten zwischen Regeln entstehender Konflikte [31]: Der imperative Charakter erfordert die Angabe einer Ausführungsreihenfolge bereits bei der Formulierung einer Anwendungs-Spezifikation.

F2DMM unterstützt die positive Variabilität in eingeschränktem Umfang: Alternativen-Mappings (vgl. Abschnitt 4.6.2) können das Multivarianten-Domänenmodell um *In-Place*-definierte Artefakte erweitern. Referenzen können auf Elemente, die nicht innerhalb der Kern-Ressource liegen, verweisen. Dennoch sind diese Konzepte eher der in Abschnitt 6.2.3 vorgestellten *Virtualität* als der positiven Variabilität zuzuordnen.

[48] Eine Familie von Sprachen zum Variabilitäts-Management, vgl. engl *A Family of Variability Management Languages*.

7 Abschließende Bemerkungen

7.1 Zusammenfassung

In dieser Master-Thesis wurde ein Ansatz zur Mapping-basierten Modellierung von Softwareproduktlinien vorgestellt. Produktlinien erlauben *organisierte Wiederverwendung* von Komponenten, die in der modellgetriebenen Softwareentwicklung durch Artefakte von Modellen repräsentiert werden. Eclipse und EMF haben sich als De-Facto-Standard für Modellierungsplattformen bzw. -rahmenwerke etabliert. Darauf aufbauende Werkzeuge erlauben die Definition von Metamodellen, zugehöriger konkreter Syntax sowie die Validierung von Modellen und Modell-zu-Modell-Transformationen.

Die Beschreibung der strukturellen sowie funktionalen Gemeinsamkeiten und Unterschiede von Mitgliedern einer Softwarefamilie erfolgt in der Feature-orientierten Domänenanalyse [35] durch Featuremodelle. Merkmalsausprägungen einzelner Mitglieder werden jeweils durch eine Featurekonfiguration beschrieben, die Entscheidungen zur Elimination der festgehaltenen Variabilität beinhalten.

Das Werkzeug **F2DMM** implementiert die Konzepte und Methoden des vorgestellten Ansatzes und stellt somit den praktischen Beitrag dieser Arbeit dar. Dem Modellierer wird ein Baumeditor zur Verfügung gestellt, in dem Elemente des Multivarianten-Domänenmodells, das als gemeinsame Basis alle optionalen oder obligatorischen Softwareartefakte enthält, auf identifizierte Softwaremerkmale abgebildet werden können. Zur Abbildung von Features und zur Formulierung weiterer Constraints wird die textuelle Anfragesprache **FEL** zur Verfügung gestellt. Durch die Auswertung von Feature-Ausdrücken ergibt sich für Mapping-Elemente nach dem Laden einer Featurekonfiguration jeweils ein Selektionszustand, der beschreibt, ob das Element in einem entsprechenden Produkt enthalten ist oder nicht. Der Editor beinhaltet eine Vielzahl von Mechanismen, um die Konsistenz zwischen den beteiligten Modellen sicherzustellen; die Sprache **SDIRL** erlaubt dem Modellierer, modellspezifische Abhängigkeitsbedingungen zu formulieren, um neben der strukturellen auch die inhaltliche Konsistenz von erzeugten Produkten zu garantieren. Unter bestimmten Voraussetzungen können inkonsistente Modelle repariert werden: Hierzu dienen unter anderem die vorgestellten Konzepte der Propagationsstrategien und Surrogate.

In Abschnitt 5 wurde ein konstruiertes Beispiel zur Evaluierung des theoretischen Ansatzes sowie der Implementierung herangezogen. Zunächst wurden repräsentative Konstellationen im Mapping-Modell erzeugt, um schließlich die Auflösung der durch sie entstehenden Konflikte durch die vorgestellten Mechanismen zu untersuchen. Im darauf folgenden Abschnitt wurden verwandte Ansätze diskutiert und mit der F2DMM-Implementierung verglichen. Dabei spielen vor allem textorientierte, sowie auf positiver Variabilität basierende Ansätze eine Sonderrolle.

7.2 Rückbezug auf die identifizierten Aspekte

In Abschnitt 1.3 wurde der Beitrag dieser Arbeit unter vier bestimmten Aspekten beschrieben, ohne die zugrundeliegenden Konzepte im Detail diskutiert zu haben. In einer abschließenden Erörterung soll nun ein Rückbezug auf diese stattfinden. Die in der Einleitung aufgestellten Behauptungen werden mit konkreten, in den Abschnitten 3 und 4 eingeführten Konzepten belegt.

7.2.1 Konsistenz

Im beschriebenen Ansatz gilt die Annahme, dass die strukturelle und inhaltliche *Konsistenz* von Produkten garantiert wird, solange das zugrundeliegende Multivarianten-Domänenmodell sowie das Mapping-Modell *wohlgeformt* sind (vgl. Abschnitt 4.8.1). Um zweiteres sicherzustellen, werden in einem Vorberechnungsschritt Abhängigkeiten zwischen Modell-Elementen identifiziert (vgl. Abschnitt 4.7.3). Ecore-Objekte sind immer von ihrem unmittelbaren *Container* abhängig. Zusätzlich wird dem Benutzer in der Sprache **SDIRL** die Möglichkeit zur Formulierung modellspezifischer *Abhängigkeitsbedingungen* gegeben (vgl. Abschnitte 3.3 und 4.5).

Ein *Abhängigkeitskonflikt* besteht, wenn sich vorberechnete Abhängigkeiten und Abhängigkeitsbedingungen widersprechen. *Ausschlusskonflikte* betreffen hingegen Werte bzw. Ziele einwertiger Attribute oder Referenzen, die sich für einen *Variationspunkt* wechselseitig ausschließen. Zur Auflösung dieser Inkonsistenzen dienen *Propagationsstrategien* (vgl. Abschnitt 4.7.4). Sie stellen die lokale Konsistenz eines von einem Konflikt betroffenen Paares von Modell-Elementen wieder her, indem deren Selektionszustände aneinander angepasst werden. *Surrogate* (vgl. Abschnitt 4.5.5) erlauben darüber hinaus die Vermeidung von *Informationsverlust*, welcher Folge etwaiger Anwendungen von Propagationsstrategien sein kann.

7.2.2 Synchronität

Die in dieser Arbeit ausgearbeiteten Methoden der Synchronisierung betreffen die Modellpaare Featuremodell/Featurekonfiguration (vgl. Abschnitt 4.2.4) und Domänenmodell/Mapping-Modell (vgl. Abschnitt 4.9.1). Beide Verfahren laufen vollautomatisch ab: Änderungen im Featuremodell wie das Umbenennen eines Softwaremerkmals werden an entsprechende Featurekonfigurationen propagiert. Auf ähnliche Weise werden Kern-Mappings strukturell synchron mit dem Kern-Domänenmodell gehalten. Einen Sonderfall stellt die Integration der **FM**/**FK**-Synchronisation in den **F2DMM**-Editor dar (vgl. Abschnitt 4.9.2): Abgeänderte Feature-Namen können sich auf annotierte Feature-Ausdrücke von Mappings auswirken. Der Benutzer wird bei der Anpassung dieser Ausdrücke unterstützt, indem ihm *automatische Umbenennungsvorschläge* unterbreitet werden.

In Fällen, in denen die Synchronität zwischen den Modellen nicht automatisch wiederhergestellt werden kann, etwa wenn eine Änderung des Kardinalitäts-Intervalls eines Features betroffene Feature-Ausdrücke ungültig macht, wird der Anwender zumindest bei der Identifikation der Konsistenzverletzung von der *Modellvalidierung* unterstützt.

7.2.3 Agilität

Alternativen-Mappings (vgl. Abschnitt 4.6.2) erlauben die Erzeugung von nicht im Multivarianten-Domänenmodell definierten Attributwerten, Referenzzielen oder durch Containment-Beziehungen ineinander verschachtelter Objekte (z. B. durch *In-Place*-Containment-Mappings). Unter gewissen Einschränkungen lässt sich so *positive Variabilität* (vgl. Abschnitt 6.5) modellieren. Zusätzlich lösen Alternativen-Mappings das Mapping-Modell von der durch das Ecore-Metamodell definierten Einschränkung für einwertige strukturelle Eigenschaften: Die Definition von *Variationspunkten* (vgl. Abschnitt 3.2.3) durch sich gegenseitig ausschließende Alternativen-Mappings ermöglicht die Angabe verschiedener Werte bzw. Ziele für einwertige Attribute bzw. Referenzen, ohne die Wohlgeformtheit abgeleiteter Produkte zu verletzen.

7.2.4 Manifestation der Variabilität

Die Abbildung zwischen Features und Domänenmodell-Artefakten erfolgt meist im Sinne einer m:n-Beziehung: Ist ein bestimmtes Feature in einer Konfiguration vorhanden, sind durch dieses abgebildete Elemente Teil des entsprechenden Produkts. In Abschnitt 3.2 dieser Arbeit wurden Konzepte diskutiert, die über diesen Zusammenhang hinausgehen.

Die notwendigen Ausdrucksmittel stellt die Anfragesprache **FEL** zur Verfügung (vgl. Abschnitt 4.4): Referenzen auf Features mit höherer Multiplizität als eins können sich etwa auf einzelne Instanzen (*Objekte*) oder auf die Gesamtheit aller Instanzen (das *Subjekt*) beziehen; FEL erlaubt entsprechend die Angabe von *Indices* bzw. *Wildcards*. *Variationspunkte* wurden bereits im vorhergehenden Unterabschnitt als Menge sich gegenseitig ausschließender Alternativen für den Wert einer strukturellen Eigenschaft diskutiert. Zusätzlich erlauben *Attribut-Constraints* und *Attribut-Ausdrücke* (vgl. Abschnitte 4.4.2 und 4.4.3) die Manifestation von Werten von Feature-Attributen (vgl. Abschnitt 3.2.5) in Produkten.

7.3 Diskussion

7.3.1 Produktlinien – pro-aktiv oder reaktiv?

Softwareproduktlinien werden häufig als Mittel zur *pro-aktiven* Wiederverwendung von Software charakterisiert. Diese idealisierte Betrachtung impliziert, dass wiederverwendbare Artefakte bereits vor der Entwicklung einzelner Produkte als Teil der Domänenentwicklung (vgl. Abschnitt 3.4) implementiert bzw. modelliert werden und kein Rückschritt nötig oder möglich ist. Für die Modellierung von Produktlinien auf Basis negativer Variabilität würde dies bedeuten, dass die Abbildung bereits auf einem vollständigen Multivarianten-Domänenmodell geschehen könnte und keinerlei Synchronisationsmechanismen notwendig wären.

F2DMM sieht von dieser Annahme ab: Zwar wird zur Erzeugung eines Mapping-Modells ein initiales Multivarianten-Domänenmodell verlangt; jedoch werden Änderungen an diesem durch den **DM/MM**-Synchronisationsmechanismus berücksichtigt (vgl. Abschnitt 4.9.1). Im Gegensatz zu vielen weiteren Featuremodellierungs-Werkzeugen erlaubt der angepasste Featurekonfigurations-Editor sogar eine nachträgliche Synchronisation zwischen **FM** und **FK** (vgl. Abschnitt 4.2.4), wodurch nicht nur die Domänenmodellierung, sondern auch die Featuremodellierung *reaktiv* erfolgen kann.

7.3.2 Generizität bezüglich des Domänen-Metamodells

F2DMM unterstützt die Abbildung von Features auf beliebige Ecore-basierte Modelle. Optional können domänenspezifische Abhängigkeitsbedingungen mit Hilfe der **OCL**-basierten Sprache **SDIRL** auf angemessener Abstraktionsebene formuliert werden.

Über das für die Entwicklung des Domänenmodells verwendete Werkzeug werden ebenfalls keine Annahmen gemacht: Verschiedene Editoren verwenden unterschiedliche konkrete Syntaxa (z.B. grafisch oder textuell), um dasselbe Modell zu repräsentierten. Das Multivarianten-Domänenmodell wird im **F2DMM**-Editor stets in der abstrakten Baumsyntax dargestellt, die sich durch die Ecore-Containment-Hierarchie ergibt und für den „domänenspezifischen Modellierer“ meist nicht die gewohnte Repräsentationsform ist. Dies wurde in diesem Ansatz aber zugunsten der erhöhten Generizität in Kauf genommen (s. auch Abschnitt 7.4.2).

7.3.3 Positive vs. negative Variabilität

In Abschnitt 6.5 wurden die *positive* und die *negative* Variabilität als gegensätzliche Konzepte zur Produktgenerierung voneinander abgegrenzt: Während erstere die Erzeugung von Produkten durch eine Menge von Operationen auf einem Referenzmodell beschreibt, entstehen bei zweiterer Produkte durch das Feature-gesteuerte Entfernen optionaler Modellartefakte.

F2DMM sieht Produktlinien zunächst aus der Perspektive der negativen Variabilität: Mit negativ evaluierenden Feature-Ausdrücken annotierte Elemente eines Multivarianten-Domänenmodells entfallen in durch eine entsprechende Featurekonfiguration beschriebenen Produkten. Alternativen-Mappings erlauben jedoch die Definition von nicht im Multivarianten-**DM** vorkommenden Artefakten für eines oder mehrere Produkte im Sinne der positiven Variabilität. Dieser Bruch mit der strikt negativen Variabilität ist sogar nötig, um Variationspunkte auf einwertigen strukturellen Eigenschaften zu realisieren (vgl. Abschnitt 3.2.3). Inwieweit die durch **F2DMM** zugelassene Mischform der beiden Arten der Variabilität Vorteile oder gar Probleme bei der Modellierung größerer Softwareproduktlinien mit sich bringt, bedarf weiterer Untersuchung.

7.4 Mögliche zukünftige Arbeit

7.4.1 Beschreibung von Teiltransformationen auf höherer Abstraktionsebene

In der vorliegenden Implementierung von **F2DMM** bzw. des Featurekonfigurations-Editors treten drei Modell-zu-Modell-Transformationen auf, die derzeit *imperativ* in Java-Quelltext verfasst sind und in zukünftigen Arbeiten durch die in Abschnitt 2.7 beschriebenen Methoden als **M2M**-Transformation auf einer höheren Abstraktionsebene beschrieben werden könnten:

1. Die Synchronisation zwischen **FM** und **FK** lässt sich als *inkrementelle* Modell-zu-Modell-Transformation auffassen: Das Featuremodell stellt das unveränderliche Quellmodell dar, während eine vorhandene oder eine neu zu erstellende Featurekonfiguration aus der Transformation als Zielmodell resultieren soll. Hierfür würden sich vor allem die in Abschnitt 2.7.2 vorgestellten deklarativen Ansätze *QVT Relations* und *Tripel-Graph-Grammatiken* (**TGG**) eignen, da sie aufgrund ihrer *Check-before-enforce*-Semantik jeweils nur eine Regel für das Binden bzw. Neuerzeugen von Elementen aus dem Zielmodell benötigen.

2. Ähnlich könnte die Synchronisation zwischen **DM** und **MM** durch eine deklarative **M2M**-Vorschrift auf höherer Abstraktionsebene beschrieben werden. Alternativen-Mappings müssten durch geeignete Vorschriften von der Synchronisation ausgeschlossen werden.

3. Die Ableitung des *konfigurierten Domänenmodells* aus einem wohlgeformten Mapping-Modell mit geladener Featurekonfiguration könnte durch einen imperativen **M2M**-Ansatz wie **ATL** (vgl. Abschnitt 2.7.1) beschrieben werden. Deklarative Ansätze kommen hier kaum in Frage: Die Auflösung von Surrogaten sowie die Behandlung von Alternativen-Mappings wäre ohne Seiteneffekte nicht realisierbar. Eine besondere Herausforderung stellt die Generizität bezüglich des Domänen-Metamodells dar: Bei der Formulierung der Transformation sind mögliche Klassen des Zielmodells, also des abzuleitenden Produkts, noch nicht bekannt, was jedoch von allen vorgestellten **M2M**-Werkzeugen gefordert wird.

In frühen Prototypen des **F2DMM**-Editors kam anstatt der textuellen Sprache **FEL** ein auf *TGG Interpreter* (vgl. Abschnitt 2.7.2, letzter Absatz) basierendes Verfahren zur Abbildung von Features auf Artefakte des Domänenmodells zum Einsatz. Aufgrund der angesprochenen mangelnden Unterstützung für Generizität wurde dieses Vorhaben jedoch wieder Fallen gelassen. Der Charakter der Tripel-Graph-Grammatik und der simultane Aufbau nach definierten Regeln findet sich jedoch in der Struktur sowie der Synchronisation des Mapping-Modells wieder.

7.4.2 Tiefere Integration in die Werkzeuge der Domänenmodellierung

Viele der in Abschnitt 6 beschriebenen Werkzeuge integrieren Werkzeuge zur Modellierung von Softwareproduktlinien tief in die Editoren der entsprechenden Domänenmodelle, um dem Modellierer eine möglichst vertraute Umgebung bei der Annotation mit Softwaremerkmalen zu geben. Bezüglich der Darstellung des Domänenmodells in seiner konkreten Syntax mussten jedoch im vorliegenden Ansatz Abstriche zugunsten der Generizität gemacht werden (vgl. Diskussion in Abschnitt 7.3.2).

Eine Unterstützung der konkreten Syntax des jeweiligen Domänenmodells wäre wie folgt möglich: Für jede zu unterstützende Diagrammart bzw. Sprache könnte ein Editor bereitgestellt werden, der das Mapping von Modell-Elementen in deren gewohnter Repräsentation erlaubt. Die von der Benutzerschnittstelle unabhängige Funktionalität des **F2DMM**-Editors, etwa Synchronisationsmechanismen, können von diesen wiederverwendet werden. Kern- und Alternativen-Mappings müssten hierbei optisch voneinander unterscheidbar sein. Der generische, auf der abstrakten Baumsyntax operierende **F2DMM**-Editor würde schließlich als Ausweich-Umgebung für Diagrammarten bzw. Sprachen dienen, für die kein syntaxspezifischer Editor implementiert wurde.

Um dies zu realisieren, müsste geprüft werden, inwieweit sich vorhandene, etwa auf **GMF**[49] basierende grafische bzw. auf *Xtext* (vgl. Abschnitt 2.5) basierende textuelle Editoren komponentenweise wiederverwenden lassen, um durch die Funktionalität des Mapping-Editors ergänzt zu werden. Auf lange Sicht ist die Integration von **F2DMM** in das am Lehrstuhl entwickelte **UML2**-Modellierungswerkzeug *Valkyrie* [11] geplant.

7.4.3 Untersuchung von Produktlinien zur Modellierung von Verhalten

Die durch die modellgetriebene Softwareentwicklung gewonnene Steigerung des Abstraktionsgrades betrifft vor allem die *Strukturmodellierung*: Datenstrukturen können durch Klassendiagramme oder domänenspezifische Modelle entworfen werden; die Generierung von Quelltext wird vielseitig unterstützt. Anders verhält es sich jedoch bei der Modellierung von *Verhalten*: Während die **UML** hierfür eine Reihe von Diagrammarten wie Zustands- oder Sequenzdiagramme vorsieht [32, Kapitel 4], siedelt *Ecore* die Beschreibung von Verhalten eine Abstraktionsebene tiefer an und lässt dem Modellierer generierte leere Methodenrümpfe ausimplementieren (vgl. Abschnitt 2.2).

Das derzeit am Lehrstuhl in Entwicklung befindliche Projekt *ModGraph* [13] befasst sich mit der Beschreibung von Verhalten durch *Graphtransformationsregeln* ([12]). Die Regeln selbst werden auf Basis eines Ecore-basierten Metamodells beschrieben. Für die Erzeugung von Regelsätzen steht ein grafischer Editor zur Verfügung.

[49] *Graphical Modeling Framework*, `http://www.eclipse.org/modeling/gmp/`

Bei der Entwicklung einer Produktlinie für einen auf eine bestimmte Anwendungsdomäne zugeschnittenen Satz an Graphtransformationsregeln könnte untersucht werden, inwieweit sich die in der Einleitung postulierte *organisierte Wiederverwendung* auch mit Verhaltensbausteinen realisieren lassen könnte. Eine entsprechende Fallstudie könnte außerdem Aufschluss über die im vorhergehenden Unterabschnitt beschriebene Integration des Ansatzes in grafische Editoren geben.

7.5 Schlusswort

Die modellgetriebene Softwareentwicklung birgt nach meiner persönlichen Einschätzung das Potenzial, sich langfristig in industriellen Entwicklungsprozessen zu etablieren. Wie auch in der Programmierung ist der Schlüssel zum Erfolg die durch die *Wiederverwendung* von Artefakten – seien es Quelltextfragmente oder Teile eines Modells – ermöglichte Steigerung der Produktivität und Senkung der Fehlerrate und der Kosten.

In Programmiersprachen wurden Konstrukte, die ebendiese Wiederverwendung erlauben, erst im Laufe der Jahre eingeführt und zum *State of the Art* erklärt. Softwareproduktlinien könnten der modellgetriebenen Entwicklung zu einer ähnlichen Produktionssteigerung verhelfen, wie Vererbung und Typparametrisierung es bei der objektorientierten Programmierung tun. Voraussetzung hierfür ist eine zunehmende Verträglichkeit des Produktlinien-Ansatzes mit den immer agiler werdenden Softwareentwicklungsprozessen.

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

Abkürzungsverzeichnis

AGG	Attribuierte Graph-Grammatik
AM	Attribut-Mapping
AOM	Aspektorientierte Modellierung
AOP	Aspektorientierte Programmierung
ATL	Atlan Transformation Language
API	Anwendungsentwicklungsschnittstelle (vgl. engl. *Application Programming Interface*)
AST	Abstrakter Syntaxbaum (vgl. engl. *Abstract Syntax Tree*)
CBFM	kardinalitätsbasierte Featuremodellierung (vgl. engl. *Cardinality Based Feature Modeling*)
CIDE	Colored Integrated Development Environment
CM	Containment-Mapping
CPA	Analyse kritischer Paare (vgl. engl. *Critical Pair Analysis*)
DM	Domänenmodell
DnD	Ziehen und Ablegen (vgl. engl. *Drag and Drop*)
DSL	Domänenspezifische Sprache (vgl. engl. *Domain Specific Language*)
EMF	Eclipse Modeling Framework
EMP	Eclipse Modeling Project
FAMILE	Features And Mappings In Lucid Evolution
FEL	Feature Expression Language
FK	Featurekonfiguration
FM	Featuremodell
FODA	Feature-orientierte Domänenanalyse
F2DMM	Feature To Domain Mapping Model
GMF	Graphical Modeling Framework
GUI	Grafische Benutzerschnittstelle (vgl. engl. *Graphical User Interface*)
KFG	Kontextfreie Grammatik
HAS	System zur Automatisierung von Wohnräumen (vgl. engl. *Home Automation System*)
LAN	Lokales Netzwerk (vgl. engl. *Local Area Network*)
MATA	Modeling Aspects using a Transformational Approach
MB	Mapping-Beschreibung
MDA	Modellgetriebene Architektur (vgl. engl. *Model Driven Architecture*)
MDPLE	Modellgetriebene Entwicklung von Softwareproduktlinien (vgl. engl. *Model Driven Product Line Engineering*)
MDSE	Modellgetriebene Softwareentwicklung (vgl. engl. *Model Driven Software Engineering*
MM	Mapping-Modell
MODPL	Modeling Product Lines
MOD2-SCM	Modular and Model driven Software Configuration Management
MOF	Meta Object Facility
MTL	Model Transformation Language
MWE	Modeling Workflow Engine
M2M	Modell-zu-Modell-Transformation (vgl. engl. *Model To Model transformation*)
OCL	Object Constraint Language

OMG	Object Management Group
OOP	Objektorientierte Programmierung
OVM	Orthogonales Variabilitätsmodell
PIM	Plattformunabhängiges Modell (vgl. engl. *Platform Independent Model*)
PLiBS	Product Line Behavior Synthesis
PLUS	Product Line UML based Software engineering
PS	Propagationsstrategie
PSM	Plattformspezifisches Modell (vgl. engl. *Platform Specific Model*)
QVT	Query View Transformation
RM	Referenz-Mapping
RP	Rückwärts-Propagation
RUP	Rational Unified Process
SDIRL	Structural Dependency Identification and Repair Language
SKMS	Softwarekonfigurations-Management-System
SM	Lösungsraum-Modell (vgl. engl. *Solution space Model*)
SPL	Softwareproduktlinie
SZ	Selektionszustand
TGG	Tripel-Graph-Grammatik
TMF	Textual Modeling Framework
UI	Benutzerschnittstelle (vgl. engl. *User Interface*)
UML	Unified Modeling Language
UML2	Unified Modeling Language, Revision 2
URI	Uniform Resource Identifier
VM	Variabilitätsmodell (vgl. engl. *Variability Model*)
VML	Variability Management Language
VML*	A Family of Variability Management Languages
VP	Vorwärts-Propagation
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Literatur

[1] Aho, Alfred V. ; Lam, Monica S. ; Sethi, Ravi ; Ullman, Jeffrey D.: *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Prentice Hall, 2006. – ISBN 0321486811

[2] Allilaire, Freddy ; Bézivin, Jean ; Jouault, Frédéric ; Kurtev, Ivan: ATL — Eclipse Support for Model Transformation. In: *Proc. of the Eclipse Technology eXchange Workshop (eTX) at ECOOP*, 2006

[3] Antkiewicz, Michal ; Czarnecki, Krzysztof: FeaturePlugin: feature modeling plug-in for Eclipse. In: *Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*. New York, NY, USA : ACM, 2004 (eclipse '04), S. 67–72

[4] Arendt, Thorsten ; Biermann, Enrico ; Jurack, Stefan ; Krause, Christian ; Taentzer, Gabriele: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In: Petriu, Dorina (Hrsg.) ; Rouquette, Nicolas (Hrsg.) ; Haugen, Aystein (Hrsg.): *Model Driven Engineering Languages and Systems* Bd. 6394. Springer Berlin / Heidelberg, 2010, S. 121–135. – ISBN 978-3-642-16144-5

[5] Atkinson, C.: *Component-Based Product Line Engineering With UML*. Addison-Wesley, 2002 (Addison-Wesley Object Technology Series). – ISBN 9780201737912

[6] Balzert, Helmut: *Lehrbuch der Software-Technik: Softwaremanagement*. 2. Heidelberg : Spektrum, 2008. – ISBN 978-3-8274-1161-7

[7] Beydeda, Sami (Hrsg.) ; Book, Matthias (Hrsg.) ; Gruhn, Volker (Hrsg.): *Model-Driven Software Development*. Springer, 2005

[8] Bézivin, Jean: On the unification power of models. In: *Software and Systems Modeling* 4 (2005), S. 171–188. – 10.1007/s10270-005-0079-0. – ISSN 1619-1366

[9] Bleek, Wolf-Gideon ; Wolf, Henning: *Agile Softwareentwicklung - Werte, Konzepte und Methoden*. dpunkt.verlag, 2008. – ISBN 978-3-89864-473-0

[10] Buchmann, Thomas: *Modelle und Werkzeuge für modellgetriebene Softwareproduktlinien am Beispiel von Softwarekonfigurationsverwaltungssystemen*, Universität Bayreuth, Dissertation, 2010

[11] Buchmann, Thomas: Valkyrie: A UML-Based Model-Driven Environment for Model-Driven Software Engineering. In: *Proceedings of the 7th International Conference on Software Paradigm Trends*, 2012

[12] Buchmann, Thomas ; Dotor, Alexander ; Westfechtel, Bernhard: Triple Graph Grammars or Triple Graph Transformation Systems?

[13] Buchmann, Thomas ; Westfechtel, Bernhard ; Winetzhammer, Sabine: MOD-GRAPH - A Transformation Engine for EMF Model Transformations. In: *Proceedings of the 6th International Conference on Software and Data Technologies*, 2011, S. 212 – 219

[14] Clayberg, Eric ; Rubel, Dan: *Eclipse: Building Commercial-Quality Plug-ins (2nd Edition) (Eclipse)*. Addison-Wesley Professional, 2006. – ISBN 032142672X

[15] Czarnecki, Krzysztof ; Eisenecker, Ulrich W.: *Generative programming: methods, tools, and applications*. New York, NY, USA : ACM Press/Addison-Wesley Publishing Co., 2000. – ISBN 0-201-30977-7

[16] Czarnecki, Krzysztof ; Kim, Chang H.: *Cardinality-Based Feature Modeling and Constraints: A Progress Report*. Oktober 2005

[17] Dotor Schumann, Alexander: *Entwurf und Modellierung einer Produktlinie von Software-Konfigurations-Management-Systemen*, Universität Bayreuth, Dissertation, 2011

[18] Dvorak, Radomil: Model transformation with Operational QVT. In: *EclipseCon'08*, 2008

[19] Eclipse.org Xtext Developer Group: *Xtext User Guide*. 2012. – URL `http://www.eclipse.org/Xtext/documentation/1_0_0/xtext.html`. – [Online; Stand 17. April 2012]

[20] Frankel, David S.: *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, 2003

[21] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John M.: *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st. Addison-Wesley Longman, 1995

[22] Giese, H. ; Wagner, R.: Incremental Model Synchronization with Triple Graph Grammars. In: *Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS), Genova, Italy*, 2006

[23] Gomaa, Hassan: *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Redwood City, CA, USA : Addison Wesley Longman Publishing Co., Inc., 2004. – ISBN 0201775956

[24] Greenyer, J.: *A Study of Technologies for Model Transformation: Reconciling TGGs with QVT*, University of Paderborn, Department of Computer Science, Paderborn, Germany, Diplomarbeit, 2006

[25] Greenyer, J. ; Kindler, E.: Comparing Relational Model Transformation Technologies: Implementing Query/View/Transformation with Triple Graph Grammars. In: *Software and Systems Modeling (SoSyM)* 9 (2009), January, Nr. 1, S. 21–46. – Published online July 15, 2009

[26] Greenyer, J. ; Rieke, J.: An Improved Algorithm for Preventing Information Loss in Incremental Model Synchronization / Software Engineering Group, Heinz Nixdorf Institute. 2011. – Forschungsbericht

[27] Groher, Iris ; Voelter, Markus: XWeave: models and aspects in concert. In: *Proceedings of the 10th international workshop on Aspect-oriented modeling*. New York, NY, USA : ACM, 2007 (AOM '07), S. 35–40. – ISBN 978-1-59593-658-5

[28] Heidenreich, Florian: Towards systematic ensuring well-formedness of software product lines. In: *Proceedings of the First International Workshop on Feature-Oriented Software Development*. New York, NY, USA : ACM, 2009 (FOSD '09), S. 69–74

[29] Heidenreich, Florian ; Şavga, Ilie ; Wende, Christian: On Controlled Visualisations in Software Product Line Engineering. In: *Proceedings of the 2nd International Workshop on Visualisation in Software Product Line Engineering (ViSPLE 2008), collocated with the 12th International Software Product Line Conference (SPLC 2008)*, September 2008. – To appear

[30] Heidenreich, Florian ; Kopcsek, Jan ; Wende, Christian: FeatureMapper: Mapping Features to Models. In: *Companion Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*. New York, NY, USA : ACM, Mai 2008, S. 943–944. – ISBN 978-1-60558-079-1

[31] Heidenreich, Florian ; Sánchez, Pablo ; Santos, João ; Zschaler, Steffen ; Alférez, Mauricio ; Araujo, João ; Fuentes, Lidia ; Ana Moreira, Uira K. amd ; Rashid, Awais: Relating Feature Models to Other Models of a Software Product Line: A Comparative Study of FeatureMapper and VML*. In: *Transactions on Aspect-Oriented Software Development VII, Special Issue on A Common Case Study for Aspect-Oriented Modeling* 6210 (2010), S. 69–114

[32] Hitz, Martin ; Kappel, Gerti: *UML @ Work*. Dpunkt Verlag, 2005. – ISBN 3898642615

[33] Jahnke, J. H. ; Zündorf, A.: Specification and Implementation of a Distributed Planning and Information System for Courses based on Story Driven Modeling. In: *Proc. of 9th International Workshop on Software Specification and Design, Ise-Shima, Japan*, 1998

[34] Jayaraman, Praveen ; Whittle, Jon ; Elkhodary, Ahmed ; Gomaa, Hassan: Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. In: Engels, Gregor (Hrsg.) ; Opdyke, Bill (Hrsg.) ; Schmidt, Douglas (Hrsg.) ; Weil, Frank (Hrsg.): *Model Driven Engineering Languages and Systems* Bd. 4735. Springer Berlin / Heidelberg, 2007, S. 151–165. – ISBN 978-3-540-75208-0

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

[36] Kästner, Christian: *Virtual Separation of Concerns: Towards Preprocessors 2.0*, University of Magdeburg, Dissertation, May 2010

[37] Königs, Alexander ; Schürr, Andy: MDI – A Rule-Based Multi-Document and Tool Integration Approach. In: *Special Section on Model-Based Tool Integration in Journal of Software and System Modeling, Academic*, Press, 2006

[38] Kruchten, Philippe: *The Rational Unified Process: An Introduction*. 3. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2003. – ISBN 0321197704

[39] Lukas, Lutz: *Erstellung eines Editors für Featuremodelle und Featurekonfigurationen*. 2012. – Vortrag zum kleinen Master-Projekt im Studiengang Computer Science, Universität Bayreuth

[40] Mendonca, Marcilio ; Wasowski, Andrzej ; Czarnecki, Krzysztof: SAT-based analysis of feature models is easy. In: *Proceedings of the 13th International Software Product Line Conference*. Pittsburgh, PA, USA : Carnegie Mellon University, 2009 (SPLC '09), S. 231–240

[41] Object Management Group: *OMG QVT Final Adopted Specification*, 2005. – URL `http://www.omg.org/spec/QVT/1.0`

[42] Object Management Group: *Meta Object Facility (MOF) Core Specification Version 2.0*, 2006. – URL `http://www.omg.org/cgi-bin/doc?formal/2006-01-01`

[43] Object Management Group: *OCL 2.2 Specification*, 2010. – URL `http://www.omg.org/spec/OCL/2.2`

[44] Object Management Group: *Documents Associated With UML Version 2.4.1*, 2011. – URL `http://www.omg.org/spec/UML/2.4.1`

[45] Pohl, Klaus ; Böckle, Günter ; Linden, Frank J. van d.: *Software Product Line Engineering: Foundations, Principles and Techniques.* Secaucus, NJ, USA : Springer-Verlag New York, Inc., 2005. – ISBN 3540243720

[46] Rausch, Andreas ; Broy, Manfred ; Bergner, Klaus ; Höhn, Reinhard ; Höppner, Stephan: *Das V-Modell XT. Grundlagen, Methodik und Anwendungen.* Heidelberg : Springer, 2007

[47] Schürr, Andy: Specification of Graph Translators with Triple Graph Grammars / RWTH Aachen. 1994. – Forschungsbericht

[48] Steinberg, Dave ; Budinsky, Frank ; Brodsky, Stephen A. ; Merks, Ed: *Eclipse Modeling Framework.* Pearson Education, 2003. – ISBN 0131425420

[49] Taentzer, Gabriele: AGG: A Graph Transformation Environment for Modeling and Validation of Software. In: Pfaltz, John (Hrsg.) ; Nagl, Manfred (Hrsg.) ; Böhlen, Boris (Hrsg.): *Applications of Graph Transformations with Industrial Relevance* Bd. 3062. Springer Berlin / Heidelberg, 2004, S. 446–453. – ISBN 978-3-540-22120-3

[50] Uhrig, Sabrina: *Korrespondenzberechnung auf Klassendiagrammen*, Universität Bayreuth, Dissertation, 2011

[51] Ullenboom, Christian: *Java ist auch eine Insel.* 6., aktualisierte und erweiterte Auflage. Bonn : Galileo Computing, 2007. – URL `http://www.galileocomputing.de/openbook/javainsel6/`

[52] Wagenknecht, C. ; Hielscher, M.: *Formale Sprachen, abstrakte Automaten und Compiler: Lehr- und Arbeitsbuch für Grundstudium und Fortbildung.* Vieweg+Teubner Verlag, 2009 (Leitfäden der Informatik). – ISBN 9783834806246

[53] Warmer, Jos ; Kleppe, Anneke: *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley, 1998. – 112 S

[54] Whittle, Jon ; Jayaraman, Praveen ; Elkhodary, Ahmed ; Moreira, Ana ; Araújo, João: MATA: A Unified Approach for Composing UML Aspect Models Based on Graph Transformation. In: Katz, Shmuel (Hrsg.) ; Ossher, Harold (Hrsg.) ; France, Robert (Hrsg.) ; Jézéquel, Jean-Marc (Hrsg.): *Transactions on Aspect-Oriented Software Development VI* Bd. 5560. Springer Berlin / Heidelberg, 2009, S. 191–237. – ISBN 978-3-642-03763-4

[55] Wunderlich, Lars: *Aspektorientierte Programmierung (AOP).* Entwickler.press, 2005. – ISBN 9783935042741

[56] Ziadi, Tewfik ; Jézéquel, Jean-Marc: *Software Product Line Engineering with the UML: Deriving Products.* Bd. pages. S. 557–588. In: *Software Product Lines* Bd. pages, Springer, 2006

[57] Ziadi, Tewfik ; Jézéquel, Jean-Marc: PLiBS: an Eclipse-based tool for Software Product Line Behavior Engineering. In: *Proc. of 3rd Workshop on Managing Variability for Software Product Lines, SPLC 2007.* Kyoto, Japan, Japan, 2007

[58] Zschaler, Steffen ; Sánchez, Pablo ; Santos, João ; Alférez, Mauricio ; Rashid, Awais ; Fuentes, Lidia ; Moreira, Ana ; Araújo, João ; Kulesza, Uirá: *VML* – a family of languages for variability management in software product lines.* 2009

Erklärung

Hiermit erkläre ich, dass ich die Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen und Hilfsmittel eingesetzt habe. Die Arbeit wurde nicht bereits zur Erlangung eines akademischen Grades eingereicht.

Ort, Datum	Unterschrift