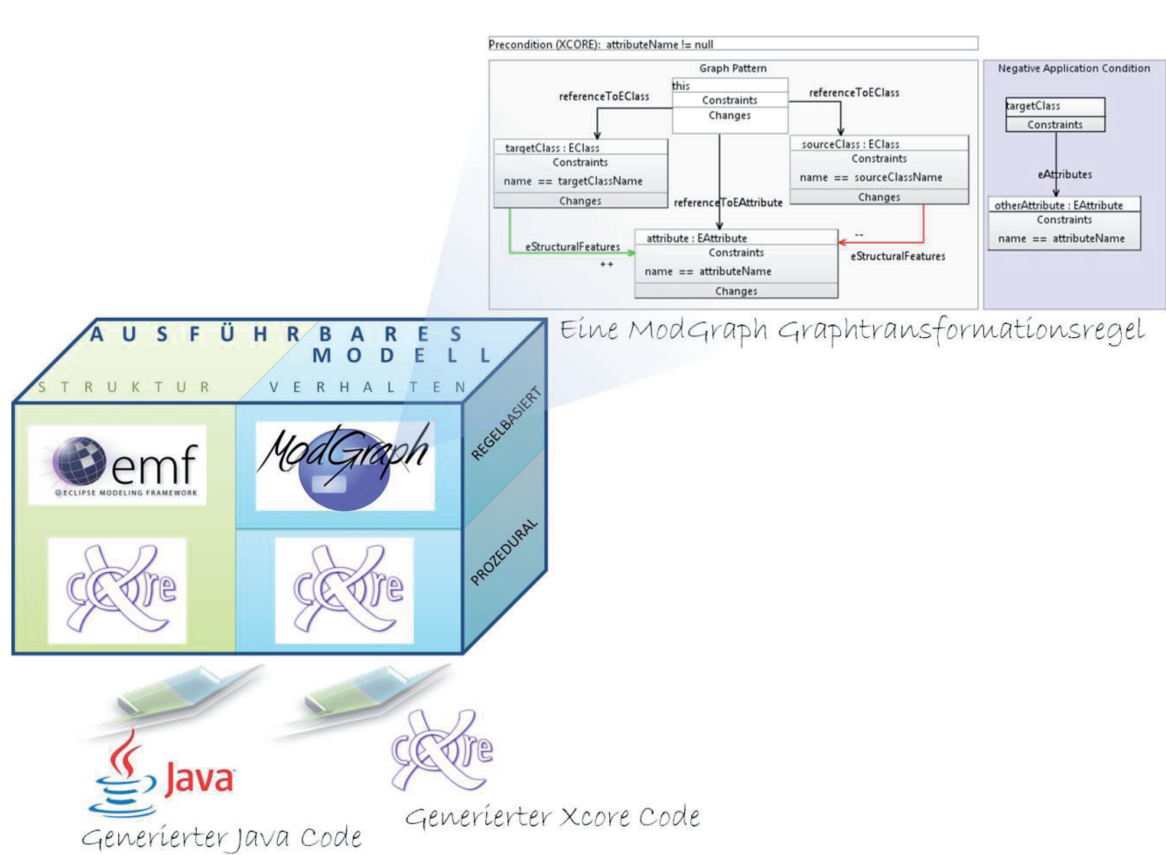
Modellgetriebene Entwicklung mit Graphtransformationen

Sabine Winetzhammer



An der Universität Bayreuth zur Erlangung des Grades eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

1. Gutachter Prof. Dr. Bernhard Westfechtel
2. Gutachter Prof. Dr. Andy Schürr

Modellgetriebene Entwicklung mit Graphtransformationen

An der Universität Bayreuth
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

von

Sabine Winetzhammer

aus Ingolstadt

1. Gutachter Prof. Dr. Bernhard Westfechtel
2. Gutachter Prof. Dr. Andy Schürr

Tag der Einreichung: 16. Dezember 2014
Tag des Kolloquiums: 13. März 2015

Kurzfassung

In dieser Arbeit wird *ModGraph* vorgestellt. Es bietet einen ganzheitlichen, leichtgewichtigen und hochgradig integrativen Ansatz zur *totalen modellgetriebenen Softwareentwicklung*. Der ModGraph-Ansatz bietet eine echte Erweiterung des EMF-Rahmenwerks, einem in Industrie und Forschung etablierten Werkzeug zur Strukturmodellierung (und anschließender Quelltextgenerierung) mit Ecore-Klassendiagrammen. Dabei steht die Vervollständigung von EMF um die - bislang fehlende - Verhaltensmodellierung im Vordergrund. Diese wird durch einen *hybriden Ansatz* aus regelbasierter und prozeduraler Verhaltensmodellierung erreicht, der *programmierte Graphtransformationsregeln* zur Spezifikation von Verhalten anbietet und gleichzeitig eine strikte Trennung der Regeln und der prozeduralen Elemente fordert. Dazu setzt sich ModGraph zum Ziel, bestehende Konzepte zu erweitern und zu nutzen, statt diese nochmals zu reimplementieren.

ModGraph konzentriert sich dabei auf den *Mehrwert der Graphtransformationsregeln im Kontext der Verhaltensmodellierung*. Sie stellen komplexe Änderungen an Objektstrukturen deklarativ dar, indem lediglich der Zustand der Objektstruktur vor und nach ihrer Ausführung angegeben wird.

In dieser Arbeit wird eine ausführbare Sprache zur Modellierung von Verhalten, basierend auf Graphtransformationsregeln *für* und *mit* EMF entwickelt, die nahtlos integriert sind. Den Ansatzpunkt bieten die strukturellen Ecore-Modelle. Jede Regel wird als Implementierung einer darin deklarierten Operation aufgefasst. Sie spezifiziert das Verhalten der Operation, die auf der Modell-Instanz ausgeführt wird. Hierbei wird die Instanz als Graph aufgefasst und eine kompakte Darstellung der Regel genutzt, die es erlaubt, Struktur und Änderungen in einem zusammengeführten Diagramm darzustellen. Dazu werden ein- und mehrwertige Knoten, die Objekte und Objektmengen repräsentieren, sowie Links, die Instanzen von Referenzen darstellen, und Pfade, die für abgeleitete Referenzen stehen, angeboten. Neben Transformationen können auch Tests und Abfragen auf den Graphen spezifiziert werden, indem Regeln ohne explizite Angabe von Änderungen modelliert werden. Zudem ist es möglich, das Graphmuster durch Bedingungen zu ergänzen. Textuelle Vor- und Nachbedingungen, sowie durch Graphen repräsentierte negative Anwendbarkeitsbedingungen, erlauben eine Einschränkung der Anwendbarkeit der Regel.

Zur Komposition der Regeln wird Xcore verwendet, das eine textuelle Syntax für Ecore anbietet und diese um die Sprachkonstrukte zur Verhaltensbeschreibung mit dem Java-nahen Xbase erweitert. Außerdem wird es zur Spezifikation von einfachen oder prozeduralen Operationen verwendet, so dass der Nutzer auswählen kann, welche Teilprobleme mit Regeln oder direkt in Xcore gelöst werden sollen. Dies führt zu einer geeigneteren Nutzung der Regeln. Zudem kann Xcore zur Strukturmodellierung, alternativ zu Ecore, verwendet werden.

Die Unterstützung des Modellierens im Großen wird außerdem durch die Einbindung eines existierenden Paketdiagrammeditors erreicht. Paketdiagramme statten EMF mit einer Möglichkeit zur Architekturmodellierung aus.

Die - zur Ausführung des Modells nötige - Zusammenführung der Struktur, der Regeln und der prozeduralen Elemente erfolgt entweder auf Modell- oder auf Quelltextebene. Dazu wird eine kompilierte Lösung verwendet, die zwei verschiedene Modell-Transformationen anbietet. Einmal wird der EMF- oder Xcore-Generator genutzt, der Java-Quelltext erzeugt, in welchen der ModGraph-Generator den aus den Regeln generierten Quelltext nahtlos integriert. Alternativ kann eine bislang einzigartige Transformation der Regeln in die Modellierungssprache Xcore stattfinden. Sie werden übersetzt und nahtlos in das bestehende Modell integriert. Das so entstandene Xcore-Modell kann interpretiert oder in Java-Quelltext übersetzt werden. Bei der Generierung nach Java (via Xcore) setzt ModGraph einen, bislang im Graphtransformationskontext nicht verfügbaren, stufenweisen Übersetzungsprozess der Regeln um, der zur Unabhängigkeit von der Zielprogrammiersprache führt.

Die Eclipse-basierte Werkzeugumgebung zum ModGraph-Ansatz bietet - neben einem intuitiv zu bedienenden grafischen Editor zur Erstellung kommentierter Regeln - eine eigene, auf die Entwicklung der Regeln zugeschnittene Ansicht, die den Nutzer in die Abläufe der Entwicklung mit ModGraph einführt und ihn anleitet (Cheat Sheets, Dashboard). Dadurch wird dem Modellierer ein einfacher, schrittweiser Übergang von der reinen Modellierung mit EMF in die Modellierung mit ModGraph ermöglicht.

Zur Evaluation der Arbeit werden zwei größere Anwendungen der ModGraph-Regeln betrachtet. Zum Einen werden propagierende Refactorings *für* und *mit* ModGraph erstellt. Dabei wird ein reflektiver Ansatz verfolgt, der komplexe Refactoring-Operationen an Ecore-Modellen auf die Regeln propagiert. Zum Anderen wird die Anwendung der Graphtransformationsregeln im Kontext der szenarienbasierten Modellierung zur Simulation von Echtzeitsystemen untersucht.

Abstract

This thesis presents ModGraph, a sound, lightweight and highly integrative approach for *total model-driven software engineering*. The ModGraph approach extends the Eclipse Modeling Framework, which is widely used in both industry and research. EMF offers structural modeling (and code generation) using Ecore class diagrams.

This work focuses on adding the missing behavior modeling, by using and extending existing concepts. It presents a *hybrid approach* for behavior modeling, using both rule-based and procedural elements, but strictly separated. For the rule-based part, *programmed graph transformation rules* are used.

ModGraph focuses on the *added value of graph transformation rules in the context of behavior modeling*. They offer a declarative style to model complex changes on object structures. Therefore one specifies the objects' state before and after the application of the rule.

This work offers a seamless integrated, executable language to model behavior with graph transformation rules *for* and *with* EMF. Each rule implements an operation defined in the Ecore model. It specifies the operations' behavior, which is executed on a model instance. Therefore the instance is considered as a graph. The rule is specified using a compact editor, which allows the specification of the structure and the changes in a merged view. To this end, ModGraph provides single- and multi-valued nodes (representing objects and sets of them, respectively) as well as links (representing instances of references) and paths (representing derived references). Besides transformations, one may specify tests and queries, by defining rules without changes. Furthermore, each rule may be constrained concerning its applicability using textual pre- and postconditions as well as graph-based graphical negative application conditions.

Rules are composed with Xcore, a textual syntax for Ecore that extends it by using the Java-like language Xbase for behavior specification. Additionally, ModGraph uses Xcore to specify simple or procedural operations. Hence, the user may decide if a rule or a procedural operation is suitable for the considered part of his problem. This strategy leads to a more concise use of the rules. Furthermore, Xcore may replace Ecore for structural modeling.

Also supporting modeling in the large, ModGraph integrates an existing package diagram editor. In this way it supports architecture modeling in EMF.

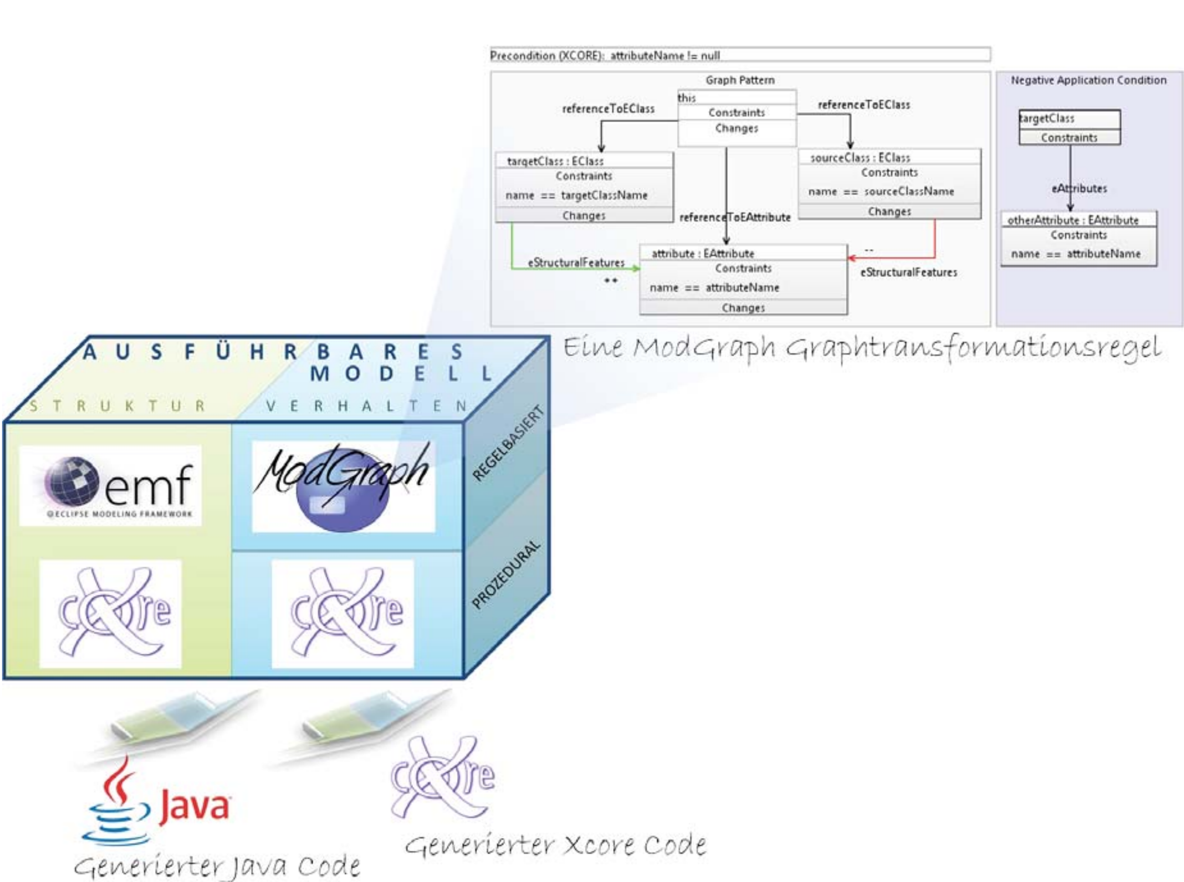
For the execution of a model specified with ModGraph, structural, rule-based and procedural models need to be merged. This may be performed on model and on code level. For the realization, a compiled solution offers two kinds of model transformations. First the EMF or Xcore generator is used to create Java code, in which the ModGraph generator injects the code generated from the rules. Alternatively, ModGraph provides a new unique approach to compile the rules into operations of the Xcore model. This

means compiling the rules to their Xcore representation and injecting it into the model. The resulting Xcore model may be interpreted or compiled. For generation, ModGraph provides a staged translation to Java (via Xcore), which is unique in the context of graph transformation tools. It provides platform independence, as the finally generated programming language is no more significant.

ModGraph's Eclipse-based tooling offers an intuitive editor for creating and commenting rules as well as a view that guides and supports the user during the modeling process (cheat sheets, dashboard). Hence, it helps the EMF user to use ModGraph's capabilities.

To evaluate the approach, this thesis considers two profoundly different use cases. The first one offers propagating refactoring operations *for* and *with* ModGraph. Therefore it uses a reflective approach that propagates the model refactorings to the rules. The second one investigates the usability of graph transformation rules in the context of scenario-based modeling to simulate real-time systems.

Grafische Kurzfassung



Inhaltsverzeichnis

I Einführung in die Thematik **1**

1 Einleitung **3**

1.1 Motivation . . . 3
1.2 Beitrag der Arbeit . . . 4
1.2.1 Kontext . . . 4
1.2.2 Zielsetzung . . . 5
1.2.3 Praktische Umsetzung . . . 7
1.3 Einführendes Beispiel: Refactoring auf Ecore-Modellen . . . 9
1.4 Bestehende Arbeiten . . . 9
1.5 Übersicht . . . 10

2 Grundlagen **13**

2.1 Überblick . . . 13
2.2 Modelle . . . 13
2.2.1 Der Modellbegriff . . . 13
2.2.2 Klassifikation von Modellen in der modellgetriebenen Softwareentwicklung . . . 14
2.2.3 Metaebenen von Modellen . . . 16
2.3 Sprachen im Rahmen der Softwareentwicklung . . . 17
2.3.1 Aufbau einer Sprache . . . 17
2.3.2 Einordnung einer Sprache in die Modellierung . . . 19
2.3.3 Domänenspezifische und allgemeine Sprachen . . . 20
2.4 Modellgetriebene Softwareentwicklung . . . 20
2.5 Modelltransformationen . . . 21
2.6 Graphtransformationen . . . 25
2.6.1 Graphen . . . 26
2.6.2 Idee einer Graphtransformation . . . 27
2.6.3 Theoretische Ansätze zur Graphtransformation . . . 29
2.7 Zusammenfassung . . . 32

II Bestehende Ansätze und Werkzeuge zur Modelltransformation 33

3 Das Eclipse Modeling Framework 35
3.1 EMF-Modelle . . . 35
3.2 Ausführen des Modells . . . 40
3.2.1 Java-Quelltext-Generierung . . . 40
3.2.2 Interpretation mit dynamischem EMF . . . 42
3.3 Hinzufügen von Verhalten . . . 42
3.4 Graphical Modeling Framework (GMF) . . . 43

4 Transformationssprachen 45
4.1 Ansätze und Werkzeuge zur Modell-zu-Modell-Transformation . . . 45
4.1.1 Xcore . . . 45
4.1.2 Atlas Transformation Language . . . 49
4.1.3 Query Views Transformation . . . 50
4.2 Ansätze und Werkzeuge zur graphbasierten Transformation . . . 51
4.2.1 Berliner Werkzeuge basierend auf dem algebraischen Ansatz . . . 51
4.2.2 Die sprachliche Einkleidung des logikorientierten Ansatzes . . . 56
4.2.3 Story-basierte Ansätze . . . 59
4.2.4 Andere Ansätze . . . 62
4.3 Fazit . . . 66

III Verhaltensmodellierung für EMF - ModGraph 69

5 ModGraphs konzeptioneller Ansatz 71
5.1 ModGraphs Architektur . . . 71
5.2 Entwurf der Graphtransformationsregeln . . . 74
5.2.1 Das Ecore-Modell als Graph . . . 74
5.2.2 Zusammenspiel von Ecore-Modell und Graphtransformationsregeln 79
5.2.3 Aufbau einer Graphtransformationsregel . . . 81
5.3 Ausführung des Modells . . . 85
5.4 Informelle Spracheinführung am Beispiel . . . 85
5.4.1 Das Bugtracker-Ecore-Modell . . . 86
5.4.2 Graphtransformationsregeln für den Bugtracker . . . 88
5.4.3 Prozedurale Elemente im Bugtracker . . . 94
5.4.4 Das ausführbare Bugtracker-Modell . . . 98
5.4.5 Zusammenfassung der Konzepte . . . 98

6 Design der Sprache für Graphtransformationsregeln 101
6.1 Syntax der Sprache für Graphtransformationsregeln . . . 101
6.1.1 Das Metamodell der Sprache . . . 101
6.1.2 Die konkrete Syntax . . . 112

6.2 Dynamische Semantik der Sprache für Graphtransformationsregeln 115
6.2.1 Ausführungslogik einer Graphtransformationsregel 116
6.2.2 Mustersuche 118
6.2.3 Anwendung der modellierten Änderungen 123

7 Integration in das EMF-Rahmenwerk 127
7.1 Konzeptuelle Integration: Wege zur Nutzung von ModGraph 127
7.2 Generatoren 133
7.2.1 Generierung von Java-Quelltext mit ModGraph 133
7.2.2 Einbinden der ModGraph-Regeln in ein Xcore-Modell 148
7.2.3 Vergleich der Modelle und Quelltexte 153
7.3 Eclipse-Integration 156
7.3.1 Die ModGraph-Perspektive 156
7.3.2 Grafischer Editor und Validierung 158
7.3.3 Dashboard 159
7.3.4 Unterstützung durch Cheat Sheets 160

IV Evaluation 163

8 Propagierendes Refactoring 165
8.1 Entstehende Problematik durch abhängige Modelle in ModGraph 166
8.2 Konzeption des propagierenden Refactorings 169
8.3 Implementierung 176
8.3.1 Architektur des propagierenden Refactorings 176
8.3.2 Beispielrefactoring: Ändern einer bidirektionalen Referenz in eine unidirektionale 178
8.3.3 Refactoring 2: Extrahieren einer Klasse 182

9 Integration mit modalen Sequenzdiagrammen 187
9.1 Theoretische Überlegungen 187
9.1.1 Beschreibung typischer Systeme am Beispiel 187
9.1.2 Modale Sequenzdiagramme und Play-out 190
9.1.3 Integration von MSDs mit Graphtransformationsregeln 194
9.2 Prototypische Werkzeugintegration von ModGraph und ScenarioTools . . 197

10 Diskussion und Abgrenzung des Ansatzes 201
10.1 Diskussion des Ansatzes 201
10.1.1 Evaluation der Regeln 201
10.1.2 Laufzeitmessungen 207
10.2 Abgrenzung des ModGraph-Ansatzes zu verwandten Arbeiten 211
10.2.1 EMF und ModGraph 212
10.2.2 ModGraph und Xcore 213
10.2.3 Modelltransformationswerkzeuge und ModGraph 214

10.3 Abgrenzung des propagierenden Refactorings 219
10.4 Abgrenzung der Integration der Regeln mit MSDs 220
10.5 Ergebnis und Ausblick 222

Literaturverzeichnis 225

A Anhang 240
A.1 Metamodell für Graphtransformationsregeln als Baum 241
A.2 Nutzung der Validierungssprache Check in ModGraph 242
A.3 Nutzung der Templatesprache Xpand in ModGraph 243
A.4 Vollständiges Bugtracker Xcore-Modell 245
A.5 Laufzeittest 249
A.6 Xtext-Grammatiken 250
A.6.1 Die Xcore-Grammatik 250
A.6.2 Die Xbase-Grammatik 253

Teil I

Einführung in die Thematik

„Wo ein Wille ist, ist auch ein Weg.”
(Sprichwort)

Inhaltsverzeichnis - Teil I

1 Einleitung **3**
1.1 Motivation . . . 3
1.2 Beitrag der Arbeit . . . 4
1.2.1 Kontext . . . 4
1.2.2 Zielsetzung . . . 5
1.2.3 Praktische Umsetzung . . . 7
1.3 Einführendes Beispiel: Refactoring auf Ecore-Modellen . . . 9
1.4 Bestehende Arbeiten . . . 9
1.5 Übersicht . . . 10

2 Grundlagen **13**
2.1 Überblick . . . 13
2.2 Modelle . . . 13
2.2.1 Der Modellbegriff . . . 13
2.2.2 Klassifikation von Modellen in der modellgetriebenen Softwareentwicklung . . . 14
2.2.3 Metaebenen von Modellen . . . 16
2.3 Sprachen im Rahmen der Softwareentwicklung . . . 17
2.3.1 Aufbau einer Sprache . . . 17
2.3.2 Einordnung einer Sprache in die Modellierung . . . 19
2.3.3 Domänenspezifische und allgemeine Sprachen . . . 20
2.4 Modellgetriebene Softwareentwicklung . . . 20
2.5 Modelltransformationen . . . 21
2.6 Graphtransformationen . . . 25
2.6.1 Graphen . . . 26
2.6.2 Idee einer Graphtransformation . . . 27
2.6.3 Theoretische Ansätze zur Graphtransformation . . . 29
2.7 Zusammenfassung . . . 32

1 Einleitung

1.1 Motivation

Mitte des letzten Jahrhunderts entwickelte Nathaniel Rochester den ersten Assembler als Maschinensprache für den damals brandneuen IBM 701.[1] Innerhalb der mittlerweile vergangenen sechs Jahrzehnte haben sich daraus viele, im Abstraktionsgrad steigende, Sprachen entwickelt - darunter auch die objektorientierten Sprachen, wie z.B. Java. Compiler übersetzen seitdem diese abstrakteren Sprachen in Maschinencode.

Parallel dazu gewannen Softwareprojekte an Größe und Langlebigkeit, was nicht zuletzt eben diesem steigenden Abstraktionsgrad der Programmiersprachen zu verdanken ist. Im Sinne einer verbesserten Wartbarkeit der gewachsenen Softwareprojekte gewann die Dokumentation an Bedeutung. Hier ließen sich sehr bald grafische Elemente auffinden.

Spätestens seit der Einführung und Standardisierung (ISO- und OMG-Standard) der Unified Modeling Language (UML), werden verbreitet Modelle zur Softwaredokumentation genutzt [58, 59]. Bei dieser sogenannten modellbasierten Softwareentwicklung existiert nur eine rein gedankliche Verbindung zwischen Modell und Quelltext. Das hat zwei gravierende Nachteile: Software ist gerade in den ersten Phasen der Entwicklung starken Änderungen unterworfen, die zu Inkonsistenzen zwischen Quelltext und dokumentierendem Modell führen können. Zudem tragen diese Modelle wenig zum Softwareentwicklungsprozess bei, da die Konsistenz zum Quelltext vom Entwickler zu gewährleisten ist und damit dessen Interpretation unterliegt [72].

Die soeben aufgezeigten Nachteile sollen durch modellgetriebene Softwareentwicklung möglichst eliminiert werden, indem die Modelle selbst ausführbar sind. Die ausführbaren Modelle sind nach Stahl et al. „nicht nur zur Dokumentation, sondern gleichzusetzen mit Code"[72]. Die Produktivität im Softwareentwicklungsprozess wird also gesteigert, indem konventionelle Programmiersprachen durch abstraktere Modelle ersetzt werden. Somit degeneriert der Code der allgemeinen Programmiersprache zu einem generierten Artefakt. Wird die Software ausschließlich mit Hilfe von Modellen entwickelt, spricht man von totaler modellgetriebener Softwareentwicklung.

Modelle sind heutzutage grafisch oder textuell. Textuelle Modelle werden meist durch domänenspezifische Sprachen dargestellt, grafische Modelle durch Diagramme.[2] Grafische Modelle heben sich weiter als ihre textuellen Pendants vom klassischen Code ab: Durch Symbole und Farben bieten sie meist eine sehr eingängige Darstellung eines Sachverhalts. Es existieren einige Ansätze, wie das Metamodell der UML [59] oder der Meta

[1] `http://www-03.ibm.com/ibm/history/exhibits/701/701_team.html`
[2] Zur Vollständigkeit sei bemerkt, dass ebenso grafische domänenspezifische Sprachen existieren.

Object Facility (MOF) Standard [60], die allgemeine grafische Modellierungssprachen definieren.

Modelle können interpretiert oder kompiliert werden. Dabei werden in den meisten Fällen struktuelle Komponenten, die überwiegend durch Klassendiagramme dargestellt werden, problemlos in Code überführt. Unterstützung zur Verhaltensmodellierung ist häufig nicht gegeben. Die daraus resultierende Lücke wird oft durch das Einfügen von handgeschriebenem Quelltext geschlossen.

Ein prominenter Vertreter dieser Vorgehensweise ist das Eclipse Modeling Framework (EMF) [73]. Hier werden Ecore-Klassendiagramme in Java-Quelltext überführt. Es wird jedoch keinerlei Verhaltensmodellierung angeboten. EMF ist in Industrie und Forschung verbreitet, so dass es als Ausgangspunkt für die in dieser Arbeit angestrebte, totale modellgetriebene Softwareentwicklung gewählt wird.

Die folgenden Kapitel stellen einen leichtgewichtigen Ansatz vor, der totale modellgetriebene Softwareentwicklung auf Basis des EMF ermöglicht, indem er zusätzlich Verhaltensmodellierung bietet. Es werden Werkzeuge bereitgestellt, um mit Hilfe von EMF, Xcore und den im Rahmen dieser Arbeit neu entwickelten Graphtransformationsregeln ausführbare Modelle zu erstellen und Code zu generieren. Der Ansatz wird im weiteren Verlauf der Einleitung skizziert und in der gesamten Arbeit ausführlich dargestellt.

1.2 Beitrag der Arbeit

1.2.1 Kontext

Die freie Programmierumgebung Eclipse[3] bietet eine solide Plattform zur Entwicklung verschiedenster Softwareprodukte. Eine auf die Modellierung zugeschnittene Variante sind die Eclipse Modeling Tools, deren Kern das Eclipse Modeling Framework (EMF) [73] bildet. Sie wurden im Rahmen des Eclipse Modeling Projects[4] entwickelt. [73]

EMF gilt als prominenter Vertreter der grafischen Modellierung und ist ein in Industrie und Forschung verbreitetes Werkzeug zur partiellen modellgetriebenen Softwareentwicklung. Es stellt strukturelle Sachverhalte in Form von Ecore-Klassendiagrammen dar. Ecore-Klassendiagramme basieren auf dem, von der Object Management Group (OMG)[5] definerten, essential Meta Object Facility (eMOF) Standard (eMOF stellt eine Untermenge des in der Motivation erwähnten MOF dar)[60]. Ecore-Klassendiagramme eigenen sich dementsprechend durch ihre große Ausdrucksmächtigkeit, bei überschaubarer Anzahl an Konstrukten, sehr gut zur Strukturmodellierung.

Ein Baumeditor sowie eine Reihe von grafischen Editoren bieten Werkzeugunterstützung für EMF an. Zudem ermöglicht EMF die Generierung von Java-Code aus den Ecore-Klassendiagrammen und die Erstellung einfacher Baumeditoren für Modellinstanzen. Code und Editoren können weiterverwendet werden, z.B. um mittels GMF[6] einen

[3] https://www.eclipse.org/
[4] http://www.eclipse.org/modeling
[5] http://www.omg.org
[6] http://www.eclipse.org/modeling/gmp

grafischen Editor zu erstellen [37]. Neben der Generierung von Code aus dem Modell kann dieses mittels dynamischem EMF interpretiert werden.

Betrachtet man EMF aus der Perspektive der modellgetriebenen Softwareentwicklung, findet sich eine Lücke: Die Verhaltensmodellierung bleibt unberücksichtigt. Der Entwickler muss auf Java-Code ausweichen, um Verhalten in ein Ecore-Modell einzubinden.

Ein relativ neuer Modellierungsansatz innerhalb der Eclipse Modeling Tools ist Xcore[7][3]. Unter dem Motto „Modellieren für Programmierer und Programmieren für Modellierer“ bietet es eine textuelle Syntax für Ecore-Klassendiagramme. Bereits das Motto verrät die Abstraktionsebene: Xcore ist weit abstrakter als Quelltext einer beliebigen Programmiersprache, dennoch ermöglicht es eine textuelle, codeähnliche Sicht auf Ecore. Deshalb wird auch in dieser Arbeit, je nach Verwendung von Xcore, der Begriff Xcore-Modell oder Xcore-Code verwendet.

Darüber hinaus erweitert Xcore die textuelle Darstellung von Ecore, um Konstrukte zur Spezifikation des Verhaltens einzelner Operationen mittels der prozeduralen Sprache Xbase [29] . Diese wird in Xcore zur Implementierung von Operationsrümpfen genutzt. Xbase bietet dementsprechend eine Möglichkeit, Verhalten in Ecore-Modelle einzubringen. Allerdings geschieht dies durch prozedurale Progammierung statt durch eine deklarative Spezifikation.

Letztendlich wird - auch hier - aus dem Xcore-Modell Java-Quelltext erzeugt, der zur Ausführung des Modells verwendet werden kann. Alternativ kann der Xcore-Interpreter verwendet werden, der dynamische Modellinstanzen unterstützt.

1.2.2 Zielsetzung

Die vorangegangenen Betrachtungen zeigen, dass innerhalb von EMF keine Möglichkeit existiert, Verhalten ohne Verlust der Abstraktionsebene zu spezifizieren [73]. An diesem Punkt setzt die vorliegende Arbeit an: EMF wird erweitert, so dass totale modellgetriebene Softwareentwicklung mit EMF möglich ist, ohne dabei „das Rad neu zu erfinden“. Dieser Ansatz wird im Folgenden als ModGraph-Ansatz bezeichnet [19, 21, 78, 80, 81].

Kern der Arbeit ist die Erweiterung von EMF um Graphtransformationsregeln zur Verhaltensmodellierung. Es wird ein leichtgewichtiger Ansatz verfolgt, der speziell für und mit EMF entwickelt wird. Dabei steht der Mehrwert der Graphtransformationsregeln im Vordergrund. Graphtransformationsregeln bieten - grafisch dargestellt - eine abstrakte Möglichkeit zur Modelltransformation. Eine Modelltransformation im EMF-Kontext kann als das Ausführen einer im strukturellen Modell definierten Operation auf einer Modellinstanz aufgefasst werden. Jede Graphtransformationsregel implementiert dementsprechend eine Operation, deren Signatur in Ecore oder Xcore definiert wurde. Die Modellinstanz wird als Graph aufgefasst, in welchem mittels einer Mustersuche Anwendungsstellen für die Transformation gesucht werden. Deshalb spezifiziert jede Graphtransformationsregel innerhalb des Graphmusters genau einen Teilgraphen, der zu ihrer Anwendung innerhalb der Modellinstanz vorhanden sein muss. Zudem werden im Graphmuster die anzuwendenden Änderungen modelliert. Die Anwendbarkeit einer

[7] `https://wiki.eclipse.org/Xcore`

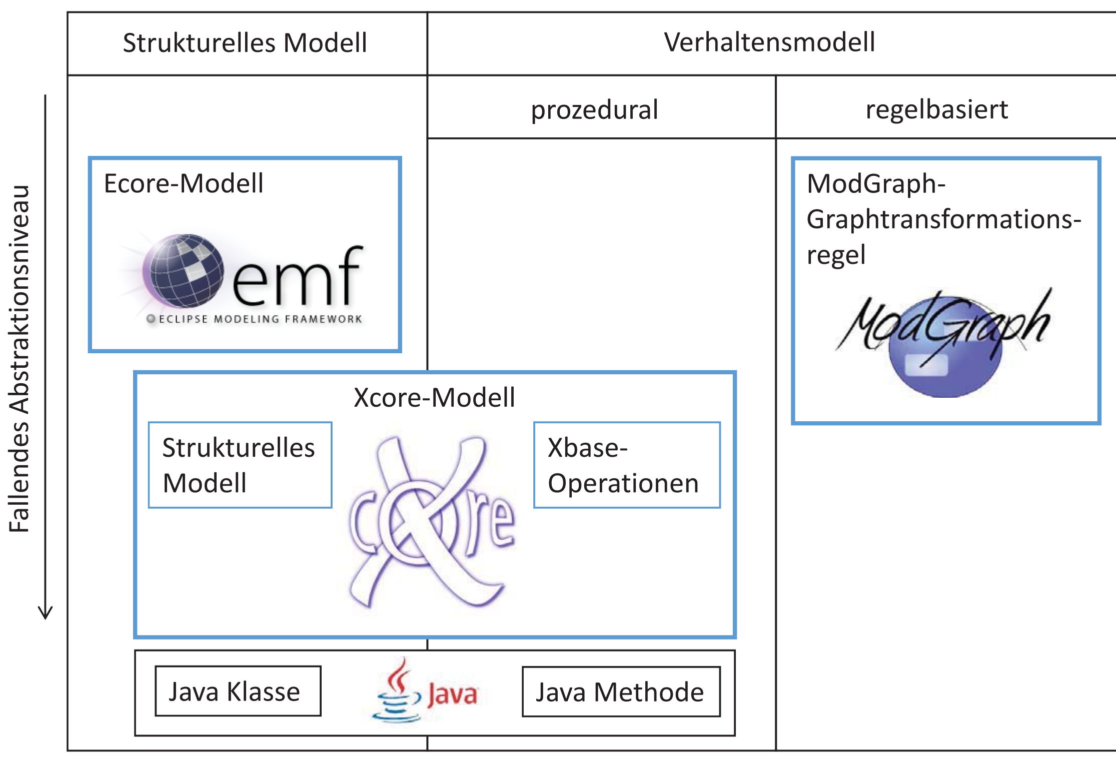


Abbildung 1.1: *Der ModGraph-Ansatz: Ein Ansatz, um mit bestehenden und neuen Werkzeugen totale modellgetriebene Softwareentwicklung für EMF zu erreichen*

Graphtransformationsregel kann zusätzlich durch Vor- und Nachbedingungen sowie negative Anwendbarkeitsbedingungen beschränkt werden.

Zur Ausführung einer durch eine Graphtransformationsregel definierten Operation existieren zwei Möglichkeiten: Zum Einen kann aus der Graphtransformationsregel Java-Quelltext generiert werden, der nahtlos in den von EMF oder Xcore generierten Code eingefügt wird. Zum Anderen kann aus einer Graphtransformationsregel Xcore-Code (bzw. Xbase-Code) erzeugt werden, der nahtlos in das bestehende Xcore-Modell eingefügt wird. Das entstandene Xcore-Modell kann dann seinerseits interpretiert oder in Java-Quelltext kompiliert werden.

Um dem Anspruch der totalen modellgetriebenen Softwareentwicklung gerecht zu werden, reichen jedoch die Graphtransformationsregeln nicht aus. Es können keine Kontrollflüsse dargestellt werden. Um dieses Defizit zu beseitigen, wird ein hybrider Ansatz zur Verhaltensmodellierung verfolgt. Dazu werden neue und bestehende Werkzeuge miteinander in Verbindung gebracht, wie in Abbildung 1.1 dargestellt ist. Die Komposition von Werkzeugen ergibt eine flexible und leichtgewichtige Umgebung, die eine strikte Trennung von Struktur und Verhaltensmodellierung anstrebt, indem jedem Werkzeug (-bestandteil) genau ein Verwendungszweck zugeordnet wird. Ecore-Klassendiagramme sowie der statische Anteil eines Xcore-Modells werden zur Strukturmodellierung verwendet. Durch die Einbindung von Xcore (und damit von Xbase) wird die Modellierung von Kontrollflüssen, stark prozeduralen oder sehr einfachen Operationen unterstützt. Da-

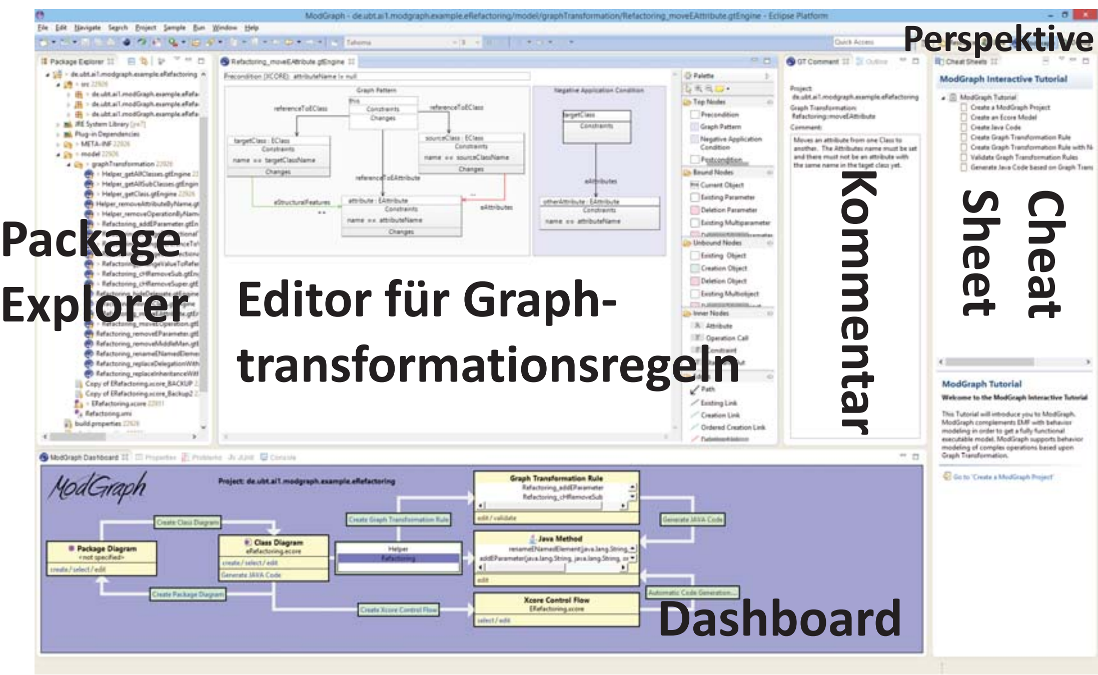


Abbildung 1.2: *Übersicht über das entstandene Werkzeug ModGraph*

mit entsteht eine zusätzliche strikte Trennung zwischen prozeduraler und regelbasierter Verhaltensmodellierung, die sich auf den Mehrwert der Graphtransformationsregeln konzentriert: Sie werden nur benutzt, wenn strukturelle Modifikationen von Modellinstanzen spezifiziert werden sollen.

Die Einbindung des am Lehrstuhl bereits vorhandenen Paketdiagrammeditors (Buchmann et al.[18]) ermöglicht zudem strukturelles Modellieren im Großen.

1.2.3 Praktische Umsetzung

Im Rahmen dieser Arbeit ist die Entwicklungsumgebung ModGraph entstanden, die es erlaubt, innerhalb von Eclipse und EMF, Software ausschließlich modellgetrieben zu entwickeln. Es handelt sich hierbei um die praktische Umsetzung des ModGraph-Ansatzes. Einen ersten Überblick gibt Abbildung 1.2. ModGraph bietet eine eigene Eclipse View. Im Editor werden Graphtransformationen erstellt, die im Kommentarreiter dokumentiert werden können. Der Eclipse Package Explorer dient der Übersicht. Ein Dashboard erleichtert den Arbeitsablauf und Cheat Sheets führen interaktiv in die Thematik ein.

ModGraph selbst wurde, soweit möglich, modellgetrieben mit EMF entwickelt: Das ModGraph-Metamodell ist eine Instanz des Ecore-Metamodells. Der Editor wurde mit Hilfe von EMF und des Graphical Modeling Framework (GMF) [37] erstellt. Die Validie-

rung von ModGraph basiert auf der aus openAchitectureWare (oAW)[8] entsprungenen Sprache Check. ModGraphs Java- und Xcore-Codegenerator basieren auf Xpand. Die Anbindung an die jeweiligen bestehenden Werkzeuge wurde in Java programmiert. Die Implementierung von ModGraph wird im Teil III dieser Arbeit genauer erläutert.

Für die Modellierung mit ModGraph haben sich durch die Anbindung an Ecore und Xcore fünf Wege aufgezeigt, wie Graphtransformationsregeln eingebunden werden können. Diese werden im Verlauf der Arbeit ausführlich dargestellt, sollen jedoch bereits hier kurz angedeutet werden (Die Namen der Wege ergeben sich aus ihrer Entstehung im ModGraph-Entwicklungsprozess):

1. **Der „klassische Weg“:** Ausgehend von einem Ecore-Klassendiagramm werden Graphtransformationsregeln und Java zur Verhaltensmodellierung verwendet.

2. **Der „neue Weg“:** Ausgehend von einem Xcore-Modell werden Graphtransformationsregeln sowie Xbase-Operationen zur Verhaltensmodellierung verwendet. Das Xcore-Modell wird in Java-Quelltext übersetzt, in den der aus den Regeln generierte Code nahtlos eingefügt wird.

3. **Der „Weg zum Neuen“:** Ein Ecore-Klassendiagramm wird in ein Xcore-Modell überführt. Danach ist es wiederum möglich, Xbase-Operationen zu verwenden. Die bereits bestehenden Regeln zum Ecore-Modell können weiterverwendet werden.

4. **Der „Weg zurück zu den Ursprüngen“:** Ein Xcore-Modell kann in ein Ecore-Klassendiagramm überführt werden, das wiederum wie im klassischen Ansatz verwendet werden kann. Der Unterschied zu diesem liegt in der Codegenerierung, da eventuell bestehendes Xbase-implementiertes Verhalten nicht verloren gehen darf.

5. **Der „unabhängige Weg“:** Die Graphtransformationsregeln können die Operationen eines Ecore-Modells, das in ein Xcore-Modell überführt wird, oder eines eigenständigen Xcore-Modells implementieren. Sie werden in Xcore-Code übersetzt und direkt in das Xcore-Modell eingefügt. Das so entstandene Xcore-Modell ist platformunabhängig insofern, dass es unabhängig von der letztendlich genutzten Programmiersprache ist. Es kann ebenso interpretiert werden.

Der für den EMF-Nutzer zunächst eingängige Weg zur totalen modellgetriebenen Softwareentwicklung ist der unter 3. aufgeführte. Hier wird die Struktur in Ecore modelliert. Die Regeln basieren auf den so definierten Operationen und Xcore wird lediglich als Kontrollflusssprache verwendet. Daher orientiert sich auch das Dashboard an dieser Vorgehensweise. Der unter 5. dargestellte Weg nimmt eine Sonderstellung ein. Die Regeln werden auf Modellebene mit dem strukturellen Modell und den prozeduralen Operationen verschmolzen. Damit ist nicht nur die Generierung von Quelltext, sondern auch die Interpretation der Xcore-Repräsentation der Regeln möglich. Grundsätzlich kann jedoch der Weg gewählt werden, der den Nutzer bei der Lösung seines Problems bestmöglich unterstützt.

[8] Die einzelnen Teilprojekte von oAW sind mittlerweile Teil des Eclipse Modeling Projects, dennoch wird hier oAW als Sammelbegriff für die Werkzeuge verwendet. Das Eclipse Modeling Project findet sich unter `http://eclipse.org/modeling/`

1.3 Einführendes Beispiel: Refactoring auf Ecore-Modellen

Bevor die ersten größeren Beispiele betrachtet werden können, wird zur Einführung der Konzepte zur Modellierung mit Graphtransformationsregeln in ModGraph das gängige Beispiel Refactoring von Ecore-Modellen betrachtet. Es handelt sich dabei um die Erweiterung einer Fallstudie zum Modellrefactoring, die im Rahmen der Bachelorarbeit von N. Dümmel entstand [27]. Sie beschäftigt sich mit dem Refactoring von Ecore-Klassendiagrammen. Alle zur Einführung abgebildeten Graphtransformationsregeln und Screenshots sind diesem Beispiel entnommen.

Die von Fowler definierten Refactoringregeln für Quelltext [33] werden dabei auf Modellebene übertragen und mit Hilfe von Ecore, ModGraph-Graphtransformationsregeln sowie Xcore modelliert. Dabei werden Graphtransformationsregeln genutzt, wenn sie einen Mehrwert für die Implementierung einer Operation bieten. Die so entstandenen Refactoringregeln für Klassendiagramme können in Eclipse eingebunden und auf beliebige Ecore-Klassendiagramme angewendet werden. In Kapitel 8 werden sie als Teil des propagierenden Refactorings wiederverwendet, das die Propagation von - durch Modell-Refactorings hervorgerufene - Änderungen auf Graphtransformationsregeln betrachtet.

1.4 Bestehende Arbeiten

In den letzten Jahren wurden verschiedene Ansätze und Werkzeuge zur Modelltransformation entwickelt. Sie divergieren beträchtlich. Einige führen endogene, überschreibende Modell-Transformationen basierend auf Graphtransformationen durch, andere sind auf exogene Modell-zu-Modell- oder Modell-zu-Text-Transformationen spezialisiert. Weitere Werkzeuge transformieren Texte. Sie nutzen das EMF-Rahmenwerk oder sind unabhängig von diesem. Aufgrund dieser Vielzahl von Werkzeugen werden hier nur die mit ModGraph in Verbindung stehenden Ansätze und Werkeuge betrachtet.

Prominente Vertreter der Modell-zu-Modell-Transformation sind ATL [45] und QVT [56]. Bei diesen handelt es sich um textuelle Sprachen. Sie dienen primär der Transformation zwischen Instanzen verschiedener Modelle. Im Gegensatz dazu strebt der hier präsentierte ModGraph-Ansatz eine Transformation einer bestehenden Modellinstanz an.

Modell-zu-Text-Transformationen sind in der Praxis weit verbreitet. Dabei werden meist Strukturdiagramme in Quelltext übersetzt. Das Verhalten bleibt unberücksichtigt. Ein prominenter Vertreter dieses Vorgehens ist EMF. Innerhalb des EMF-Rahmenwerks wird Java-Code aus einem Ecore-Klassendiagramm generiert. Dabei erzeugt der EMF-Codegenerator lediglich Methodensignaturen. Das Verhalten der modellierten Operationen muss der Benutzer in Java implementieren. An dieser Stelle setzt der hier vorgestellte ModGraph-Ansatz an: Er erweitert EMF um eine Möglichkeit, Operationen zu implementieren. Dazu werden Graphtransformationsregeln genutzt. ModGraph ist komplementär zu EMF, es erweitert EMF.

Xcore - die parallel zu ModGraph entstandene, textuelle Syntax für EMF - bietet neben einer textuellen Darstellung von Ecore-Klassendiagrammen eine Möglichkeit zur Verhaltensmodellierung. Bei der dazu verwendeten Sprache Xbase handelt es sich allerdings um eine textuelle und prozedurale Sprache, nicht - wie es bei ModGraph der Fall ist - um eine grafische und regelbasierte Sprache. Damit eignet sich Xbase sehr gut zur Kontrollflussmodellierung und zur Implementierung einfacher Operationen. Die Stärke von ModGraph ist hingegen die deklarative, regelbasierte und übersichtliche Implementierung komplexer struktureller Transformationen, die den durch eine Graphtransformationsregel gebotenen Mehrwert optimal nutzen.

Wie bereits angedeutet, basiert ModGraph auf einer Theorie zur Graphtransformation. Beispiele für weitere prominente Vertreter dieser Art sind PROGRES [70], Fujaba [83] und Henshin [7]. Einige dieser Werkzeuge sind in der EMF-Welt beheimatet. Dennoch unterscheidet sich der ModGraph-Ansatz von allen graphbasierten Ansätzen insbesondere durch seine leichtgewichtige Architektur: Er wurde *für* und *mit* EMF entwickelt und ist damit spezialisiert auf die EMF-Welt. Er ergänzt EMF um eine Möglichkeit, Verhalten zu modellieren. Zudem ist der hier präsentierte Ansatz der Einzige, der sich auf den Mehrwert der Graphtransformationsregeln konzentriert, um das EMF-Rahmenwerk zu verbessern. Die im ModGraph-Ansatz angestrebte Integration von Ecore, Graphtransformationsregeln und Xcore ermöglicht totale modellgetriebene Softwareentwicklung.

Bezüglich der Ausführbarkeit der Modelle werden Werkzeuge mit Interpreter- und Compilerlösungen angeboten. ModGraph verfolgt eine Compilerlösung. In allen anderen Werkzeugen mit Compilern werden die Graphtransformationsregeln in den Quelltext einer Programmiersprache übersetzt. ModGraph ermöglicht die Erzeugung von Xcore-Code, der sich auf einer höheren Abstraktionsebene als Quelltext einer Programmiersprache befindet. ModGraph bietet zwar keinen eigenen Interpreter an, jedoch ist das generierte Xcore-Modell interpretierbar, womit eine indirekte Interpretation der Graphtransformationsregeln gewährleistet ist. Da Xcore selbst wiederum Code erzeugt, verfolgt ModGraph durch die Generierung nach Xcore einen, bislang im Kontext der Graphtransformationsregeln nicht verfügbaren, stufenweisen Übersetzungsprozess der Regeln, der zur Unabhängigkeit von der generierten Zielprogrammiersprache führt.

Die Werkzeuge und Ansätze zur Graphtransformation werden in Teil II, Kapitel 4.2, der sich mit bestehenden Arbeiten beschäftigt, näher dargestellt. Eine detaillierte Abgrenzung zu dem hier vorgestellten Forschungsbeitrag findet sich in Teil III, Abschnitt 10.2.

1.5 Übersicht

Die vorliegende Arbeit ist in vier Teile gegliedert. Die Einführung in die Thematik wird mit dem Kapitel über Grundlagen zu Modellen und deren Transformation, sowie einer Einführung in die Graphtransformation und des Aufbaus von Sprachen fortgeführt.

Der zweite Teil beschäftigt sich mit bereits vorhandenen Ansätzen und Werkzeugen zur Modelltransformation. Er geht dazu auch auf verwandte Arbeiten ein. Nach einer Betrachtung des EMF-Rahmenwerks, werden - für den ModGraph-Ansatz relevante -

Transformationsprachen untersucht.

Im dritten Teil der Arbeit wird der Forschungsbeitrag dieser Dissertation erläutert. Zunächst wird die Einbindung der Graphtransformationsregeln in das EMF-Rahmenwerk auf konzeptioneller Ebene untersucht und ein erstes größeres Beispiel eingeführt. Es folgt eine detaillierte Darstellung der Architektur des entstandenen Werkzeugs für die ModGraph-Graphtransformationsregeln. Dazu wird die in dieser Arbeit erstellte Sprache für ModGraph-Regeln erläutert und deren Integration in das EMF-Rahmenwerk betrachtet. Neben der Erläuterung der statischen und dynamischen Semantik der Sprache wird ebenso eine Beschreibung der ModGraph-Generatoren gegeben. Außerdem wird die Integration in Eclipse (inklusive Perspektive) Editor und angebotener Unterstützung bei der Modellierung gezeigt.

Im vierten und letzten Teil wird die Arbeit evaluiert. Dazu wird das propagierende Refactoring, ein reflektiver Ansatz erläutert, der Refactorings auf Ecore-Modellen und deren Propagation auf die Regeln *für* und *mit* ModGraph erlaubt. Zudem wird die komplementäre Integration mit einem szenarienbasierten Werkzeug beschrieben. Dabei werden die Regeln in Verbindung mit modalen Sequenzdiagrammen genutzt und es wird die Simulation der so modellierten Echtzeitsysteme mittels Play-out betrachtet. Zum Abschluss dieses Teils wird das Werkzeug ModGraph evaluiert, diskutiert und von anderen Werkzeugen abgegrenzt. Zuletzt werden die Ergebnisse der Arbeit zusammengefasst und zukünftige Erweiterungsmöglichkeiten betrachtet.

2 Grundlagen

2.1 Überblick

Dieses Kapitel stellt die Grundlagen, die zum Verständnis des ModGraph-Ansatzes nötig sind, dar. Es beschäftigt sich dazu einerseits mit Modellen, andererseits mit Graphen. Beginnend mit den Modellen wird der Frage einer adäquaten Modelldefinition nachgegangen sowie die Frage, wie ein in der modellgetriebenen Softwareentwicklung verwendetes Modell klassifiziert werden kann, beantwortet. Im Folgenden wird geklärt, was unter einem Metamodell zu verstehen ist und wie Beziehungen zwischen Modellen im Bezug auf Metaebenen charakterisiert werden. Die Betrachtung von Modellierungssprachen bildet den nächsten Abschnitt. Hier wird zunächst erläutert, wie sich Syntax- und Semantikdefinition gestalten. Anschließend wird die Verbindung zwischen Modellen und Sprachen hergestellt, indem gezeigt wird, wie sich diese Definitionen als (Meta-)Modell interpretieren lassen. Der Begriff modellgetriebene Softwareentwicklung wird im Anschluss erläutert. Der folgende Abschnitt widmet sich der Erklärung der Transformationen. Dabei werden Kriterien zur Klassifikation einer Transformation aufgezeigt, wie z.B. die Realisierung der Transformation und der Aufbau von Transformationsregeln. Den Abschluss des Grundlagenkapitels bildet der Abschnitt zur Graphtransformation. Dabei wird unter Anderem der Begriff Graph definiert und erläutert, was im Allgemeinen unter einer Graphtransformation zu verstehen ist. Zudem werden die drei gängigen theoretischen Ansätze zur Graphtransformation skizziert.

Als Literaturgrundlage für den Modellteil dieses Kapitels dienen Czarnecki [23], Kühne [47], Stachowiak[71] und Stahl et al.[72]. Zur Erklärung von Graphen und deren Transformationsmöglichkeiten werden Ehrig [30], Rozenberg citeRozenberg1997, Schürr [68] und Schürr / Westfechtel [69] als Literaturquellen genutzt.

2.2 Modelle

2.2.1 Der Modellbegriff

Befragt man Menschen, was ein Modell sei, werden je nach persönlichem Hintergrund verschiedene Antworten gegeben: maßstabsgetreue Nachbauten realer Gegenstände, Vorleben von Verhaltensweisen, vereinfachte Darstellungen komplexer Sachverhalte, Konstruktionspläne, usw.

Eine Systematisierung dieser sehr unterschiedlichen Meinungen bietet Herbert Stachowiak. Als Vorreiter der Modelltheorie ordnete er Modellen gemeinsame Eigenschaften zu, die er in seinem Standardwerk „Allgemeine Modelltheorie“ [71] vorstellt. Dazu betrachtet

er Modelle als Abbilder der Wirklichkeit, die diese auf notwendige Weise verkürzen. Die genaue Ausprägung des Modells hängt dabei vom Modellierer und dessen Arbeitsgebiet ab, jedoch erfüllen die Modelle immer drei Merkmale:

1. Abbildung
   (Jedes Modell ist eine Abbildung eines natürlichen oder künstlichen Originals.)

2. Verkürzung
   (Ein Modell vereinfacht das Original. Es werden nicht alle Eigenschaften übernommen, nur diejenigen, die für den Modellnutzer relevant sind.)

3. Pragmatismus
   (Modelle erfüllen ihre Ersatzfunktion hinsichtlich der, im betrachteten Fall, relevanten Eigenschaften des Originals. Dabei kann ein Modell stellvertretend für verschiedene Originale stehen. Genauso kann es für ein Original mehrere Modelle geben, wobei jedes Modell einem anderen Zweck dient.)

Unter Berücksichtigung der Arbeiten von Stachowiak und im Kontext der modellgetriebenen Softwareentwicklung definiert Kühne in [47] ein Modell folgendermaßen:

Ein Modell ist eine Abstraktion eines (realen oder sprachbasierten) Systems, mit Hilfe dessen Vorhersagen oder Rückschlüsse gezogen werden können. [47]

Stachowiaks allgemeine Merkmale von Modellen bleiben erhalten, jedoch ist zu berücksichtigen, dass sein „Original“ Kühnes „System“ entspricht. Es wird in der modellgetriebenen Softwareentwicklung als virtuelles Produkt und damit als fertige Software, betrachtet. Diese fertige Software kann wiederum als Modell angesehen werden, scharfe Grenzen zwischen Modell und System sind damit inexistent. Dementsprechend kann, innerhalb dieses Kontexts, Jean Bézivin recht gegeben werden:

Alles ist ein Modell [13].

Für den weiteren Verlauf dieses Dokuments ist Bézivins Modelldefinition jedoch zu generalisiert. Sofern der Begriff Modell im Folgenden verwendet wird, gilt Kühnes Definition des Modellbegriffs.

2.2.2 Klassifikation von Modellen in der modellgetriebenen Softwareentwicklung

Zur Nutzung eines Modells in der modellgetriebenen Softwareentwicklung ist eine weitere Konkretisierung des Modellbegriffs nötig. Abbildung 2.1 zeigt Kriterien zur Klassifikation, die im Folgenden besprochen werden.

Das Modell als abstrahiertes System aufzufassen, ist bereits aus der Definition nach Kühne im vorangegangenen Abschnitt ersichtlich. Dabei erfasst das Modell entweder einen realen Sachverhalt oder die universellen Eigenschaften eines Systems.

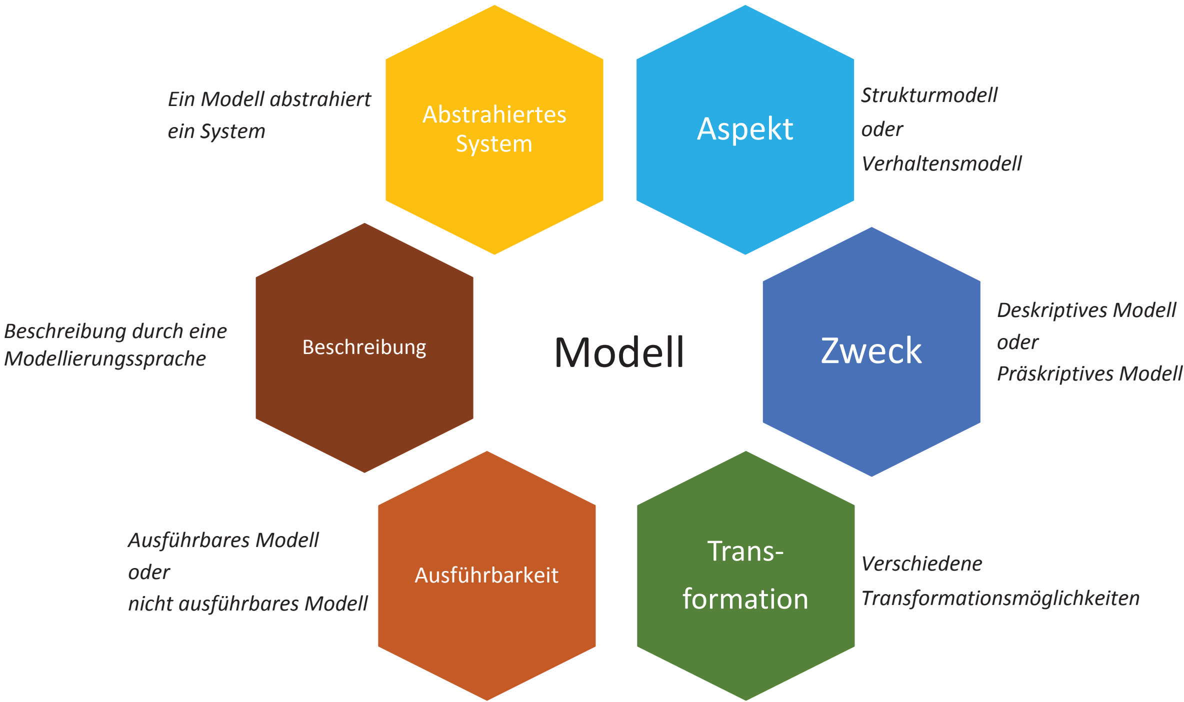


Abbildung 2.1: *Klassifikation von Modellen in der modellgetriebenen Softwareentwicklung*

Der Aspekt, unter welchem ein System beschrieben wird, ist ebenfalls ausschlaggebend. Dabei werden Strukturmodelle und Verhaltensmodelle unterschieden.

Strukturmodelle bilden die statischen Eigenschaften eines Softwaresystems ab. Sie definieren die Architektur der Software, indem die Elemente eines Systems und deren statische Beziehungen zueinander modelliert werden. Häufige Vertreter sind Klassendiagramme und Paketdiagramme.

Verhaltensmodelle bilden das dynamische Verhalten der Software ab. Dabei unterscheidet man klassisch zwischen Ablauf- und Zustandsmodellen. Ablaufmodelle geben den zeitlichen Ablauf einer Systemveränderung an. Typische Beispiele sind Kommunikations- und Sequenzdiagramme der UML. Zustandsmodelle zeigen den Zustand eines Modells und gegebenenfalls dessen Änderungen. Beispiel hierfür sind Zustandsdiagramme der UML. Im Rahmen des ModGraph-Ansatzes werden zudem regelbasierte Modelle genutzt. Diese basieren auf Graphtransformationsregeln und stellen ein spezielles Zustandsmodell dar.

Der Zweck eines Modells beschreibt, ob es deskriptiver oder präskriptiver Natur ist. Deskriptive Modelle beschreiben ein existierendes System. Ein Beispiel hierfür ist ein - zu Dokumentationszwecken - anhand einer existierenden Software erstelltes Klassendiagramm. Präskriptive Modelle beschreiben ein zu erstellendes System, beispielsweise die zu erstellende Software. Dabei entsteht das Modell vor dem Produkt. [47]

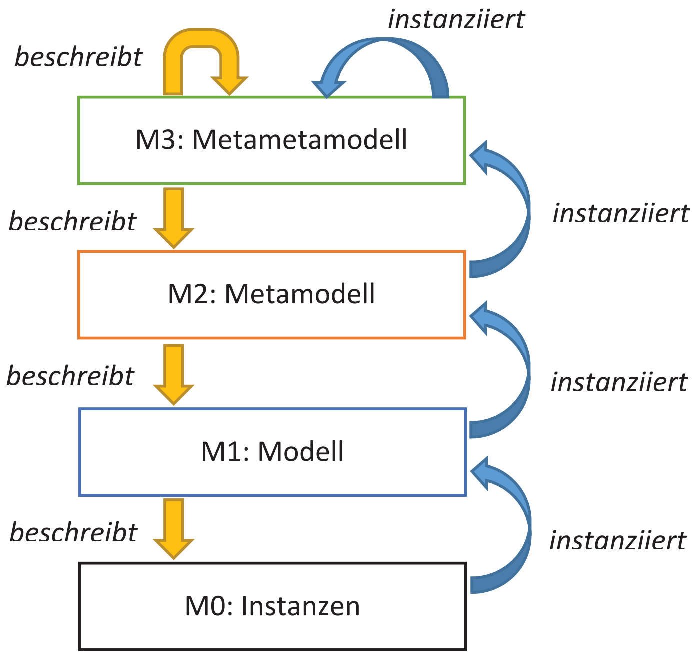


Abbildung 2.2: *Metaebenen der OMG (analog zu [72])*

Transformationen bieten eine Möglichkeit Modelle und Texte, die Modelle repräsentieren, ineinander zu überführen. Sie werden in Kapitel 2.5 ausführlicher besprochen.

Beschreibung und Ausführbarkeit eines Modells sind eng miteinander verknüpft. Die Beschreibung eines Modells erfolgt mittels einer Modellierungssprache, deren Syntax wohldefiniert ist. Eine wohldefinierte Semantik der Sprache ist wünschenswert, jedoch nur zwingend, wenn das Modell ausgeführt werden soll. In Kapitel 2.3 werden die Modellierungssprachen, im Folgenden kurz Sprachen genannt, näher betrachtet.

2.2.3 Metaebenen von Modellen

Eine weiteres Merkmal eines Modells, das bezüglich der Modellierung betrachtet werden muss, ist die Metaebene, auf der sich ein Modell befindet. Hier gilt:

Ein Modell von Modellen ist ein Metamodell. [47]

Das Metamodell definiert demnach die Konzepte, mittels derer die Modelle erstellt werden. Es beschreibt, welche Elemente in den instanziierenden Modellen vorkommen und wie diese miteinander in Beziehung stehen. Zusätzlich können im Metamodell Bedingungen definiert werden, die das Modell einzuhalten hat. Zusammenfassend ist der Zweck des Metamodells die zugehörigen Modelle konform zu halten. [72]

Durch die Metaebenen kann eine Modellhierarchie aufgebaut werden. Diese hierarchischen Beziehungen zwischen Modellen und deren Metamodell sind Instanz-Beziehungen.

Diesbezüglich hat die OMG Metaebenen für Modelle definiert, die in Abbildung 2.2 dargestellt sind. Referenzpunkt ist das Modell, aus welchem der Quelltext generiert wird. Es wird mit M1 bezeichnet. Die Instanzebene M0, die reale Objekte repräsentiert, bildet die unterste Ebene. Diese Ebene wird auch Datenebene genannt und kann nicht weiter instanziiert werden. Die Instanzen M0 instanziieren ein Modell M1. M1 selbst ist eine Instanz seines Metamodells M2. Dies bedeutet, dass M2 die Konzepte vorgibt, die in M1 benutzt werden dürfen, um M0 zu beschreiben. Diese Konzepte sind Instanzen des Metamodells von M2, hier M3 genannt. Hier zeigt sich ein interessanter Fallstrick bei der Definition des Begriffs Metamodell: M2 nimmt einerseits die Rolle des Metamodells von M1 ein und ist gleichzeitig ein Modell von M3, hat also wiederum ein Metamodell. Der Begriff des Metamodells ist daher keinesfalls absolut zu verstehen. Genauso wenig darf er im Sinne einer Generalisierung verstanden werden. Das Modell instanziiert immer sein jeweiliges Metamodell: M1 ist Instanz von M2, M2 Instanz vom M3. Betrachtet man die Beziehung eines Modells zu dem Metamodell seines Metamodells, spricht man von dem Metametamodell des Modells. M3 ist das Metametamodell von M1.

Durch Hinzufügen weiterer Meta-Präfixe ließe sich so eine beliebige Modellhierarchie aufbauen. Dabei kann diese theoretisch auf beliebiger (Meta-)Ebene begonnen werden. In der Praxis ist es meist üblich, nicht über die Metametaebene hinauszugehen. So ist M3 in sich selbst definiert. In einigen praktischen Anwendungen sind bereits drei Ebenen ausreichend. In diesem Fall wird eine Modellhierarchie aufgebaut, innerhalb derer sich das Metamodell M2 selbst definiert.

Prominente Beispiele der Selbstdefinition sind MOF und das bereits in der Einleitung erwähnte Ecore [73]. Im Falle der Ecore-Definition handelt es sich um eine dreistufige Modellhierarchie: Ecore ist sein eigenes Metamodell, also in sich selbst definiert. Modelle sind Instanzen von Ecore, die wiederum instanziiert werden können. Dies wird in Abschnitt 3.4 genauer betrachtet. Die OMG-Definition der Metaebenen wird in den folgenden Kapiteln genutzt.

2.3 Sprachen im Rahmen der Softwareentwicklung

2.3.1 Aufbau einer Sprache

Jede natürliche Sprache besteht aus Sätzen. Jeder Satz ist nach bestimmten Regeln aufgebaut, die durch Worte, deren Bedeutung und die Grammatik dieser Sprache vorgegeben sind.

Analog verhält es sich mit den Sprachen der Softwareentwicklung. Um mit einer Sprache modellgetrieben entwickeln zu können, gibt man fünf Schichten zur Definition der Sprache an. Dabei spielt es nur bedingt eine Rolle, ob diese Sprache grafisch oder textuell ist. Die Schichten werden im Folgenden vorgestellt und sind in Abbildung 2.3 dargestellt. Dabei werden zunächst nur die Sechsecke, welche die Sprachdefinition repräsentieren und die erläuternden, kursiven Texte betrachtet. Die weiteren Bestandteile der Abbildung dienen der Einordnung einer Sprache in die Modellierung und werden in Abschnitt 2.3.2 erklärt. Grundsätzlich ist jeder Sprache eine eigene Syntax und eine

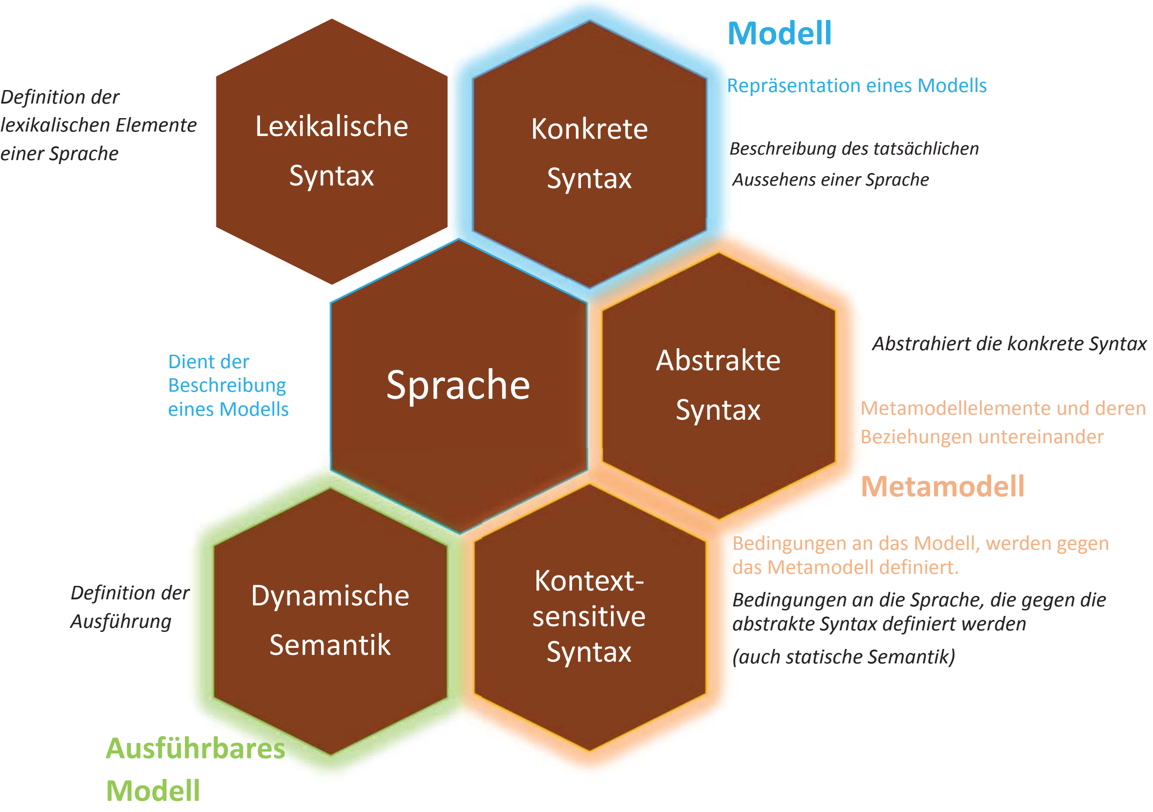


Abbildung 2.3: *Definition einer Sprache und deren Einordnung in die Modellierung*

eigene Semantik zugeordnet. Beide lassen sich, wie folgt, weiter aufspalten.

Die lexikalische Syntax gibt die lexikalischen Elemente einer Sprache vor, beispielsweise Schlüsselwörter einer Sprache. Dabei werden die einzelnen lexikalischen Elemente auf ihre Validität geprüft. Die lexikalische Syntax wird oft durch reguläre Ausdrücke dargestellt.

Die konkrete Syntax beschreibt das tatsächliche Aussehen einer Sprache. Durch sie entscheidet sich, ob die Sprache textuell oder grafisch gestaltet wird. Die konkrete Syntax wird im textuellen Fall meist durch kontextfreie Grammatiken festgelegt. Im Falle einer grafischen Sprache werden oft Beispieldiagramme genutzt.

Die abstrakte Syntax abstrahiert die konkrete Syntax. Dabei werden konkrete Schlüsselwörter eliminiert und damit nur noch die Konzepte erhalten. Demnach können beliebig viele konkrete Syntaxen zu einer abstrakten Syntax definiert werden. Die abstrakte Syntax wird im textuellen Fall durch Baumgrammatiken oder Klassendiagramme festgelegt. Grafisch sind mehrheitlich Klassendiagramme zur Definition möglich.

Die kontextsensitive Syntax, auch statische Semantik genannt, legt Bedingungen an

die Wohlgeformtheit der Sprache fest. Diese Bedingungen werden auch Constraints genannt. Sie werden gegen die abstrakte Syntax definiert und sind somit von ihr abhängig. Die kontextsensitive Syntax wird oft mittels Attributgrammatiken oder OCL-Ausdrücken vorgegeben. Diese werden in der Object Constraint Language (OCL) formuliert.[1] Man spricht in diesem Zusammenhang auch von kontextsensitiver Korrektheit einer Sprache.

Die dynamische Semantik definiert das Verhalten des Programms zur Laufzeit. Sie ist optional. Sobald jedoch eine dynamische Semantik existiert, ist die Sprache ausführbar. Die dynamische Semantik wird durch eine Funktionssemantik (denotationelle Semantik), durch eine operationale Spezifikation oder durch einen Text, geschrieben in einer natürlichen Sprache, festgelegt.

Nach dieser Definition stellt sich die Frage, wann ein Ausdruck, also ein programmiersprachlicher Satz, als „zu einer Sprache gehörig“ eingestuft werden kann. Dazu muss dieser syntaktisch korrekt sein. Ist eine dynamische Semantik definiert, muss er auch dieser genügen.

2.3.2 Einordnung einer Sprache in die Modellierung

Im vorangegangenen Abschnitt wurde gezeigt, wie eine Sprache definiert wird. Davon ausgehend werden in diesem Abschnitt Parallelen zu Modellen gezogen. Abbildung 2.3 zeigt, wie im vorangegangenen Abschnitt besprochen, die einzelnen Ebenen der Sprachdefinition. Zudem sind die zugehörigen Modellebenen farblich markiert. Diese werden hier gesondert besprochen. Dabei dient [72] als Literaturquelle.

Die abstrakte Syntax enthält die Konzepte der Sprache. Im Sinne der Modellierung sind die Konzepte im Metamodell angesiedelt. Damit entspricht die abstrakte Syntax den Metamodellelementen und ihren Beziehungen untereinander. Um den Begriff des Metamodells vollständig auf eine Sprache abzubilden, fehlt die Komponente der Bedingungen. Diese findet sich in der kontextsensitiven Syntax, da sie Bedingungen an die Wohlgeformtheit der Modelle festlegt. Die Bedingungen werden an die Elemente des Metamodells und deren Beziehungen definiert. Sie sind damit Bedingungen an das Modell und dienen insbesondere der Vermeidung von Modellierungsfehlern. Ein typischer Vertreter unter den Modellierungssprachen sind OCL-Bedingungen, die mit Hilfe von Metamodellelementen deklariert werden. Zusammengefasst gilt demnach:

Die abstrakte und die kontextsensitive Syntax bilden das Metamodell der Sprache.

Die konkrete Syntax dient als „konkrete Ausprägung der textuellen oder grafischen Konstrukte, mit denen modelliert wird“[72]. Sie bildet die Schnittstelle zum Modellierer, indem sie ihm die zur Verfügung stehenden Konstrukte vorgibt. Diese Konstrukte sind in der Modellierung im Modell angesiedelt. Damit kann folgende Parallele gezogen werden:

Die konkrete Syntax entspricht der Repräsentation eines Modells.

[1] Nähere Informationen zu OCL finden sich unter `http://www.omg.org/spec/OCL/2.3.1/`.

Betrachtet man nun die dynamische Semantik, kann eine Aussage über die Ausführbarkeit von Modellen gemacht werden:

Sobald die Sprache ausführbar ist, ist es auch das Modell.

Ein solches Modell nennt man *formales Modell*. Dies bedeutet einerseits, dass ein Modell, welches durch eine Sprache mit definierter dynamischer Semantik beschrieben wird, immer ausführbar ist. Andererseits stellt ein formales Modell „irgendeinen Aspekt der Software vollständig"[72] dar, d.h. es ist klar festgelegt, worüber mit Hilfe des Modells Aussagen getroffen werden können.

2.3.3 Domänenspezifische und allgemeine Sprachen

Nachdem die Eigenschaften von Sprachen definiert sind und die Brücke zu den Modellen geschlagen wurde, werden in diesem Abschnitt die Sprachen als Ganzes näher betrachtet. Programmiersprachen und Modellierungssprachen sind heute fester Bestandteil der Softwareentwicklung. Sie agieren strikt nach der in Abschnitt 2.3.1 beschriebenen Definition. Beide können entweder zur Gruppe der allgemeinen Sprachen oder der Domänenspezifischen Sprachen, kurz DSLs genannt, zugeordnet werden. Ihre Darstellung ist textuell, grafisch oder hybrid. Hybrid bedeutet hier, dass die betreffende Sprache sowohl textuelle als auch grafische Elemente aufweist.

Eine allgemeine Sprache kann zu jedem Zweck verwendet werden. Typische Vertreter hierfür sind Java und alle gängigen C-Dialekte.

Eine domänenspezifische Sprache (DSL) wurde für eine bestimmte Domäne entwickelt und wird innerhalb dieser vom einem Domänenspezialisten eingesetzt. DSLs sind dementsprechend spezialisierter als allgemeine Programmiersprachen. Sie besitzen in der Regel eine wohldefinierte Syntax und Semantik und sind oft ausführbar, indem sie auf allgemeine Programmiersprachen abgebildet oder interpretiert werden. DSLs sind damit sehr gut zur modellgetriebenen Softwareentwicklung geeignet. Im Hinblick auf die Definition der modellgetriebenen Softwareentwicklung, kann der Begriff des formalen Modells definiert werden:

„Ein formales Modell ist ein Modell, das in einer DSL geschrieben wurde."[72]

Diese Definition wird im folgenden Abschnitt für die Definition der modellgetriebenen Softwareentwicklung benötigt.[2]

2.4 Modellgetriebene Softwareentwicklung

Thomas Stahl et al. definieren in ihrem Buch „Modellgetriebene Softwarentwicklung" den gleichnamigen Begriff wie folgt:

[2] Es sei hierbei erwähnt, dass ein formales Modell ebenso in einer allgemeinen Sprache verfasst werden darf.

„Modellgetriebene Softwareentwicklung (Model Driven Software Development) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen.“ [72]

Die modellgetriebene Softwareentwicklung zielt darauf ab, den Entwicklungsprozess zu vereinfachen und die Softwarequalität zu steigern, indem Software auf einer höheren Abstraktionsebene erstellt wird. Dabei soll nicht nur die Entwicklungsgeschwindigkeit der Software erhöht, sondern auch eine einheitliche Softwarearchitektur verfolgt werden. Dies geschieht durch eine einheitliche Übersetzung aller Komponenten des Modells.

Um diese ambitionierten Ziele verfolgen zu können, soll zunächst die Definition der modellgetriebenen Softwareentwicklung näher betrachtet werden. Diese wirft folgende Fragen auf: *„Was ist ein formales Modell?“*, *„Wie kann man lauffähige Software erzeugen?“* und *„Wie funktioniert diese Automatisierung?“*.

Die Frage nach den Eigenschaften eines formalen Modells wurde aufgrund ihrer Komplexität bereits in den vorangegangenen Abschnitten erklärt, da zunächst der Modellbegriff selbst geklärt werden musste.

Ausführbare Modelle werden durch Generierung oder Interpretation zur Ausführung gebracht. Ein Generator erzeugt Quelltext einer konventionellen Programmiersprache (z.B. Java, C#) aus dem Modell. Dieser Quelltext kann ausgeführt werden. Der Generator ist, wie auch der Compiler, Teil des Build-Prozesses der Software. Ein Interpreter ist eine Software, die ein Modell zur Laufzeit einliest und dabei abhängig von dessen Inhalt verschiedene Aktionen ausführt. Die entscheidende Gemeinsamkeit ist, dass beide das Modell auf dem Rechner ausführbar machen und damit Software lauffähig zu machen.[72]

Automatisierung bedeutet, dass das Modell nicht als Spezifikation zur manuellen Implementierung oder Dokumentation verwendet wird, sondern die Rolle des Quelltextes einnimmt. Die bei einer Generator-Lösung erzeugten Quelltexte sind lediglich temporäre Artefakte, eine Art Build-Zwischenprodukte. Änderungen werden immer an den Modellen vorgenommen und auf den Quelltext propagiert. Damit sind Modell und generierter Quelltext immer einheitlich und aktuell. [72]

2.5 Modelltransformationen

Transformationen sind ein wichtiger Bestandteil der Softwareentwicklung. „In erster Linie dienen Modelltransformationen dazu, Lücken zwischen verschiedenen Abstraktionseben zu überbrücken.“, stellen Stahl et al. in [72] im Bezug auf die modellgetriebene Softwareentwicklung fest. Im Folgenden werden die für diese Arbeit wichtigen Merkmale besprochen. Sie sind in Abbildung 2.4 dargestellt. Dabei wird gezeigt, was - unter welchen Umständen - transformiert werden kann und wie dies vonstatten geht. Ebenso werden die Beziehungen zwischen Quelle und Ziel einer Transformation erläutert. Der Artikel „Feature-based Survey of Model Transformation Approaches“ [23] von Czarnecki und Helsen dient als Literaturgrundlage zur Klassifikation von Transformationen. Für eine detailliertere Darstellung sei ebenfalls auf diesen Artikel verwiesen.

Abbildung 2.4: *Klassifikation von Modelltransformationen*

Realisierungen von Transformationen können auf Modellen oder Texten, die in der Regel Modelle repräsentieren, basieren. Hierbei unterscheidet man vier Transformationsarten: Text-zu-Text, Text-zu-Modell, Modell-zu-Text und Modell-zu-Modell.

Text-zu-Text-Transformationen überführen einen beliebigen Text in einen anderen Text. Text-zu-Modell-Transformationen analysieren einen Text und rekonstruieren daraus ein Modell.

Modelltransformationen werden im Folgenden genauer besprochen, da sie zum Verständnis des ModGraph-Ansatzes beitragen.

Modell-zu-Modell-Transformationen werden häufig genutzt, um ein Modell in ein Anderes zu überführen. Dabei gilt das allgemeine Modelltransformationsmuster aus Abbildung 2.5. Quellmodell und Zielmodell, in der Abbildung blau umrandet, sind Instanzen ihrer Metamodelle (gelb umrandet). Die Transformation wird ebenfalls als Modell aufgefasst (daher ist sie ebenfalls blau umrandet) und ist konform zu ihrem Transformations-Metamodell (gelb umrandet). Es beschreibt die abstrakte Syntax der Transformationssprache. Die Metamodelle instanziieren alle das selbe Metametamodell (grün umrandet), das sich selbst instanziiert. Damit geht die Transformationsdefinition konform mit den von der OMG definierten Metaebenen. Die Ausführung einer Transformation ist durch ein oranges Oval dargestellt. Sie interagiert mit den Modellen, indem das Quellmodell gelesen wird und gemäß der Transformationsvorschriften des Transformationsmodells in das Zielmodell übersetzt wird [45]. Dabei benutzt das Transformationsmodell Elemente aus Quell- und Ziel-

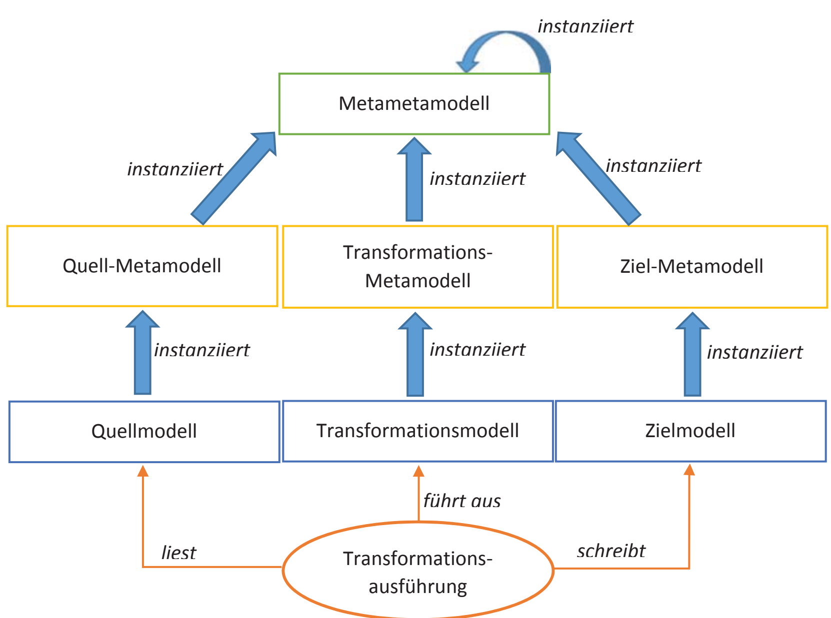


Abbildung 2.5: *Allgemeines Transformationsmuster der Modell-zu-Modell-Transformation (analog zu [45])*

metamodell, um die Transformationsregeln aufzubauen.

Modell-zu-Text-Transformationen überführen Modelle in Texte. Dabei wird für jedes Modellelement der korrespondierende Text erzeugt. Dies geschieht in der Praxis durch Templates. Die Transformation nutzt Metamodellelemente des Quellmodells und schreibt konkrete Elemente des Zieltextes. Auf diese Art und Weise wird oft die eingangs zitierte „Lücke zwischen verschiedenen Abstraktionsebenen“ geschlossen.

Regeln, auch Transfomationsregeln genannt, beschreiben die kleinsten Einheiten einer Transformation. Die zum Verständnis dieser Arbeit nötigen Eigenschaften einer Regel, wie deren Domäne und Anwendbarkeitsbedingungen, werden hier besprochen. Für eine ausführlichere Besprechung der Eigenschaften sei wiederum auf [23] verwiesen.

Domänen sind fester Bestandteil einer Regel. Eine Domäne beschreibt, wie auf ein Modellelement zugegriffen werden kann. Normalerweise existieren innerhalb einer Regel eine Quell- und eine Zieldomäne, aber auch mehrere Domänen sind möglich. Letzteres tritt zum Beispiel beim Verschmelzen von Modellen auf. Dabei sind mehrere Quellmodelle zu einem Zielmodell zusammen zu führen. Die hier wichtigen Eigenschaften einer Domäne sind deren Sprache und Muster.

Die Sprache einer Domäne beschreibt die möglichen Strukturen eines Modells innerhalb dieser. Sie entspricht, wie in 2.3, der Beschreibung des Modells. Eine Transformation, deren Quell- und Zielmodell Instanzen des selben Metamo-

dells sind, wird als *endogene Transformation* bezeichnet, wohingegen solche, deren Quell- und Zielmodell nicht dem selben Metamodell entsprechen, als *exogene Transfomationen* bezeichnet werden.

Muster innerhalb einer Domäne stellen Fragmente eines Modells dar. Dabei werden String-, Term-, und Graphmuster unterschieden. Stringmuster werden in Texttransformationen verwendet, Term- und Graphmuster in Modell-zu-Modell-Transformationen. Sie können durch die abstrakte oder konkrete Syntax der jeweils zugehörigen Quell- und Zielsprache repräsentiert werden. Muster können grafisch oder textuell dargestellt werden, je nach Definition der konkreten Syntax

Die Multidirektionalität einer Regel gibt an, ob diese in verschiedenen Richtungen ausgeführt werden kann, sprich, ob von Quell- zu Zielmodellen und umgekehrt transformiert werden kann.

Anwendbarkeitsbedingungen sind ebenfalls Bestandteil einer Regel. Dabei können positive und negative Anwendbarkeitsbedingungen angegeben werden. Diese müssen zur Regelausführung zwingend erfüllt sein.

Intermediäre Strukturen entstehen, wenn eine Regel zu ihrer Ausführung Zusatzinformationen benötigt. Diese Strukturen sind oft temporär und können nicht in den zu transformierenden Modellen gespeichert werden.

Quell- und Zielbeziehungen einer Transformation setzen ihre Quelle und ihr Ziel bezüglich der Sprache und des Modells in Beziehung.

Wird ein Modell in ein anderes Modell überführt, spricht man von einer *Ein-Ausgabe-Transformation*. Sind die Metamodelle von Quelle und Ziel verschieden, so spricht man von einer *exogenen Ein-Ausgabe-Transformation*, sind die Metamodelle gleich, von einer *endogenen Ein-Ausgabe-Transformation*. Endogene Transformationen bleiben innerhalb der selben Sprache, da sie der selben kontextfreien Syntax folgen.

Wird ein Modell durch Ausführung einer Transformation verändert, spricht man von einer *überschreibenden Transformation*. Es erfolgt eine Aktualisierung des bestehenden Modells durch diese Transformation.

Die Richtung einer Ein-Ausgabe-Transformation gibt an, ob die Transformation uni- oder multidirektional ist. Ist eine Transformation gerichtet, so sind es alle Regeln innerhalb dieser. Multidirektionale Transformationen sind in der Praxis meist bidirektional.

Unidirektionale Transformationen sind solche, die nur in eine Richtung erfolgen, die also lediglich aus Quellmodellen Zielmodelle erzeugen. Zielmodelle können weiter bearbeitet und modifiziert werden, die Änderungen haben aber keine unmittelbare Auswirkung auf das oder die Quellmodelle.[54]

Multidirektionale Transformationen sind im Gegensatz dazu in beiden Richtungen möglich. Es erfolgt eine Kopplung der Modelle, so dass Änderungen vom Zielmodell

unmittelbar im Quellmodell nachvollzogen werden. [54]

Im Rahmen der strikt modellgetriebenen Softwareentwicklung werden schwerpunktmäßig unidirektionale Vorwärtstransformationen benötigt.

Die Granularität einer Transformation bestimmt, ob es sich um eine Batchtransformation oder eine inkrementelle Transformation handelt. Eine inkrementelle Transformation ist in der Lage, ein Update auf einem bereits bestehenden Zielmodell auszuführen. Dabei können benutzerdefinierte Änderungen im Zielmodell erhalten oder überschrieben werden. Bei inkrementellen Transformationen spielt die Nachvollziehbarkeit der Änderungen durch Traces der Transformation eine Rolle. Eine Trace speichert die Vorgehensweise bei der Transformation. Dabei wird festgehalten, wie jedes Element des Quellmodells auf eines des Zielmodells abgebildet wurde. Für die genaue Funktion von Traces sei auf [23] verwiesen. Eine Batchtransformation generiert das Zielmodell jedes Mal komplett neu.

Spezifikationen beschreiben formale Eigenschaften von Transformationen. Ein Beispiel für eine Spezifikation sind Vor- und Nachbedingungen in OCL, die an die Transformation gestellt werden. Nur einige Transformationsansätze besitzen einen zugehörigen Spezifikationsmechanismus.

Zusammengefasst beschreiben Transformationen, wie ein Element der Quelldomäne in die Zieldomäne überführt wird. Dabei ist es erlaubt, dass Quell- und Zieldomäne identisch sind. Die Regeln können modular sein und unter Umständen wiederverwendet werden. Es wurde gezeigt, welche Eigenschaften Transformationsregeln erfüllen. Sie werden in einer DSL verfasst, die den in Abschnitt 2.3 besprochenen Eigenschaften unterliegt. Die Frage nach der konkreten Syntax einer Regel bleibt bislang unberücksichtigt. Beispiele hierfür finden sich in Teil II, der die bestehenden Ansätze zur Modelltransformation beschreibt.

2.6 Graphtransformationen

Im vorangegangenen Abschnitt wurden allgemeine Eigenschaften einer Modelltransformation dargestellt. Dieser Abschnitt spezialisiert den vorangegangenen, indem die bereits genannten Eigenschaften einer Transformation weiterhin gelten, Modelle jedoch zusätzlich als Graphen aufgefasst werden, die transformiert werden können. Vorteil dieses Ansatzes ist seine deklarative Herangehensweise. Modelle, aufgefasst als Graphen, werden transformiert, indem der Ausgangs- und Endzustand eines Teilgraphen angegeben wird. Historisch gesehen sind Graphtransformationen auf Modellen aus Graphgrammatiken entstanden, die Ende der 60er Jahre Schneider [38] und Pfalz/Rosenfeld [61] unabhängig voneinander als Verallgemeinerung von Stringgrammatiken eingeführt haben.[3] Davon ausgehend haben sich drei Hauptrichtungen entwickelt: der algorithmi-

[3] Eine Graphgrammatik beschreibt die Struktur komplexer Graphen. Sie besteht aus Produktionen, die aus einem Startgraphen, auch Axiom genannt alle Graphen einer Klasse erzeugen können.

sche, der algebraische und der logikorientierte Ansatz zur Graphtransformation, die im Folgenden skizziert werden.

2.6.1 Graphen

Bevor Graphen transformiert werden, wird hier zunächst geklärt, wie ein Graph definiert ist. Dazu wird die mengentheoretische Definition eines gerichteten Graphen aus Schürr und Westfechtel [69] gewählt:

Definition 1: Graph

„Ein gerichteter, markierter Graph (im folgenden auch kurz als Graph bezeichnet) über zwei Mengen L_V, L_E ist ein Tripel $G = (V, E, l)$ mit folgenden Konponenten:

(1) V(“vertices“) ist eine (endliche) Menge von Knotenbezeichnern.

(2) $E \subseteq V \times L_E \times V$ ist eine Menge von Kanten (“edges“), wobei L_E eine (endliche) Menge von Kantenmarkierungen (“edge labels”) ist.

(3) $l : V \rightarrow L_V$ ist die Knotenmarkierungsfunktion; dabei ist L_V eine (endliche)Menge von Knotenmarkierungen (“vertex labels”).“

Ein Graph setzt sich demnach aus markierten Knoten (**V**ertex) und Kanten (**E**dges) zusammen. Knoten werden durch Kanten verbunden. Zudem ist es möglich diese Definition um Knotenattribute zu erweitern.

Statt des vollständigen Graphens können auch Ausschnitte aus diesem betrachtet werden, wobei man Teil- und Untergraphen unterscheidet. Die folgenden Definitionen dieser sind ebenso [69] entnommen.

Definition 2: Teilgraph

„Seien $G, G' \in G = (L_V, L_E)$. Dann heißt G Teilgraph von G’ (in Zeichen : $G \subset G'$) $\Leftrightarrow$

(1) $V \subseteq V'$

(2) $E \subseteq E'$

(3) $l = l'|_V$ (d.h. $\forall v \in V : l(v) = l'(v)$)“

Definition 3: Untergraph

„Es sei $G \subset G'$. Dann heißt G Untergraph von G’ (in Zeichen : $G \subseteq G'$) $\Leftrightarrow$
G enthält alle Kanten aus G’, die Knoten aus V verbinden:
$E = E'|_V$ (d.h. $\forall e' \in E' : s(e') \in V \wedge t(e') \in V \Rightarrow e' \in E$
Dabei seien $s, t : E \rightarrow V$ Funktionen, die Quell- bzw. Zielknoten einer Kante liefern (‘source’ bzw. ‘target’)“

Eine weitere wichtige Eigenschaft eines Graphen, die im Lauf der Arbeit genutzt werden wird, ist die Existenz von Pfaden in einem Graphen. Pfade sind bestimmte Teilgraphen, die einen Anfangs- und Endknoten verbinden. Die folgende Beschreibung eines Pfades folgt der in Diestel [25], wurde jedoch an die obenstehende Graphdefinition angepasst.

Definition 4: Pfad im Graph

Ein Pfad ist ein nicht-leerer Graph G=(V,E,l), für den gilt:
Sei $V = \{v_1, v_2, v_3, ...v_n\}$*, dann ist* $E = \{(v_1, l_{E1}, v_2), (v_2, l_{E2}, v_3), ..., (v_{n-1}, l_{E(n-1)}, v_n)\}$.

Desweiteren können Graphen homomorph oder isomorph aufeinander abgebildet werden. Dies ist nach [69] wie folgt definiert:

Definition 5: Homomorphismus

„Seien $G, G' \in G(L_V, L_E)$*. Ferner sei* $h : V \to V'$*. Die Funktion h heißt (Graph-) Homomorphismus von G nach G' (in Zeichen: G* $\xrightarrow{\sim}$ *G')* $\Leftrightarrow$
h erhält die Markierungen von Knoten sowie Endknoten, Orientierung und Markierung von Kanten :

(1) $\forall v \in V : l'(h(v)) = l(v)$

(2) $\forall v_1, v_2 \in V \forall el \in L_E : (v_1, el, v_2) \in E \Rightarrow (h(v_1), el, h(v_2)) \in E'$ “

Damit heißen G und G' homomorph, wenn ein Homomorphismus von G nach G' existiert.

Definition 6: Isomorphismus

„Es sei G $\xrightarrow{\sim}$ *G'. h heißt Isomorphismus von G nach G' (in Zeichen G* $\xrightarrow{\approx}$ *G')* $\Leftrightarrow$

(1) $h : V \to V'$ *ist injektiv und surjektiv*

(2) $h^{-1} : V' \to V$ *ist ein Homomorphismus von G' nach G*

Ferner heißen G und G' isomorph (in Zeichen G $\cong$ *G')* $\Leftrightarrow$
es existiert ein Isomorphismus von G nach G' “

2.6.2 Idee einer Graphtransformation

Grundsätzlich ist der Sinn einer Graphtransformationsregel das Verändern eines bestehenden Graphen G, im Folgenden Wirtsgraph genannt, in einen Graphen G'. Es handelt sich um eine deklarative Transformation, die eine nach ihren Regeln veränderte Version von G erzeugt. Die Idee der Graphtransformationsregel wird nun möglichst allgemein, unabhängig von den im nächsten Abschnitt dargestellten Theorien, charakterisiert: Jede Graphtransformationsregel setzt sich aus einer linken und einer rechten Regelseite

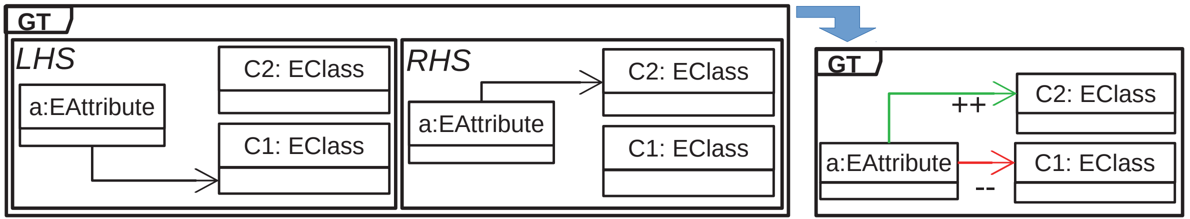


Abbildung 2.6: *Zwei Arten der Darstellung einer Graphtransformationsregel*

zusammen. Abbildung 2.6 zeigt dies schematisch an einer Regel, die ein Attribut von einer Klasse in die andere Klasse verschiebt. Die linke Seite (LHS, left-hand side in Abb. 2.6) definiert das Graphmuster, das im Wirtsgraphen gefunden werden muss, um die Regel anwenden zu können. Die rechte Seite (RHS, right-hand side in Abb. 2.6) definiert, wodurch das gefundene Graphmuster im Wirtsgraphen ersetzt wird. Knoten und Kanten, die sowohl auf der linken als auch auf der rechten Regelseite vorkommen, werden erhalten. Knoten und Kanten, die nur auf der rechten Regelseite vorkommen, werden erstellt und solche, die nur auf der linken Regelseite vorkommen, gelöscht. Heutzutage ist es üblich, die linke und rechte Regelseite in einem Diagramm darzustellen. Schematisch ist dies in Abbildung 2.6 rechts zu sehen. Die Knoten und Kanten werden je nach Zugehörigkeit zur einen oder anderen Seite gekennzeichnet. Elemente, die ausschließlich auf der linken Seite auftreten, werden mit $--$ und rot markiert. Elemente die nur auf der rechten Seite auftreten werden mit $++$ und grün markiert. Beidseitig vorkommende Elemente werden nicht explizit markiert. Zudem können Anwendbarkeitsbedingungen an eine Regel gestellt werden. Diese sind entweder positiver oder negativer Natur, müssen also unbedingt oder dürfen keinesfalls im Wirtsgraphen zu finden sein. Im Zuge der Transformation kann es vorkommen, dass der Graph nicht verändert wird. In diesem Spezialfall spricht man von einem Graphtest, der ursprünglich als identische Ersetzung definiert wurde, oder einer Graphabfrage, die Strukturen im Graphen ermittelt.

Die konkrete Anwendung von Graphtransformationsregeln auf einen Graphen kann in einzelne Schritte unterteilt werden: Im ersten Schritt findet man unter Einhaltung aller Anwendbarkeitsbedingungen das Muster im Wirtsgraphen, das jeweils zur linken Seite der Regel homomorph oder isomorph ist. Im zweiten Schritt wird es durch die rechte Seite ersetzt und dabei der Kontext des zu ersetzenden Teilgraphen bestimmt.

Bereits im ersten Schritt lassen sich zwei potentielle Probleme in Form von Nichtdeterminismen finden: Erstens kann eine Regel unter Umständen an mehreren Stellen angewendet werden, so dass eine Stelle zufällig gewählt werden muss. Zweitens könnte es mehrere Regeln geben, die an derselben Stelle angewendet werden können. Dieses Problem wird in der Praxis, je nach Einsatzgebiet der Transformation, auf unterschiedliche Weise gehandhabt. Im zweiten Schritt ist von einer Ersetzung der Teilgraphen die Rede. Wie dies vonstatten geht, hängt von der zugrunde gelegten Theorie ab. Daher ist es sinnvoll, die gängigen drei theoretischen Ansätze im nächsten Abschnitt zu skizzieren.

2.6.3 Theoretische Ansätze zur Graphtransformation

Algorithmischer Ansatz

Der algorithmische Ansatz basiert auf Mengentheorie. Graphen werden durch Mengen von Knoten und Kanten beschrieben. Diese können markiert und attributiert werden. Da Transformationen damit auf Mengen operieren, werden sie mit Hilfe von mengentheoretischen Operatoren definiert. Zudem können Parameter an Transformationen übergeben werden. Grundsätzlich „steht die Angabe eines Verfahrens im Vordergrund, das eine Graphersetzung als Folge von Mengenoperatoren abwickelt.“ [68] Es handelt sich um einen praxisorientieren Ansatz.

Der Ablauf einer Transformation gestaltet sich, grob skizziert, wie folgt: Zuerst wird der zu ersetzende Untergraph (linke Seite der Transformationsregel) identifiziert und dessen Kontext ermittelt. Darauffolgend wird der Untergraph gelöscht und der neue Untergraph (rechte Seite der Transformationsregel) an dessen Stelle eingefügt. Im nächsten Schritt wird der neue Untergraph in den Wirtsgraphen eingebettet. Dies geschieht mittels Einbettungsüberführungsregeln. Die Einbettungsüberführungsregeln beschreiben, wie die rechte Seite der Graphtransformationsregel in den bestehenden Wirtsgraphen eingefügt wird. Einbettungsüberführungsregeln können global (eine Regel für alle Graphproduktionen) und lokal (je Graphproduktion eine Regel) definiert werden. In jedem Falle beziehen sie den Kontext der Ersetzung ein. Handelt es sich um eine attributierte Transformation, werden die Attribute abgeleitet oder aus Parametern zugewiesen. [68, 69]

Die so definierte Transformation wurde später auf Teilgraphen ausgeweitet. Genauere Beschreibungen des algorithmischen Ansatzes finden sich in [69] und [66].

Algebraischer Ansatz

Basierend auf der Kategorientheorie werden im algebraischen Ansatz Graphtransformationsregeln deklarativ beschrieben. Deklarativ bedeutet, dass zur Darstellung der Wirkung einer Regel entsprechende Bedingungen vor der Transformation an ihre linke Seite und nach der Transformation an ihr rechte Seite gestellt werden. Zentrales Element ist hierbei die Kategorie, eine totale, assoziative, zweistellige Verknüpfung. Pushout-Diagramme stellen die Transformation für eine Kategorie von Graphen dar. Sie legen deklarativ fest, wie das Ergebnis einer Graphersetzung auszusehen hat. Ein Pushout ist das Ergebnis der Vereinigung zweier Graphen, die hier Hypergraphen genannt werden. Die Hypergraphen werden mittels totaler Morphismen vereinigt.

Es existieren zwei Pushout-Ansätze. Der klassische Double-Pushout-Ansatz agiert auf der Kategorie von Graphen und benutzt Graphersetzungsregeln, deren Semantik mit Hilfe zweier Pushout-Diagramme beschrieben wird. Abbildung 2.7 zeigt den Ablauf einer Transformation. PO1 und PO2 markieren die beiden Pushouts. L und R stellen die linke bzw. rechte Seite der Transformationsregel dar. G ist der Graph, in dem die Ersetzung vorgenommen werden wird. H ist der abgeleitete Graph, das Ergebnis der Transformation und gleichzeitig das Pushout des PO2. K bzw. D repräsentieren den gemeinsamen Durchschnitt, die Gemeinsamkeiten von L und R bzw. von G und H. Die Morphismen g,

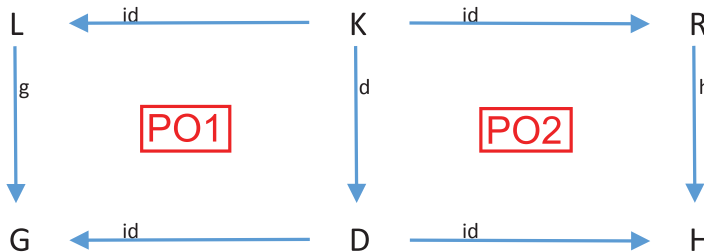


Abbildung 2.7: *Der klassische algebraische double-pushout Ansatz*

d und h überführen die Graphen ineinander. Bei dieser Art der Transformation werden die Knoten und Kanten des Graphen, die im Bild im Durchschnitt K der linken und rechten Regelseite liegen, identisch ersetzt.

Alternativ dazu wurde der Single-Pushout-Ansatz entwickelt. Während der Double-Pushout-Ansatz totale Morphismen verwendet, werden im Single-Pushout-Ansatz partielle Morphismen zur Abbildung von L nach R bzw. G nach H genutzt. Der Single-Pushout-Ansatz reduziert die Anzahl der zur Graphtransformation benötigten Pushouts auf 1. [66]

Für eine genauere Erklärung des algebraischen Ansatzes, inklusive der Pushout-Ansätze, sei auf das „Handbook of Graph Grammars and Computing by Graph Transformation“, Band 1 [66] und auf die „Fundamentals of Algebraic Graph Transformation“ [30] verwiesen.

Logikorientierter Ansatz

Dieser Hybridansatz der vorangegangenen Ansätze definiert den Graphersetzungsbegriff mittels Prädikatenlogik erster Stufe neu und greift dabei die identische Ersetzung von Knoten des algebraischen Ansatzes, sowie die Einbettungsüberführungsregeln des algorithmischen Ansatzes auf. Da es sich um einen praxisnahen Ansatz handelt, greift dieser einige neue Aspekte auf, die den vorangegangenen Ansätzen fehlen. Vor allem bietet er eine Möglichkeit, Eigenschaften von schematreuen Graphen statisch zu beschreiben und den Erhalt dieser Eigenschaften im Zuge einer Graphveränderung zu prüfen. Grundsätzlich gilt, dass dieser Ansatz den algebraischen Ansätzen maximal ähnelt ohne dabei gegenüber dem algorithmischen Ansatz an Ausdrucksfähigkeit zu verlieren.

Graphtransformationsregeln und Graphen werden durch prädikatenlogische Formelmengen dargestellt. Die Transformation selbst wird als Manipulation der Formelmenge verstanden. Zentraler Begriff ist hierbei das Graphschema. Es ordnet einer Klasse von Graphen Integritätsbedingungen zu, beschreibt demnach deren internen Aufbau. Unter einer Klasse von Graphen versteht man Graphen mit gleicher Signatur. Hierbei werden Graphen als gerichtete, attributierte, knoten- und kantenmarkierte Graphen, kurz gakk-Graph aufgefasst. Zur Transformation muss der Graph, wie im Übrigen auch seine Ersetzungsregel, dem Schema treu sein (schematreuer Graph bzw. schematreue Graphersetzungsregel). Ist dies der Fall, wird die Transformation durchgeführt, indem zuerst eine

Ansatz / Merkmal	Algorithmische Ansätze	Algebraische Ansätze	Logikbasierte Ansätze
Theorie	Mengentheorie	Kategorientheorie	Prädikatenlogik
Abbild	Teilgraph	Teilgraph	Teilgraph
Einbettung	Überführungsregel	Id. Ersetzung	Beide Konzepte
Anwendung	Sequenziell, parallel	Invertiert, Verschmolzen, parallel, sequenziell	Sequenziell, parallel
Erstellen	Kantenbüschel, Knoten	Kanten, Knoten	Kantenbüschel, Knoten
Löschen	Kantenbüschel, Knoten	Kanten, Knoten NUR im Kontext	Kantenbüschel, Knoten
Attributierbarkeit	Intrinsisch	Intrinsisch	Abgeleitet, intrinsisch
Kontrollstrukturen	Vorgesehen	Vorgesehen	Vorgesehen

Tabelle 2.1: *Vergleich der theoretischen Ansätze zur Graphtransformation*

Teilgraphensuche auf dem Wirtsgraphen ausgeführt wird, der gefundene Teilgraph ersetzt und mittels einer Einbettungsüberführungsfunktion eingebettet wird, Attribute auf neue Knoten transferiert und deren Werte berechnet werden.

Für eine detaillierte Beschreibung des logikbasierten Ansatzes sei auf [68] verwiesen.

Vergleich

Zum Ende des Theorieabschnitts sollen die einzelnen Ansätze verglichen werden. Die wichtigsten Eigenschaften der Ansätze sind in Tabelle 2.1 zusammenfasst. Die drei Ansätze operieren letztendlich alle auf Teilgraphen, obwohl der algorithmische Ansatz ursprünglich auf Untergraphen definiert war. Die Einbettung in den Wirtsgraphen findet auf komplett unterschiedliche, hier nicht weiter gewichtete, Arten statt.

Ein großer Vorteil des algorithmischen Ansatzes besteht in seiner Ausdrucksmächtigkeit bei gleichzeitig intuitiver Verständlichkeit. Komplexe Graphtransformationsregeln, die beispielsweise das Löschen von Kanten mit unbestimmtem Kontext oder das Löschen, Kopieren und Erzeugen ganzer Kantenbüschel erlauben, sind damit formal beschreibbar. Der algorithmische Ansatz ist jedoch bei der Regelanwendung auf eine sequenzielle oder parallele und gerichtete Anwendung beschränkt. Er eignet sich damit gut, um Graphtransformationsregeln mit Kontrollstrukturen zu steuern.

Aufgrund der mathematischen Fundierung des algebraischen Ansatzes können Regeln parallelisiert, verschmolzen oder invertiert werden. Gleichzeitig ist die Ausdrucksmächtigkeit der Regeln geringer, da keine Kantenbüschel unbekannter Kardinalität behandelt werden können. Zudem können Knoten nur im Kontext aller ihrer Kanten gelöscht werden. Dies bedeutet, dass alle Kanten, die mit dem zu löschenden Knoten in Verbindung stehen, auf der linken Seite der Regel zu finden sein müssen.[4]

[4]Dies gilt nicht für den Single Pushout Ansatz. Nähere Informationen hierzu finden sich in [66]

Der bisherige Vergleich zeigt, dass viele der Stärken des algebraischen Ansatzes den Schwächen des algorithmischen Ansatzes entsprechen und umgekehrt. Daraus kann gefolgert werden, dass eine restriktivere mathematische Spezifikation eine geringere Ausdrucksmächtigkeit zur Folge hat, während eine größere Ausdrucksmächtigkeit eine weniger restriktive mathematische Spezifikation erfordert. Die klassische Form des algebraischen Ansatzes ist laut [69] weniger ausdrucksfähig als die des mengentheoretischen Ansatzes. Beide Ansätze unterstützen die Beschreibung graphverändernder Operationen, bieten jedoch keine Hilfsmittel an, die die statische Beschreibung abgeleiteter Eigenschaften bestimmter Graphklassen zulassen. Genauso wenig ist es ihnen möglich, die Erhaltung bestimmter Eigenschaften einer Graphklasse vor und nach einer Transformation zu prüfen.

Diese Defizite versucht der logikbasierte Ansatz zu eliminieren, indem er eine Hybridlösung aus beiden Ansätzen bietet. Der Vorteil des logikbasierten Ansatzes liegt in der Fähigkeit komplexe Regeln über den Aufbau von Graphen zu gestatten, die Graphschemata. Dabei kommen beliebige Bedingungen zum Einsatz. Es kann z.B. festgelegt werden, wie Attribute von verschiedenen Knoten zueinander in Beziehung stehen dürfen. Zudem werden abgeleitete Grapheigenschaften unterstützt: abgeleitete Kanten werden durch Pfade mit einer Pfadmarkierung dargestellt und Attribute können abgeleitet werden, indem ihr Wert berechnet statt zugewiesen wird. Der logikbasierte Ansatz versucht demnach die Vorteile der beiden Ansätze zu vereinen. Dies gelingt nicht immer. Logikbasierte Regeln können zwar auf Kantenbüscheln operieren, sie sind jedoch nicht invertierbar.

2.7 Zusammenfassung

In diesem Abschnitt wurden die Grundlagen zum weiteren Verständnis des ModGraph-Ansatzes gelegt. Ausgehend von der Definition eines Modells wurden dessen Eigenschaften erklärt und die Metaebenen der Modelle erläutert. Sprachen bildeten den nächsten Teil des Grundlagenkapitels. Sie wurden definiert, indem die syntaktischen und semantischen Ebenen beschrieben und in den Kontext der Modelle eingeordnet wurden. Die darauf folgende Definition der modellgetriebenen Softwareentwicklung stellte die im weiteren Verlauf genutzte Vorgehensweise dar: Modelle werden als Softwareartefakte erster Klasse angesehen, die den Entwicklungsprozess steuern. Transformationen spielten in diesem Kapitel eine große Rolle, da sie zum Verständis des ModGraph-Ansatzes, der Graphtransformationen für EMF bietet, essentiell sind. Dementsprechend durfte auch ein Kapitel über diese nicht fehlen, das neben einer allgemeinen Definition eines Graphen den Aufbau einer Graphtransformationsregel erklärte und die theoretischen Ansätze skizzierte. Dabei wurde festgestellt, dass eine restriktivere mathematische Spezifikation eine geringere Ausdrucksmächtigkeit zur Folge hat und umgekehrt.

Teil II

Bestehende Ansätze und Werkzeuge zur Modelltransformation

„Man muss das Rad nicht neu erfinden!"
(Redensart)

Wenn jeder das Rad neu erfinden wollte...

...würden wir heute noch auf Bäumen sitzen!

Inhaltsverzeichnis - Teil II

3 Das Eclipse Modeling Framework **35**
3.1 EMF-Modelle 35
3.2 Ausführen des Modells 40
3.2.1 Java-Quelltext-Generierung 40
3.2.2 Interpretation mit dynamischem EMF 42
3.3 Hinzufügen von Verhalten 42
3.4 Graphical Modeling Framework (GMF) 43

4 Transformationssprachen **45**
4.1 Ansätze und Werkzeuge zur Modell-zu-Modell-Transformation 45
4.1.1 Xcore 45
4.1.2 Atlas Transformation Language 49
4.1.3 Query Views Transformation 50
4.2 Ansätze und Werkzeuge zur graphbasierten Transformation 51
4.2.1 Berliner Werkzeuge basierend auf dem algebraischen Ansatz . . . 51
4.2.2 Die sprachliche Einkleidung des logikorientierten Ansatzes 56
4.2.3 Story-basierte Ansätze 59
4.2.4 Andere Ansätze 62
4.3 Fazit 66

3 Das Eclipse Modeling Framework

Modellgetriebene Softwareentwicklung ist heute eine wohlbekannte und akzeptierte Methode, Software zu entwickeln. Dabei haben sich zahlreiche Ansätze und Werkzeuge zur Modell-zu-Modell- und Modell-zu-Text-Transformation entwickelt. Dieser Teil der vorliegenden Arbeit beschäftigt sich mit den Ansätzen, die zum ModGraph-Ansatz in Beziehung stehen.

Hier wird das Eclipse Modeling Framework (EMF) betrachtet, das die Grundlage für viele weitere Werkzeuge bildet und im ModGraph-Ansatz eine zentrale Rolle spielt. Der Fokus liegt dabei auf dem Aufbau eines EMF-Modells und der Generierung von Quelltext aus diesem Modell. Die Ausführungen hierzu basieren auf der zweiten Auflage des Buchs „EMF Eclipse Modeling Framework" von Steinberg et al. [73].

Das Eclipse Modeling Framework (EMF) ist ein in Forschung und Industrie weit verbreitetes Rahmenwerk zur modellgetriebenen Softwareentwicklung. Es ist Teil des Eclipse Modeling Projects (EMP), das wiederum Teilprojekt des Open-Source-Softwareprojekts Eclipse zur Entwicklung der gleichnamigen Werkzeugumgebung ist. EMF setzt Modellierungskonzepte und ihre Implementierung in direkte Relation zueinander. Dazu werden UML, Java und XML vereinigt, indem Überführungsmöglichkeiten zwischen den Sprachen geboten werden, wie in Abbildung 3.1 dargestellt ist. Den zentralen Vereinigungspunkt bildet das EMF-Modell. Dieses kann aus einer UML-, Java- oder XML-Spezifikation abgeleitet bzw. in diese drei Sprachen übersetzt werden. EMF wird in diesem Kapitel auf Grundlage von Steinberg et al.[73] näher betrachtet. Das aktuelle Eclipse EMF-Projekt findet sich unter `http://www.eclipse.org/modeling/emf/`. Die Installation der aktuellen EMF-Version erfolgt in Eclipse.

3.1 EMF-Modelle

EMF-Modelle werden analog zu den von der OMG definierten Metaebenen (siehe Abschnitt 2.2.3, Abbildung 2.2) hierarchisch aufgebaut. Zu ihrer Modellierung wird Ecore

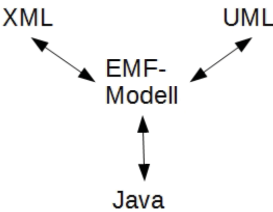


Abbildung 3.1: *EMF vereinigt UML, Java und XML (Zeichnung analog zu [73])*

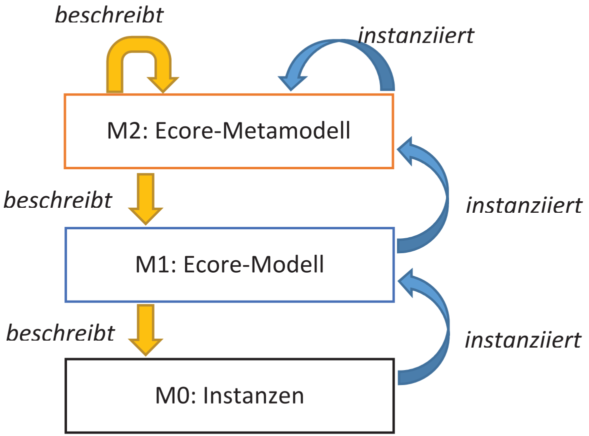


Abbildung 3.2: *Metaebenen in Ecore*

verwendet. Da bereits das Ecore-Metamodell in sich selbst definiert ist, entfällt die Meta-Metaebene und die Modellhierarchie vereinfacht sich zu der in Abbildung 3.2 gezeigten. Es handelt sich hierbei um eine dreistufige Hierarchie, beginnend mit dem Ecore-Metamodell. Dessen Instanzen sind die in der Modellierung mit EMF genutzten Ecore-Modelle. Ecore-Modelle können wiederum instanziiert werden zu Ecore-Modellinstanzen, im Folgenden Instanzen genannt, die konkrete Objekte darstellen. Die drei Ebenen werden im Folgenden näher betrachtet.

Das Ecore-Metamodell

Das Ecore-Metamodel (M2 in Abbildung 3.2) stellt die abstrakteste Ebene dar. Es ist das Metamodell, das dem EMF-Rahmenwerk zugrunde liegt. Das Ecore-Metamodell ist ebenfalls ein EMF-Modell und basiert auf dem von der OMG definerten eMOF-Standard, der eine Untermenge der MOF bildet [60]. Abbildung 3.3 zeigt das Metamodell, abgesehen von der Klasse EObject, von der alle Klassen erben. Es handelt sich hierbei um ein selbstbeschreibendes Klassendiagramm. Abstrakte Klassen sind grau hinterlegt und durch kursive Schrift gekennzeichnet, konkrete Klassen sind gelb hinterlegt. Abgeleitete Referenzen sind blau-grün markiert, direkte Referenzen sind schwarz dargestellt. Letztere weisen an einigen Stellen Enthaltenseins-Beziehungen, die durch einen schwarzen Diamanten gekennzeichnet sind, auf. Für alle Enthaltenseins-Beziehungen in einem EMF-Modell gilt das Prinzip der Exklusivität[1].

Bevor näher auf das abgebildete Metamodell eingegangen wird, wird zunächst die Klasse EObject betrachtet, deren Eigenschaften alle Modellelemente erben. EObject ist zum Einen eine Klasse, die generische Operationen bietet, welche den Zugriff auf Objekte regeln. Beispiele hierfür sind eClass(), das den Typ einer Klasse zurück gibt sowie die reflektiven Akzessoren eGet(...) und eSet(...). Zum Anderen bietet EObject einen Mechanismus zur Notifikation an. Änderungen an Objekten werden dem Observer-Muster folgend propagiert.

[1]Exklusivität bedeutet, dass es genau einen Container für ein Objekt gibt.

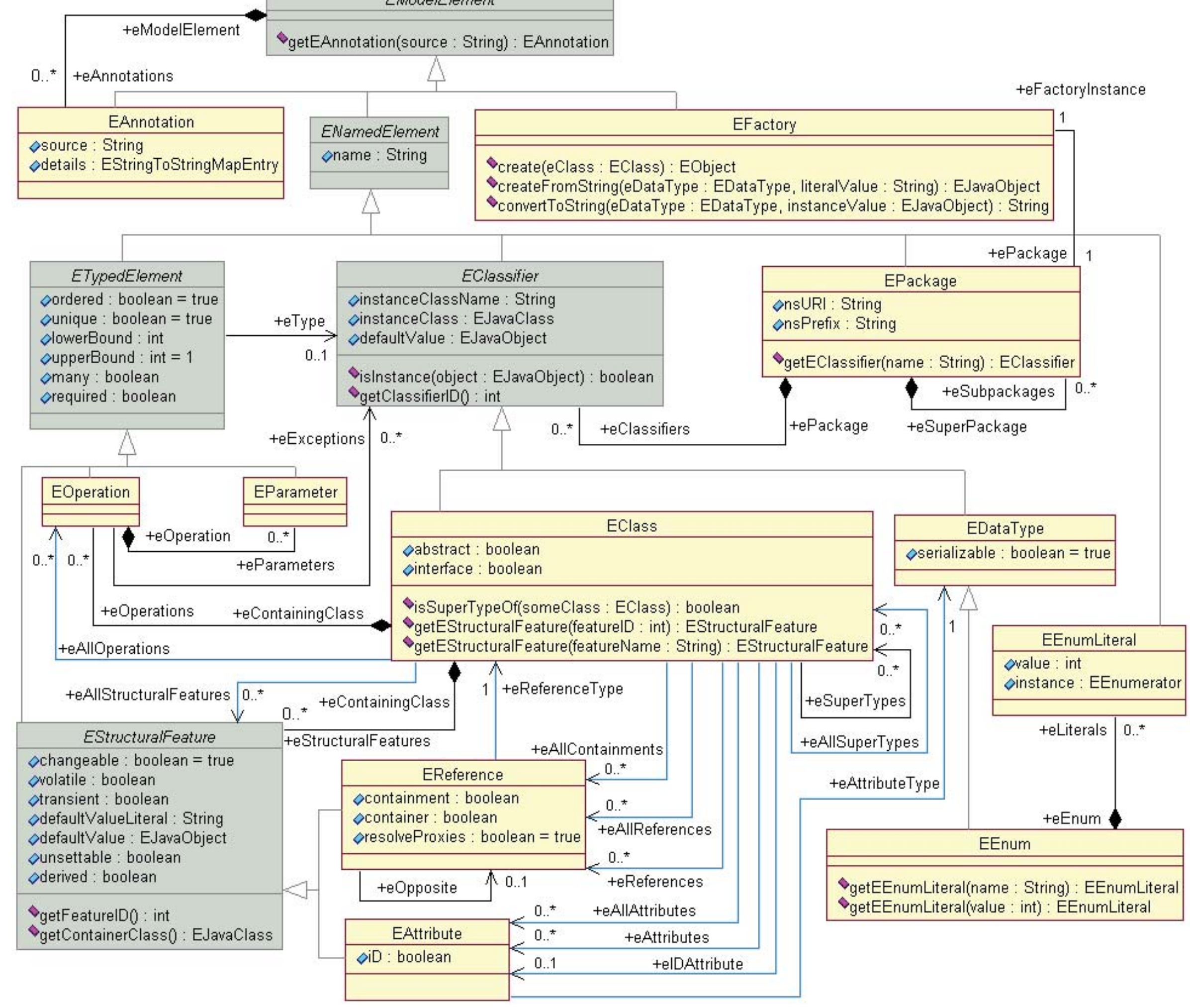


Abbildung 3.3: *Das Ecore-Metamodell (aus [1])*

Klassen werden durch die in Abbildung 3.3 zentral dargestellte Klasse EClass modelliert. Dabei können die Klassen sowohl abstrakt (Attribut abstract) sein, als auch als Interface (Attribut interface) deklariert werden. Klassen können in beliebig geschachtelten Paketen (EPackage) gruppiert, annotiert (EAnnotation) und in Fabriken (EFactory) erzeugt werden. Jede Klasse kann benannt, attribuiert und referenziert werden. Zudem können Operationen innerhalb der Klasse spezifiziert werden und die Angabe von Oberklassen für jede Klasse ist möglich (Referenz eSuperTypes). Hierbei ist Mehrfachvererbung erlaubt. Das Attribut name zur Benennung der Klasse erbt EClass - wie die meisten Klassen in Ecore - von der abstrakten Klasse *ENamedElement*. Attribute (EAttribute), Operationen (EOperation) und Referenzen (EReference) der Klasse sind im Ecore-Metamodell als eigenständige Klassen modelliert und durch Enthaltenseins-Beziehungen der Klasse zugeordnet (Referenzen eStructuralFeatures und eOperations). Zu ihrer Beschreibung müssen zunächst ihre Oberklassen detaillierter betrachtet werden:

Attribute und Referenzen werden zur abstrakten Klasse der strukturellen Eigenschaften einer Klasse zusammengefasst (*EStructualFeature*). Strukturelle Eigenschaften können mit den Attributen veränderbar (changeable), volatil (volatile), transient (transient), unsetzbar (unsettable) und abgeleitet (derived) gekennzeichnet werden. Dabei werden abgeleitete Attribute aus bestehenden Daten berechnet. Sie sind daher typischer weise transient, werden also nicht gespeichert. Zudem besteht die Möglichkeit, Standardwerte zu vergeben, mit welchen die strukturellen Eigenschaften initialisiert werden (Attribut defaultValue) und welche in den zugehörigen Literalen gespeichert werden (Attribut defaultValueLiteral). Strukturelle Eigenschaften sind immer genau einer Klasse zugeordnet (Referenz eContainingClass), während Klassen beliebig viele strukturelle Eigenschaften aufweisen können (Referenz eStructuralFeatures). Diese strukturellen Eigenschaften sind wiederum typisierte Elemente, denn die Klasse *EStructualFeature* erbt von der ebenfalls abstrakten Klasse *ETypedElement*. Typisierte Elemente können geordnet (ordered), eindeutig (unique), mehrwertig (many) und erforderlich (required) sein, sowie eine obere und untere Schranke bezüglich ihrer Multiplizität aufweisen. Ihre Typisierung erfolgt über die Klasse EClassifier, einer Oberklasse von EClass, und ihren Namen erben typisierte Elemente wiederum von der Klasse ENamedElement. Neben all diesen Gemeinsamkeiten von Referenzen und Attributen weisen beide einige zusätzliche Eigenschaften auf. Referenzen sind unidirektional, können Enthaltenseins-Beziehungen aufweisen (Attribute containment und container) und es besteht die Möglichkeit eine entgegengesetzte Referenz zu spezifizieren (Referenz eOpposite), so dass dieses Paar eine bidirektionale Referenz repräsentiert. Zudem sind Referenzen immer vom Typ genau einer Klasse (abgeleitete Referenz eReferenceType), nämlich derer, auf die sie verweisen. Attribute können im Gegensatz dazu nur über Datentypen (EDataType) typisiert werden. Datentypen sind hier sowohl primitive Datentypen (Ganzzahlen, Zeichenketten, etc.) als auch Aufzählungsdatentypen, die mittels der Klasse EEnum spezifiziert werden können. Zudem kann ein Datentyp eine beliebige Java-Klasse repräsentieren. Damit besteht vollständige Unterstützung zur Modellierung von Klassen und deren statischen Eigenschaften.

Operationen werden im Ecore-Metamodell durch die Klasse EOperation modelliert. Sie erben alle Eigenschaften der typisierten Elemente (*ETypedElement*). Operationen beinhalten beliebig viele Parameter, die durch die Klasse EParameter modelliert werden und jeweils in genau einer Operation enthalten sein dürfen (Referenzen eParameters und eOperation). Parameter sind ebenfalls typisierte Elemente. Darüber hinaus kann eine Operation beliebig viele Ausnahmen vom Typ eines EClassifiers werfen (Referenz eExceptions).

An dieser Stelle zeichnet sich bereits ein zentraler Punkt des ModGraph-Ansatzes ab: EMF bietet zwar Möglichkeiten zur Spezifikation einer Operation, das eigentliche Verhalten der Operation kann jedoch mit Ecore nicht modelliert werden. Wie diese „Lücke“ im EMF-Rahmenwerk geschlossen werden kann, zeigt Teil III dieser Arbeit. Für eine weitergehende Erläuterung des Ecore-Metamodells sei auf Steinberg et al. [73], Kapitel 5, verwiesen.

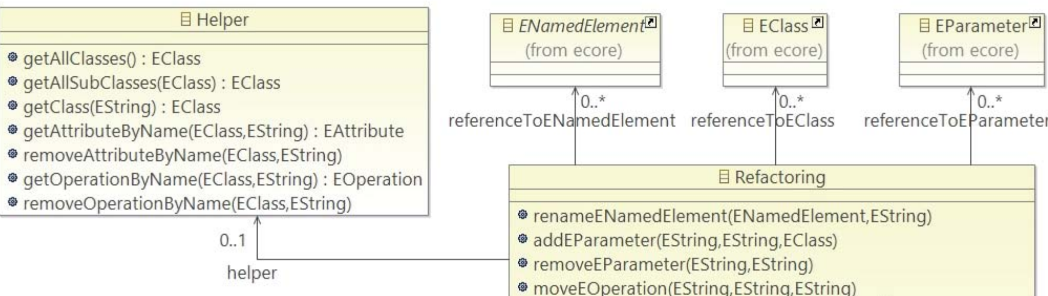


Abbildung 3.4: *Das Refactoring Ecore-Modell*

Das Ecore-Modell

Nach dieser ausführlichen Betrachtung der relevanten Abschnitte des Ecore-Metamodells, wird im Folgenden seine Instanzebene untersucht: Ecore-Modelle (M1 in Abbildung 3.2) modellieren strukturelle Sachverhalte in Form von Ecore-Klassendiagrammen. Sie werden in jedem mit EMF realisierten Projekt benötigt. Als Instanzen des Ecore-Metamodells bieten sie alle eben vorgestellten Konzepte an. Abbildung 3.4 zeigt einen Ausschnitt des Ecore-Modells am Beispiel Refactoring auf Klassendiagrammen. Hierbei modelliert die Klasse Refactoring Methoden zum Refactoring von Ecore-Modellen. Dazu ist es nötig, Klassen aus dem Ecore-Metamodell zu referenzieren (z.B.EClass), die in Abbildung 3.4 durch einen kleinen Pfeil rechts oben markiert sind. Außerdem wird eine interne Hilfsklasse (Helper) referenziert, die ihrerseits Methoden bereitstellt.

Ecore-Modelle sind hierarchisch aufgebaut und weisen immer eine Baumstruktur auf. Dies bedeutet auch, dass für jedes Modell ein globaler Behälter, das Wurzelobjekt des Baums, existiert. Dieser entspricht immer dem Paket, in welchem sich die Elemente befinden. Dadurch lassen sie sich auf einfache Weise in Ressourcen speichern. Zur Ressource muss lediglich das Wurzelobjekt des Ecore-Modells hinzugefügt werden, um Persistenz zu erreichen. Damit werden alle enthaltenen Objekte automatisch der Ressource hinzugefügt.

Instanzen des Modells

Instanzen dieser Ecore-Modelle bilden die unterste der Ebenen (M0) aus Abbildung 3.2. Sie stellen konkrete Sachverhalte dar. Instanzen werden entweder direkt im Quelltext oder mittels (der generierten) Editoren erstellt und bearbeitet. Sie dürfen beliebige Klassen des Ecore-Modells beliebig oft instanziieren. Da dies auch das Wurzelobjekt des Ecore-Modells betrifft, werden die Instanzen in der Regel durch einen Wald repräsentiert. Instanzen werden ebenfalls in Ressourcen gespeichert. Um Persistenz zu erreichen, müssen hier alle, in der Instanz enthaltenen, Wurzelobjekte zur Ressource hinzugefügt werden.

Ressourcen

Eine Ressource kann mittels eines Behälters für Ressourcen, der Ressourcenmenge, erzeugt werden. Jede Ressource repräsentiert einen physikalischen Speicherort, eine Datei. Objekte können durch ressourcenübergreifende Referenzen verbunden werden, sofern alle Resourcen innerhalb derselben Ressourcenmenge liegen. Diese Referenzen werden bei Bedarf aufgelöst, d.h. die Objekte werden in den Speicher geladen. Dies wird dynamisches Laden von Ressourcen innerhalb einer Ressourcenmenge genannt. Die Notwendigkeit, Objekte dynamisch zu laden, ergibt sich aus der Serialisierung der EMF-Modelle. EMF-Ressourcen werden im XMI-Format serialisiert und müssen somit in den Speicher geladen werden, um Zugriff zu ermöglichen. Im Refactoring-Beispiel wird dynamisches Laden benötigt, um die aus dem Ecore-Modell importierten Klassen in das Refactoring-Modell zu laden.

3.2 Ausführen des Modells

Im weiteren Vorgehen kann jede Instanz des Ecore-Metamodells, und damit jedes Ecore-Modell, mittels dynamischem EMF interpretiert oder die Generierung von Java-Quelltext aus dem Modell angestoßen werden. Dazu ist ein valides Modell zwingend erforderlich. Dieses wird durch die EMF-Validierung sichergestellt. Sie erlaubt es Bedingungen an Modellelemente direkt innerhalb des Modells zu stellen. Diese ergeben sich aus dem Aufbau des Ecore-Metamodells oder befinden sich in Annotationen des Ecore-Modells. Die Überprüfung des Modells erfolgt mit dem Ecore-Validator. Weitere Details zur Validierung finden sich in Steinberg et al. [73], Kapitel 18 und 21.

3.2.1 Java-Quelltext-Generierung

Die Generierung von Java-Quelltext erfolgt mit einem Zwischenschritt: Das Ecore-Modell wird in ein Generator-Modell transformiert, das zusätzliche, für die Quelltexterzeugung nötige Informationen enthält. Das Generator-Modell ist ein Wrapper für das Ecore-Modell. Es stellt sicher, dass das Ecore-Modell frei von Zusatzinformationen zur Erstellung von Quelltext bleibt, da in EMF eine strikte Trennung von Modell und Zusatzinformation gewünscht ist. Es stellt im Kontext der Modelltransformation eine intermediäre Struktur dar. Aus diesem Generator-Modell wird der Quelltext erzeugt. Dabei können, neben dem Quelltext für das Modell selbst, ein einfacher Editor und Tests generiert werden.

Der - durch die Modell-zu-Text-Transformation des Modells nach Java - entstandene Quelltext besteht aus drei Paketen: Das erste Paket enthält für jede Klasse im Ecore-Modell ein Interface. Dieses erweitert EObject[2], die zentrale Klasse in Ecore.[3] Das zweite

[2]Alternativ kann auch ein minimale und damit effizientere Variante von EObject, MinimalEObjectImpl, implementiert werden.

[3]Dies tritt im Ecore-Metamodell nicht auf; Auf Java-Ebene handelt es sich um ein Interface.

Paket enthält Klassen, die mindestens eines dieser Interfaces implementieren. Die Implementierung von mehreren Interfaces erlaubt die Umsetzung von Mehrfachvererbung in Java[4]. Generierte Klassen beinhalten die modellierten statischen Sachverhalte sowie Akzessoren. Für modellierte Operationen werden jeweils leere Methodenrahmen generiert. Zudem werden eine Factory-Klasse, die Methoden zur Instanziierung der Klassen beinhaltet, und eine Paketklasse, die Akzessoren für die Ecore-Metadaten bereit stellt, erzeugt. Das dritte Paket stellt Hilfsklassen, sog. Utility-Klassen zur Verfügung, die hier nicht näher betrachtet werden.

Neben dem Quelltext, der aus dem Modell erzeugt wird, besteht die Möglichkeit, Quelltext für einen Editor zu generieren. Hierzu wird EMF-Edit verwendet. Es erlaubt die Generierung von Editoren aufbauend auf dem Ecore-Modell. Diese Editoren basieren auf JFace und erlauben das Editieren und Anzeigen von Instanzen des Modells (inklusive Undo, Redo, Copy, Paste, Drag&Drop). Außerdem wird ein Property-Sheet erzeugt, innerhalb dessen die Eigenschaften der instanziierten Objekte betrachtet und verändert werden können. Der generierte Quelltext für Editoren setzt sich aus zwei generierten Projekten zusammen: dem Edit- und dem Editor-Projekt. Somit entsteht ein vollwertiger Multipage-Editor mit Project-Wizard, Property-Sheet und Baumeditor. Details zu dessen Aufbau finden sich in Steinberg et al. [73], Kapitel 3. Ein solcher Editor kann auf relativ einfacher Ebene zum Beispiel mittels GMF zu einem vollwertigen grafischen Editor ausgebaut werden (siehe Abschnitt 3.4).

Eine weitere Möglichkeit besteht in der Generierung von Tests. Hierbei generiert der EMF-Generator JUnit-Testfälle für das EMF-Projekt. Da diese im weiteren Verlauf für den ModGraph-Ansatz nicht von Belang sind, wird auch hier auf Steinberg et al. [73] verwiesen.

Der EMF-Nutzer hat nun die Möglichkeit den generierten Quelltext zu verändern, um z.B. modellierten Operationen Verhalten hinzuzufügen. Hierzu bietet EMF zusätzliche Unterstützung, z.B. durch die Klasse `EcoreUtil`[5], die unter Anderem Möglichkeiten zum Erzeugen, Löschen und Deserialisieren von Objekten bietet. Damit diese Änderungen bei erneuter Generierung nicht verloren gehen, werden Annotationen im Quelltext genutzt. Der EMF-Generator annotiert jede generierte Methode mit einem `@generated`. Diese Methoden werden bei jedem Generierungsdurchlauf automatisch erneuert. Löscht der EMF-Nutzer die Annotation oder ergänzt diese zu einem `@generated NOT`, wird die Methode vom Generator ignoriert und die benutzerdefinierten Änderungen bleiben erhalten. Alternativ kann durch anhängen von „Gen“ an den Java-Methodennamen die Generierung umgeleitet und damit die Methode erhalten werden. Manuell eingefügte Methoden, die nicht Teil des Modells sind, bleiben immer erhalten, da sie keine `@generated` Annotation tragen.

Wichtig ist an dieser Stelle, dass der EMF-Generator so angelegt ist, dass der finale Quelltext aus handgeschriebenen und generierten Anteilen besteht.

---

[4] Java unterstützt selbst keine Mehrfachvererbung.

[5] Die Dokumentation zur Klasse `EcoreUtil` findet sich unter: `http://download.eclipse.org/modeling/emf/emf/javadoc/2.7.0/org/eclipse/emf/ecore/util/EcoreUtil.html`

3.2.2 Interpretation mit dynamischem EMF

Zur Interpretation eines Modells werden dynamische Instanzen erzeugt und mit dem EMF-eigenen Interpreter ausgeführt. Man spricht hierbei von dynamischem EMF. Dabei ist es nicht nötig, Quelltext aus dem Modell zu generieren. Objekte werden als Instanzen von `EObject`[6] durch Aufruf der - in `EObject` definierten - reflexiven Methoden erzeugt. Diese haben das gleiche Verhalten wie die generierten Methoden, sind jedoch in ihrer Ausführung deutlich langsamer. Daher werden auch zur Interpretation der generierten Klassen genutzt, sofern Quelltext generiert wurde. Sollten darin bereits Operationen implementiert sein, besteht im reinen EMF trotzdem keine Möglichkeit, das Verhalten der modellierten Operationen auszuführen.

3.3 Hinzufügen von Verhalten

Ecore-Modelle bieten - den letzten beiden Abschnitten zufolge - durch ihre große Ausdrucksmächtigkeit, bei überschaubarer Anzahl an Konstrukten, eine sehr gute Möglichkeit zur Strukturmodellierung eines EMF-Modells. Verhaltensmodellierung bleibt im reinen EMF jedoch unberücksichtigt: Es können lediglich Operationsdeklarationen ohne Rumpf spezifiziert werden. EMF unterstützt auf diese Weise die modellgetriebene Softwareentwicklung nur partiell. Um dem Modell dennoch Verhalten hinzuzufügen, existieren verschiedene Wege:

Der Entwickler kann zum Einen direkt im generierten Java-Quelltext den Methodenrumpf implementieren. Damit dieser bei erneuter Generierung erhalten bleibt, muss das mittels einer Java-Annotation dem EMF-Generator mitgeteilt werden (siehe Abschnitt 3.2.1). Eine Alternative dazu ist die Annotation der entsprechenden Operation im Generator-Modell. Der in Java implementierte Methodenrumpf ist innerhalb dieser Annotation hinterlegt. Er wird bei der Generierung der Java-Methode übernommen. Die dritte und letzte Möglichkeit bietet OCLInEcore [2], nämlich die Einbettung der OCL in das Ecore-Modell. Dabei werden die Operationen im Ecore-Modell annotiert. Die Annotationen beinhalten OCL-Ausdrücke, die den Methodenrumpf spezifizieren.

Die ersten beiden Möglichkeiten basieren auf der direkten Eingabe von Quelltext. Sie können daher nicht als modellgetrieben betrachtet werden. Die verbleibende Alternative besteht in der Nutzung von OCL. Zwar ist OCL wesentlich abstrakter als gewöhnlicher Quelltext, jedoch handelt es sich hierbei um eine seiteneffektfreie Sprache. Sie erlaubt Abfragen auf dem Modell, allerdings keinerlei Änderungen an diesem. Es gibt demnach zwar drei Wege, Verhalten in ein Modell einzubinden, jedoch kein Verfahren beliebiges Verhalten auf Modellebene zu spezifizieren.

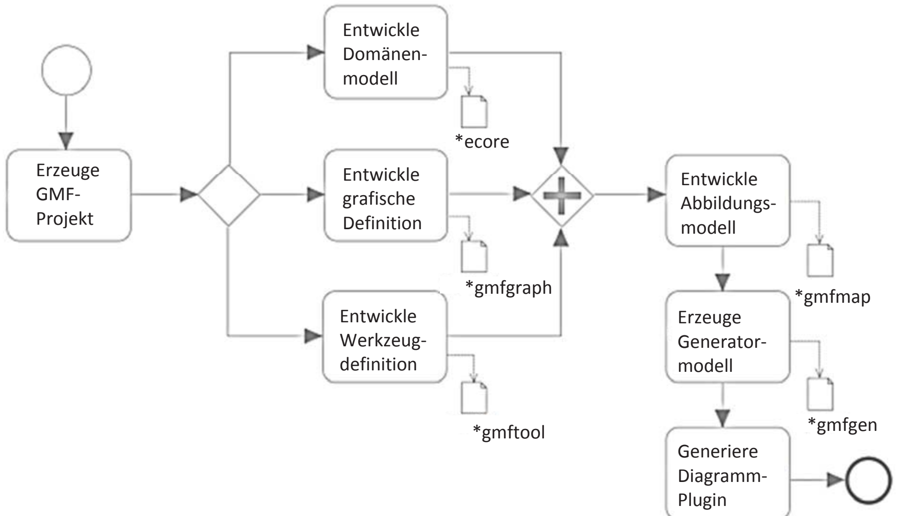


Abbildung 3.5: *Zusammenarbeit von EMF und GMF*[7]

3.4 Graphical Modeling Framework (GMF)

EMF ist die Grundlage vieler Werkzeuge der Eclipse-Welt. Eines davon ist das Graphical Modeling Framework (GMF). Viele Ecore-Modelleditoren bauen auf GMF auf, so dass der im EMF-Kontext arbeitende Nutzer quasi zwangsläufig mit GMF konfrontiert ist. Es ist Teil des Graphical Modeling Projects (GMP). Da der ModGraph-Ansatz im weiteren Verlauf sowohl den GMF-basierten Editor für Ecore-Modelle favorisiert, sowie selbst GMF im EMF-Kontext nutzt, soll GMF im Folgenden anhand von [37] sowie der aktuellen Online-Dokumentation (`http://www.eclipse.org/modeling/gmp/`) skizziert werden.

GMF basiert auf EMF und unterstützt die automatische Generierung von grafischen Editoren für EMF-Metamodelle. Dazu müssen, neben dem EMF-Metamodell, GMF-Konfigurationsmodelle erstellt werden, die unter anderem Einstellungen für die grafischen Elemente für den Editor und deren Bindung an die Metamodellelemente enthalten. GMF basiert auf dem Graphical Editing Framework (GEF)[37], „welches sozusagen als Zielarchitektur (zusammen mit der sog. GMF-Runtime) für den GMF-Generator fungiert."[72]

Die Zusammenarbeit von EMF und GMF ist in Abbildung 3.5 skizziert. Zunächst wird das Ecore-Modell benötigt, das Domänenmodell. Daneben werden die grafischen Elemente, die später Verwendung finden sollen, in einem weiteren Modell definiert: Unabhängig vom Ecore-Modell werden Elemente spezifiziert, die grafisch einen Knoten, eine Kante oder einen Text darstellen. Diese repräsentieren später Ecore-Modellelemente. Ein drittes Modell beinhaltet die Werkzeug-Definition. Dabei werden die Aktionen des grafischen Editors festgelegt. Hierzu zählt auch die Erzeugung neuer Objekte. Eine Palette grup-

[6] Genauer von der auf Interpretation spezialisierten Klasse `DynamicEObjectImpl`.
[7] Übersetzt aus: `http://wiki.eclipse.org/Graphical_Modeling_Framework/Tutorial/Part_1`

piert die möglichen Aktionen. Diese drei Modelle werden nun im Abbildungs-Modell vereint. Es stellt die Zusammenhänge zwischen Modellelement, grafischem Element und dem Werkzeug zu deren gemeinsamer Erzeugung dar. Aus diesem Abbildungs-Modell wird nun ein GMF-Generator-Modell erzeugt. Analog zum Generator-Modell in EMF handelt es sich auch hierbei um eine intermediäre Struktur. Darin werden zusätzliche, für die Quelltext-Generierung notwendige Informationen gespeichert. Mit Hilfe des Generator-Modells wird letztlich der Quelltext für den grafischen Editor erzeugt. Detailliertere Informationen zu GMF sind finden sich in [37] und auf der oben genannten GMF Homepage. Interaktive Tutorials werden nach der Installation in Eclipse angeboten.

4 Transformationssprachen

In diesem Kapitel werden Modell-zu-Modell-Transformationssprachen untersucht. Den Anfang bildet Xcore, eine textuelle Syntax für Ecore-Modelle, die diese um Verhaltensmodellierung erweitert, indem es die prozedurale Ausdruckssprache Xbase integriert. Der Abschnitt über Xcore verlässt sich (aus Mangel an wissenschaftlichen Publikationen) auf die inzwischen sehr ausführliche Internetdokumentation von Xcore [3]. Die Ausführungen zu Xbase stützen sich auf Effinge et al. [28].

In je einem weiteren Abschnitt dieses Kapitels werden die Atlas Transformation Language (ATL) [45] und die Query View Transformation (QVT) [56] beschrieben, die denselben operationalen Kontext zur Modelltransformation aufweisen. Grund dafür ist, dass es sich um zwei Einreichungen auf den QVT-RFP der Object Management Group (OMG), einem Aufruf zum Finden einer Modell-zu-Modell-Transformationssprache handelt. Beide nutzen MOF [60] als Metametamodell und können Instanzen von Ecore-Modellen transformieren. Die Ausführungen zu ATL basieren auf den Arbeiten von Jouault et.al. [44, 45], die Ausführungen zu QVT sind dem OMG-Standard entnommen [56].

Im Abschluss dieses Teils werden bereits bestehenden Ansätze und Werkzeuge zur graphbasierten Transformation beschrieben. Dabei werden gängige Werkzeuge vorgestellt, die später mit ModGraph verglichen werden (siehe Kapitel 10.2). Den Anfang bilden AGG, Tiger und Henshin, die den algebraischen Ansatz praktisch umsetzen, gefolgt von PROGRES, der sprachlichen Einkleidung des logikorientierten Ansatzes. Danach werden die Story-basierten Ansätze betrachtet. Fujaba bildet sowohl hier, als auch chronologisch gesehen den Anfang, gefolgt von MDELab und eMoflon. Der letzte Abschnitt behandelt GReAT, GrGen.NET und Viatra2, die ebenfalls graphbasierte Werkzeuge sind, sich jedoch in keine der oben genannten Kategorien einordnen lassen. Als Literaturgrundlagen der einzelnen Abschnitte dienen wissenschaftliche Veröffentlichungen zu den Ansätzen und Werkzeugen, die im jeweiligen Abschnitt genannt werden.

4.1 Ansätze und Werkzeuge zur Modell-zu-Modell-Transformation

4.1.1 Xcore

Unter dem Motto „Modellieren für Programmierer und Programmieren für Modellierer“ zielt Xcore [3] darauf ab, die Lücke zwischen Programmierung und Modellierung zu schließen, indem es die Vorteile beider Ansätze vereint. Xcore wurde parallel zum ModGraph-Ansatz entwickelt. Es ist eine domänenspezifische Sprache (DSL), die eine

erweiterte konkrete Syntax für Ecore anbietet. Diese ermöglicht neben der Modellierung der Struktur auch die Beschreibung von Verhalten durch Integration der Ausdruckssprache Xbase [28]. So entsteht eine vollwertige Programmiersprache mit Java-naher Syntax.

Xcore-Modelle

Der Aufbau eines Xcore-Modells ähnelt dem eines Ecore-Modells sehr (siehe Abschnitt 3.1), da es sich um einen Nachbau der entsprechenden Klassen handelt. Zudem enthält jedes Xcore-Modell ein zugehöriges Ecore-Modell sowie ein Generator-Modell. Beide werden aus dem Xcore-Modell abgeleitet.

Xcore ist in Xtext [29, 4], einem freien Rahmenwerk zur Entwicklung von externen, textuellen DSLs spezifiziert. Bei der Erstellung der Sprache wird entweder ausgehend von der konkreten Syntax (der Xtext-Grammatik) der Sprache ihre abstrakte Syntax (ein Ecore-Modell) abgeleitet oder - möchte man ein bestehendes Ecore Modell nutzen- eine konkrete Syntax für dieses definiert. Zudem wird weitere Sprachinfrastruktur bereitgestellt (Editor, Parser, Compiler, ...). Ein Ausschnitt aus der Xtext-Grammatik zur Xcore-Definition ist in Auflistung 4.1 abgebildet, die vollständige Grammatik befindet sich in Anhang A.6.1.

Der in Auflistung 4.1 dargestellte Abschnitt zeigt die Definition einer Klasse (XClass Definition) und deren Eigenschaften. Die Notation ähnelt der erweiterten Backus Naur Form (EBNF), die unter anderem in [67] besprochen wird. Jeder Klasse können beliebig viele Annotationen zugeordnet sein, sie darf zum Interface deklariert werden und hat einen Namen. Zudem wird Generizität und Vererbung unterstützt. Sie kann als Behälter für eine Java-Klasse oder einen -Datentyp dienen. Jede Klasse beinhaltet beliebig viele Elemente (XMember Definition), die entweder Operationen, Referenzen oder Attribute sind.

Eine konkrete Spezifikation einer Klasse in Xcore ist am Beispiel Refactoring in Auflistung 4.2 gezeigt. Diese Auflistung ist lediglich eine andere Sicht auf das bereits in Kapitel 3.4, Abbildung 3.4, als Klassendiagramm dargestellte Refactoring-Modell. Die Klasse Refactoring (Schlüsselwort class) definiert, analog zu der im vorangegangenen Kapitel 3.4, Referenzen (Schlüsselwort refers) auf Klassen des Ecore Metamodells sowie auf eine interne Hilfsklasse.

```
1  ...
2  XClass:
3  {XClass}
4  (annotations+=XAnnotation)*
5  ((abstract?='abstract'? 'class') | interface?= 'interface') name = ID
6  ('<' typeParameters+=XTypeParameter (',' typeParameters+=XTypeParameter)* '>')?
7  ('extends' superTypes+=XGenericType (',' superTypes+=XGenericType)*)?
8  ('wraps' instanceType=JvmTypeReference) ?
9  '{'
10 (members+=XMember)*
11 '}';
12 XMember:
13 XOperation |XReference | XAttribute;
14 ...
```

Auflistung 4.1: *Ausschnitt aus der Xcore Grammatik*

```
...
class Refactoring {
    refers Helper helper
    refers ENamedElement[] referenceToENamedElement
    refers EClass[] referenceToEClass
    refers EParameter[] referenceToEParameter
    ...
    op void renameENamedElement(ENamedElement element , String newName)
    op void addEParameter(String operationName , String parameterName ,
            EClass paramType)
    ...
```

Auflistung 4.2: *Ausschnitt aus dem erRefactoring Xcore-Modell*

Operationen (Schlüsselwort op) definieren Methoden zum Refactoring von Ecore-Klassendiagrammen.

An dieser Stelle sind die Analogien zu Ecore klar ersichtlich: Vergleicht man diese Auflistung mit Abbildung 3.4, so wird klar, dass es sich um den Aufbau derselben Strukturen handelt. Darüber hinaus wird bei der praktischen Nutzung von Xcore deutlich, dass jedes Element auf das entsprechende Ecore-Element abgebildet wird. So wird jede durch class gekennzeichnete Klasse als EClass aufgefasst. Im Übrigen ist es möglich, bereits existierende Ecore-Modelle problemlos in Xcore-Modelle zu überführen.

Auf dieser Ebene ist Xcore noch nichts anderes als die textuelle Einkleidung von Ecore. Allerdings unterstützt Xcore die Spezifikation von Verhalten durch Einbindung der prozeduralen Ausdruckssprache Xbase. Xbase ist Teil von Xtext und wurde speziell zur Wiederverwendung in beliebigen Xtext-basierten Sprachen, wie Xcore, erstellt. Die Xtext-Grammatik zur Spezifikation von Xbase befindet sich in Anhang A.6.2. Xbase ist eine statisch typisierte Programmiersprache und ist eng verzahnt mit dem Java-Typ-System. Sie bietet eine prozedurale, Java-nahe, ausdrucksorientierte Syntax. Auflistung 4.3 zeigt exemplarisch die Implementierung der im Refactoring-Beispiel genutzten Methode addEParameter, die einer Operation im Ecore-Modell einen Parameter hinzufügt.

Xbase-Ausdrücke stellen Kontrollstrukturen, die aus den gängigen Programmiersprachen bekannt sind, und programmiersprachliche Ausdrücke bereit. In Auflistung 4.3, Zeile 3, wird eine Schleife zum Finden der Operation durchlaufen. In Zeile 4 wird eine Bedingung überprüft, die ausschließt, dass der Parameter bereits vorhanden ist.

Jeder Ausdruck hat einen Rückgabewert. Er kann beliebig mit anderen Ausdrücken, Operatoren, Methodenaufrufen, usw. kombiniert werden, um komplexere Ausdrücke zu

```
op void addEParameter(String operationName, String parameterName, EClass paramType){
        var EOperation op1 = null
        for (_op1 : referenceToEOperation.filter(e|e.name == operationName)) {
            if(! ( _op1.getEParameters.findFirst(e|e.name == parameterName)!=null))
                op1 = _op1
        }
        val newParameterNameValue = parameterName
        var newParameter = EcoreFactory::eINSTANCE.createEParameter()
        newParameter.name = newParameterNameValue
        op1.getEParameters.add(newParameter)
        newParameter.EType = paramType
        referenceToEParameter.add(newParameter)
    }
```

Auflistung 4.3: *Einfache Xbase Implementierung der Methode* addEParameter

erhalten. Diese ermöglichen das Implementieren jeder Xcore definierten Operation sowie die Spezifikation von abgeleiteten Eigenschaften und die Redefinition von Datentypen. Datentypen können zudem inferiert werden. So reicht es, bei der Deklaration einer Variablen lediglich das Schlüsselwort var bzw. val (bei finalen Variablen) voranzustellen, siehe Zeilen 7 und 8 in Auflistung 4.3. Der Typ der Variablen wird durch Dependency Injection mit Google Guice[1] ermittelt.

Des Weiteren wird Dependency Injection genutzt, um Typen von Rückgabewerten und von Lambda-Ausdrücken festzustellen. Lambda-Ausdrücke definieren anonyme Funktionen auf den aktuell gültigen Elementen. Sie können zum Beispiel durch einen einzeiligen Ausdruck ein Element einer gegebenen Liste anhand seiner Eigenschaften identifizieren. (In Java wäre hier eine (meist mehrzeilige) Schleife zu programmieren.) Die filter-Funktion innerhalb der for-Schleife (Zeile 3), sowie die findFirst-Funktion innerhalb der if-Bedingung (Zeile 4), die jeweils Namensgleichheit prüfen, sind zwei Beispiele für einen Lambda-Ausdruck.

Auch besteht in Xbase die Möglichkeit, Operatoren zu überladen. Ihre Auswertung geschieht durch Funktionsaufrufe mit operator-spezifischer Signatur. Im Gegensatz zu Java können demnach beliebige Operatoren auf beliebigen Typen definiert werden. Beispielsweise kann Objektgleichheit mit == festgestellt werden (Zeilen 3 und 4, Vergleich von String-Objekten innerhalb der Lambda-Ausdrücke).

Xbase bietet zudem eine kleine Standard-Bibliothek mit erweiterten Methoden an. Diese können aufgerufen werden, als würden sie zu einem bestimmten Typ gehören, sind jedoch extern definiert. Dabei wird das Objekt, auf dem die Methode aufgerufen wird, zum ersten Übergabeparameter der externen Methode. Eine weitere Eigenschaft von Xbase ist die Modellinferenz. Methoden- und Konstruktoraufrufe werden mit dem Java-Typ-Modell verbunden, das alle Java Sprachkonstrukte unterstützt. Xbase nutzt dies bei der Auflösung von Typen und Methoden sowie zur Festlegung von Gültigkeitsbereichen von Variablen. Diese Typinferenz erlaubt die Generierung von validen Java-Quelltext-Fragmenten für Xcore-Ausdrücke. Damit ist es möglich, dass Sprachen - die Xbase erweitern - ausführbaren Java-Quelltext erzeugen und diesen benutzen. Zum Beispiel wird in Auflistung 4.3, Zeile 8 die zuvor generierte Fabrikmethode zur Erzeugung eines Parameters in Ecore aufgerufen, in Zeile 9 wird durch Nennung des Attributs name implizit dessen Akzessor-Methode aufgerufen.

Ausführen eines Modells

Parallel zur Spezifikation des Xcore-Modells wird dieses validiert und ausführbarer Java-Quelltext generiert, der auf der Grundlage der Modellinferenz aufbaut. Eine Interpretation mit dem Xcore-Interpreter ist ebenfalls möglich. Dieser nutzt die Tatsache, dass jedes Xcore-Modell ein Ecore-Modell (und ein Generator-Modell) beinhaltet. Beide Modelle werden aus dem Xcore-Modell abgeleitet. Das Ecore-Modell kann, wie in 3.2.2 beschrieben, interpretiert werden. Der Xbase-Interpreter ermöglicht zusätzlich die Interpretation von Verhalten.

[1]Näheres zu Google Guice findet sich unter: `https://github.com/google/guice`

Nutzt man also Xcore mit dem integrierten Xbase, so erhält man ein ausführbares Modell, das eine Modell-zu-Modell-Transformation spezifiziert, indem es die Ausführung von Methoden auf Instanzen des Modells ermöglicht. Es bietet damit eine tatsächliche Abstraktion zu Java an, denn der Modellierer muss sich nicht mehr mit Quelltext beschäftigen.

Für weitere Informationen zu Xcore sei auf die Xcore-Homepage [3] verwiesen. Die Installation der aktuellen Xcore-Version erfolgt in Eclipse.

4.1.2 Atlas Transformation Language

Die Atlas Transformation Language (ATL) ist eine domänenspezifische, textuelle Sprache zur Modell-zu-Modell-Transformation. Sie ist Teil der Atlas Model Management Architecture (AMMA) Platform, die von der Atlas Group (INRIA & LINA) entwickelt wird. ATL wird nun anhand von Jouault et al. [44, 45] dargestellt.

ATL ist eine hybride Sprache. Sie unterstützt deklarative und imperative Sprachkonstrukte. Dabei wird der Ansatz des allgemeinen Transformationsmusters der MDSD verfolgt, der bereits in Abbildung 2.5, Unterkapitel 2.5 besprochen wurde. ATL konkretisiert dieses Muster mit MOF [60] als Metametamodell.[2]

Jede ATL-Transformation ist aus Modulen (ATL Modules) aufgebaut. Jedes Modul beinhaltet eine Signatur (ATL Header), die Quell- und Zielmodelle definiert, sowie Importe, Transformationsregeln und Helfer. Letztere sind Anfragen, die mit Hilfe von OCL spezifiziert werden. Des Weiteren können innerhalb eines Moduls Bibliotheksfunktionen verwendet werden. Grundsätzlich werden in ATL zwei Arten der Transformation unterschieden: Der normale Modus und der Verfeinerungsmodus.

Der gebräuchlichere normale Modus unterstützt unidirektionale Ein-Ausgabe-Batch-Transformationen. Diese instanziieren das ATL-Metamodell und sind entweder endogener oder exogener Natur. Bei Ausführung der Transformation wird mindestens ein Quellmodell in mindestens ein Zielmodell überführt. Quellmodelle sind hierbei nur lesbar, Zielmodelle können nur geschrieben werden. Die Realisierung bidirektionaler Transformationen erfolgt, indem ein Paar unidirektionaler Transformationen erstellt wird.

Transformationsregeln sind das elementare Konstrukt um die Transformationslogik in ATL auszudrücken. Deklarative Regeln heißen Matched Rules. Es existieren verschiedene Arten von Matched Rules, die sich in ihrer Anwendungshäufigkeit unterscheiden. Zur ihrer genaueren Erklärung sei auf Jouault et al. [45] verwiesen. Alle Matched Rules bestehen aus einem Quell- und einem Zielmuster. Die beiden Muster binden Modellelemente. Damit setzen sie die Modellelemente in Relation. Matched Rules können Vererbungshierarchien aufbauen.

In einigen Fällen sind die deklarativen Eigenschaften von ATL nicht ausreichend. Sie können durch imperative Eigenschaften ergänzt werden. Dazu bietet ATL verschiedene Konzepte an: native Operationsaufrufe, Called Rules und Aktionsblöcke. Native Operationsaufrufe können in einer beliebigen, externen Sprache geschrieben werden. Für

[2]Da das Ecore-Metamodell eine Teilmenge des MOF darstellt, kann es ebenfalls als Metametamodell genutzt werden.

Called Rules und Aktionsblöcke ist in ATL eine eigene Subsprache definiert. Called Rules werden explizit mit Parametern aufgerufen. Aktionsblöcke befinden sich innerhalb einer Regel. Sobald imperative und deklarative Sprachkonstrukte genutzt werden, spricht man von einer hybriden ATL-Transformation.

Neben dem soeben dargestellten Ein-Ausgabe-Modus unterstützt ATL überschreibende Transformationen, indem eine modifizierte Kopie des Originals angelegt wird. Der sog. Verfeinerungsmodus unterstützt die Definition von Matched Rules. Dabei ist es nicht nötig, einen vollständigen Satz von Transformationsregeln zu definieren. Alle unveränderten Elemente des Quellmodells werden automatisch in das Zielmodell kopiert. Das Löschen eines Elements muss daher explizit angegeben werden. Es entsteht folglich der Eindruck, das Quellmodell wäre modifiziert worden, obwohl jede Transformationsregel auf dem Originalmodell ausgewertet wird und auf einer Kopie dessen angewendet wird. Eine solche Transformation ist bereits aufgrund ihres Aufbaus endogen.

Weitere Informationen zu ATL finden sich in Jouault et al. [44, 45]. Die aktuelle Version der ATL-Entwicklungsumgebung ist unter `http://www.eclipse.org/atl/` zu finden und wird via Eclipse installiert.

4.1.3 Query Views Transformation

Der QVT-Ansatz bildet den von der OMG definierten Standard für Modelltransformationssprachen. Er wird nun anhand der OMG-Spezifikation [56] skizziert. QVT folgt ebenfalls dem allgemeinen Transformationsmuster aus Unterkapitel 2.5, Abbildung 2.5 mit MOF als Metametamodell. Insgesamt werden in QVT drei Sprachen mit unterschiedlicher Abstraktionsebene definiert. Die abstrakteste Sprache ist die deklarative Sprache QVT-Relational (QVT-R). Auf etwas geringerer Abstraktionsebene folgt QVT-Operational (QVT-O), eine prozedurale Sprache, wiederum gefolgt von QVT-Core, einer deklarativen Sprache auf einer niedrigen Abstraktionsebene. Dabei wird QVT-Core primär zur Semantikdefinition von QVT-R verwendet und wird deswegen hier nicht näher betrachtet. QVT-O kann als eigene Sprache oder als Ergänzung zu QVT-R gesehen werden.

In QVT-R werden verschiedene Ausführungsszenarien unterstützt. Transformationen sind entweder uni- oder bidirektional. Auf semantischer Ebene wird zwischen überprüfenden und durchführenden Transformationen unterschieden. Überprüfend bedeutet hier, dass die Transformation die Konsistenz zwischen den beteiligten Modellen prüft und Beziehungen zwischen diesen herstellt. Durchführende Transformationen erzeugen oder aktualisieren ein Modell. An jeder Transformation sind zwei Modelle beteiligt. Dabei wird für jedes Modell ein Parameter definiert, der einen Namen und einen Typ aufweist.

Auf höchster Ebene sind Transformationen Konsistenzrelationen zwischen einer Menge von Modellen. Sie können endogen oder exogen sein. Zudem kann es sich um überschreibende oder Ein-Ausgabe-Transformationen handeln. Transformationen sind inkrementell oder Batchtransformationen.

Hauptunterschied zu ATL ist hier die Multidirektionalität der Regeln. Diese werden automatisch angewendet und dabei implizit geordnet durch ihre Abhängigkeiten untereinander.

QVT-O, die prozedurale Sprache unter den QVT-Sprachen, unterliegt gegenüber QVT-R einigen Einschränkungen. Sie unterstützt weder bidirektionale noch inkrementelle oder überprüfende Transformationen. Dennoch bietet QVT-O einige interessante Aspekte: Jede Transformation operiert hier auf einem oder mehreren Modellen, wobei jedes Modell einer Transformation als Eingabe, Ausgabe oder Ein- und Ausgabe der Transformation zugewiesen werden kann. Zur Spezifikation von imperativen Konstrukten wird in QVT-O imperatives OCL verwendet. Damit werden funktionale (seiteneffektfrei) und prozedurale Erweiterungen (mit Seiteneffekten) unterstützt. Es können Kontrollflüsse spezifiziert und Ausnahmen behandelt werden. Zudem können intermediäre Daten für die Transformation definiert und genutzt werden. Wiederverwendbare Mapping-Regeln definieren die Transformationen, die nach einem Modulkonzept aufgebaut sind.

Jede Transformation operiert auf einer Menge von Modellparametern. Jeder Parameter hat eine Richtung. Eingabeparameter bestimmen das Quellmodell, Ausgabeparameter das Zielmodell und Ein-Ausgabe-Parameter führen zur Modifikation des Modells. Zudem hat jeder Parameter einen Modelltyp, der für eine Menge von Metamodellen steht und festlegt, welche Objekttypen in der Transformation benutzt werden können. (Objekte sind hier Instanzen der Typen, die im Metamodell deklariert wurden.)

Innerhalb des QVT-O-Metamodells ist eine Transformation als Klasse mit Eigenschaften und Operationen definiert. Diese Klasse kann instanziiert und aufgerufen werden. Gleichzeitig kann eine Transformation als Paket aufgefasst werden, innerhalb dessen neue Typen erstellt oder existierende erweitert werden. Diese Deklarationen werden zur Definition von intermediären Daten, die während der Transformationsausführung benötigt werden, genutzt. Den Einstiegspunkt in eine QVT-O-Transformation bildet eine einzigartige Main-Methode, die typischerweise den Aufruf einer Root-Mapping-Operation beinhaltet. Alle weiteren Mapping-Operationen müssen ebenfalls explizit aufgerufen werden.

Weitere Einzelheiten bezüglich QVT sind der Spezifikation der OMG [56] zu entnehmen. Eine Entwicklungsumgebung für QVT-R namens Medini QVT findet sich unter `http://projects.ikv.de/qvt`, eine für QVT-O ist, innerhalb des Eclipse Model-to-Model-Transformation-Projekts, unter `http://projects.eclipse.org/projects/modeling.mmt.qvt-oml` zu finden und wird via Eclipse installiert.

4.2 Ansätze und Werkzeuge zur graphbasierten Transformation

4.2.1 Berliner Werkzeuge basierend auf dem algebraischen Ansatz

Das Attributed Graph Grammar System (AGG), das Tiger EMF Model Transformation Framework (EMF Tiger) und dessen Nachfolger Henshin haben Eines gemeinsam: Alle drei Werkzeuge basieren auf dem algebraischen Ansatz zur Graphtransformation. Sie wurden in der genannten Reihenfolge unter Federführung der Technischen Universität Berlin an verschiedenen Standorten entwickelt. Dabei können alle Modelle letztendlich

in AGG übersetzt, dargestellt und analysiert werden. Die Werkzeuge werden hier gemeinsam betrachtet.

Das Attributed Graph Grammar System (AGG) ist eine hybride Programmiersprache, die grafische Graphtransformationsregeln und Java Quelltext nutzt. Die formale Grundlage für AGG bildet der algebraische Ansatz zur Graphtransformation, wie er in Abschnitt 2.6 skizziert wurde, genauer gesagt der Single-Pushout-Ansatz[3]. Der vorliegende Text basiert auf dem Handbook of Graph Grammars and Graph Transformation, Band 2, Kapitel 14 von Ermel et al. [32].

Ein AGG-Programm besteht zum Einen aus benutzerdefinierten Java-Klassen, zum Anderen aus einer Graphgrammatik. Die Graphgrammatik setzt sich aus einem Startgraphen und einer Menge von Regeln zusammen, deren Aufbau und Ausführung im Folgenden betrachtet werden. Explizite Kontrollflüsse sind in AGG nicht vorgesehen. Die Regelanwendung ergibt sich durch eine AGG-inhärente Regelanwendungsstrategie.

Graphen werden in AGG als simple Graphen dargestellt. Dies bedeutet, dass sie aus Knoten und Kanten bestehen, die Objekte des Graphen oder Graphobjekte genannt werden. Jedes Objekt des Graphen kann zur weiteren Klassifikation beliebig typisiert werden, indem es mit einer Markierung versehen wird. Zudem gibt es die Möglichkeit Objekte zu attribuieren. Diese Attribute dürfen über jedem zulässigen Java-Typ definiert sein. Jede Kante repräsentiert zudem eine gerichtete Verbindung zwischen zwei Knoten, die Quell- und Zielknoten genannt werden.

Eine Graphtransformation kann als Transition des Wirtsgraphen angesehen werden. Der Vorzustand eines Teilgraphen des Wirtsgraphen entspricht dabei der linken Regelseite, der Nachzustand der rechten Regelseite. Die Objekte innerhalb einer Regel werden als Variablen aufgefasst, die instanziiert werden, sobald die Regel auf einen konkreten Wirtsgraphen angewendet wird. Die Anwendung der Regel erfolgt dabei analog zu einer einseitigen Bedingung einer Programmiersprache. Die linke Seite der Regel stellt die Bedingung dar. Das Prüfen der Bedingung entspricht einer Mustersuche nach einer totalen Übereinstimmung, einem totalen Graphmorphismus, der linken Regelseite im Wirtsgraphen. Wird keine totale Übereinstimmung gefunden, ist die Regel nicht anwendbar, da ihre Anwendbarkeitsbedingung nicht erfüllt ist. Werden mehrere Übereinstimmungen gefunden, so wird eine davon ausgewählt. Dies geschieht kontextabhängig entweder zufällig oder durch den Benutzer. Ist die Übereinstimmung gefunden und damit die Bedingung erfüllt, so wird der Übergang zur rechten Regelseite ausgeführt. Hierfür wird eine partielle Übereinstimmung der rechten Regelseite benötigt, um die zu löschenden Knoten von den zu erhaltenden zu unterscheiden. Zudem besteht die Möglichkeit, innerhalb von rechten Seiten Knoten zu vereinigen. Bei der Anwendung der Regel (siehe 2.6) bleibt der Kontext, d.h. alle Elemente des Wirtsgraphen, die zur Musterinstanz gehören. Kanten, deren Quell- oder Zielknoten fehlt, werden hängende Kanten genannt. Sie werden sofort entfernt. Attribute von Graphobjekten können innerhalb einer Regel verändert werden, indem ihnen direkt neue Werte zugewiesen werden. Alternativ kann dies durch Aufruf be-

[3]Genauere Informationen zum Single-Pushout-Ansatz finden sich im Handbook of Graph Grammars and Graph Transformation, Band 1, Kapitel 4 [66].

liebiger Java-Methoden geschehen. Allerdings muss der Rückgabetyp der Java-Methode dem Attributtyp entsprechen.

Zusätzlich können zu jeder Regel negative Anwendbarkeitsbedingungen (NACs) hinzugefügt werden. Diese können analog zur linken Seite der Graphtransformationsregel spezifiziert werden. Während die linke Seite, wie bereits gezeigt, eine positive Anwendbarkeitsbedingung darstellt, dürfen die in den NACs geforderten Bedingungen auf keinen Fall zutreffen. Formal gesehen darf es also keine Abbildung einer NAC auf den Wirtsgraphen geben. Sind mehrere NACs angegeben, darf keine von ihnen zutreffen.

Aufgrund der SPO-Fundierung des Ansatzes ist jede AGG-Transformation komplett, minimal und lokal: Transformationen führen jeden Effekt aus, der in der Transformation spezifiziert wurde (komplett). Es finden auch keine zusätzlichen Änderungen statt (minimal), ausgenommen das Entfernen der hängenden Kanten. Dies geschieht letztlich genau in dem Teil des Wirtsgraphen, für den der Match gefunden wurde (lokal). Es handelt sich dabei um eine überschreibende Transformation.

Da AGG selbst in Java geschrieben ist, ist es plattformunabhängig. Regeln können interpretiert und schrittweise betrachtet werden. Es besteht keine Möglichkeit, Quelltext aus den Regeln zu generieren. Die aktuelle Version von AGG kann unter `http://user.cs.tu-berlin.de/~gragra/agg/` heruntergeladen werden. Dort befindet sich auch die Online-Dokumentation zu AGG.

Das Tiger EMF Model Transformation Framework (EMF Tiger) ist die auf EMF-Modelle spezialisierte Variante des AGG-basierten Werkzeugs TIGER. TIGER ist ein Akronym für „TransformatIon-based Generation of modeling EnviRonments“. Das Werkzeug wurde ursprünglich zur Generierung von grafischen Editoren für Eclipse gebaut, basierend auf Graphtransformationsregeln und dem Graphical Editing Framework (GEF) [37]. Hierbei wurden AGG-Graphtransformationsregeln zur exogenen Ein-Ausgabe-Transformation einer grafischen Sprache in eine andere verwendet.

EMF Tiger spezialisiert den Gedanken der in TIGER realisierten Transformation. Es setzt seinen Fokus auf die Strukturmodifikation von EMF-Modellen. EMF Tiger bietet ein Rahmenwerk zur endogenen, überschreibenden EMF-Modelltransformation basierend auf AGG-Graphtransformationen und wird nun anhand von Biermann et al. [15] dargestellt.

Eine Transformation besteht in EMF Tiger aus einer Regelmenge, welche die Regeln zur Transformation enthält, sowie einem Verweis auf ein bestimmtes EMF-Modell. Es ermöglicht die grafische Spezifikation von AGG-Graphtransformationsregeln, deren Elemente über dem zugehörigen EMF-Modell typisiert sind. Ist zudem eine optionale Startstruktur als Einstiegspunkt in die Transformation definiert, spricht man von einer EMF-Grammatik. Analog zu AGG wird die linke Seite der Regel in EMF Tiger als Vorbedingung für die Anwendung der Regel aufgefasst, die rechte Seite der Regel als Nachbedingung, die nach der Regelanwendung gelten muss. Attribute, die in den Regeln auftreten, müssen im zugehörigen Ecore-Modell definiert sein und dürfen, wie im Abschnitt über AGG beschrieben, verwendet werden. Zudem können auch hier negative Anwendbarkeitsbedingungen (NACs) definiert werden, welche die Regelanwendung

einschränken. Sie stellen in EMF Tiger Objektstrukturen dar, die bei Ausführung der Regel keinesfalls auftreten dürfen.

Die Attribute betreffend enthalten linke Regelseiten und NACs Bedingungen und Variablen, die durch Attributwerte ausgedrückt werden. NACs können bereits in der linken Regelseite genutzte Variablen wiederverwenden, oder neue Variablen, die als Eingabeparameter deklariert wurden. Variablen gelten immer global innerhalb einer Regel. Rechte Regelseiten können zusätzlich Java-Ausdrücke enthalten. In diesen darf wiederum jede gültige Variable vorkommen.

Die beiden Nichtdeterminismen, die in Abschnitt 2.6 bereits besprochen wurden, werden in EMF Tiger eingeschränkt. Die Anzahl der Übereinstimmungen einer Regel kann durch Eingabeparameter limitiert werden. Diese stellen bereits gebundene Objekte des Objektgraphen dar. Zudem kann eine Art Kontrollfluss die Reihenfolge der Regelanwendung steuern. Dieser Kontrollfluss wird entweder durch Java oder durch die Einführung von Ebenen gesteuert, die den Regeln hinzugefügt werden. Bezüglich ihrer semantischen Ausführung werden alle Regeln einer Ebene so lange wie möglich angewendet. Danach wird in die nächste Ebene gewechselt.

Die Anwendung einer Menge von Regeln, auch Transformation genannt, auf ein EMF-Modell erfolgt entweder Schritt für Schritt oder indem alle Regeln so oft wie möglich genutzt werden. Wird bei der Mustersuche ein Muster mehrmals im Objektgraphen gefunden, so muss die Anwendungsstelle der Regel entweder vom Benutzer oder zufällig ausgewählt werden.

Die Ausführung einer Transformation kann durch Interpretation oder durch Generierung von Quelltext und Ausführung desselben erfolgen. Bei der Interpretation wird AGG genutzt.[4] Dazu wird das EMF-Modell zunächst in einen Graphen umgewandelt. Die Anwendung der Transformation selbst erfolgt in AGG. Der transformierte Graph wird wieder in ein EMF-Modell übersetzt. Dieses muss unter Umständen konsistent gemacht werden, indem sichergestellt wird, dass die - von EMF geforderte - Waldstruktur des Modells auch nach der Transformation erhalten ist. Es erfordert das Löschen aller Objekte, die nicht innerhalb des Baumes liegen.

Eine andere Möglichkeit zur Ausführung der Transformation ist die Übersetzung in Java-Klassen, die auf die generierten EMF-Klassen verweisen. Dabei werden für jede Regel zwei Klassen generiert - Eine für die konsistente Ausführung der Regel, die Andere für die Mustersuche. Um eine Regel anzuwenden, muss eine Instanz der Ausführungsklasse erzeugt und dieser separat ein Objekt der Modellinstanz, auf der die Regel ausgeführt werden soll, übergeben werden. Da diese Vorgehensweise manuelles Schreiben von Java-Quelltext erfordert, können Java-Kontrollstrukturen verwendet werden um die Ausführungsreihenfolge der Regeln zu steuern.

Die aktuelle Version von EMF Tiger findet sich unter `https://www.uni-marburg.de/fb12/informatik/arbeitsgebiete/swtechnik/softwaretechnik/taentzer/eclipse_emv`.

[4]Zudem sind weitere Analysen mittels AGG auf dem Graphen möglich.

Henshin - das japanische Wort für Transformation - bezeichnet hier ein Werkzeug zur endogenen, überschreibenden Transformation von EMF-Modellen.[5] Es wird im Folgenden basierend auf Arendt et al. [8] betrachtet.

Henshin benutzt Graphtransformationsregeln als elementare Bestandteile einer Transformation, welche durch Transformationseinheiten strukturiert werden können. Es basiert ebenfalls auf dem algebraischen Ansatz und operiert direkt auf EMF-Modellen. Henshin ist insofern der Nachfolger von EMF Tiger. Mit Henshin es möglich, die Regeln in AGG zu übersetzen, so dass diese nach Konflikten und Abhängigkeiten bezüglich ihrer Anwendung und ihrer Terminierung untersucht werden können.

Henshin erweitert die aus EMF Tiger bekannten Konzepte deutlich. Die Regeln werden durch Anwendbarkeitsbedingungen, Vor- und Nachbedingungen sowie durch flexible Attributberechnungen basierend auf Java oder Java-Skript ergänzt. Zudem werden Transformationseinheiten zur Definition von Kontrollstrukturen bereitgestellt. Diese modularisieren die Regelanwendung.

Das Henshin-Transformations-Metamodell ist ein EMF-Modell, welches das Ecore-Metamodell zur Typisierung verwendet. Jede Graphtransformationsregel besteht, wie in Abschnitt 2.6 beschrieben, aus einer linken und einer rechten Regelseite, die in Henshin zusammengeführt dargestellt werden. Dabei dürfen Attributbedingungen an die komplette Regel gestellt werden. Knoten referenzieren Klassen, Kanten die Referenzen und Attribute ebendiese im Ecore-Modell. Klassen aus dem Ecore-Metamodell werden zur Typisierung der Elemente der Graphtransformationsregel verwendet: Typen von Knoten einer Transformationsregel sind vom Typ EClass, Typen von Kanten vom Typ EReference und Typen von Attributen vom Typ EAttribute.

Die Regeln werden auf den sogenannten EMF-Graphen angewendet. Dieser gruppiert die Elemente vom Typ EObject eines Modells, die zur Mustersuche zur Verfügung stehen. Wird ein Element aus dem EMF-Graphen entnommen, so steht es nicht mehr für die Mustersuche zur Verfügung, kann jedoch in einem anderen Kontext weiterhin verwendet werden.

Um die Regel an der korrekten Stelle des Graphen ausführen zu können, werden positive und negative Anwendbarkeitsbedingungen definiert. Neben der linken Regelseite und den NACs erlaubt Henshin die Definition von logischen Ausdrücken erster Ordnung über Graphbedingungen. Diese sind als atomar anzusehen. Sie fordern die (Nicht)-Existenz bestimmter Muster innerhalb des Modells und können auch Bedingungen über Bedingungen definieren. Dabei spricht man von geschachtelten Bedingungen. In Henshin können Graphen mit einer Formel annotiert werden, die die Anwendbarkeitsbedingungen beinhaltet. Es handelt sich dabei entweder um einen logischen Ausdruck oder um eine Erweiterung der Graphstruktur durch zusätzliche Knoten und Kanten. Eine Regel kann genau dann angewendet werden, wenn alle Anwendbarkeitsbedingungen erfüllt sind.

Die Regeln können in Transformationseinheiten gruppiert werden, die die Regelausführung durch Kontrollstrukturen steuern. Die kleinste Transformationseinheit ist die Regel selbst. Dies ist mit einer einmaligen Regelausführung gleichzusetzen. Zudem exis-

[5] Henshin könnte auch zur exogenen Transformation verwendet werden. Da dies nicht in Zusammenhang mit dem ModGraph-Ansatz steht, wird von einer näheren Betrachtung abgesehen.

tieren Transformationseinheiten zur nichtdeterministischen, zur priorisierten, zur sequenziellen, zur bedingten und zur abgezählten Regelanwendung. Abgesehen von den Regeln selbst kann jede dieser Transformationseinheiten weitere Untereinheiten beinhalten. Zudem besteht die Möglichkeit, Werte zwischen den Transformationseinheiten mittels Parametern auszutauschen. So kann der Objektfluss zwischen den verschiedenen Regeln und Einheiten kontrolliert werden und es können komplexe Transformationen parametrisiert werden. Die Anzahl der Parameter pro Einheit ist beliebig. Jeder Parameter kann einen Wert oder ein EObject beinhalten. ParameterMappings definieren, wie die Parameter einer Transformationseinheit an deren Untereinheiten weitergegeben werden. Die Anwendbarkeit einer Einheit ist gegeben, wenn sie erfolgreich ausgeführt werden kann. Dabei ist „erfolgreich" vom Typ der Einheit abhängig. Beispielsweise ist eine sequenzielle Einheit nur dann erfolgreich, wenn alle ihre Untereinheiten in der gegebenen Reihenfolge erfolgreich abgearbeitet werden konnten. Eine Einheit terminiert, wenn sie erfolgreich oder gar nicht angewendet wurde.

Amalgamationseinheiten sind spezielle Transformationseinheiten. Sie spezifizieren „für-alle"-Operationen auf einem Modell und beinhalten keine weiteren Untereinheiten. Amalgamationseinheiten bestehen aus genau einer Kernregel und mehreren Multiregeln. Sie definieren die Interaktion dieser beiden mittels eines Interaktionsschemas. Die Einbettung einer Kernregel in eine Multiregel erfolgt, indem Abbildungen zwischen der linken Seite der Kernregel und der Multiregel definiert werden. Semantisch wird dabei die Kernregel genau einmal abgebildet und dieser Fund als gemeinsame partielle Abbildung für die Multiregeln genutzt, die ihrerseits möglichst oft abgebildet werden. Damit werden die Änderungen der Kernregel genau einmal angewendet, die der Multiregeln jedoch so oft wie möglich. Eine Amalgamationseinheit ist damit nur anwendbar, wenn ihre Kernregel anwendbar ist.

Die Henshin-Entwicklungsumgebung basiert auf Eclipse und EMF. Sie bietet neben dem von EMF-generierten Baumeditor einen GMF-basierten grafischen Editor an. Dieser unterstützt, im Gegensatz zu den AGG- und EMF Tiger-Editoren, die zusammengeführte Darstellung der Regeln, wie schematisch in Abschnitt 2.6, Abbildung 2.6 rechts dargestellt. Zudem bietet Henshin die Integration von NACs in diese Darstellung. Zur Ausführung der Modelle steht ein Interpreter zur Verfügung. Eine Quelltextgenerierung ist nicht vorgesehen.

Die aktuelle Henshin Entwicklungsumgebung und deren Dokumentation findet sich unter `http://www.eclipse.org/henshin/`. Es handelt sich dabei um ein Eclipse-Projekt.

4.2.2 Die sprachliche Einkleidung des logikorientierten Ansatzes

PROGRES, ein Akronym für "PROgrammed Graph REwriting Systems" [31] (dt. programmierte Graphersetzungssyteme), ist die sprachliche Einkleidung des logikorientierten Ansatzes. Sie wird nun basierend auf dem Handbook of Graph Grammars and Graph Transformation, Band 2, Kapitel 13 von Schürr et. al [70] und Schürr [68] näher erläutert.

Die Sprache PROGRES wurde ursprünglich zum „Rapid Prototyping" mit Graphtransformationssystemen entwickelt und mit drei Designzielen erstellt: (1) Unterschei-

dung von Aktivitäten zur Datendefinition und Datenmanipulation sowie die Nutzung von Graphschemata zur Typprüfung von Graphtransformationsregeln. (2) Kein alleiniger Verlass auf das regelbasierte Paradigma für alle Anwendungen. Zusätzliche Unterstützung der imperativen Programmierung mit Kontrollstrukturen. (3) Berücksichtigung des Nichtdeterminismus von Regeln (und Kontrollstrukturen) durch Tiefensuche und Backtracking.[68]

Um diese Ziele zu erreichen, kombiniert PROGRES regelbasierte, objektorientierte, deduktive und aktive Datenbanksysteme, sowie imperative Programmierkonzepte. Als Datenmodell dienen gerichtete, attribuierte, knoten- und kantenmarkierte Graphen. Dabei entsteht eine hybride Sprache, die Text mit grafischen Elementen für die rechte und linke Seite einer Graphtransformationsregel vereint. Jedem grafischen Element ist eine analoge textuelle Darstellung zugeordnet, die alternativ genutzt werden kann. Die grafischen Elemente dürfen wiederum Textblöcke zur Definition von zusätzlichen Anwendbarkeitsbedingungen enthalten.

Teilgraphentests und Teilgraphersetzungen bilden die atomaren Einheiten aus welchen sich Graphtransformationen zusammensetzen. Zudem werden Kontrollstrukturen aus der imperativen Programmierung verwendet. Diesen wird teilweise nichtdeterministischer Charakter zugeordnet, so dass sie die regelorientierte Ausführung von Graphtransformationen unterstützen.

Teilgraphentests werden in PROGRES als Graphabfragen bezüglich eines Subgraphen aufgefasst. Grundsätzlich gibt ein solcher Test einen boolschen Wert zurück, der den Testerfolg widerspiegelt. Teilgraphentests dürfen Ein- und Ausgabeparameter beinhalten, die für Knoten oder Knotenmengen stehen. Zur Definition sind Pfade, Attribute, Kanten, Knoten und negative Knoten verfügbar, auf die im Weiteren noch eingegangen wird.

Teilgraphersetzungen bilden die rechten Regelseiten der Transformation. Sie repräsentieren atomare Transformationen und erzeugen und modifizieren schematreue Graphen. Teilgraphersetzungen werden analog zu einem Teilgraphentest spezifiziert, allerdings lassen sie keine negativen Knoten und Kanten zu.

Graphtransformationen bestehen demnach aus einer linken Seite, dem Teilgraphentest und einer rechten Seite der Teilgraphersetzung (siehe 2.6). Zudem können in PROGRES lokale Einbettungsüberführungsfunktionen erstellt werden, die die Verbindung des neuen Teilgraphen mit dem verbleibenden Wirtsgraphen modifizieren. Dabei können Kanten gelöscht, kopiert oder ummarkiert werden.

Bei der Ausführung einer Graphtransformation werden nach Teilgraphentest, Teilgraphersetzung und Einbettungsüberführung die Attribute transferiert. Dazu werden den eigenständigen Attributen der Knoten Werte zugewiesen. Zudem sind Vor- und Nachbedingungen an die Graphtransformationsregeln als Zusicherungen möglich.

Grundsätzlich wird dabei jeder Graph in einer Datenbank repärasentiert. Dies bedeutet, dass alle Operationen zur Manipulation einer Datenbank als Auswahl und Transformation eines Teilgraphen angesehen werden. Dies bezeichnet man als Transaktion. Analog zu Datenbanksystemen folgen sie dem ACID-Prinzip. Im Einzelnen besteht eine Transaktion also aus der Identifikation eines Teilgraphen, die durch einen Teilgraphentest beschrieben wird und den Transformationsschritten, die durch die Graphtransformati-

onsregel beschrieben werden.

Die Ausführung jeder Graphtransformationsregel wird durch imperative, (nicht-)deterministische Kontrollstrukturen gesteuert. Es ist ebenso möglich, ganze Teile einer PROGRES-Spezifikation als Transaktion zu markieren, so dass diese aus Regeln und deren zugehörigem Kontrollfluss besteht. Da bei Fehlschlagen eines Bestandteils der Transaktion diese insgesamt rückgängig gemacht werden muss, unterstützt PROGRES eine Prolog-ähnliche Tiefensuche und Backtracking. Ein einfacher Abbruch würde hier nicht ausreichen, da die Graphsuche nichtdeterministische Anteile beinhaltet.

Bei genauerer Betrachtung der Syntax zur Spezifikation von Graphtranformationsregeln stellen sich typisierte Knoten und Kanten als Basiselemente zur Definition von Graphklassen heraus. Knoten sind über Knotenklassen typisiert. Knotenklassen definieren demnach die gemeinsamen, statischen Eigenschaften der Knotentypen. Zudem vererben sie diese Eigenschaften an die Knotentypen. Knotenklassen können Vererbungshierarchien aufbauen, wobei Mehrfachvererbung erlaubt ist. Zum besseren Verständnis können Knotenklassen mit abstrakten Klassen und Knotentypen mit finalen, konkreten Klassen der objektorientierten Programmierung verglichen werden. Außerdem besteht in PROGRES die Möglichkeit, neben einfachen Knoten Mengenknoten zu definieren. Diese Mengenknoten können auf der rechten Seite einer Regel lediglich identisch ersetzt werden. Zudem können alle Knoten einer Regel optional oder obligatorisch sein.

Kanten sind als binäre Relationen zwischen Knoten definiert. Sie sind ebenfalls typisiert, indem Kantentypen zwischen Knotentypen bzw. Knotenklassen definiert werden. Kantentypen sind gerichtet, können aber in beide Richtungen durchlaufen werden. Zudem sind ihnen Kardinalitäten zugeordnet. Diese limitieren durch Angabe oberer und unterer Schranken die Anzahl der Kanten pro Knoten. Dabei kann eins oder beliebig viele für die obere Schranke und null oder eins für die untere Schranke angegeben werden.

Knoten können - im Gegensatz zu Kanten - typisierte Attribute beinhalten. Es existieren drei Möglichkeiten zur Typisierung. Sie erfolgt entweder über einem der drei in PROGRES definierten Standardtypen (integer, boolean, string) oder über einem Graphtypen, d.h. einer Menge von zusammengehörigen Knoten- und Kantentypdeklarationen. Dabei sollten Graphtypen bevorzugt werden. Alternativ kann ein Attribut über einem Typ, der in der Wirtsprogrammiersprache C definiert und importiert wurde, getypt werden. Zudem können Funktionen importiert werden, die im Zusammenhang mit dem Datentyp stehen. Analog zu den Kanten dürfen Attribute Kardinalitäten aufweisen. Attribute modellieren die atomaren Eigenschaften von Knotenobjekten. Hierzu werden drei Arten von Attributen bezüglich ihrer Wertzuweisung unterschieden. Intrinsische Attribute sind diejenigen, die ihren Wert klassisch zugewiesen bekommen. Sie können als Schlüsselattribute deklariert werden, d.h. sie identifizieren den Knoten eindeutig und werden indiziert. Abgeleitete Attribute berechnen ihren Wert aus den Werten anderer Attribute. Meta-Attribute beinhalten Knoteneigenschaften, die den selben Wert für alle Instanzen des Knotentyps haben. Sie definieren objektorientiert gesprochen statische Attribute. Attribute können zudem Integritätsbedingungen an Knoten angeben.

Zur Definition von Graphtransformationsregeln stehen zusätzlich Pfaddeklarationen als Sprachkonstrukte zur Verfügung. Diese dienen der Definition von abgeleiteten Beziehungen zwischen Knoten. Sie werden in Graphtests und Graphtransformationsregeln

als Anwendbarkeitsbedingungen genutzt. Pfade beinhalten Pfadausdrücke, die textueller oder grafischer Natur sind. Grafische Pfade können durch alle Konstrukte, die für Graphtests (sprich linke Seiten von Regeln) zugelassen sind, beschrieben werden. Die Pfadausdrücke werden in drei Kategorien unterteilt: Kantenoperatoren, Attributbedingungen und Restriktionen. Pfadausdrücke können mittels Operatoren zu komplexeren Pfadausdrücken zusammengefasst werden. Zudem gibt es bedingte und iterierende Pfadausdrücke.

Die Konsistenz von Graphen kann sowohl durch globale Integritätsbedingungen als auch durch Bedingungen auf Knoten geprüft werden. Integritätsbedingungen prüfen das Modell auf Konsistenz gegenüber Invarianten. Spezielle Integritätsbedingungen enthalten Reparaturaktionen. Sie werden aktive Integritätsbedingungen genannt. Diese Reparaturaktionen werden automatisch ausgeführt, sobald die zugehörige Bedingung verletzt ist und damit ein verbotener Graphzustand erreicht wird.

PROGRES unterstützt die Definition modularer Pakete, die die Regeln und Tests gruppieren. Diese dürfen sich durch Import-Beziehungen referenzieren. Die Import-Beziehungen sorgen dafür, dass ein anderes Paket auf die öffentlichen Resourcen eines Pakets zugreifen kann, nicht jedoch auf die privaten Resourcen.

Die Entwicklungsumgebung bietet einen syntaxgesteuerten Editor für textuelle und grafische Elemente sowie eine Validierungsfunktion, die textuelle Komponenten und Typverträglichkeitsregeln prüft. Zur Ausführung einer PROGRES-Spezifikation stehen ein Interpreter und ein inkrementeller Compiler zur Verfügung. Der Interpreter bietet einen Debugger an, der schrittweise (vorwärts und rückwärts) die Transformation ausführt. Der Compiler erzeugt lauffähigen C- oder Java-Quelltext.

4.2.3 Story-basierte Ansätze

Die Werkzeuge Fujaba, MDELab und eMoflon nutzen alle drei den sog. Story-basierten Ansatz, welcher, bezüglich der in Abschnitt 2.6 skizzierten Ansätzen unter der Kategorie algorithmischer Ansatz anzusiedeln ist. Es handelt sich hierbei um einen Ansatz, der programmierte Graphersetzungssysteme nutzt. Das UML-nahe Fujaba bildet historisch gesehen den Anfang. MDELab und eMoflon realisieren den Story-basierten Ansatz für EMF auf verschiedene Arten. Während eMoflon auf Fujaba aufbaut und dieses erweitert, ist MDELab eine Reimplementierung, die sich auf die EMF-basierte Interpretation von Modellen spezialisiert hat. Die Werkzeuge werden im Folgenden beginnend mit Fujaba erläutert.

Fujaba, ein Akronym für „From Uml to Java And Back Again“, ist eine objektorientierte Modellierungsumgebung, die UML-Klassendiagramme zur Modellierung der statischen Struktur sowie Storydiagramme zur Verhaltensmodellierung bietet. Fujaba wird hauptsächlich an der Universität Kassel entwickelt und in diesem Abschnitt anhand von Norbisrath et al. [55] und Zündorf [83] erklärt.

Fujaba verwendet UML-Klassendiagramme zur Strukturbeschreibung und Storydiagramme zur Beschreibung des Verhaltens. Ein Storydiagramm realisiert genau eine Methode aus dem Klassendiagramm. Ein Storydiagramm basiert auf einem UML-Aktivitätsdiagramm und stellt daher ein Kontrollflussdiagramm dar. Der Kontrollfluss hat einen

definierten Startpunkt und einen oder mehrere definierte Endpunkte. Dazwischen befinden sich Aktivitäten, die durch Kontrollflusselemente verbunden sind. Kontrollflusselemente sind zunächst Sequenzen, While-Schleifen und Wenn-Dann-Bedingungen (analog zu den UML-Aktivitätsdiagrammen). Zudem können Ausnahmen durch spezielle Ausnahme-Kontrollflusselemente behandelt werden und Rückgabewerte definiert werden. Letztere werden am Endpunkt des Storydiagramms definiert. Hier können unter anderem rekursiv Methoden aufgerufen werden.

Aktivitäten werden in der Regel einmal ausgeführt. Eine spezielle Aktivität ist die For-Each-Aktivität. Diese führt ihren Inhalt und eventuell als zugehörig gekennzeichnete folgende Aktivitäten solange aus, bis keine Anwendungsmöglichkeit mehr besteht.

Der einfachste Bestandteil einer Aktivität in einem Storydiagramm ist ein Statement, das ein reines Java-Quelltextfragment beinhaltet. Da die Statements innerhalb der Aktivität beliebig angeordnet sein können, ist es nötig, deren Ausführungsreihenfolge durch eine fortlaufende Nummerierung sicherzustellen, sofern sie Operationsaufrufe darstellen. Handelt es sich um eine reine Statementaktivität, also ein Java-Quelltextfragment, so ist keine Nummerierung nötig. Zudem dürfen Aktivitäten boolsche Bedingungen enthalten, die als Abfrage an den aktuellen Zustand des Modells dienen. Sie haben zwei ausgehende Kontrollflusskanten für den Erfolg bzw. das Scheitern dieser Bedingung. Selbiges gilt für komplexere Aktivitäten. Sie werden Story-Muster genannt und beinhalten Objekt-Muster, die nichts anderes als Graphtransformationsregeln darstellen. Die Objekte innerhalb der Story-Muster werden zur Laufzeit auf die aktuelle Objektstruktur abgebildet. Gelingt dies, werden die spezifizierten Aktionen ausgeführt und das Story-Muster wird über die Kante, welche den Erfolgsfall repräsentiert, verlassen.

Die Objekte innerhalb der Story-Muster werden in gebundene und ungebundene Objekte unterschieden, die durch Links verbunden sind. Objekte sind über den Klassen im UML-Klassendiagramm typisiert, Links über deren Assoziationen. Wird ein Objekt in einem Story-Muster gebunden, so kann es in jedem darauffolgenden als gebunden angenommen und durch seinen Namen eindeutig identifiziert werden. Bereits zu Anfang ist das this-Objekt gebunden, welches - analog zu Java - dem Objekt entspricht, auf dem die Methode aufgerufen wurde. Das gilt ebenfalls für die Parameter der Methode. Ungebundene Objekte werden während der Mustersuche gebunden. Objekte können Attributbedingungen und -zuweisungen enthalten. Darüber hinaus können auf jedem Objekt - mittels Java-Statements - Methoden aufgerufen werden. Jedes Objekt und jeder Link kann erzeugt, erhalten oder - unabhängig von seinem Kontext - gelöscht werden. Neben den Links dürfen, analog zu PROGRES, Pfade definiert werden. Fujaba stellt zur Definiton der Pfadausdrücke eine eigene Ausdruckssprache bereit. Zudem besteht die Möglichkeit, negative Objekte und Links einzuführen. Sie bilden eine Alternative zur NAC, stellen also ebenfalls nicht erlaubte Sachverhalte dar. Fujaba bietet mehrwertige und optionale Objekte an, die lediglich erhalten oder gelöscht werden dürfen.

Fujabas grafische Modellierungsumgebung wurde zunächst außerhalb von Eclipse entwickelt. Zudem bietet Fujaba4Eclipse eine in Eclipse integrierte Variante. Fujaba erzeugt durch Modell-zu-Text-Transformationen lauffähigen Java-Quelltext aus den Diagrammen. Eine Möglichkeit zur Interpretation der Diagramme besteht ebenso.

Fujaba und dessen Dokumentation ist unter `http://www.fujaba.de/` verfügbar.

eMoflon nutzt und erweitert Fujaba. Es ist ein Werkzeug zur Erstellung von Werkzeugen auf Basis von uni- und bidirektionalen Modelltransformationen. Dazu wird eine Umgebung zur Spezifikation von Klassendiagrammen, Story-basierten Graphtransformationsregeln und Tripelgraphgrammatiken bereitgestellt. eMoflon wird am Fachgebiet Echtzeitsysteme (ES) der Technischen Universität Darmstadt entwickelt und im Folgenden anhand von Anjorin et al. [6] und Leblebici et al. [48] erklärt. eMoflon ist in zwei Frontends zur Metamodellierung und einem Backend, um Quelltext zu generieren und mit diesem zu arbeiten, modularisiert. Es stellt ein textbasiertes Frontend auf Basis von Eclipse und ein grafisches Frontend, das auf dem kommerziell UML-Werkzeug Enterprise Architect aufbaut, zur Verfügung.

Das Enterprise-Architect-Frontend erlaubt das Erstellen von Metamodellen mit UML-Klassendiagrammen, nutzt Storydiagramme zur unidirektionalen Modelltransformation und Tripelgraphgrammatiken zur bidirektionalen Modelltransformation. Zudem bietet das Frontend ein Interface zum Backend: Klassendiagramme, Storydiagramme und Tripelgraphgrammatiken werden in einem Ecore- bzw. EMF-konformen XMI persistiert. Es dient als Austauschformat zwischen Frontend und Backend.

Das eMoflon-Backend besteht aus mehreren Eclipse-Plugins. Hier wird eine EMF konforme Quelltextgenerierung aus den Klassendiagrammen, den Storydiagrammen und den Tripelgraphgrammatiken ermöglicht.

Die Verarbeitung der Daten aus dem Frontend beginnt mit dem Laden der serialisierten Modelle. Im nächsten Schritt werden die Tripelgraphgrammatiken in eine Menge von Storydiagrammen umgewandelt. Diese erzeugten und die geladenen Storydiagramme werden zusammen mit dem Klassendiagramm in Fujaba umgewandelt. Fujaba's Modeltransformationseinheit CodeGen2, die mit EMF-kompatiblen Templates rekonfiguriert wurde, wird nun genutzt, um Quelltext zu erzeugen. Zudem wird aus dem geladenen Klassendiagramm mittels der EMF-Codegenerierung (siehe Abschnitt 3.4) Quelltext erzeugt. Die entstandenen Quelltexte werden automatisch verschmolzen, so dass ein lauffähiges Programm entsteht.

Die aktuelle Version von eMoflon sowie die Dokumentation dazu finden sich unter `http://www.moflon.org/`.

MDELab beinhaltet, neben anderen Werkzeugen, ein Graphtransformationswerkzeug namens SDMTools, das Ecore-Klassendiagramme zur Beschreibung der statischen Struktur nutzt. Es bietet einen grafischen Editor und einen Interpreter für Storydiagramme, die hier anhand von Giese et al. [36] skizziert werden. MDELab wird am Hasso-Plattner-Institut in Potsdam entwickelt und ist von Beginn an für Eclipse und EMF entworfen worden. Es ist eine Reimplementierung von Fujaba, die UML-Klassendiagramme durch Ecore-Klassendiagramme ersetzt und OCL zur Spezifikation von Bedingungen unterstützt.[6] Zudem ist keine Quelltext-Generierung vorgesehen, so dass Java-Statements, die sich auf generierten Quelltext beziehen würden, ausfallen. Sie werden durch spezielle Aktionsknoten ersetzt. Das Storydiagramm-Metamodell ist selbst ein Ecore-Modell. Storydiagramme sind Instanzen dieses Metamodells und die Objekte im Storydiagramm

[6] Eine OCL-Integration gab es zudem in einem prototypischen Ansatz für Fujaba, der es nicht bis zum Release Stadium geschafft hat [74].

sind über einem Ecore-Klassendiagramm typisiert. Storydiagramme in MDELabs SDM-Tools sind auf Interpretation spezialisiert. Der Interpreter nutzt ausschließlich die - in dynamischem EMF verwendeten - generischen Methoden. Die Interpretation der Storydiagramme über einem EMF-Modell entspricht einer endogenen, überschreibenden Transformation. Um diese anzustoßen, muss explizit eine Methode des Interpreters aufgerufen werden. Dieser werden das Storydiagramm, das interpretiert werden soll, sowie die Parameter der Methode, die es implementiert und das Kontextobjekt übergeben. Dies bedeutet, dass der Endanwender auf den Interpreter beschränkt ist, um mit Modellinstanzen zu arbeiten. Eine Generierung von Quelltext ist hier nicht vorgesehen, Möglichkeiten zum Debuggen sind vorhanden.

Die aktuelle Version von MDELab kann unter `https://www.hpi.uni-potsdam.de/giese/public/mdelab/` heruntergeladen werden. Dort befindet sich auch die Dokumentation.

4.2.4 Andere Ansätze

In diesem Abschnitt werden weitere Ansätze zur Graphtransformation behandelt, die sich in keine der eben eingeführten Kategorien eingliedern lassen.

GReAT, ein Akronym für „Graph REwriting And Transformation“, ist eine grafische Sprache, die es erlaubt, Graphtransformationen zwischen verschiedenen domänenspezifischen Modellierungssprachen zu erstellen. Sie wird hier anhand von Agrawal et al. [5] und Balasubramanian et al. [9] erläutert. GReAT besteht aus drei Sprachen: die erste Sprache dient der Spezifikation von Graphmustern, die zweite der Erstellung von Transformationsregeln und die dritte Sprache wird zur Sequenzierung beziehungsweise zur Spezifikation von Kontrollflüssen verwendet. Die Ein- und Ausgabemodelle der Transformation werden zusätzlich durch UML-Klassendiagramme definiert, die als Metamodelle agieren. Zudem erlaubt GReAT die Erstellung von heterogenen Klassendiagrammen, die die Beziehungen zwischen Ein- und Ausgabemodellen herstellen.

Eine Transformation wird in GReAT über einer Menge von Ein- und Ausgabemodellen und einem heterogenen Klassendiagramm spezifiziert. Die Transformationsregel selbst setzt sich aus mehreren Bestandteilen zusammen, die im Folgenden skizziert werden. Das Graphmuster der Regel ist aus Knoten, die über Klassen des zugehörigen Eingabemodells, und Kanten, die über Assoziationen dieses Modells typisiert sind, aufgebaut. Beiden dürfen beliebig Kardinalitäten zugewiesen werden. Es besteht die Möglichkeit NACs zu spezifizieren. All diese Knoten und Kanten werden zur Mustersuche für den Wirtsgraphen verwendet. Die Aktionen innerhalb der Regel definieren, ob ein Element gebunden, gelöscht oder erzeugt werden soll. Eingangs- bzw. Ausgangsinterfaces bestehen aus Ports und dienen zur Übergabe von Graphobjekten an die Regel bzw. zur Weitergabe von Graphobjekten an andere Regeln.

Ein Wächter ist ein boolscher Ausdruck, der bestimmt, ob die Aktionen der Regel nach einer erfolgreichen Mustersuche auf dem Wirtsgraphen ausgeführt werden sollen. Durch Attributabbildungen können übereinstimmende Attribute der Knoten und Kanten neue Werte erhalten. Zusätzliche Bedingungen legen fest, welche während der Mustersuche gefundenen Übereinstimmungen im Wirtsgraphen transformiert werden.

Die Sprache zur Kontrollflussspezifikation stellt verschiedene Möglichkeiten bezüglich der Ausführungsreihenfolge der Regel bereit. Die sequenzielle und rekursive Ausführung entsprechen den aus Programmiersprachen bekannten Kontrollstrukturen. Zusätzlich kann eine Regel nichtdeterministisch, hierarchisch oder bedingt ausgeführt werden. Nichtdeterministische Ausführung bedeutet, dass eine Regel aus einer vorgegebenen Regelmenge ausgewählt wird. Hierarchische Ausführung bedeutet, dass die Regeln zuvor in hierarchisch aufgebaute, strukturierte Blöcke gruppiert wurden. Es existieren zwei Arten von Blöcken, die sich in der Verarbeitung der ihnen übergebenen Graphobjekte unterscheiden. Die bedingte Anwendung besteht aus Testblöcken, die wiederum verschiedene Fallunterscheidungsblöcke beinhalten, die solange ausgeführt werden, bis der erste Fall erfolgreich ist.[7]

Zur Ausführung der Transformationen stehen ein Interpreter, ein darauf aufbauender Debugger und ein C++-Generator zur Quelltextgenerierung zur Verfügung. Diese sind auch in der aktuellen Version von GReAT enthalten. Sie findet sich unter `http://www.isis.vanderbilt.edu/tools/GReAT`.

GrGen.NET ist ein domänen-unabhängiges Graphersetzungssystem. Es basiert auf dem algebraischen Ansatz, ist jedoch am Karlsruher Institute of Technology (KIT) entstanden und lässt sich, wie im Folgenden klar wird, nicht den Berliner Ansätzen aus Abschnitt 4.2.1 zuordnen. Bei der Entwicklung von GrGen.NET standen die Ausdrucksmächtigkeit der Sprache, die Ausführungsgeschwindigkeit und die Benutzerfreundlichkeit sowie die korrekte Umsetzung des algebraischen Ansatzes, genauer des SPO-Ansatzes (siehe [66], Kap. 4), im Mittelpunkt. Es wird hier anhand von Jakumeit et al.[43] und Geiß et al. [35] betrachtet.

GrGen.NET setzt sich aus zwei Komponenten zusammen: dem Graphersetzungssystem GrGen und einer Umgebung zum Rapid Prototyping, die sich wiederum aus der Konsole GrShell und dem Graph-Viewer yComp zusammensetzt.

Die Einbindung von GrGen bietet - in Verbindung mit seiner Ausführungsumgebung libGr - die Basisfunktionalität des Systems. Zur Beschreibung von Graphen nutzt GrGen attribuierte, getypte und gerichtete Multigraphen, die Mehrfachvererbung auf Knoten- und Kantentypen sowie gerichtete und ungerichtete Kanten erlauben. Die Definition der Typen erfolgt dabei gesondert. Die Graphtransformation selbst wird in einer textuellen DSL dargestellt. Graphtransformationen in GrGen bieten Graphtests - die je aus einem zu prüfenden Graphmuster bestehen - und Graphtransformationsregeln an. Dabei ist auch hier eine Regel aus einem Test (einem Graphmuster) und einem Anwendungsteil aufgebaut. Das Graphmuster beinhaltet typisierte Knoten und Kanten. (Bei Kanten ist die Typ-Angabe optional.) Dabei ist die zusätzliche Spezifikation von NACs, Typ- und Attributbedingungen möglich. Zur weiteren Einschränkung ihrer Anwendbarkeit sind die Transformationen parametrisierbar, so dass bestimmte Objekte innerhalb des Graphmusters bereits vorgegeben sind. Des Weiteren es ist möglich, die Mustersuche homomorph zu gestalten, um zwei Knoten im Graphmuster auf dasselbe Objekt im Wirtsgraphen abzubilden.

[7] Dies ist analog zu den Mehrfachverzweigungen mit switch-case in objektorientierten Programmiersprachen zu sehen.

Bezüglich des Anwendungsteils einer Regel unterscheidet GrGen Modifikations- und Ersetzungs-Blöcke. Ein Modifikations-Block erhält bei seiner Anwendung alle Graphelemente, die nicht explizit gelöscht werden sollen. Bei Anwendung eines Ersetzungs-Blocks werden nur die explizit auf der rechten Seite angegebenen Elemente erhalten. Kommen in einem Ersetzungs-Block neue Elemente vor, so wird dies als Deklaration der Elemente aufgefasst und sie werden erzeugt. Zudem können in beiden Blockarten Attributwerte neu berechnet und Elementen ein neuer Typ zugewiesen werden. Bei der Erzeugung neuer Elemente kann deren Typ dynamisch festgelegt werden, indem ein Typ eines anderen bei der Mustersuche gebundenen Elements verwendet wird. Zusätzlich kann jede Regel Elemente, die nicht gelöscht wurden, zurückgeben.

GrGen bietet Möglichkeiten zur Steuerung der Ausführung der Regeln an. Dazu werden logische und reguläre Ausdrücke verwendet. Außerdem können innerhalb der Kontrollflusssprache Variablen deklariert werden, die den Rückgabewert einer Regel speichern und diesen gegebenenfalls als Parameter an eine andere Regel übergeben.

Der GrGen-Generator liest Dateien, die Graph-Modelle und Graphtransformationsregeln beinhalten, ein und erzeugt daraus effiziente C#-Programme. Diese Programme nutzen die GrGen-Laufzeit-Bibliotheken, welche unter anderem eine Sprache zur Kontrolle der Regelanwendung anbietet. Graphmodelle, Regeln und Sequenzen von Regeln können von jeder .NET-Sprache aus aufgerufen werden. Des Weiteren bietet GrGen.NET einen grafischen Debugger und die interaktive oder dateibasierte Ausführung der Regeln, die auch schrittweise erfolgen kann.

Die aktuelle Version von GrGen.NET sowie die Dokumentation des Werkzeugs finden sich unter `http://www.info.uni-karlsruhe.de/software/grgen/`.

Viatra2 oder VIsual Automated model TRAnsformations 2 vereint Graphtransformationen mit abstrakten Zustandsautomaten, um graphbasierte Modelle zu manipulieren. Viatra2 wird im Folgenden anhand von Várró und Balogh [76, 10] beschrieben.

Hauptziel bei der Entwicklung des Viatra2-Transformations-Rahmenwerks ist die Unterstützung von präziser modellbasierter Systementwicklung unter Zuhilfenahme von unsichtbar bleibenden formalen Methoden. Viatra2 besteht selbst aus drei textuellen DSLs: Die erste dient der Metamodellierung, die zweite der Musterspezifikation und die verbleibende dritte der regelbasierten Modelltransformation mit Graphtransformationsregeln und abstrakten Zustandsautomaten.

Die Metamodellierungssprache von Viatra2 baut auf dem „Visual and Precise Metamodelling“-Ansatz von Várró und Pataricza [77] auf, der Modelle einheitlich mittels Modellelementen, die in Entitäten und Beziehungen unterschieden werden, darstellt. Da für diesen Ansatz lediglich eine abstrakte Syntax definiert wurde, wird die passende konkrete Syntax namens Viatra Textual Metamodeling Language (VTML) durch Viatra2 bereitgestellt. In VTML ist jedes Element mittels eines qualifizierten Namens modellweit eindeutig bezeichnet und zudem durch seinen Namen eindeutig innerhalb seines Containers. VTML unterstützt die Vererbung von Modellelementen und deren Typisierung mittels anderer Modellelemente. Zudem sind Entitäten immer innerhalb eines Containers. Referenzen können Kardinalitäten zugeordnet werden und sie dürfen beliebige Modellelemente als Quelle und Ziel aufweisen. Durch diesen Aufbau können MOF-Modelle simuliert werden. Viatra2 wird zudem in den Kontext der für den ModGraph-Ansatz

relevanten Werkzeuge aufgenommen.

Die Sprache Viatra Textual Command Language (VTCL) zur Spezifikation von Modelltransformationen in Viatra2 beinhaltet Graphmuster, die als atomare Einheit aufgefasst werden und Bedingungen an ein Modell stellen. VTCL unterstützt dabei einfache und negative Graphmuster, die in einer Prolog-ähnlichen Syntax spezifiziert werden. Dies bedeutet, dass alle Graphmuster einen Namen, Parameter und einen Rumpf besitzen. Der Rumpf darf wiederum Elemente aus VTML sowie negative Graphmuster enthalten. Dabei darf ein negatives Graphmuster kein weiteres negatives Muster beinhalten. Zudem können Bedingungen innerhalb des Musters gestellt werden. Ein Graphmuster kann ein anderes aufrufen oder mehrere alternative Rümpfe spezifizieren. Letzteres ist insbesondere bei der Nutzung rekursiver Graphmuster von Bedeutung: Der erste Rumpf enthält normalerweise das Abbruchkriterium, alle weiteren bilden rekursive Aufrufe.

Die Sprache zur Erstellung von Modelltransformationen enthält sowohl Möglichkeiten zur Modell-zu-Modell-Transformation, als auch zur Modell-zu-Text-Transformation. Bei einer Modell-zu-Modell-Transformation werden Graphtransformationsregeln zur Spezifikation von elementaren Modellmanipulationen und abstrakte Zustandsautomaten als Kontrollflüsse zur Steuerung der Regelausführung verwendet.

Viatra2 bietet eine textuelle Darstellung der klassischen Regelnotation an. Zudem existiert eine eine zusammengeführte Notation (siehe Abschnitt 2.6). Bei Nutzung der klassischen Notation wird die linke Regelseite durch eines der soeben erklärten Graphmuster dargestellt. Hier besteht die Möglichkeit, dass Transformationsregeln sich Graphmuster teilen. Sie dienen als Vorbedingung, während die rechte Seite klassisch als Nachbedingung aufgefasst wird. Rechte Seiten enthalten keine negativen Graphmuster. Beide Seiten dürfen weitere eigenständig definierte Muster aufrufen. Zudem darf eine Regel Aktionen beinhalten, die nach einer erfolgreichen Transformation ausgeführt werden.

Abgesehen von klassischen Transformationen über in VTML definierten Modellen unterstützt Viatra2 generische Transformationen. Diese agieren auf den im „Visual and Precise Metamodelling“-Ansatz definierten Elementen: Entität und Beziehung.

Kontrollstrukturen werden in Viatra2 mit Hilfe von abstrakten Zustandsautomaten dargestellt. Diese können Graphtransformationen (wiederholt) aufrufen. Sie bieten zudem eine Möglichkeit der Definition von Variablen und des Aufrufs beliebiger Java-Methoden. Darüber hinaus können sehr einfache Modellmanipulationen mittels der Zustandsautomaten spezifiziert werden.

Modell-zu-Text-Transformationen werden in Viatra durch print-Anweisungen realisiert, die programmiersprachliche Ausdrücke oder Text enthalten. Diese werden gemäß der vorgegebenen Programmiersprache formatiert. Die Anweisungen werden innerhalb der Zustandsautomaten in die Transformation eingebunden.

Viatra2 ist als Eclipse-Plugin verfügbar. Download und Dokumentation finden sich unter `http://www.eclipse.org/viatra2/`.

4.3 Fazit

Hier wurden mit dem ModGraph-Ansatz in Verbindung stehende Ansätze und Werkzeuge besprochen. Dabei wird auf das Eclipse Modeling Framework (EMF) eingegangen. Es bietet eine solide und in Industrie und Forschung anerkannte Möglichkeit, Software teilweise modellgetrieben zu entwickeln. Dazu werden Ecore-Modelle verwendet, die eine ausgereifte und mächtige Möglichkeit zur Strukturmodellierung bei überschaubarer Anzahl an Konstrukten darstellen. EMFs Modell-zu-Text-Transformationsmöglichkeiten ermöglichen eine Quelltextgenerierung aus den Ecore-Klassendiagrammen an. Die Spezifikation des Verhaltens auf Modellebene bleibt jedoch unberücksichtigt.

Zudem werden Modell-zu-Modell-Transformationsansätze beschrieben. Dabei bietet Xcore neben einer textuellen Syntax für Ecore, eine Möglichkeit dem Modell Verhalten hinzuzufügen durch Integration der Java-nahen Sprache Xbase. Dies ermöglicht die Generierung von voll funktionsfähigem Java-Quelltext. Xbase ist jedoch relativ Java-nah, so dass an dieser Stelle Einbußen bezüglich des Abstraktionsniveaus entstehen. Dennoch ist es abstrakt genug, um komplexe Zusammenhänge übersichtlich darzustellen, indem zum Beispiel Lambda-Ausdrücke unterstützt werden. Diese erlauben die Nutzung vordefinierter Filterfunktionen um komplexe Schleifen (in Java mehrere Zeilen Quelltext) sehr kompakt (ein bis zwei Zeilen) darzustellen. Es ist sehr gut geeignet, um einfache Änderungen oder prozedurale Abläufe darzustellen. Komplexe, strukturverändernde Operationen an einem Modell können mit Xcore spezifiziert werden, jedoch ist der Aufwand deutlich höher als mit Graphtransformationsregeln. Ein wichtiger Grund dafür ist, dass in Xcore die Mustersuche, die zum Auffinden der Objekte in der Modellinstanz benötigt wird (siehe Abschnitt 6.2.2), explizit angegeben werden muss, während alle Graphtransformationsprachen diese implizit mitgeben. Im weiteren Verlauf dieses Kapitels wurden ATL und QVT skizziert, die ebenfalls EMF-Modelle transformieren.

Den Abschluss bilden Werkzeuge, die Graphtransformationen anwenden. Bei einigen Werkzeugen, wie AGG, liegt der Fokus auf der theoretischen Fundierung, andere Werkzeuge wie PROGRES, Fujaba und Viatra2 sind eher praxisorientiert. Werkzeuge, wie eMoflon und MDELab nutzen das EMF-Rahmenwerk. Die Art der Steuerung der Ausführungsreihenfolge der Transformationen divergiert. Nur einige Werkzeuge erlauben eine gezielte Steuerung durch die explizite Angabe von Kontrollflüssen. Viele bevorzugen grafische Elemente zur Darstellung der Transformation. Die Regeln werden entweder interpretiert oder zu ihrer Ausführung in eine Programmiersprache übersetzt.

Betrachtet man die existierenden Ansätze als Einheit, so wird eines deutlich: grafische Verhaltensmodellierung in EMF auf Modellebene ist verbesserungswürdig. Es existiert bislang kein leichtgewichtiges Werkzeug, das nahtlos in EMF integriert ist, mit EMF gebaut wurde und gleichzeitig die Modellierung von Verhalten mit Graphtransformationsregeln anbietet. Bisher sind es größtenteils eigenständige Werkzeuge (z.B. Fujaba), die an EMF angepasst wurden. EMF-kompatibler Quelltext aus den Regeln wird entweder in Java über Umwege (z.B. mittels der Fujaba-Quelltextgenerierung in eMoflon) oder durch das manuelle Einbinden von generiertem Quelltext (z.B. in EMF Tiger) erreicht. Es existiert kein Werkzeug, das die Beschreibung von Verhalten auf Modellebene direkt

in und mit EMF ermöglicht und gleichzeitig die Quelltextgenerierung unterstützt.

Wünschenswert ist demnach ein Ansatz und ein daraus resultierendes Werkzeug zur direkten Unterstützung und Erweiterung von EMF, ohne „das Rad neu zu erfinden“. Dies wird den folgenden Teilen der Arbeit vorgestellt.

Teil III

Verhaltensmodellierung für EMF - ModGraph

„Ein Bild sagt mehr als 1000 Worte."
(Fred R. Barnard)

Oder sagen 1000 Worte...

...doch mehr als ein Bild?

Inhaltsverzeichnis - Teil III

5 ModGraphs konzeptioneller Ansatz **71**
5.1 ModGraphs Architektur 71
5.2 Entwurf der Graphtransformationsregeln 74
5.2.1 Das Ecore-Modell als Graph 74
5.2.2 Zusammenspiel von Ecore-Modell und Graphtransformationsregeln 79
5.2.3 Aufbau einer Graphtransformationsregel 81
5.3 Ausführung des Modells 85
5.4 Informelle Spracheinführung am Beispiel 85
5.4.1 Das Bugtracker-Ecore-Modell 86
5.4.2 Graphtransformationsregeln für den Bugtracker 88
5.4.3 Prozedurale Elemente im Bugtracker 94
5.4.4 Das ausführbare Bugtracker-Modell 98
5.4.5 Zusammenfassung der Konzepte 98

6 Design der Sprache für Graphtransformationsregeln **101**
6.1 Syntax der Sprache für Graphtransformationsregeln 101
6.1.1 Das Metamodell der Sprache 101
6.1.2 Die konkrete Syntax 112
6.2 Dynamische Semantik der Sprache für Graphtransformationsregeln 115
6.2.1 Ausführungslogik einer Graphtransformationsregel 116
6.2.2 Mustersuche 118
6.2.3 Anwendung der modellierten Änderungen 123

7 Integration in das EMF-Rahmenwerk **127**
7.1 Konzeptuelle Integration: Wege zur Nutzung von ModGraph 127
7.2 Generatoren 133
7.2.1 Generierung von Java-Quelltext mit ModGraph 133
7.2.2 Einbinden der ModGraph-Regeln in ein Xcore-Modell 148
7.2.3 Vergleich der Modelle und Quelltexte 153
7.3 Eclipse-Integration 156
7.3.1 Die ModGraph-Perspektive 156
7.3.2 Grafischer Editor und Validierung 158
7.3.3 Dashboard 159
7.3.4 Unterstützung durch Cheat Sheets 160

5 ModGraphs konzeptioneller Ansatz

Dieser Teil der Arbeit präsentiert einen Ansatz und ein daraus resultierendes Werkzeug zur direkten Unterstützung und Erweiterung von EMF, ohne „das Rad neu zu erfinden“. Die Neuentwicklung soll den aus den vorangegangenen Kapiteln geforderten, auf EMF abgestimmten, leichtgewichtigen Ansatz umsetzen, der sich auf den Mehrwert der erreicht werden kann konzentriert. Dieser Mehrwert muss eine echte Erweiterung von EMF darstellen, welche die Verhaltensmodellierung direkt in EMF anbietet. Diese Kombination von Verhaltens- und Strukturmodellierung soll die Erstellung ausführbarer Modelle ermöglichen. Graphtransformationsregeln sind hierfür grundsätzlich gut geeignet, sollten jedoch in ihrer Anwendung durch Kontrollflüsse gesteuert werden. Dazu ist die Wiederverwendung von bestehenden Werkzeugen und Infrastrukturen innerhalb der Eclipse-Welt, wie Xcore mit Xbase, denkbar.

Bezieht man in diese Anforderungen diejenigen aus Buchmann et al. [20] mit ein, so bestätigt sich die Forderung nach einer Steuerung der Regeln. Die Anforderungen aus [20] wurden anhand einer Analyse von einigen großen Graphtransformations-basierten Modellen verschiedener Anwendungsdomänen gewonnen. Es wird deutlich, dass zur Verhaltensmodellierung regelbasierte und prozedurale Elemente angeboten werden sollten. Der Nutzer kann damit selbst wählen, welche Elemente zur Lösung der vorliegenden Problematik adäquat sind. Zudem wird aus den Analysen deutlich, dass ein Kontrollfluss in einer konzisen, textuellen Notation verfasst werden sollte. Dieser sollte sich außerdem an der strukturierten Programmierung orientieren. Auf diese Weise können Graphtransformationsregeln mit prozeduralen Elementen gesteuert werden. Sie gelten weiterhin als intuitiver und - im Vergleich zu einer textuellen Notation - als einfacher zu verstehen.

Ein weiterer Schluss, der aus Teil II der Arbeit (insbesondere aus Kapitel 3.4) gezogen werden kann, ist der Mangel an Möglichkeiten zur Modellierung im Großen mit EMF. Wann immer ein System basierend auf EMF erstellt wird, muss sofort mit Klassendiagrammen modelliert werden. Die Einbindung eines Werkzeugs zur Architekturmodellierung auf höherer Abstraktionsebene wäre ein weiterer Mehrwert für EMF.

Aus diesen Überlegungen wurde der Ansatz zur **Mod**elltransformation mit **Graph**transformationen - kurz ModGraph-Ansatz - und das zugehörige Werkzeug ModGraph entwickelt. Beide sollen in diesem Teil der Arbeit basierend auf eigenen Veröffentlichungen [19, 21, 78, 80, 81, 82], beschrieben werden.

5.1 ModGraphs Architektur

In Kapitel 3.4 wurde gezeigt, welche Möglichkeiten EMF bietet und welchen Limitierungen es unterliegt. Hierbei hat sich gezeigt, dass EMF Klassendiagramme zur strukturel-

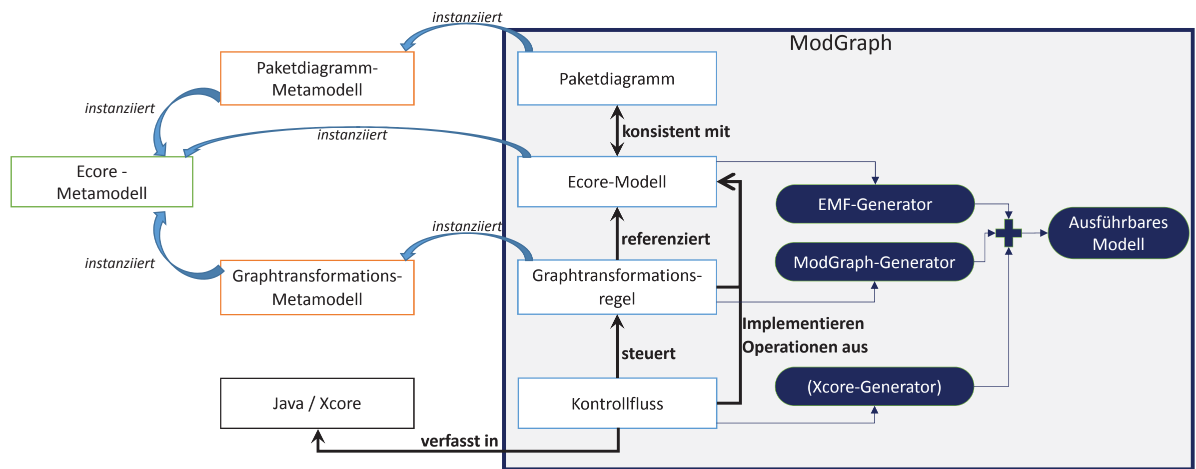


Abbildung 5.1: *Überblick über die Architektur des ModGraph-Ansatzes*

len Modellierung unterstützt und Quelltext aus diesen erzeugen kann. Diese Fähigkeiten werden im Folgenden genutzt und erweitert.

Der hier vorgestellte ModGraph-Ansatz schlägt eine Erweiterung von EMF in zwei Dimensionen vor, um die oben aufgezählten Erweiterungswünsche an EMF zu erfüllen. Einerseits soll das Modellieren im Großen mit Paketdiagrammen unterstützt, andererseits die Modellierung von Verhalten mit Graphtransformationsregeln und Kontrollstrukturen ermöglicht werden. Abbildung 5.1 zeigt die Architektur des ModGraph-Ansatzes und des damit verbundenen Werkzeugs. Diese werden im Folgenden, sofern nicht explizit unterschieden, mit ModGraph bezeichnet.

Die Modellierung im Großen wird durch die Einführung von Paketen und damit Paketdiagrammen, die eine gröbere Sicht auf das System darstellen, erreicht. Die Integration der Paketdiagramme in den Ansatz ist in Abbildung 5.1, oben im Bild, zu sehen. Ein Paketdiagramm beschreibt die Architektur eines Systems, indem Pakete und deren Abhängigkeiten untereinander spezifiziert werden. Diese Pakete gruppieren Klassen aus Ecore-Modellen. Hierfür werden private und öffentliche Pakete sowie der Import von Elementen angeboten.

Auf Werkzeugebene wird der bereits am Lehrstuhl existierende Paketdiagrammeditor (Buchmann at al. [18]) in ModGraph integriert, der zu Beginn des ModGraph-Projektes bereits die Gruppierung von Ecore-Klassen in Pakete ermöglichte. Paketdiagramme sind Instanzen des Paketdiagramm-Metamodells, das wiederum das Ecore-Metamodell instanziiert. Der Paketdiagramm-Editor stellt einige Methoden zur Verfügung, welche die Konsistenz zwischen ihm und den zugehörigen Klassendiagrammen sicherstellen. Dazu zählen vorwärts, rückwärts und inkrementelles Round-Trip-Engineering sowie eine zusätzliche Validierung. Für eine detaillierte Beschreibung des Paketdiagramm-Editors sei auf Buchmann et al. [18] verwiesen.

Da sich die Problematik des Modellierens im Großen durch die Einbindung von bereits existierenden Arbeiten lösen lässt, liegt der Fokus dieser Arbeit auf der Verhaltensmodellierung. Sie soll hauptsächlich durch die Einbettung von Graphtransformationsregeln in

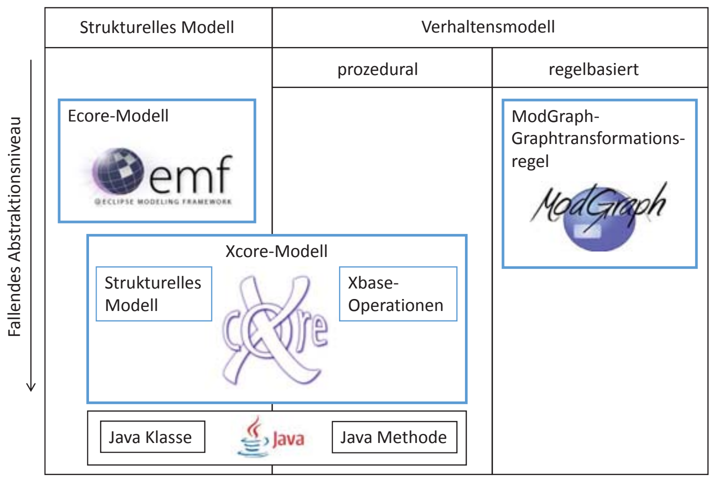


Abbildung 5.2: *Verwendete Modellierungskonzepte und deren Nutzung im ModGraph-Ansatz*

das EMF-Rahmenwerk erreicht werden. Dabei ist es entscheidend sich auf den Mehrwert zu konzentrieren den Graphtransformationsregeln bieten. Dies beinhaltet die Fähigkeit komplexe, strukturelle Änderungen deklarativ und kompakt zu beschreiben (siehe Abschnitt 2.6). Der untere Teil von Abbildung 5.1 zeigt die Architektur. Man beachte, dass es sich im Bezug auf Ecore um einen Hybridansatz handelt: Beim strukturellen Modell wird Ecore unmittelbar als Modellierungssprache genutzt. Bei dem mit Graphtransformationsregeln spezifizierten Verhaltensmodell wird Ecore für die Metamodellierung genutzt.

Da es keine Möglichkeit gibt Kontrollstrukturen direkt innerhalb der Regeln zu spezifizieren, wird hier zunächst Java, später das - parallel zu ModGraph entwickelte - Xcore verwendet. Die Verhaltensmodellierung ist damit in einen prozeduralen und einen regelbasierten Anteil aufgeteilt. Somit verfolgt ModGraph einen hybriden Ansatz zur Verhaltensmodellierung. Eine Übersicht hierzu ist in Abbildung 5.2 dargestellt. Prozedurales Verhalten wird in Xcore, genauer mit den darin enthaltenen Xbase-Operationen, oder in Java spezifiziert, während regelbasiertes Verhalten mit Graphtransformationsregeln spezifiziert wird. Zudem besteht bei der Verwendung von Xcore die Möglichkeit, das statische Modell in Xcore, statt durch ein Ecore-Klassendiagramm zu spezifizieren.

Einer der Vorteile dieses hybriden Ansatzes ist die Steuerbarkeit der Regeln durch einen konzisen und textuellen Kontrollfluss, der sich an der strukturierten Programmierung orientiert. Man spricht in diesem Zusammenhang von einem programmierten Graphersetzungssystem. Des Weiteren wird Xcore aufgrund der in [20] gefundenen Fakten, dass grafisch dargestellte Kontrollflüsse keinen Abstraktionsgewinn bieten, als Kontrollflusssprache ausgewählt.

Die strikte Trennung von prozeduralen und regelbasierten Anteilen trägt zudem zur Übersichtlichkeit des Modells bei. Jede Regel ist einzeln definiert und trotzdem nahtlos in das Modell integriert. Diese Integration wird durch einen Aufrufmechanismus zwischen der Regel und ihrem Kontrollfluss realisiert. Jede Regel implementiert eine Operation. Der Kontrollfluss implementiert eine andere Operation und ruft die durch

die Regeln implementierten Operationen auf. Zudem können aus den Regeln Operationen aufgerufen werden, die einfache Änderungsoperationen beschreiben. Dabei handelt es sich um Operationen wie Datumsabfragen oder Berechnungen, die so einfach sind, dass ihre Implementierung mit einer Regel dem sprichwörtlichen „mit Kanonenkugeln auf Spatzen schießen“ entsprechen würde. Es entsteht eine Symbiose von regelbasierter und prozeduraler Modellierung.

Ein weiterer Vorteil des hybriden Ansatzes ist in der strikten Trennung von prozeduralen und regelbasierten Komponenten zu finden. Die Regeln können plattformunabhängig - im Sinne von unabhängig von einer Programmiersprache - modelliert werden. Sie enthalten dadurch keine Quelltextfragmente, wie sie zum Beispiel in Fujaba [83] gebräuchlich sind. Gleichzeitig führt das Fehlen von Quelltextfragmenten zu einer Erhöhung des Abstraktionsniveaus der Regel. Damit ist auch die generierte Zielsprache zunächst frei wählbar. Eine gewöhnliche Programmiersprache ist dabei genauso möglich wie eine Modellierungssprache. ModGraphs hybrider Ansatz favorisiert, wie in den Abbildungen 5.1 und 5.2 gezeigt, Java als gewöhnliche Programmiersprache und Xcore als textuelle Modellierungssprache. Er erfordert eine Reihe von Generatoren, die in Abbildung 5.1 dunkelblau dargestellt sind. Diese erzeugen das ausführbare Modell. Dazu wird entweder ausführbarer Java-Quelltext aus den Modellen erzeugt oder das Xcore-Modell erweitert, indem die Regel in eine Xbase-Operation transformiert wird. Beide Varianten liefern ein ausführbares (Gesamt-)Modell, das nicht weiter nachbearbeitet werden muss. Die Interaktion der Modelle und Generatoren wird im Folgenden genauer erläutert. Unter anderem wird geklärt, wie eine Graphtransformationsregel mit dem Ecore-Modell durch Referenzierung zusammenspielt und wie der Kontrollfluss zur Steuerung der Regeln verwendet werden kann.

Bereits jetzt wird jedoch eines klargestellt: Das Ziel von ModGraph besteht darin, Komponenten zu entwickeln, die in eine umfassende Modellierungsumgebung integriert sind. Dies erfolgt durch eine nahtlose Integration von ModGraph in das EMF-Rahmenwerk. Diese spiegelt sich in der Architektur des Gesamtsystems wieder.

5.2 Entwurf der Graphtransformationsregeln

In Teil I, Abschnitt 2.6, wurde das Konzept einer Graphtransformation und deren Anwendung auf einen Graphen zur strukturellen Änderung desselben beschrieben. Die folgenden Ausführungen bauen auf Abschnitt 2.6 auf und beschäftigen sich mit der Integration von Graphtransformationsregeln in EMF und dem Aufbau dieser Regeln.

5.2.1 Das Ecore-Modell als Graph

Graphtransformationsregeln operieren im Allgemeinen auf attribuierten Graphen. Das Datenmodell attribuierter Graphen besteht, wie aus der Definition des Graphen in Abschnitt 2.6.1 ersichtlich, aus Knoten, die Attribute tragen und Kanten. Jede Kante verbindet zwei Knoten. Dabei ist eine Kante im einfachsten Fall als Tripel, bestehend aus

Quellknoten, Typ und Ziel definiert. Die Kanten sind damit gerichtet, bleiben jedoch bidirektional traversierbar.

ModGraph nutzt Ecore zur strukturellen Modellierung und hat sich zum Ziel gesetzt, die Regeln auf Instanzen von Ecore-Modellen anzuwenden. Damit ist das im Weiteren genutzte Datenmodell auf Ecore, d.h. auf ein objektorientiertes Datenmodell, festgelegt. Dies wirkt sich auf die Graphtransformationsregeln aus. Um Graphtransformationsregeln auf Modellinstanzen anwenden zu können, muss zunächst untersucht werden, wie die Instanzen als Graphen aufgefasst werden können und welchen Einschränkungen sie gegenüber dem allgemeinen attribuierten Graphen unterliegen.

Eine Modellinstanz lässt sich als Graph auffassen, indem man die Objekte (Instanzen der Klassen) als Knoten und die Links (Instanzen der Referenzen) als Kanten interpretiert. Objekte und deren Attribute lassen sich eins zu eins auf Knoten und Knotenattribute abbilden. Beziehungen weisen jedoch einige Besonderheiten auf, die in den ModGraph-Regeln berücksichtigt werden müssen.

In Ecore werden den Referenzen Multiplizitäten zugeordnet. Sie stellen Beziehungen zwischen Objekten dar, die u.U. auch Enthaltenseinsbeziehungen repräsentieren. Diese Beziehungen können geordnet sein. Zudem wird zwischen uni- und bidirektionalen Referenzen unterschieden. Daraus ergeben sich verschiedene Konsistenzbedingungen[1] an die Links:

K1 *Referenzielle Integrität*: Quell- und Zielobjekt eines Links müssen existieren.

K2 *Bidirektionalität*: Bei bidirektionalen Referenzen muss zu jedem Vorwärts- ein Rückwärtslink existieren.

K3 *Exklusivität*: Jedem Objekt muss ein eindeutiger Behälter zugeordnet sein.

K4 *Zyklenfreiheit*: Enhaltensseinsbeziehungen dürfen keine Zyklen bilden.

K5 *Transitives Löschen*: Wird ein Objekt gelöscht, müssen alle enthaltenen Objekte ebenfalls gelöscht werden.

K6 *Einhaltung der Multiplizität*: Die Anzahl der Links darf die durch die Multiplizität der Referenz angegebenen Schranken nicht verlassen.

Wie diese Konsistenzbedingungen bei der Anwendung einer Graphtransformationsregel eingehalten werden, wird nun untersucht.

Unidirektionale Referenzen

Unidirektionale Referenzen sind nur in eine Richtung navigierbar. Sie werden verwendet, um Modelle effizienter zu gestalten, indem unnötige Rückrichtungen eingespart werden. Außerdem erfordert die Definition einer unidirektionalen Referenz lediglich eine Erweiterung der Quellklasse, während die Zielklasse unverändert bleibt.

[1] Die relevanten Textstellen zu den Konsistenzbedingungen K3-K5 in den Standards sind: MOF 2.4 Beta, S. 39, UML Infrastructure 2.4, S. 98 bzw. 113.

Das sonst übliche Datenmodell des attribuierten Graphen sieht nur bidirektionale (gerichtete) Beziehungen vor. Damit ergeben sich zwei Probleme:

P1 Das Löschen eines Knotens im Graphdatenmodell beinhaltet das Entfernen aller adjazenten Kanten. Für das Löschen eines Objekts müssen demnach auch die einlaufenden, nicht navigierbaren Kanten gelöscht werden.

P2 Bei der Mustersuche, die zur Ausführung der Regel nötig ist, können Knoten und damit die zugehörigen Objekte nicht erreicht werden, wenn sie in der falschen Richtung mit einem bekannten Objekt verbunden sind.

P1 muss gelöst werden, um die referenzielle Integrität (**K1**) des Links zu gewährleisten. Bezüglich des Löschens stellt EMF eine passende Hilfsklasse (`EcoreUtil`) zur Verfügung, die eine Methode zum Löschen eines Objekts und seines Kontexts bereit stellt. Damit werden auch alle verbundenen Links gelöscht.

P2 kann nur gelöst werden, indem jede Regel bereits vor der Mustersuche sicherstellt, dass alle zu suchenden Objekte erreichbar sind. Dies ist in der von ModGraph genutzten navigierenden Mustersuche begründet, die Parameter und das aktuelle Objekt als Ausgangspunkt der Suche nutzt. Dazu wird die Validierung verwendet. Sie stellt sicher, dass eine Kante oder ein Pfad (eine Folge von Kanten) existiert, so dass jedes Objekt von einem bekannten Objekt aus erreichbar ist.

Bidirektionale Referenzen

Bidirektionale Referenzen sind in beide Richtungen navigierbar. Sie entsprechen insofern einer Kante im Graphen. Allerdings sind sie in Ecore streng genommen aus zwei aufeinander verweisenden (und damit zu einem Paar zusammengefassten) unidirektionalen Referenzen aufgebaut. Damit ergibt sich folgendes Problem:

P3 Die beiden einzelnen Links müssen konsistent gehalten werden. Es dürfen keinerlei inkonsistente Operationen auf den Links ausgeführt werden.

P4 Vorwärts- und Rückwärtslink können innerhalb einer Regel inkonsistent spezifiziert werden.

P3 ist durch die Quelltextgenerierung in EMF bereits gelöst. Wird ein Link verändert, so wird der gegenüberliegende ebenso angepasst. Die Konsistenzbedingung der Bidirektionalität (**K2**) ist demnach nicht verletzt. **P4** tritt auf, sobald beide Richtungen der Referenz in der Regel durch Links verschiedener Status eingezeichnet sind. So kann der Vorwärtslink nicht erstellt oder erhalten werden, während der Rückwärtslink gelöscht wird. Eine solche Regel ist nicht anwendbar. Dieses Problem wird durch die Validierung der Regeln überprüft. Dabei dürfen Links grundsätzlich nicht zweimal eingezeichnet werden. Da Vorwärts- und Rückwärtslink sich gegenseitig bedingen, werden sie als Instanzen derselben bidirektionalen Referenz aufgefasst. Sie dürfen daher nicht gleichzeitig zwischen dem selben Objektpaar in einer Regel vorkommen.

Damit ist auch die Mustersuche von **P4** betroffen. Sie fordert die Erreichbarkeit der unbekannten Knoten von bekannten aus, durch Navigation der Links. ModGraph erlaubt die Rückwärtsnavigation eines Links, sofern der (nicht eingezeichnete, jedoch implizit gegebene) entgegengesetzte Link existiert. So spielt auch bei der Mustersuche die Richtung des Links, der die bidirektionale Referenz instanziiert, keine Rolle.

Multiplizitäten der Referenzen

In Graphen dürfen Knoten durch beliebig viele Kanten verbunden werden, sofern Kantentypen nur den Quell- und Zieltyp von Knoten einschränken. Wird die mengentheoretische Definition des Graphen verwendet, so wird dem Wirtsgraphen einfach eine weitere Kante als Tripel aus Quelle, Typ und Ziel hinzugefügt oder eben entfernt.

In Ecore sind den Referenzen jedoch einzuhaltende Multiplizitäten (**K6**) zugeordnet, was zu folgenden Problemen führt:

P5 Die maximale Multiplizität einer Referenz darf durch die Anzahl der Links nicht überschritten werden.

P6 Die minimale Multiplizität einer Referenz darf nicht unterschritten werden.

Ecore unterscheidet bei der Codegenerierung bezüglich der maximalen Multiplizität in **P5** lediglich zwischen einwertigen und mengenwertigen Referenzen. Mengenwertige Referenzen dürfen beliebig oft auftreten. Daher können sie beliebig oft gesetzt werden und verhalten sich wie die oben genannten Kanten im Graphen. Einwertige Referenzen jedoch werden durch Zuweisungen manipuliert, die einen Seiteneffekt aufweisen. Bei jeder Zuweisung geht der vorher zugewiesene Link verloren. Dies wird in ModGraph toleriert und unterstützt: Ein zusätzlicher explizit als zu löschend eingezeichneter Link ist nicht erforderlich.

Allerdings dürfen eine einwertige Referenz instanziierende Links nicht mehrmals aus einem Objekt der Regel auslaufen (auch nicht, wenn sie nur implizit durch ihren jeweiligen entgegengesetzten Link eingezeichnet sind), es sei denn, die Ersetzung wird explizit angegeben. Das mehrmalige Auftreten einwertiger Links wird durch die Validierung erkannt.

EMF garantiert nicht, dass die minimalen Multiplizitäten nicht unterschritten werden. Deshalb wird in ModGraph ebenfalls nicht versucht, **P6** zu garantieren.

An dieser Stelle sei angemerkt, dass EMF bei der Modellierung mit Ecore beliebige Zahlenwerte für die oberen und unteren Schranken zulässt. Dies ist ein seltener Sonderfall. ModGraph prüft hier lediglich, dass die Anzahl der Links einer Regel die obere Schranke nicht überschreitet und überlässt die weitere Verwaltung EMF.

Geordnete Referenzen

In EMF sind derzeit alle mengenwertigen Referenzen geordnet. Sie werden als Listen implementiert. In vielen Fällen ist die Reihenfolge der Links nicht relevant und es wird das

Standardverhalten von EMF genutzt, neue Links am Ende der Liste einzufügen. Manchmal ist die Reihenfolge jedoch wichtig, woraus sich für die Regeln folgendes Problem ergibt:

P7 Die Position beim Einfügen des Links in eine Liste, die eine geordnete mehrwertige Referenz repräsentiert, ist nicht definiert.

ModGraph bietet zur Lösung von **P7** das Konzept der geordneten Links an. Dabei kann angegeben werden, ob ein Link am Anfang, am Ende, an einer bestimmten Stelle oder vor bzw. nach einem angegebenen Element (das durch seinen Namen vorgegeben wird) eingefügt wird.

Enthaltenseinsbeziehungen

EMF unterstützt Containment-Referenzen, die Enthaltenseinsbeziehungen darstellen. Sie dürfen keinerlei Zyklen enthalten (**K4**). Da sie jedem Objekt höchstens einen Container zuweisen, unterliegen sie ebenfalls der Exklusivität (**K3**). Außerdem definieren sie, welche Objekte transitiv gelöscht werden müssen (**K5**). Dadurch entstehen einige Probleme, die in den Regeln berücksichtigt werden müssen:

P8 Ein Objekt kann potenziell in mehrere Container eingefügt werden.

P9 Durch die Erzeugung von zugehörigen Links können Zyklen in den Enthaltenseinsbeziehungen entstehen.

P10 Ist ein Objekt durch eine Enthaltenseinsbeziehung mit einem zu löschenden anderen Objekt verbunden, muss es ebenso gelöscht werden.

Zur Lösung von **P8** wird die Eindeutigkeit des Containers in ModGraph zunächst analog zur Eindeutigkeit einwertiger Referenzen behandelt. Dies ist eine geeignete Methode, solange auch das strukturelle Modell nur einen möglichen Container erlaubt. Da das Ecore-Modell jedoch die Modellierung mehrerer eingehender Containment-Referenzen in eine Klasse erlaubt, muss die Exklusivität des Containers bei Objekten weitergehend geprüft werden: Es dürfen keinesfalls mehrere Links, die verschiedene - auf Modellebene erlaubte - Containment-Referenzen instanziieren, in ein Objekt eingehen, da ansonsten die Exklusivität (**K3**) verletzt ist. Dies muss zur Laufzeit geprüft werden, da sie nicht zwingend in der Regel vorkommen.

Obwohl der direkte Container des Objekts eindeutig ist, können sich trotzdem Zyklen in der Containment-Hierarchie aufbauen (**P9**). Um diese zu erkennen, sind statische und dynamische Überprüfungen notwendig. Die Regel wird auf Zyklenfreiheit validiert, indem geprüft wird, ob die eingezeichneten Links bereits zu einem Zyklus führen. Ist dies der Fall, ist die Regel nicht ausführbar. Da in der Regel nicht zwingend der gesamte Kontext eines Objekts angegeben ist, muss die Zyklenfreiheit zur Laufzeit nochmals auf der gesamten Instanz geprüft werden. Dies geschieht in einem letzten Schritt der Mustersuche. Damit ist die Zyklenfreiheit eine implizite Anwendbarkeitsbedingung für die Regel.

Bleibt **P10** zu betrachten, zu dessen Lösung das transitive Löschen erfolgreich verwirklicht werden muss. Sind Objekte durch eine Enthaltenseinsbeziehung verbunden und wird ein Container gelöscht, so müssen alle enthaltenen Objekte ebenso gelöscht werden (**K5**). Die Klasse `EcoreUtil` stellt eine Möglichkeit hierfür bereit, die in ModGraph genutzt wird. Damit hat das Löschen eines Objekts einen Seiteneffekt auf andere Objekte. Dies wirkt sich auch auf die Konsistenz der Graphtransformationsregeln aus. Ein zu löschender Knoten wird über einer Container-Klasse typisiert, ein zu erhaltender Knoten über einer Klasse innerhalb des Containers. Sind beide durch einen zu löschenden, die Containment-Referenz instanziiernden Link verbunden, ist die Regel inkonsistent. Dies kann durch die ModGraph Validierung statisch geprüft werden. Es wird gefordert, dass der Container und das beinhaltete Objekt denselben Status aufweisen. Das alleinige Löschen des Containment-Links löscht die Objekte im Übrigen nicht,damit diese dann einem anderen Container zugeordnet werden können.

Die Regel wird ebenso inkonsistent, wenn ein in der Containment-Hierarchie über einem anderen stehendes Objekt gelöscht wird, das Objekt innerhalb des Containers jedoch erhalten werden soll und in der Regel zwischen beiden Objekten keine Verbindung besteht. Dies kann statisch nicht geprüft werden. Die Prüfung findet also nach der Mustersuche statt, indem geprüft wird, ob ein Pfad aus Containment-Referenzen zwischen dem zu löschenden und dem zu erhaltenden Objekt existiert. Ist dies der Fall, muss die Regelausführung scheitern. Man kann dementsprechend auch diese Konsistenzbedingung als implizite Anwendbarkeitsbedingung der Regel auffassen.

Fazit

Nach eingehender Betrachtung der Unterschiede wird klar, dass die Referenzen eines Modells eine komplexere Betrachtung erfordern als Kanten in einem Graphen. Die Einhaltung der Konsistenzbedingungen führt gerade bei großen Modellen zu einer Erhöhung der Laufzeit. Dies begründet sich einerseits im Löschen, andererseits in der Prüfung auf nicht erlaubte Zyklen bei Enthaltenseinsbeziehungen. Wird ein Objekt gelöscht, müssen die enthaltenen Objekte gelöscht werden. Ebenso müssen alle einlaufenden Kanten entfernt werden. Dazu ist es notwendig, das gesamte Modell zu untersuchen, was letztendlich bei großen Modellen viel Zeit in Anspruch nimmt.

Um hohe Laufzeiten zu vermeiden, wäre ein konfigurierbarer ModGraph-Generator denkbar, der es dem Nutzer überlässt, welche der Konsistenzbedingungen eingehalten werden. Damit könnte der Nutzer die Laufzeit direkt beeinflussen, indem er die Anzahl der während der Generierung der Regeln zu berücksichtigenden Konsistenzbedingungen selbst wählt.

5.2.2 Zusammenspiel von Ecore-Modell und Graphtransformationsregeln

Um Graphtransformationsregeln in EMF verwenden zu können, müssen Modell-Instanzen (wie eben gezeigt) als Graphen aufgefasst werden, die über dem Modell typisiert sind. Betrachtet man die Metaebenen der OMG für EMF-Modelle (siehe Abschnitt 3.4,

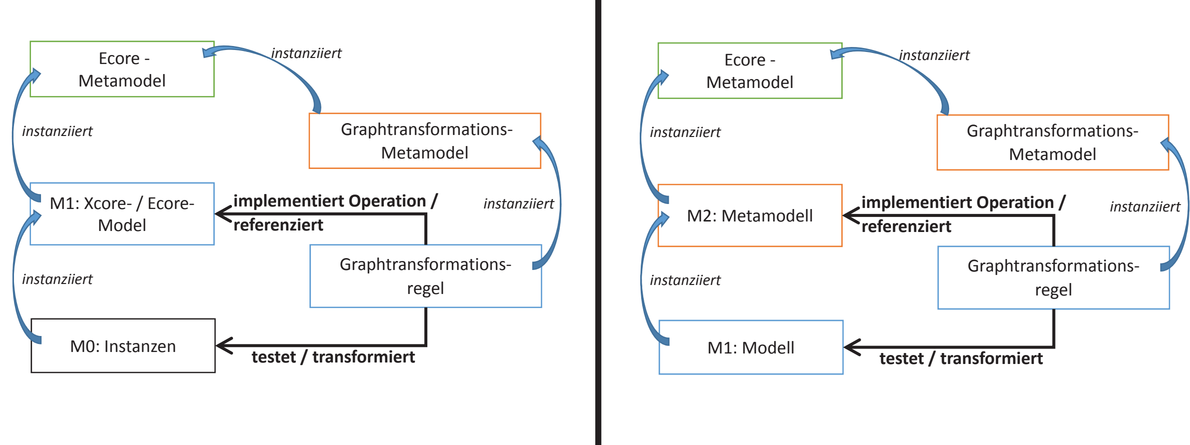


Abbildung 5.3: *M0-Anwendung (links) und M1-Anwendung (rechts) von Graphtransformationsregeln mit EMF*

Abbildung 3.2) und berücksichtigt, dass auch das Ecore-Metamodell ein Ecore-Modell ist, wird die Regelanwendung auf zweierlei Ebenen ermöglicht. Sie sind im Folgenden nach der Metaebene benannt, auf der die Regeln Anwendung finden.

Die M0-Anwendung einer Graphtransformationsregel

Die M0-Anwendung einer Graphtransformationsregel führt eine endogene, überschreibende Transformation von Instanzen (Metaebene M0), die über einem Modell (Metaebene M1) typisiert sind, durch. Sie ist in Abbildung 5.3 auf der linken Seite dargestellt.

Dabei wird ein über dem Ecore-Metamodell typisiertes Xcore- oder Ecore-Modell erstellt. Dieses Modell dient als Typgraph. Es gibt die Typen der Elemente der Graphtransformationsregel vor und repräsentiert die abstrakte Syntax der Modellinstanz.

Die Transformationen stellen Änderungen an der Instanz dar. Diese werden in EMF durch Aufrufe von - im zugehörigen Xcore- oder Ecore-Modell deklarierten - Operationen ermöglicht. Daher wird in ModGraph eine Graphtransformationsregel als Implementierung genau einer solchen Operation einer Klasse des Ecore-Modells betrachtet.

Das Zusammenspiel der Graphtransformationsregeln und des zugehörigen Ecore-Modells ist ebenfalls in Abbildung 5.3 auf der linken Seite dargestellt. Jede Regel referenziert die Operation im Modell, die sie implementiert. Zudem sind die Elemente der Regel über den Elementen des Ecore-Modells typisiert, was zu einer Referenzierung dieser Elemente führt.

Anwendung findet eine Regel auf einer Instanz. Diese kann auf eine bestimmte Bedingung hin untersucht oder transformiert werden. Eine Transformation besteht hierbei aus strukturellen Änderungen, wie der Erzeugung oder Löschung von Objekten oder Links, Attributzuweisungen oder weiteren Operationsaufrufen. Dabei ist zu beachten, dass es sich um geschachtelte Operationsaufrufe handelt, da die Regel selbst für den Rumpf einer Operation steht.

Eine Graphtransformationsregel ist zudem immer eine Instanz des Graphtransformations-Metamodells, das in Abschnitt 6.1.1 genauer erläutert wird und seinerseits eine Instanz des Ecore-Metamodells darstellt. Die hier betrachteten Graphtransformationsregeln sind daher *für* und *mit* EMF erstellt.

Ein Beispiel für diese Art der Anwendung wird im nachfolgenden Kapitel 5.4 gegeben.

Die M1-Anwendung einer Graphtransformationsregel

Die M1-Anwendung einer Regel ist in Abbildung 5.3 auf der rechten Seite dargestellt. Basierend auf dem Ecore-Metamodell können weitere Metamodelle erstellt werden. Ein Beispiel hierfür ist das Graphtransformations-Metamodell in ModGraph.

Die auf diesen Metamodellen aufbauenden Modelle können ebenso mit Regeln transformiert werden. Dabei wird Ecores reflektierender Ansatz genutzt: Das Modell wird als Instanz des Metamodells aufgefasst. Damit kann eine Instanz (die selbst auf Metaebene M1 einzuordnen ist) analog zur M0-Anwendung transformiert werden, indem die Regeln Operationen des Metamodells implementieren. Weiterhin kann dadurch auch das Ecore-Metamodell selbst genutzt werden, da jedes Ecore-Modell (Metaebene M1) als Instanz des Ecore-Metamodells aufgefasst werden kann. Damit ist es möglich, Ecore-Modelle mit Regeln zu transformieren. Man kann demnach argumentieren, dass es sich hierbei um einen Spezialfall der M0-Anwendung handelt.

Ein interessanter Fall tritt auf, wenn existierende Metamodelle, wie das Ecore-Metamodell, genutzt werden sollen. Diese bieten nicht zwingend Operationen an, die durch Regeln implementiert werden können. Um dennoch Regeln spezifizieren zu können, wird ein solches Modell in ein Wrapper-Modell verpackt, das es gänzlich importiert und Klassen hinzufügt, die die Operationen zur Verfügung stellen. Diese Vorgehensweise wird auch in Kapitel 8 verfolgt.

5.2.3 Aufbau einer Graphtransformationsregel

Jede Regel besteht aus Vorbedingungen, Nachbedingungen, einem Graphmuster und negativen Anwendbarkeitsbedingungen. Die Bedingungen bilden das Kontraktmodell der Regel. Ihr schematischer Aufbau ist in Abbildung 5.4 gezeigt. Die Graphtransformationsregeln werden grafisch dargestellt. Die Erstellung des dazu nötigen Editors für ModGraph wird in Abschnitt 7.3 genauer betrachtet.

Das Graphmuster bildet die Kernkomponente der Regel. Es besteht aus einem statischen und einem dynamischen Teil. Das Graphmuster entspricht einer zusammengefügten Ansicht von linker und rechter Regelseite einer Graphtransformation, wie sie in Abschnitt 2.6 beschrieben wurde. Darin wird das zu suchende Objekt-Muster in der Instanz (statischer Teil) und alle an diesem vorzunehmenden Änderungen (dynamischer Teil) spezifiziert. Durch die zusammengeführte Modellierung von linker und rechter Regelseite ist es zudem möglich, Graphtests zu spezifizieren, indem keine Änderungen angegeben werden. Hier wird geprüft, ob ein Muster in einer Instanz vorliegt. Analog können Graphanfragen erstellt werden. Sie lesen Informationen über Objekte, Attribute und Links aus und geben das Ergebnis zurück. Außerdem ähnelt ein Graphmuster

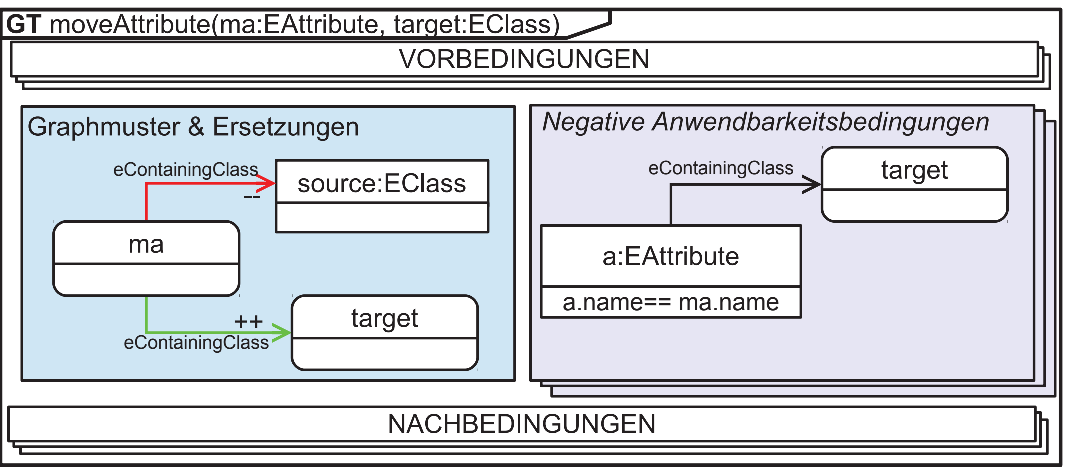


Abbildung 5.4: *Schematischer Aufbau einer Graphtransformationsregel in ModGraph am Beispiel moveAttribute(EAttribute ma, EClass target)*

einem UML-Kommunikationsdiagramm (siehe [42]), welches auch aus einem statischen Teil - dem zugrundeliegenden Objektdiagramm - und einem dynamischen Teil - den Änderungsoperationen - besteht.

Die in Abbildung 5.4 beispielhaft abgebildete Regel zeigt die Implementierung der oft genutzten - dem Leser höchstwahrscheinlich bekannten - Refactoring-Operation moveAttribute(EAttribute ma, EClass target) auf einem Modell. Diese M1-Anwendung der Regel zeigt die Verschiebung eines Attributs ma von einer Klasse source in eine andere Klasse target. Die Transformation wurde über dem Ecore-Metamodell (siehe Kapitel 3.4) spezifiziert. Da dieses keine eigenen Operationen modelliert, wird ein Wrapper-Modell genutzt, welches in Kapitel 8 dargestellt wird und die Refactoring-Operationen bereitstellt.

Vorbedingungen werden oberhalb, Nachbedingungen unterhalb der Kernkomponente der Regel angegeben. Sie sind konjunktiv verknüpft und werden in OCL oder Xcore spezifiziert. Die Einbindung von OCL- und Xcore-Ausdrücken erhöht die Ausdrucksfähigkeit der Regel gegenüber einer rein graphbasierten Darstellung enorm. Negative Anwendbarkeitsbedingungen (NACs) werden analog zum Graphmuster grafisch dargestellt. Sie beinhalten verbotene Objekt-Muster. Im Beispiel aus Abbildung 5.4 darf die Klasse target kein Attribut enthalten, das den Namen des zu verschiebenden Attributs trägt. Treten mehrere NACs auf, so werden diese ebenfalls konjunktiv verknüpft. Dies bedeutet, dass keine der NACs zutreffen darf.

Eine Alternative zu den NACs, um Verbote durch Graphen darzustellen, sind negative Objekte und Links. Sie werden hier, aufgrund ihrer geringeren Ausdrucksstärke, nicht genutzt. Die größere Ausdrucksmächtigkeit der NACs führt jedoch zu Redundanzen. Da zwei Graphen modelliert werden, können Knoten doppelt auftreten. Dies wird einerseits durch die Wiederverwendbarkeit der gebundenen Knoten, andererseits durch die gebundenen Knoten der NAC begünstigt, die speziell eingeführt werden, um ungebundene Knoten im Graphmuster zu referenzieren. Hierbei wird genutzt, dass das Graphmuster immer vor der NAC auf die Instanz abgebildet wird.

An dieser Stelle sei bemerkt, dass Knoten im Allgemeinen letztendlich auf das selbe Objekt abgebildet werden dürfen. ModGraphs Abbildung der Graphen auf die Instanz ist also homomorph. Die Objekt-Muster im Graphmuster und in den NACs werden bei der in Abschnitt 6.2.2 erklärten Mustersuche auf die Modellinstanz abgebildet.

Knoten, die Objekte repräsentieren, sind gebunden oder ungebunden, ein- oder mehrwertig. Mehrwertige Knoten bieten eine kompakte Schreibweise zur Behandlung von Mengen typgleicher Objekte an (und sparen damit Schleifen über Regeln ein). Zudem sind optionale und obligatorische Knoten erlaubt.

Gebundene Knoten stehen stellvertretend für alle vor Ausführung der Operation bereits bekannten Objekte. Da ModGraphs Regeln parametrisiert sind, handelt es sich neben dem aktuellen Objekt, das in Java-Analogie mit `this` gekennzeichnet wird und auf dem die Methode aufgerufen wird, um die über einer Klasse typisierten, ein- und mehrwertigen Parameter einer Operation. Da ihr Typ bereits festgelegt ist, wird er in der Regel nicht näher spezifiziert, es sei denn, es findet eine Typumwandlung des Parameters statt. Parametertypen können alternativ zur Klasse auch Datentypen sein. In diesem Fall werden Parameter innerhalb der Regel in Bedingungen an die (Knoten der) Regel oder in Wertzuweisungen genutzt. In Abbildung 5.4 sind die Parameter der Operation die Zielklasse `target` und das zu verschiebende Attribut `ma`.

Ein- oder mehrwertige ungebundene Knoten stehen stellvertretend für Instanzen von Klassen. Sie müssen während der Mustersuche in der Instanz gefunden werden und können als Rückgabeparameter der Regel und damit der Operation dienen, indem sie mit << out >> gekennzeichnet werden. Alternativ könnten auch alle (ungebundenen) Objekte der Regel zurückgegeben werden. Dies erscheint jedoch hier nicht sinnvoll, da Operationen einen Rückgabewert haben und die Regeln diese implementieren.

Knoten, ausgenommen der das aktuelle Objekt repräsentierende, sind optional oder obligatorisch. Während obligatorische Knoten auf die Instanz abgebildet werden müssen, können optionale bei Anwendung der Regel fehlen. Ein optionaler Knoten wird mit << optional >> gekennzeichnet. Da EMF letztlich Java-Code generiert, der keine optionalen Parameter unterstützt, werden diese als möglicherweise `null`-wertige Parameter umgesetzt.

Die Knoten können durch zwei Arten von Kanten, Links und Pfade, verbunden werden. Links stehen für Instanzen von Referenzen. Ein Link, der zwei einfache Objekte verbindet, ist ebenfalls einfach. Einer, der einen mehrwertigen Knoten als Quelle oder Ziel hat, wird in der Instanz unter Umständen auf mehrere Instanzen der zugehörigen Referenz abgebildet. Beginnt oder endet ein Link an einem optionalen Knoten, ist er ebenso optional. Die explizite Kennzeichnung eines Links als optional ist nicht vorgesehen. Um geordnete Referenzen korrekt zu instanziieren, werden geordnete Links angeboten. Diese erlauben das Einfügen des Links an einer vorgegebenen Position.

Pfade sind abgeleitete Referenzen. Sie haben keinen Status und sind mit einem Pfadausdruck markiert. Dieser erhöht ebenfalls die Ausdrucksfähigkeit der Regel. Ein Pfadausdruck wird auf dem Quellobjekt des Pfads ausgewertet und gibt genau die Objekte zurück, welche die im Ausdruck angegebene Bedingung erfüllen. Zur Angabe eines Pfades zwischen zwei Knoten ist die Existenz einer Referenz zwischen den zugehörigen Klassen nicht notwendig. Damit erhöhen sie das Abstraktionsniveau der Regel.

Außerdem dürfen keine Links - und auch keine Pfade - zwischen mehrwertigen Knoten gezogen werden, da es schwierig ist, in diesem Fall eine eindeutige und sinnvolle Semantik zu definieren.

Die Knoten können Bedingungen und Änderungen enthalten. Bedingungen sind entweder OCL-Ausdrücke oder werden direkt an die Werte der Attribute des Knotens gestellt. Änderungen sind in Form von Wertzuweisungen an Attribute oder Operationsaufrufen zulässig.

Um Änderungen vornehmen zu können, tragen Objekte und Links des Graphmusters einen Status. Dieser ist entweder erzeugen (++ Markierung), erhalten (keine Markierung) oder löschen (−− Markierung). Parameter dürfen dabei nicht erzeugt werden, da sie bereits bei Aufruf der Regel bekannt sind. Das aktuelle Objekt ist per Definition zu erhalten. Zu erzeugende Knoten sind immer einwertig, obligatorisch und werden über einer nicht-abstrakten Klasse typisiert. Würde man die Erzeugung optionaler oder mehrwertiger Knoten zulassen, müsste festgelegt werden, wie viele Objekte in Abhängigkeit von welchen Bedingungen erzeugt werden sollen.

In Abbildung 5.4 sind alle Knoten zu erhalten und tragen daher keine Markierung. Die zwei Links, welche die Referenz eContainingClass des Ecore-Metamodells instanziieren, sind jedoch statusbehaftet. Der Link zwischen ma und source wird gelöscht, derjenige zwischen ma und target wird erzeugt. Der Link und die Objekte in der NAC in Abbildung 5.4 bleiben erhalten, da NACs keine Veränderungen, sondern Verbote spezifizieren.

Der Ablauf einer Transformation wird als eine atomare Aktion aufgefasst, die aus folgenden Schritten aufgebaut ist: Vorbedingungen werden geprüft, bevor das Graphmuster betrachtet wird. Die negativen Anwendbarkeitsbedingungen werden während der Mustersuche nach einem erfolgreichen Fund des Graphmusters in der Instanz geprüft. Ist keine NAC erfüllt, werden die im Graphmuster definierten Änderungen ausgeführt. Die Nachbedingungen werden erst nach erfolgreicher Anwendung der Änderungen überprüft.

Dabei kann die Regel an verschiedenen Stellen scheitern. Sind die Vorbedingungen nicht erfüllt, ist die durch das Graphmuster vorgegebene Objektstruktur nicht auffindbar oder das innerhalb einer NAC spezifizierte Muster auffindbar, ist die Regel nicht ausführbar. Die Instanz bleibt damit unverändert. Semantisch gesehen, gibt es im Übrigen einen Unterschied zwischen einer Vorbedingung und dem durch das Graphmuster gegebene, zur Ausführung der Regel benötigte Objektmuster. Zwar stellen beide positive Anwendbarkeitsbedingungen an die Regel, allerdings sind die Vorbedingungen auf die bei Aufruf der Regel bekannten Objekte beschränkt. Im Graphmuster können jedoch beliebige Muster, die während der Mustersuche geprüft werden, als Bedingung an die Regel gestellt werden.

Scheitert eine Nachbedingung, ist die Instanz inkonsistent. Da die Regeln nicht invertierbar sind und an dieser Stelle kein Backtracking mit ModGraph möglich ist, muss die Konsistenz vom Nutzer wiederhergestellt werden. Dazu bietet sich die Definition von Reparaturregeln an.

5.3 Ausführung des Modells

Zur Ausführung eines Modells ist seine Validität zwingend erforderlich. EMF bietet bereits eine Validierung für Ecore-Klassendiagramme an. Die Validierung der zugehörigen Graphtransformationsregeln übernimmt der ModGraph-Validator. Dieser ist zum einen Teil des Editors und verhindert bereits bei der Erstellung der Regeln durch eine Live-Validierung grobe syntaktische Fehler, wie zum Beispiel das Ziehen eines zu erzeugenden Links zwischen zwei zu löschenden Knoten. Zum anderen validiert er durch eine Batch-Validierung jede Regel gegen das Graphtransformations-Metamodell und gegen das ihr zugeordnete Ecore-Modell. Nach dieser Validierung wird lauffähiger Java-Quelltext aus dem Ecore-Modell und aus den Graphtransformationsregeln erzeugt oder das Modell nach Xcore übersetzt.

Der EMF-Generator unterstützt die Generierung von Java-Quelltext aus Ecore-Modellen. Für benutzerdefinierte Methoden werden lediglich Rahmen erzeugt, die der Programmierer ausfüllen muss.[2] Hier setzt ModGraph an: Für Methoden wird ausführbarer Code erzeugt, der auf Graphtransformationsregeln basiert. Dieser wird direkt in den EMF-generierten Quelltext eingebettet. Der Aufbau einer so erzeugten Methode wird in Abschnitt 7.2.1 genauer erläutert, da hier zunächst das Konzept der Mustersuche erklärt werden muss. Zur Steuerung der Ausführung der Graphtransformationsregeln werden entweder Kontrollstrukturen direkt in Java implementiert oder in Xcore modelliert und mittels des Xcore-Generators in Java-Methoden übersetzt, wie in Abbildung 5.1 dargestellt ist.

Zudem besteht die Möglichkeit, aus den Regeln Xcore-Code zu generieren, der in ein bestehende Xcore-Modell eingefügt wird, die Kontrollflüsse in Xbase zu beschreiben und damit ein ausführbares Xcore-Modell zu erzeugen, das sowohl interpretiert als auch in Java-Quelltext übersetzt werden kann. Auf diese Weise können die Graphtransformationsregeln zu Test- und Debug-Zwecken indirekt interpretiert werden.

Die Codegeneratoren erzeugen insgesamt ausführbaren, EMF-kompatiblen Java-Quelltext. Dieser kann beispielsweise anschließend mit einer grafischen Oberfläche versehen und dadurch für den Endbenutzer nutzbar gemacht werden. Denkbar ist hierbei die Verwendung des Model-View-Controller-Entwurfsmusters zur Kopplung von grafischer Oberfläche und generiertem Quelltext.

5.4 Informelle Spracheinführung am Beispiel

Zur informellen Einführung der Sprache für Graphtransformationsregeln in ModGraph steht eine Reihe kleinerer Beispiele zur Verfügung, die im Rahmen der Werkzeugentwicklung erstellt wurden. Dabei wurde auf die Funktionalität der Beispiele und die sinnvolle Nutzung von Graphtransformationsregeln Wert gelegt. Grafische Oberflächen

[2]EMF bietet zur Ausführung seiner Modelle einen Quelltext-Generator und einen Interpreter an. Da die reine EMF-Interpretation die Ausführung von benutzerdefinierten Operationen nicht vorsieht, wird hier die Integration von ModGraph in den EMF-generierten Quelltext betrachtet.

existieren nicht. Die größeren dieser kleinen Beispiele finden sich auf der ModGraph-Homepage (`http://btn1x4.inf.uni-bayreuth.de/modgraph/homepage`). Sie modellieren einen Online-Kalender (ähnlich zu Mozilla-Lightning[3]), einen Bugtracker (ähnlich zu Bugzilla[4]) zur Verwaltung von Fehlern in einer Software und eine Versionsverwaltung, die das Concurrent Versions System [22] darstellt.

Eines dieser Beispiele, der Bugtracker, wird im Folgenden vorgestellt. Dabei werden die wichtigsten Sprachbestandteile der hier entwickelten Sprache für Graphtransformationsregeln und deren Vorteile erklärt. Das Beispiel zielt darauf ab, möglichst viele Eigenschaften von ModGraph abzudecken.

Die statische Struktur des hier betrachteten Beispiels Bugtracker wird zunächst mit einem Ecore-Modell erstellt, dessen modellierte Operationen mit Graphtransformationsregeln implementiert werden. Die prozeduralen Elemente werden zunächst in Java programmiert, später in Xcore modelliert. Das Beispiel begleitete demnach große Teile des Entwicklungsprozesses von ModGraph.

5.4.1 Das Bugtracker-Ecore-Modell

Den Ausgangspunkt der Modellierung bildet das Ecore-Modell, welches als Klassendiagramm im Abbildung 5.5 dargestellt ist. Es enthält eine zentrale Klasse BugTracker, die als globaler Container des Modells dient. Sie beinhaltet Nutzer (Klasse User), die Mitglieder in mindestens einer Gruppe (Klasse Group) sind. Jeder Nutzer hat diverse Namen und eine Email-Adresse. Gruppen haben einen eindeutigen Namen und eine Beschreibung. Zudem verwaltet der Bugtracker Projekte (Klasse Project). Jedem Projekt ist ebenfalls ein eindeutiger Name und eine Beschreibung zugeordnet.[5] Die Projekte sind für Nutzer bestimmter Gruppen zugänglich (Referenz accesibleProjects). Jedes Projekt hat einen Projektleiter aus der Gruppe der Nutzer (Referenz leader). Projekte bestehen aus Tickets (Klasse Ticket), welche die für das Projekt bekannten Fehler beinhalten. Tickets enthalten eine - durch den Bugtracker verwaltete - eindeutige Nummer und das Datum ihrer Erstellung. Des Weiteren beinhalten sie Revisionen (Klasse TicketRevision), haben einen Berichterstatter (Referenz reporter) und dürfen kommentiert werden (Klasse Comment). Unter den Revisionen ist immer eine aktuellste Revision zu finden (Referenz currentRevision), alle anderen Revisionen befinden sich in der Historie des Tickets (geordnete Referenz history). Jeder Revision sind ein Titel, eine Beschreibung, das Datum ihrer Erzeugung, eine Dringlichkeit und ein aktueller Status zugeordnet. Dringlichkeit und Status werden durch die Aufzählungsdatentypen Severity und Status modelliert. Außerdem ist zu jeder Revision eine beauftragte Person (Referenz assignee) und ihr Ersteller (Referenz creator) bekannt. Jeder Kommentar eines Tickets hat einen Titel, einen Inhalt und ein Erzeugungsdatum, sowie einen Nutzer als Autor (Referenz author).

Bezüglich des Verhaltens des Bugtrackers werden hier Operationen lediglich deklariert.

---

[3] `https://addons.mozilla.org/de/thunderbird/addon/lightning/`

[4] `https://bugzilla.mozilla.org/`

[5] Dieses Modell ist nicht immer optimal modelliert. Beispielsweise können hier Beschreibung und Name in einer gemeinsamen Oberklasse zusammengefasst werden. Der Hintergedanke dabei ist, später Refactoring-Operationen (siehe Kapitel 8) auf diesem Modell demonstrieren zu können.

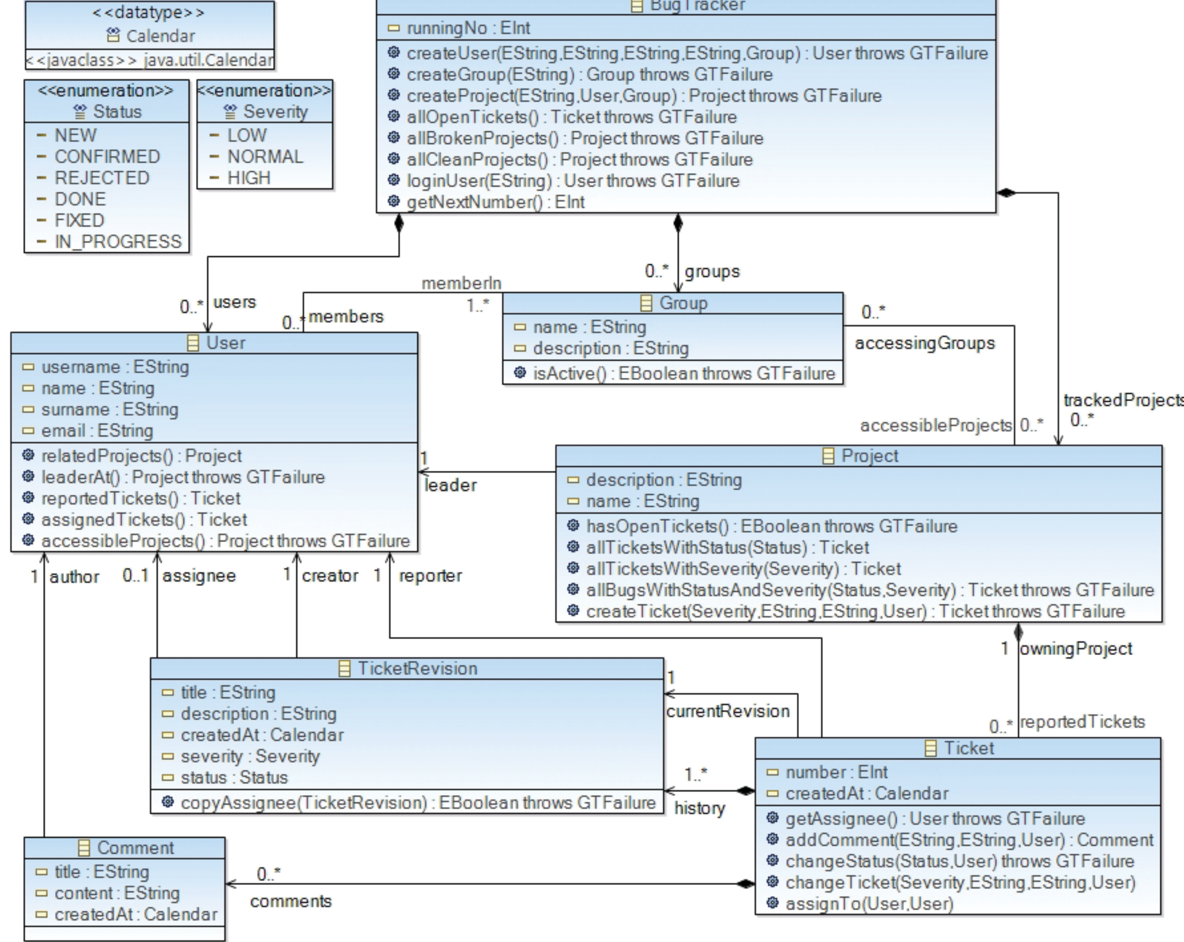


Abbildung 5.5: *Ecore-Modell eines einfachen Bugtrackers*

Eine Implementierung findet nicht statt. Die Klasse BugTracker selbst deklariert Operationen zur Erzeugung von Gruppen, Projekten und Benutzern sowie zur Anmeldung der Nutzer. Zudem werden Operationen zur Abfrage aller offenen Tickets und von Projekten mit bestimmten Eigenschaften angeboten. Der Benutzer kann die ihm zugeordneten, die ihm zugänglichen und die von ihm geleiteten Projekte abfragen, sowie die von ihm erstellten Tickets und diejenigen, für die er zuständig ist. Eine Gruppe gibt Auskunft über ihre Aktivität. Das Projekt verwaltet seine Tickets und die ihm zugeordneten Fehler. Zudem bieten Tickets und deren Revisionen Methoden zur Verwaltung ihrer selbst an. Einige dieser Operationen werden im Folgenden näher betrachtet.

Aus diesem Bugtracker-Ecore-Modell kann Quelltext generiert werden. Dieser bildet die modellierte statische Struktur ab. Außerdem bietet er Akzessoren für die modellierten Attribute und konsistenzerhaltende Operationen für die Beziehungen zwischen den aus den Klassen instanziierten Objekten, d.h. für die Links (Instanzen der Referenzen). Beispielsweise werden Instanzen bidirektionaler Referenzen bei jeder Änderung konsistent für beide Richtungen aktualisiert.

```java
...
public class ProjectImpl extends MinimalEObjectImpl.Container
                                                implements Project {
...
    /**
    * <!-- begin-user-doc --> <!-- end-user-doc -->
    * @generated
    */
    public Ticket createTicket(Severity severity, String title,
                                  String description, User reporter){
        // TODO: implement this method
        // Ensure that you remove @generated or mark it @generated NOT
        throw new UnsupportedOperationException();
    }
    ...
}
```

Auflistung 5.1: *EMFs Lücke: Verhalten muss in reinem EMF in Java beschrieben werden; Ausschnitt aus dem EMF-generierten Quelltext für die modellierte Klasse* Project *des Bugtracker Ecore-Modells*

Die Umsetzung einer Operation in eine Java-Methode ist in Auflistung 5.1 dargestellt. Sie zeigt am Beispiel von createTicket(...) den generierten Quelltext. Dabei wird der Rumpf der Methode mit einer Ausnahme befüllt und der Modellierer in einem Kommentar dazu aufgerufen, die Methode per Hand in Java zu implementieren.

5.4.2 Graphtransformationsregeln für den Bugtracker

EMFs Aufruf, die Methoden per Hand in Java zu implementieren, ist der ursprüngliche Ansatzpunkt dieser Arbeit. Ecore bietet keine Möglichkeit, Verhalten zu modellieren. Hier setzt ModGraph mit den Graphtransformationsregeln an. Diese beschreiben das Verhalten der Operation deklarativ auf Modellebene. Sie kommen zum Einsatz, wenn komplexe strukturelle Änderungen am Modell vorgenommen werden oder es auf eine komplexe Bedingung hin getestet wird. Damit werden die Regeln nur genutzt, wenn sie einen echten Mehrwert bieten. Im Vergleich zu dem geforderten Java-Quelltext ist dies ein echter Gewinn bezüglich des Abstraktionsniveaus. Der Modellierer muss sich dabei nicht mit der Mustersuche innerhalb der Instanz beschäftigen. Er gibt lediglich, wie im Folgenden gezeigt wird, den gewünschten Vor- und Nachzustand der Instanz an.

Zudem sind die Regeln sehr leicht verständlich, da sie größtenteils grafisch dargestellt werden und dabei intuitiv farbkodiert sind. Zu erstellende Elemente sind beispielsweise immer grün und mit ++ markiert, zu löschende immer rot und mit −−. Außerdem weisen die Regeln immer den gleichen Aufbau auf: Das Graphmuster ist zentral links angeordnet, die NACs rechts daneben. Die Vorbedingungen befinden sich oberhalb beider und die Nachbedingungen darunter. Dies erleichtert dem Nutzer die Handhabung.

Implementiert man eine Methode, wie die eben betrachtete createTicket, mit einer Graphtransformationsregel, ist ein Eingriff in das Ecore-Modell nötig, der auch bereits in Abbildung 5.5 zu sehen ist: Jeder Operation, die durch eine Regel implementiert ist, wird eine Ausnahme des Typs GTFailure zugeordnet. Diese wird benötigt, sobald eine

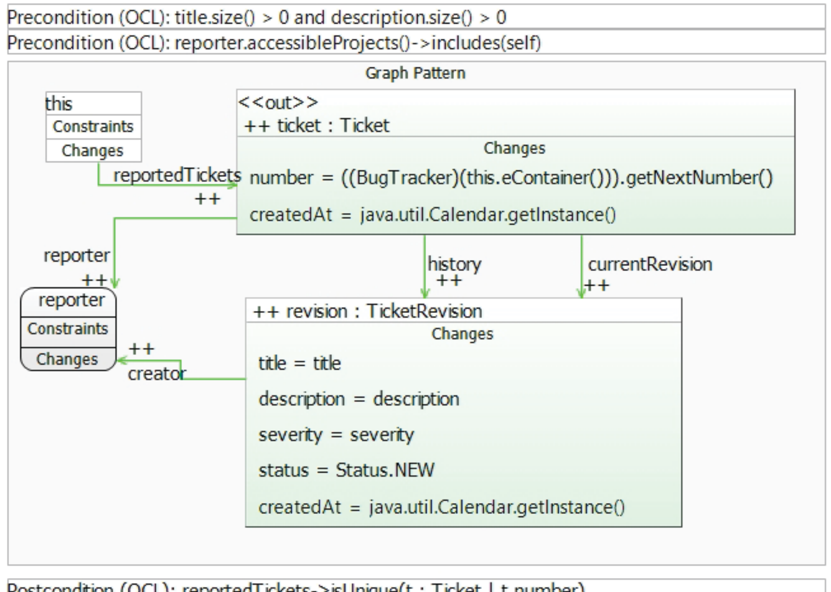


Abbildung 5.6: *ModGraph-Regel zur Erzeugung eines Tickets innerhalb eines Projekts im Bugtracker; Implementierung der Operation createTicket((severity:Severity, title:EString, description:EString, reporter:User):Ticket der Klasse Project*

Bedingung der Regel verletzt ist. Sind Vorbedingungen, Graphmuster oder NAC Ursache der Ausnahme, ist die Regel nicht ausführbar. Ist das Scheitern der Nachbedingung Grund für das Auslösen der Ausnahme, ist die Regel bereits angewendet, jedoch ist die Instanz inkonsistent. Da die Regeln selbst nicht invertierbar sind, muss das Scheitern der Regel durch den Kontrollfluss, aus dem die Regel aufgerufen wurde, behandelt werden. Dieser kann beispielsweise alternative Regeln oder, im Falle des Scheiterns der Nachbedingung, Reparaturregeln definieren, um die Ausnahme zu behandeln.

Welche Bedingungen an eine Regel gestellt werden können und wie diese formuliert werden, wird nun exemplarisch an einigen Regeln gezeigt. Des Weiteren wird die Definition von Strukturänderungen in der Regel erläutert.

Implementierung der Operation createTicket der Klasse Project

Abbildung 5.6 zeigt die Implementierung der Operation createTicket als Graphtransformationsregel. Zunächst stellen OCL-Vorbedingungen sicher, dass die übergebenen Zeichenketten für den Titel und die Beschreibung des Tickets nicht leer sein dürfen und der Berichterstatter Zugriff auf das Projekt hat, welchem das Ticket zugeordnet werden soll. Ist die Vorbedingung verletzt, wird eine Ausnahme ausgelöst. Die Instanz bleibt unverändert. Ist sie nicht verletzt, wird das Graphmuster auf die Instanz abgebildet.

Das Graphmuster beinhaltet zunächst links oben ein aktuelles Objekt, das mit this gekennzeichnet ist und in diesem Fall eine Instanz der Projektklasse darstellt. Außerdem beinhaltet es links unten den nicht-primitiven Parameter reporter, der stellvertretend für eine Instanz der Klasse User steht. Beide Knoten werden nicht verändert, weil ihr Chan-

ges-Bereich, in welchem die Änderungen spezifiziert werden, leer ist. Sie werden auch nicht mit Bedingungen versehen, da ihr Constraints-Bereich, welcher der Spezifikation von Bedingungen an einen Knoten dient, ebenfalls leer ist. Der Parameter reporter ist grau hinterlegt und damit zu erhalten. Zu erhaltende oder zu löschende obligatorische Knoten im Graphmuster müssen zur Ausführung der Regel auf Objekte abgebildet werden. Dies wird vor Anwendung der Änderungen geprüft. Sie stellen letztlich eine weitere Vorbedingung an die Anwendbarkeit der Regel dar, bei deren Scheitern wiederum eine Ausnahme ausgelöst wird.

Die beiden anderen Knoten werden erzeugt. Sie sind daher mit ++ und einem grünen Hintergrund gekennzeichnet. Da keine Bedingungen an einen neu erzeugten Knoten gestellt werden können, fehlt hier die Möglichkeit, Bedingungen zu spezifizieren. Oben rechts in Abbildung 5.6 wird eine Instanz der Klasse Ticket erzeugt. Dieser werden ein Erstellungsdatum und eine durch die Klasse BugTracker verwaltete, eindeutige laufende Nummer zugewiesen. Die Zuweisung erfolgt innerhalb des Knotens, im Changes-Bereich. Zusätzlich ist dieser Knoten mit $<<$ *out* $>>$ gekennzeichnet. Dies bedeutet, dass er als Rückgabewert der Operation dient. Unten rechts wird eine Revision dieses Tickets erzeugt und initialisiert. Hierbei wird dem Attribut title der Wert des gleichnamigen primitiven Parameters der Operation zugewiesen. Analog geschieht dies für die Beschreibung und die Dringlichkeit. Der Status bekommt das Literal Status.NEW (für ein neues Ticket) zugewiesen und das Datum der Revisionserstellung wird vermerkt.

Erzeugte Objekte müssen in ihren Kontext eingebettet werden. Dazu werden alle Beziehungen zwischen den existierenden und neuen Objekten erstellt. Die Links werden jeweils mit dem Namen der Referenz des Modells gekennzeichnet, die sie instanziieren. Sofern sie erzeugt werden, sind sie zusätzlich grün und mit ++ markiert. Dem Projekt wird das Ticket zugeordnet (Link reportedTickets), welchem wiederum ein Benutzer als Berichterstatter (Link reporter) zugeordnet wird. Zudem wird der Ticketrevision der erstellende Benutzer zugewiesen (Link creator). Die Revision wird einerseits als aktuelle Revision (Link currentRevision) des Tickets gekennzeichnet, andererseits direkt in die Historie (Link history) aufgenommen. Obwohl die Referenz geordnet ist, wird ein nicht geordneter Link eingezeichnet, da es sich hier um das erste Objekt im mehrwertigen Link handelt.

Unten in Abbildung 5.6 ist eine Nachbedingung angegeben. Sie muss nach Anwendung der im Graphmuster spezifizierten Änderungen gültig sein. In der betrachteten Regel ist gefordert, dass die laufende Nummer des Tickets eindeutig ist. Ist die Nachbedingung nicht erfüllt, scheitert die Regel. Die Änderungen an der Instanz sind allerdings bereits vorgenommen worden. So liegt es am Modellierer, diese Ausnahme im Kontrollfluss geeignet zu behandeln, indem beispielsweise Reparaturaktionen definiert werden.

Aus dieser Regel kann nun Quelltext generiert werden, der direkt in den EMF-generierten eingefügt wird. Bevor dies jedoch genauer betrachtet wird, werden einige weitere Operationen mit Graphtransformationsregeln implementiert, um die Einführung in die Sprache zu vertiefen.

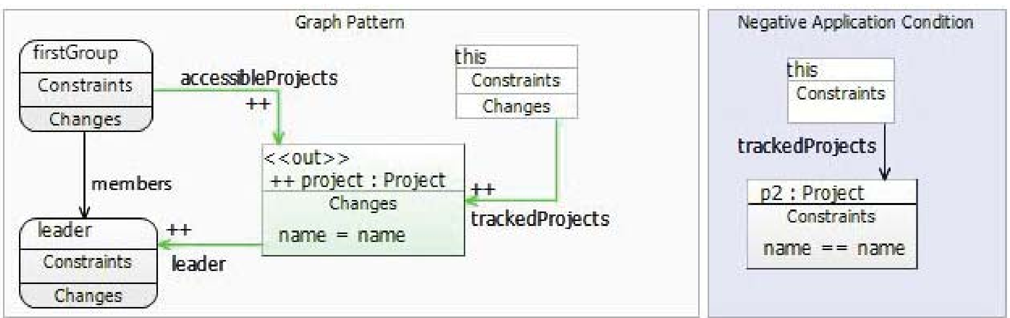


Abbildung 5.7: *ModGraph-Regel zur Erzeugung eines Projekts im Bugtracker; Implementierung der Operation* createProject(name:EString, leader:User, firstGroup:Group):Project *der Klasse* BugTracker

Implementierung der Operation createProject der Klasse BugTracker

Zur korrekten Funktionalität des Bugtrackers ist es notwendig, vor der Erzeugung von Tickets zunächst ein Projekt zu erstellen, dem die Tickets, wie soeben beschrieben, zugeordnet werden müssen.

Die Regel zur Implementierung der Operation createProject(...), die diese Aufgabe übernimmt, ist in Abbildung 5.7 gezeigt. Sie besteht aus einem Graphmuster und einer negativen Anwendbarkeitsbedingung, die ebenfalls als Graph dargestellt wird. Im Graphmuster befinden sich zwei nicht-primitive Parameter der Methode, firstGroup oben links und leader unten links. Der Link members zwischen diesen beiden ist schwarz gefärbt und damit zu erhalten. Er stellt eine positive Anwendbarkeitsbedingung an die Regel und könnte im Übrigen auch als Vorbedingung textuell spezifiziert werden. Hier zeigt sich die Äquivalenz von Vorbedingungen und vorgegebenem Graphmuster deutlich. Der Benutzer, der durch den Parameter leader repräsentiert wird, muss Mitglied der Gruppe, die durch firstGroup dargestellt wird, sein. Ist dies der Fall, so wird ein neues Projekt mit dem der Operation übergebenen Namen als Instanz der Klasse Project erzeugt. Der hierzu genutzte, ungebundene und zu erstellende Knoten wird zusätzlich als Rückgabewert der Methode durch $<< out >>$ markiert.

Das neue Projekt wird in seinen Kontext eingebettet, indem die Links zur Gruppe, dem Projektleiter und dem Bugtracker selbst - wie im vorangegangenen Abschnitt beschrieben - erzeugt werden. Allerdings werden die im Graphmuster spezifizierten Änderungen nur ausgeführt, wenn der Bugtracker kein Projekt enthält, das bereits den übergebenen Namen trägt. Dieser Fakt kommt in einer negativen Anwendbarkeitsbedingung (NAC) zum Ausdruck. Sie beschreibt Sachverhalte, die zur Regelanwendung keinesfalls gegeben sein dürfen. Jede in einer Regel vorkommende NAC stellt ein Verbot dar. Ihre Knoten sind immer einwertig und enthalten nur Bedingungen. In Abbildung 5.7 wird gefordert, dass das aktuelle Objekt this - als Instanz der Klasse BugTracker - kein Projekt p2 enthalten darf, das bereits denselben Namen trägt, welcher dem neuen Projekt zugewiesen werden soll.

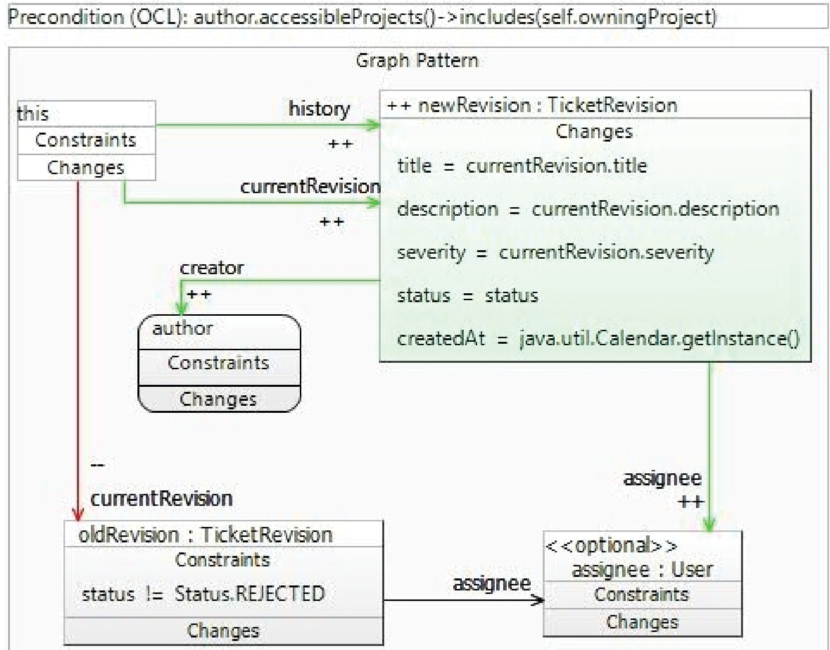


Abbildung 5.8: *ModGraph-Regel zur Änderung des Status eines Tickets im Bugtracker; Implementierung der Operation changeStatus(status:Status, author:User) der Klasse Ticket*

Implementierung der Operation changeStatus der Klasse Ticket

Betrachtet man die Evolution eines erzeugten Tickets im Bugtracker, so kommt es vor, dass der Status eines bestehenden Tickets geändert werden muss, z.B. weil das beschriebene Problem gelöst ist.

Die Regel zur Implementierung der zugehörigen Operation changeStatus ist in Abbildung 5.8 gezeigt. In ihr werden sowohl OCL als auch programmiersprachliche Konstrukte (z.B. !=) verwendet.

Zunächst muss die aktuelle (letzte) Revision des Tickets ermittelt werden. Diese ist durch einen ungebundenen Knoten mit der Bezeichnung oldRevision im Graphmuster dar-

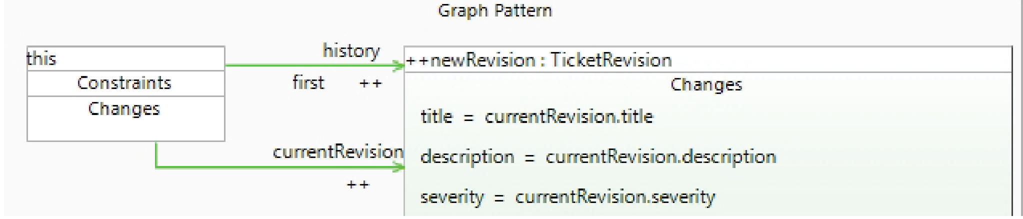


Abbildung 5.9: *Ausschnitt aus der alternativen Modellierung der ModGraph-Regel zur Implementierung der Operation changeStatus(status:Status, author:User) der Klasse Ticket*

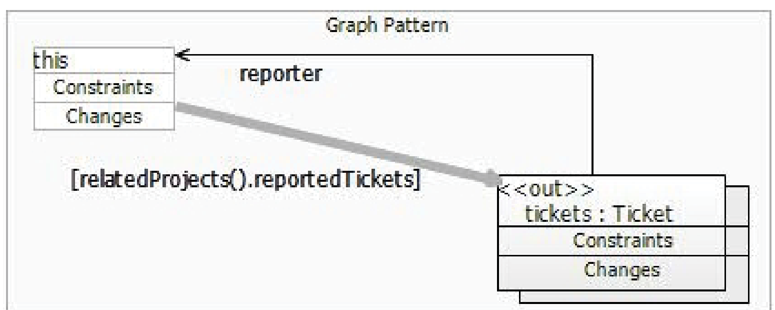


Abbildung 5.10: *ModGraph-Regel zur Zuweisung eines Tickets an einen Benutzer im Bugtracker; Implementierung der Operation* reportedTickets():Ticket[0..*] *der Klasse* User

gestellt, welcher erhalten werden soll. Er ist daher grau hinterlegt. Das Objekt, welches durch diesen Knoten repräsentiert wird, muss zur Laufzeit der Regel ermittelt werden. Dabei muss die Bedingung an den Knoten eingehalten werden. Die vorangegangene aktuelle Revision darf keinesfalls den Status zurückgewiesen haben. Diese Anforderung ist durch eine Bedingung direkt an den die Revision repräsentierenden Knoten gestellt und daher im Constraints-Bereich des Knotens zu finden. Sie kann - ebenso wie der existierende Link der Regel createProject - als positive Anwendbarkeitsbedingung an die Regel angesehen werden. Ausgehend von diesem Objekt wird eine beauftragte Person ermittelt. Hierbei ist es zulässig, dass kein Nutzer zuständig ist. Dies wird durch die Markierung << *optional* >> am Knoten erreicht, die diesen als optionalen Knoten definiert.

Zur Änderung des Status wird, ähnlich zur Regel der Operation createTicket, eine neue Revision des Tickets erzeugt und in ihren Kontext eingebettet. Auffällig ist dabei die Referenz assignee, die den zuständigen Nutzer festlegt. Die implementierenden Links enden in einem optionalen Knoten und werden deswegen als optional angenommen. Zudem muss der Link der aktuellen Revision umgesetzt werden. Dies geschieht hier durch explizites Löschen und erneutes Setzen des Links zur aktuellen Revision. Der zu löschende Link zur vorangehenden aktuellen Revision ist, wie alle zu löschenden Links, rot und mit $--$ markiert. Der Link history wird trotz der Instanziierung der geordneten Referenz wiederum als nicht explizit geordnet angegeben. Hierbei wird ausgenutzt, dass alle Referenzen in EMF implizit geordnet sind. Die erzeugte Revision wird am Ende der bereits in history enthaltenen Revisionen eingefügt. Eine alternative Variante, welche die neue Revision vor allen bereits bekannten einfügt, ist ausschnittsweise in Abbildung 5.9 dargestellt. Hier wird ein geordneter Link genutzt, der durch seine Kennzeichnung mit first das gewünschte Verhalten ermöglicht.

Implementierung der Operation reportedTickets der Klasse User

Ist ein Nutzer eines Bugtrackers sehr aktiv, so kann es für ihn von Vorteil sein, sich alle von ihm erstellten Tickets zur Ansicht ausgeben zu lassen. Dies wird durch die Operation reportedTickets modelliert, deren implementierende Regel in Abbildung 5.10 gezeigt ist. Die Operation weist einen mehrwertigen Rückgabeparameter des Typs Ticket

```
1  public class BugTrackerImpl extends MinimalEObjectImpl.Container
2          implements BugTracker {
3  ...
4      protected static final int RUNNING_NO_EDEFAULT = 0;
5  ...
6      protected int runningNo = RUNNING_NO_EDEFAULT;
7  ...
8      /**
9       * <!-- begin-user-doc --> Gibt die laufende
10      * Nummer für Tickets zurück <!-- end-user-doc -->
11      * @generated NOT
12      */
13     public int getNextNumber() {
14         return ++runningNo;
15     }
16 ...
17 }
```

Auflistung 5.2: *Java Implementierung der Methode* `getNextNumber()` *der Klasse BugTracker*

auf. Die Regel beinhaltet neben dem bereits bekannten Knoten für das aktuelle Objekt this einen mehrwertigen, ungebundenen Knoten. Er ist durch eine verdoppelte, leicht verschobene Silhouette dargestellt. Mehrwertige Knoten repräsentieren mehrere Objekte gleichen Typs. Hierbei reicht auch die Typisierung über eine gemeinsame Oberklasse aus. Zudem sind neben den ungebundenen Knoten auch gebundene mehrwertige Knoten möglich, die mehrwertige, nicht-primitive Parameter der Operation repräsentieren.

Der hier abgebildete Knoten wird durch die Markierung mit $<< out >>$ als Rückgabewert der Operation definiert. Er repräsentiert alle Tickets, die von genau dem Benutzer, der durch das aktuelle Objekt repräsentiert ist, erstellt wurden. Dies wird durch den zu erhaltenden Link reporter vom Ticket zu dessen Ersteller gefordert. Der Link ist jedoch unidirektional, da die instanziierte Referenz es auch ist. Damit kann hier nicht vom Nutzer (this) zu den Tickets navigiert werden, da auch sonst keine direkte Verbindung zwischen einem Nutzer und seinen Tickets besteht. Deshalb nutzt man einen Pfad. Dieser wird als dicker, grauer Pfeil dargestellt und verbindet zwei Knoten. Er stellt eine abgeleitete Referenz dar und muss einen Pfadausdruck tragen. Der Pfadausdruck ist hier in OCL verfasst und wird auf dem Quellobjekt, hier dem Benutzer, ausgewertet. Er gibt eine Menge von Zielobjekten, hier Tickets, zurück. Er ruft die Operation relatedProjects() auf, die alle mit dem Nutzer zusammenhängenden Projekte zurück gibt und holt sich aus diesen Projekten alle Tickets. Man beachte, dass dies genau der Fall eines geschachtelten Operationsaufrufes ist, da die Regel selbst eine Operation implementiert. Die durch den Pfad ermittelte Ticketliste enthält jedoch mehr als die vom aktuellen Nutzer erstellten Tickets. Sie wird durch den Link zum aktuellen Objekt nochmals gefiltert, bevor die resultierenden Objekte dem mehrwertigen Knoten zugewiesen werden.

5.4.3 Prozedurale Elemente im Bugtracker

Im vorangegangenen Abschnitt wurden Beispiele für die Implementierung von Operationen auf Modellebene betrachtet. Dabei zeigt sich, dass Graphtransformationsregeln sehr gut zur Modellierung von komplexen Strukturveränderungen an der Bugtracker-Instanz

```
1
2 package de.ubt.ai1.modgraph.bugmodel
3 ...
4 class Group {
5     String name
6     String description
7     refers User[] members opposite memberIn
8     refers Project[] accessibleProjects opposite accessingGroups
9     op EBoolean isActive() throws GTFailure
10 }
11 ...
12 class BugTracker {
13     int runningNo
14     contains User[] users
15     contains Project[] trackedProjects
16     contains Group[] groups
17     op User createUser(String nickname , String surname , String name ,
18     String email , Group group) throws GTFailure
19     op Group createGroup(String name) throws GTFailure
20     op Project createProject(String name , User leader ,
21     Group firstGroup) throws GTFailure
22     op User loginUser(String nickname) throws GTFailure
23     op unique Ticket[] allOpenTickets() throws GTFailure
24     op unique Project[] allCleanProjects() throws GTFailure
25     op unique Project[] allBrokenProjects() throws GTFailure
26
27 // Handgeschriebener prozeduraler Quelltext
28
29
30     op void init() throws GTFailure{
31         runningNo = 0
32         var group = createGroup("Initiatoren")
33         var user = createUser("erster", "Der Erste","Benutzer","erster@xyz.org",group)
34         createProject("erstes Projekt", user, group)
35     }
36     op int getNextNumber() {
37         runningNo = runningNo + 1
38         return runningNo
39     }
40 }   ...
```

Auflistung 5.3: *Ausschnitt aus dem generierten Xcore-Modell der Bugtrackers: Klassen Group und BugTracker mit Ergänzung um die Methoden zur Berechnung der laufenden Nummer für Tickets*

geeignet sind. Prozedurale Elemente werden von den Regeln selbst nicht angeboten. Einzige Ausnahme bilden hier die Operationsaufrufe innerhalb der Regel. Sie stellen geschachtelte Operationsaufrufe - eine Art prozeduralen Kontrollfluss - dar. Dadurch dürfen sich die Regeln einerseits gegenseitig aufrufen, andererseits auf einfache Operationen zurückgreifen. Beispielsweise wird in der Regel createTicket eine prozedurale Methode des Bugtrackers aufgerufen, die laufende Nummern an die Tickets verteilt. Weiterhin ruft die Regel reportedTickets die durch die Regel relatedProjects implementierte gleichnamige Operation auf.

Die aus createTicket aufgerufene Methode getNextNumber() hat stark prozeduralen Charakter. In der abgebildeten Regel (siehe Abbildung 5.6) wird daher vorausgesetzt, das sie in Java geschrieben ist (siehe Auflistung 5.2). Da in ModGraph durch die Integration von Xcore totale modellgetriebene Softwareentwicklung ermöglicht wird, soll diese

Methode nun ebenfalls auf Modellebene spezifiziert werden. So wird sie, den Richtlinien des ModGraph-Ansatzes folgend, in Xcore verfasst. Xcore mit Xbase wird in ModGraph als prozedurale Sprache zur Unterstützung und Steuerung der Regeln gewählt, da nach den Ergebnissen von Buchmann et al. [20], eine graphische Sprache keinerlei Vorteile in der prozeduralen Verhaltensmodellierung bringt. Gleichzeitig bietet Xcore eine gut verständliche, kompakte und konzise textuelle Syntax. Zudem folgt der Einsatz von Xcore der Linie des ModGraph-Ansatzes, die vorgibt, dass hier eine echte Erweiterung von EMF stattfinden soll, ohne EMF unnötig zu belasten. Dazu werden bereits bestehende Sprachbestandteile wiederverwendet. Die resultierende, strikte Trennung von prozeduraler und regelbasierter Verhaltensmodellierung in Kombination mit deren symbiotischer Zusammenarbeit ist dabei als Vorteil zu sehen, weil diese unter anderem zur Übersichtlichkeit des Modells beiträgt. Dies wird im Folgenden deutlich.

Zur Spezifikation von prozeduralen Anteilen des Modells muss zunächst das Ecore-Modell des Bugtrackers in ein Xcore-Modell überführt werden. Dies ist problemlos möglich, da das zum Ecore-Modell gehörende Generator-Modell - durch Nutzung seines Export-Rahmenwerks - in ein Xcore-Modell transformiert werden kann.

Das erzeugte Xcore-Modell ist in Auflistung 5.3 ausschnittsweise gezeigt. Die Klasse BugTracker ist bereits um die Ermittlung der laufenden Nummer (Zeilen 36-39) erweitert. Diese wird jedem Ticket, unabhängig von seiner Projektzugehörigkeit, zugewiesen. Dazu wird im Bugtracker eine Variable initialisiert, welche die Nummer des nächsten zu verwaltenden Tickets speichert (Zeile 13) sowie eine Methode, die diese ermittelt (Zeilen 36-39). Sie wird nun auch aus der Regel createTicket aufgerufen. Um die modellierte Operation korrekt aufzurufen, muss die Zuweisung innerhalb der Regel verändert werden in den Xbase-Ausdruck number = ((eContainer().eContainer()) as BugTracker).getNextNumber(), da Xbase keine Java-Typumwandlungen unterstützt, jedoch seine eigenen durch das Schlüsselwort as anbietet.

Neben diesen einfachen Methoden kommt kaum ein Programm ohne die aus der strukturierten Programmierung bekannten Kontrollstrukturen aus. Eine einfache Sequenz zur Initialisierung des Bugtrackers ist in Auflistung 5.3 in Zeilen 30-35, gezeigt. Hier wird zuerst eine Gruppe namens „Initiatoren“, danach ein Benutzer mit Benutzernamen „Erster“ und zuletzt ein Projekt namens „erstes Projekt“ angelegt. Zudem werden Schleifen und Bedingungen benötigt. Angenommen, ein Nutzer des Bugtrackers möchte mehrere Tickets zu einem Projekt auf einmal erstellen, wäre es von Vorteil, eine Schleife zur Verfügung zu haben. Möchte ein Nutzer ein Ticket zu einem noch nicht existenten Projekt in den Bugtracker einpflegen, ist es für ihn bequemer, wenn bei der Erstellung eines Tickets das zugehörige neue Projekt mit angelegt werden kann. Dazu wäre es vorteilhaft, die Regel nicht einfach scheitern zu lassen, sondern eine Möglichkeit anzubieten, das Projekt direkt an dieser Stelle zu erstellen. Hier kommt Xcore als Kontrollflusssprache ins Spiel. In Auflistung 5.4 wird die Operation createProjectAndTickets der Klasse BugTracker dargestellt. Diese wird dem Bugtracker-Xcore-Modell als komplexe Operation hinzugefügt.

Die Operation legt zunächst ein neues Projekt durch Aufruf der Regel createProject an (Zeile 6). Scheitert dies, wird geprüft, ob ein Projekt mit identischem Namen bereits existiert (Zeile 10). Dies ist genau dann der Fall, wenn die Regel an ihrer NAC ge-

```
op boolean createProjectAndTickets(String projectName , User projectLeaderAndReporter,
        Group projectsfirstGroup, Severity[] ticketSeverity,
                EString[] ticketTitle, EString[] ticketDescription) {
    var Project project = null
    try {
        project = createProject(projectName, projectLeaderAndReporter,
                      projectsfirstGroup)
    }
    catch(de.ubt.ai1.modgraph.gt.failure.GTFailure failure){
        project = this.trackedProjects.findFirst[Project p | p.name == projectName]
        if(project==null)
            return false
        else { /*Notify user, that a project with the given name already exists
                    * and will be used if he doesn't cancel the operation */
        }
    }
    var int i = 0
    try{
        if(ticketSeverity.size == ticketTitle.size == ticketDescription.size){

            while (i < ticketSeverity.size) {
                project.createTicket(ticketSeverity.get(i), ticketTitle.get(i),
                         ticketDescription.get(i), projectLeaderAndReporter)
                i = i + 1
        }
        else throw new de.ubt.ai1.modgraph.gt.failure.GTFailure("Arraysizes unequal")
        }catch (de.ubt.ai1.modgraph.gt.failure.GTFailure failure) {
            return false
        }
    }
    return true
}
```

Auflistung 5.4: *Xcore als Kontrollflussprache zur korrekten Kombination von Projekt und Ticketerstellung*

scheitert ist. Hierzu wird eine interne Xbase-Funktion verwendet, die das erste Projekt mit dem angegebenen Namen zurückliefert. Da die Regel zur Erzeugung eines Projekts festlegt, dass Namen eindeutig sein müssen, ist es auch das einzige Projekt dieses Namens. Ist die Regel anderweitig gescheitert, beispielsweise durch einen mit dem Wert `null` übergebenen Parameter, false zurückgegeben (Zeile 12). In einem nächsten, aus Gründen der Übersichtlichkeit nicht gelisteten Schritt, wird der Nutzer benachrichtigt, dass die Tickets in ein bereits bestehendes Projekt eingepflegt werden, sofern er an dieser Stelle nicht widerspricht. Anschließend werden die Tickets erzeugt. Da die Titel, die Beschreibungen und die Dringlichkeit separat übergeben werden, muss zunächst deren Konsistenz überprüft werden (Zeile 19): Sie müssen die gleiche Länge haben. Ist dies erfüllt, werden die Tickets in einer Schleife erstellt (Zeilen 21-25). Dabei werden die Einträge der Listen ihrer Position gemäß zu Tickets zusammengefasst. Ist dies erfolgreich, gibt die Methode true zurück, andernfalls false. Auffällig ist hier, dass jede der mit ModGraph implementierten Operationen innerhalb eines Blocks zur Ausnahmebehandlung aufgerufen wird (try-catch-Blöcke Zeilen 5-16 und 18-29). Dies ist notwendig, da jede Regel bei ihrem Scheitern eine Ausnahme auslöst. Diese wird hier durch die aufrufende Methode behandelt, da sie selbst einen Wahrheitswert zurückgibt.

5.4.4 Das ausführbare Bugtracker-Modell

Nachdem das gesamte Modell - den im Vorangegangenen vorgestellten Methoden folgend - modelliert wurde, kann es ausgeführt werden. Dazu stehen verschiedene Wege zur Verfügung, die in Kapitel 6.2 genauer besprochen werden. Im soeben betrachteten Bugtracker kann einerseits Java-Quelltext aus dem Xcore-Modell erzeugt werden, in welchen der ModGraph-generierte Quelltext nahtlos eingefügt wird. Andererseits kann das Xcore-Modell erweitert werden, indem die Regeln in mit Xbase implementierte Operationen des Modells übersetzt und damit nahtlos in das Modell eingefügt werden.

Dazu ist die Vereinigung der Modelle und damit die Erstellung des ausführbaren Modells auf Quelltext-Niveau und auf Modell-Niveau möglich. Außerdem ist nicht nur die Generierung von voll funktionsfähigem Quelltext möglich, sondern auch die Interpretation des Xcore-Modells. Diese ist gleichzusetzen mit einer indirekten Interpretation der Regeln, was zu Test- und Debug-Zwecken sehr nützlich ist. Zudem bietet diese Art der Interpretation eine kostengünstige Alternative zur Implementierung eines eigenen Interpreters für Graphtransformationsregeln, ohne dabei Einbußen in der Funktionalität hinzunehmen.

Das Bugtracker-Modell könnte, sobald der Quelltext generiert ist, mit einer Weboberfläche oder einer andern grafischen Oberfläche versehen und eingesetzt werden. Da die Logik vollständig implementiert ist, bietet sich das Model-View-Controller-Muster zur Erstellung der Oberfläche für den Endanwender an.

5.4.5 Zusammenfassung der Konzepte

Soeben wurden die in ModGraph zur Spezifikation eines ausführbaren Modells genutzten Konzepte vorgestellt. Dabei wurden insbesondere die Graphtransformationsregeln am Beispiel Bugtracker gezeigt. Die Konzepte werden hier zusammengefasst.

Nachdem der Nutzer ein strukturelles Modell unter Verwendung von Ecore oder Xcore definiert hat, steht ihm im ModGraph-Ansatz die Möglichkeit offen, das Verhalten mittels Graphtransformationsregeln zu modellieren.

Jede Graphtransformationsregel implementiert eine Operation des Ecore-Modells. Sie modelliert das Verhalten der Operation. Durch die Nutzung der zusammengeführten Darstellung für die linke und rechte Regelseite kann die Regel einen Test, eine Anfrage oder eine Transformation darstellen. Tests implementieren Boolesche Operationen. Diese stellen eine Bedingung an die Instanz. Ihre Ausführung führt immer zu einem Wahrheitswert, der Aufschluss gibt, ob die betrachtete Instanz die Bedingung erfüllt. Anfragen ändern die Instanz nicht, geben jedoch ein einzelnes Objekt oder eine Menge von Objekten zurück. Transformationen führen, wie der Name sagt, Änderungen auf den Instanzen durch. Sie dürfen, wie die Abfragen, einen Rückgabewert aufweisen.

Die Kernkomponente einer Graphtransformationsregel ist das Graphmuster. Es gibt das zu suchende Objektmuster innerhalb der Modellinstanz, sowie die Änderungen, welche an diesem vorgenommen werden sollen, an. Dazu werden Graphen genutzt. Die Knoten stehen stellvertretend für Objekte der Instanz, die Kanten stellen die Beziehungen zwischen ihnen dar. Zur Modellierung stehen verschiedene Knoten- und Kantenar-

ten zur Verfügung. Knoten werden in einwertige und mehrwertige sowie in gebundene und ungebundene unterschieden. Einwertige Knoten stehen für Objekte und mehrwertige Knoten für Mengen von Objekten gleichen Typs. Zudem wird zwischen obligatorischen und optionalen Knoten unterschieden. Während ein obligatorischer Knoten zur Ausführung der Regel auf ein Objekt der Instanz abgebildet werden muss, kann das Objekt zum optionalen Knoten in der Instanz fehlen. Dies geschieht genau dann, wenn keine Übereinstimmung zwischen dem Knoten und den Objekten der Instanz gefunden werden kann. Optionale Knoten werden mit << optional >> markiert.

Gebundene Knoten stellen nicht-primitive Parameter der Operation dar. Sie repräsentieren ein Objekt, keinen Datentyp. Ist ein solcher Knoten optional, wird er als unter Umständen `null`-wertig aufgefasst. Parameterknoten werden in der Regel ohne ihren Typ abgebildet, es sei denn, sie unterliegen einer Typumwandlung. Dann wird der abweichende Typ angezeigt.

Ein besonderer gebundener Knoten ist derjenige für das aktuelle Objekt. Er wird in Anlehnung an Programmiersprachen immer mit `this` benannt und repräsentiert das Objekt, auf welchem die Operation aufgerufen wird. Dieser Knoten ist immer obligatorisch.

Neben den gebundenen Knoten werden ungebundene Knoten im Graphmuster angeboten. Diese müssen zur Laufzeit durch die Mustersuche gefunden werden und sind über Klassen typisiert. Einwertige ungebundene Knoten stehen für Objekte der betrachteten Modellinstanz, wohingegen mehrwertige ungebundene Objekte Mengen von Objekten gleichen Typs repräsentieren.

Allen Knoten im Graphmuster, ausgenommen dem das aktuelle Objekt repräsentierende, ist ein Status zugewiesen. Dieser ist entweder zu erhalten, zu löschen oder zu erzeugen. Zu erhaltende Knoten werden grau hinterlegt, zu löschende rot und mit $--$ markiert, zu erzeugende grün und mit ++ gekennzeichnet. Der Status des Knotens für das aktuelle Objekt kann nicht verändert werden. Es ist implizit zu erhalten. Nicht-primitive Parameter repräsentierende Knoten können entweder den Status zu erhalten oder zu löschen tragen. Ungebundene einfache Knoten dürfen alle drei Status annehmen. Sie können zu erhalten, zu löschen oder zu erzeugen sein. Einwertige Knoten müssen bei ihrer Erzeugung über konkrete Klassen (nicht abstrakt und kein Interface im EMF-Kontext) typisiert werden. Ungebundene, mehrwertige Knoten dürfen nur erhalten oder gelöscht werden. Statusbehaftete Knoten dürfen ebenso als Rückgabeparameter der Operation gekennzeichnet werden. Dies geschieht durch eine Markierung mit << out >>. Sind mehrere Knoten konform zur implementierten Operation als Rückgabeparameter markiert, werden diese (als Liste) zurückgegeben. An zu erhaltende und zu löschende Knoten dürfen Bedingungen gestellt werden. Diese sind textuell und können als OCL- oder Xcore-Bedingung formuliert werden. Zudem sind Bedingungen an Attributwerte möglich. Knoten, die erhalten oder erzeugt werden, können Attributzuweisungen und Operationsaufrufe enthalten.

Knoten können durch Kanten verbunden werden. Dazu stehen Links und Pfade zur Auswahl. Links sind Instanzen der Referenzen im Modell. Sie dürfen zwischen Knoten gezogen werden, die Objekte des Typs der Quell- und Zielklasse der Referenz repräsentieren. Natürlich sind auch Knoten, die auf Objekte einer Unterklasse der Quell- oder Zielklasse verweisen, zulässig. Ein Link wird durch den Namen der Referenz gekenn-

zeichnet. Zudem kann jedem Link einer der drei bekannten Status zugeordnet werden. Dabei ist es unzulässig, widersprüchliche Status an Knoten und an die mit ihnen verbundenen Links zu vergeben. So kann beispielsweise in einen zu löschenden Knoten kein zu erzeugender Link eingehen. Des Weiteren dürfen keine Links - und auch keine Pfade - zwischen mehrwertigen Knoten gezogen werden. Mehrwertige zu erzeugende Links können geordnet sein. Sie geben an, an welcher Stelle das Objekt in den Link aufgenommen wird. Dabei sind first, last und die direkte Angabe der Position als Index zugelassen sowie das Einfügen vor oder nach einem namentlich vorgegebenen Objekt. Pfade stellen abgeleitete Referenzen dar und werden mit einem Pfadausdruck beschriftet, der entweder in OCL oder Xcore geschrieben werden kann. Dieser Pfadausdruck wird auf dem durch den Quellknoten des Pfads dargestellten Objekt ausgewertet und liefert mindestens ein Zielobjekt zurück. Stehen mehrere Objekte zur Auswahl, wird aber nur eines benötigt, so wird das erste gefundene genutzt.

Jede Regel kann durch Bedingungen erweitert werden. Dies können einerseits textuelle Vor- und Nachbedingungen sein, andererseits dürfen durch Graphen dargestellte negative Anwendbarkeitsbedingungen (NACs) verwendet werden, die verbotene Graphmuster darstellen. Die textuellen Bedingungen werden in OCL oder Xcore bzw. Xbase spezifiziert. NACs werden durch Graphen dargestellt. Sie stellen Verbote dar. Dies bedeutet, dass sie eine Art Graphtest spezifizieren, der zur Ausführung der Regel fehlschlagen muss. Innerhalb der NACs ist nur die Verwendung einwertiger Knoten zulässig, da es ausreicht, wenn ein Knoten das modellierte Verbot erfüllt. Zudem sind die Knoten und Kanten der NACs statuslos.

Die Ausführung der Regeln kann mit Kontrollflüssen gesteuert werden. Sie werden als Xcore-Operationen programmiert. Diese rufen geeignet die durch Regeln implementierten Operationen auf. Dabei sind alle gängigen Konstrukte wie Schleifen und Bedingungen verfügbar. Zudem besteht die Möglichkeit, einfache oder prozedurale Operationen direkt in Xcore zu implementieren. Diese können durch einen Operationsaufruf innerhalb der Regeln genutzt werden. Dies kann der reine Aufruf der Operation sein oder die Zuweisung ihres Rückgabewertes an ein Attribut. Damit arbeiten die Xcore-Operationen und die Regeln symbiotisch.

6 Design der Sprache für Graphtransformationsregeln

Nach der informellen Einführung soll die Sprache der Graphtransformationsregeln in ModGraph nun formal dargestellt werden. Bereits in Kapitel 2, Abschnitt 2.3, wurde gezeigt, welche syntaktischen und semantischen Komponenten nötig sind, um eine (ausführbare) Sprache vollständig zu beschreiben. Dieser Sprachdefinition folgend, wird hier zuerst die Syntax der Sprache für Graphtransformationsregeln in ModGraph dargestellt. Ihre dynamische Semantik wird gesondert in Abschnitt 6.2 untersucht, da sie die gesamte Ausführungslogik der Sprache beinhaltet.

Ziel dieses Kapitels ist die präzise Beschreibung der ausführbaren Sprache zur Spezifikation der Graphtransformationsregeln.

6.1 Syntax der Sprache für Graphtransformationsregeln

In diesem Abschnitt wird zunächst das Metamodell der Sprache erläutert. Darauffolgend wird in Abschnitt 6.1.2 die konkrete Syntax anhand der im Editor für Graphtransformationen zur Verfügung stehenden Elemente erklärt. Hier wird gleichzeitig die lexikalische Syntax betrachtet.

6.1.1 Das Metamodell der Sprache

Die abstrakte Syntax und die kontextsensitive Syntax bilden zusammen das Metamodell der Sprache (siehe Kapitel 2.3, Abbildung 2.3). Zur folgenden Erklärung der abstrakten Syntax wird das Ecore-Metamodell für Graphtransformationsregeln beschrieben, zur Erklärung der kontextsensitiven Syntax werden die zusätzlich an das Modell gestellten Bedingungen aufgeführt, welche sich in der Validierung der Regeln widerspiegeln.

Die abstrakte Syntax

Die Komponenten des Ecore-Metamodells für Graphtransformationsregeln und deren Beziehungen untereinander entsprechen der abstrakten Syntax der Sprache für Graphtransformationsregeln und definieren gleichzeitig die verfügbaren Sprachkonstrukte.

Dieses Ecore-Metamodell für Graphtransformationsregeln wurde von Grund auf als **Werkzeugmetamodell** angelegt. Der Fokus liegt dabei auf einer möglichst einfachen Umsetzung in den GMF-Editor, indem die spezifischen Anforderungen des GMF-basierten Werkzeugs bereits während der Modellerstellung berücksichtigt sind. Das Modell

stellt für jedes später im Editor vorhandene Element eine eigene Klasse bereit. Dies führt zu einem etwas redundant wirkenden Metamodell, erleichtert jedoch die Abbildung auf die grafischen Elemente ungemein, da diese annähernd eins zu eins unter Zuhilfenahme einiger weniger OCL-Ausdrücke erfolgt.

Die Alternative wäre ein minimales, werkzeugunabhängiges Metamodell, das einen Graphen modelliert und dabei nur unter Zuhilfenahme vieler OCL-Ausdrücke auf die grafischen Elemente abgebildet werden kann. Diese Möglichkeit erscheint bis heute nicht attraktiv, da die OCL-Ausdrücke sehr schnell unübersichtlich werden. Der Kompromiss, welcher zwischen Redundanzen und einer Vielzahl von OCL-Ausdrücken gefunden werden musste, fiel zugunsten eines einfachen Abbildungsmodells und damit der Redundanzen aus.

Die Metamodell-Beschreibung in den nächsten Abschnitten folgt [21]. Das Metamodell wird zum besseren Verständnis ausschnittsweise erklärt, die Übersicht des gesamten Metamodells ist in Abbildung 6.1.1 zu finden.

Hauptkomponenten einer ModGraph-Graphtransformationsregel

Abbildung 6.2 zeigt die Hauptkomponenten einer hierarchisch aufgebauten ModGraph-Regel. Jeder Regel (Klasse GraphTransformationRule) wird ein Name zugeordnet. Sie referenziert die durch sie implementierte Operation (Referenz implementedOperation) und deren Klasse (Referenz correspondingClass).[1] Aufgrund der Kompatibilität zu Xcore und Ecore verweisen beide Referenzen auf EObject. Zudem unterliegen beide einer Typverengung. Die generischen Datentypen EXOperation bzw. EXClass werden genutzt, um zulässige Objekte vorzugeben. Im Falle der Operation sind nur Objekte vom Typ EOperation oder XOperation zugelassen. Bei Klassen sind nur EClass oder XClass erlaubt. Dies wird bei Erstellung der jeweiligen Regel festgelegt.[2]

Jeder Graphtransformationsregel wird eine Ausnahme (Klasse GTFailure) zugeordnet, die im Falle des Scheiterns der Regel ausgelöst wird. Weiterhin besteht die Möglichkeit der Erstellung eines Kommentars (Klasse GTComment) zu jeder Regel. Der Inhalt des Kommentars wird als Zeichenkette gespeichert.

Einer Regel ist genau ein Graphmuster (Klasse GTGraphPattern) zugeordnet. Es stellt den zu suchenden Teilgraphen, also ein Objekt-Muster, in der als Graph aufgefassten EMF-Instanz dar, sowie die Änderungen, die an diesem vorgenommen werden. Graphmuster werden durch die enthaltenen Elemente (Interface GPElement) spezifiziert.

Ein Graphmuster kann durch grafische und textuelle Anwendbarkeitsbedingungen ergänzt werden. Da die Anwendbarkeitsbedingungen grundsätzlich verschieden sind, werden nicht alle in einer gemeinsamen Oberklasse zusammengefasst. Textuelle Bedingungen (Interface GTCondition) sind Vor- und Nachbedingungen an die gesamte Graphtransformationsregel, die durch die Klasse GTPrecondition und Klasse GTPostcondition modelliert

[1] Obwohl die Klasse (als Container der Operation) nicht direkt referenziert werden müsste, wird - im Hinblick auf die Resourcenverwaltung im Werkzeug - dieser Link zum schnelleren Zugriff explizit gesetzt.

[2] Generizität kann mit dem grafischen Editor nicht abgebildet werden, ist jedoch im Anhang A.1, der das Metamodell im Baumeditor darstellt, ersichtlich.

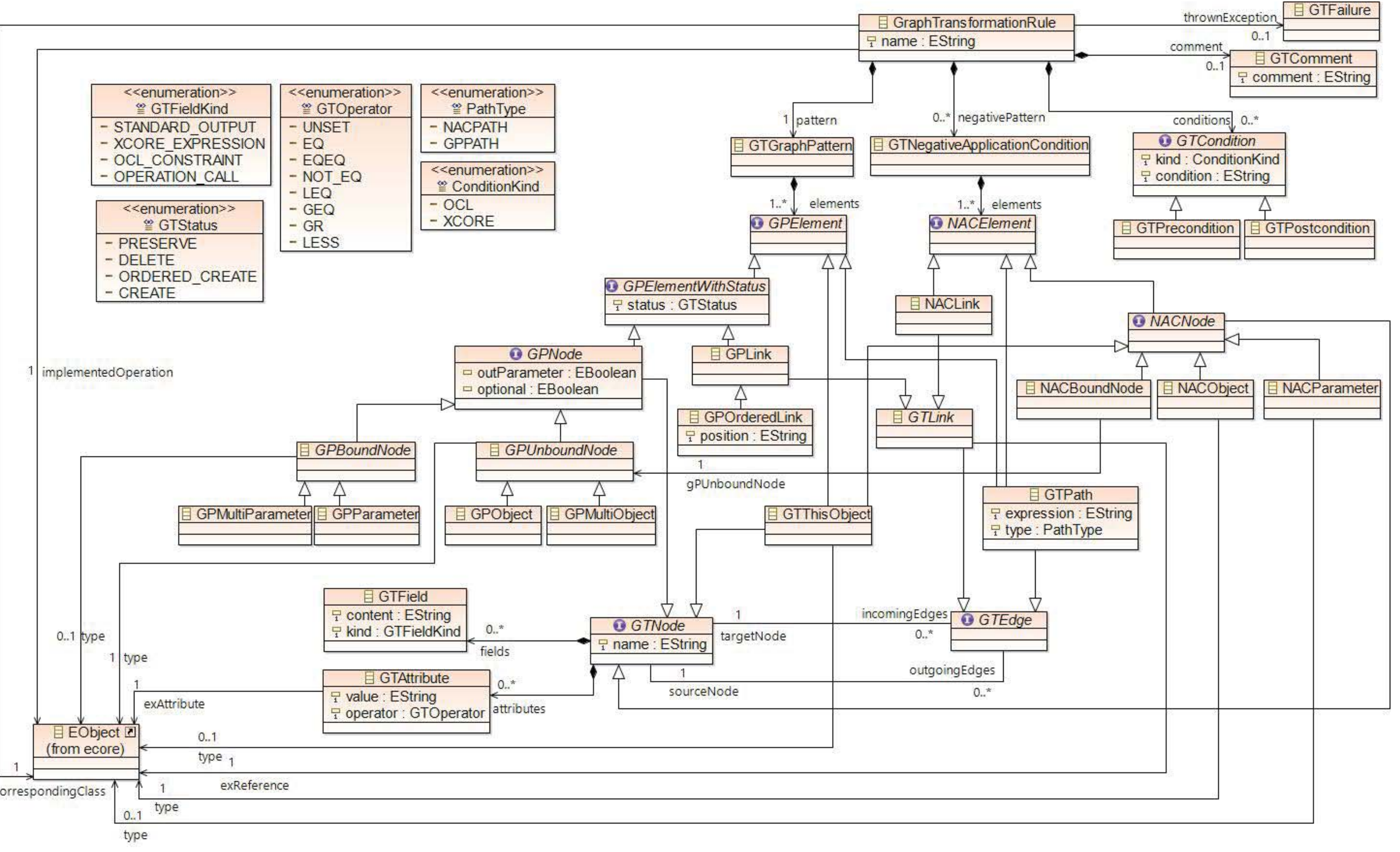


Abbildung 6.1: *Überblick - Das gesamte ModGraph Metamodell für Graphtransformationen*

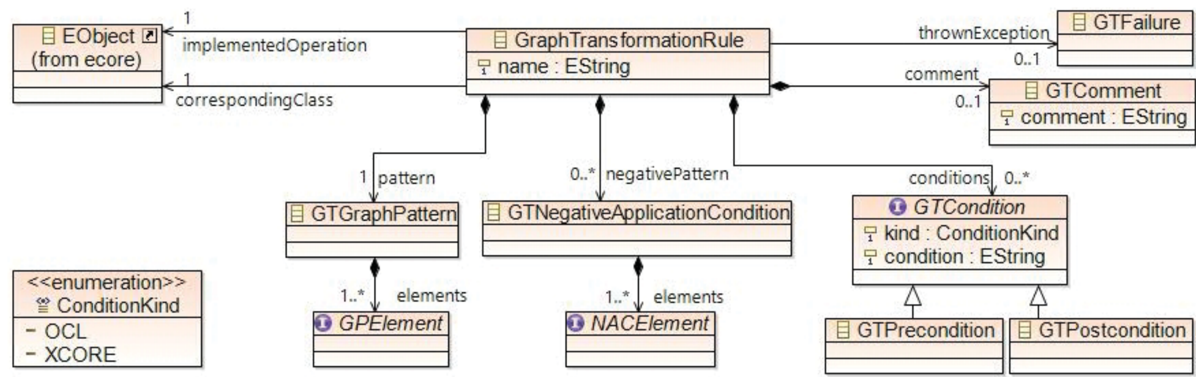


Abbildung 6.2: *Ausschnitt aus dem ModGraph-Metamodell für Graphtransformationen: Die Hauptkomponenten einer hierarchisch aufgebauten ModGraph-Regel*

werden. Der Typ der Bedingung, modelliert durch den Aufzählungsdatentyp ConditionKind, der dem Attribut GTCondition.kind zugewiesen wird, gibt an, ob diese in OCL oder Xcore formuliert ist. Die Bedingung selbst wird als Zeichenkette repräsentiert (Attribut GTCondition.condition). Außerdem kann das Graphmuster mit negativen Anwendbarkeitsbedingungen (Klasse GTNegativeApplicationCondition) versehen werden. Sie werden grafisch durch Objekt-Muster - analog zum Graphmuster - dargestellt. Die darin enthaltenen Elemente (Interface NACElement) spezifizieren Sachverhalte, die keinesfalls in der Modell-Instanz auftreten dürfen. Jeder Regel können beliebig viele, konjunktiv verknüpfte, negative Anwendbarkeitsbedingungen zugeordnet werden, die insgesamt nicht zutreffen dürfen, wenn die Regel ausgeführt werden soll.

Aufbau des Graphmusters

Abbildung 6.3 zeigt das Metamodell des Graphmusters. Es ist der einzige zwingende Bestandteil einer ModGraph-Graphtransformationsregel. Es besteht, wie in Abschnitt 5.2.3 beschrieben, aus einem statischen Teil und einem dynamischen Teil. Somit steuert das Graphmuster den Ablauf der Transformation durch die dargestellten Elemente (Interface GPElement und dessen implementierende Klassen).

Konkret werden im Graphmuster Elemente mit und ohne Status unterschieden. Klassen, die statusbehaftete Elemente modellieren, implementieren das Interface GPElementWithStatus, welches von GPElement erbt. Klassen, die statuslose Elemente modellieren, implementieren direkt das Interface GPElement.

Der Status eines Elements gibt an, ob dieses erhalten, gelöscht oder erzeugt wird. Dies wird durch den Aufzählungsdatentyp GTStatus modelliert, der die Werte PRESERVE für zu erhaltende Elemente, DELETE für zu löschende Elemente und CREATE bzw. ORDERED_CREATE für zu erzeugende Elemente annehmen kann. Der Status ORDERED_CREATE ist für die Erzeugung geordneter Links reserviert und wird später genauer betrachtet.

Das Graphmuster besteht aus mindestens einem gebundenen Knoten. Dieser Knoten ist entweder das statuslose, aktuelle Objekt (Klasse GTThisObject) oder ein statusbe-

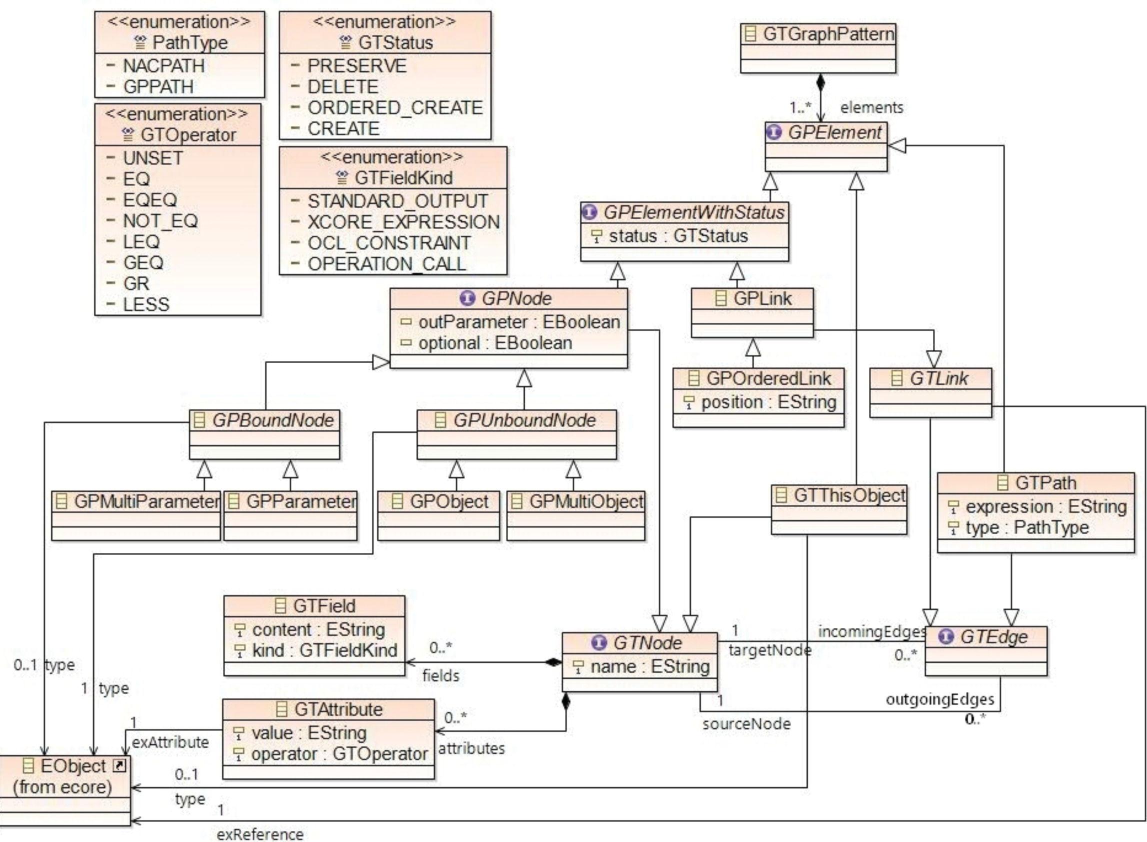


Abbildung 6.3: *Ausschnitt aus dem Metamodell für Graphtransformationen: Aufbau des Graphmusters*

hafteter Übergabeparameter der Methode, der über einer Klasse typisiert ist (Klasse GPBoundNode) und im Folgenden nicht-primitiver Parameter der Methode genannt wird. Die Übergabeparameter werden, je nach Wertigkeit, in mehrwertige Parameter (Klasse GPMultiParameter) und einwertige Parameter (Klasse GPParameter) unterteilt. Als Status stehen ihnen Erhalten und Löschen zur Verfügung. Die Erzeugung eines Parameters innerhalb einer Graphtransformationsregel ist nicht möglich, da dies dem Konzept der Regel widerspricht: Regeln implementieren das Verhalten einer Operation. Daher müssen ihnen alle Parameter der durch sie implementierten Operation zwingend bekannt sein, selbst wenn diese kein Objekt referenzieren (Wert `null`). Parameter, die kein Objekt zu referenzieren, werden als optionale Parameter markiert. Das zugehörige modellierte Attribut optional findet sich im Interface GPNode, das von der abstrakten Klasse GPBoundNode implementiert wird. Diese ist ihrerseits die Oberklasse der die Parameter repräsentierenden Klassen. Zudem besteht die Möglichkeit, Typumwandlungen auf Parametern zu spezifizieren. Das wird durch Anpassung der Referenz type zwischen GPBoundNode und EObject erreicht. Diese Referenz muss für gebundene Knoten nicht zwingend gesetzt sein, da ihr Typ implizit gegeben ist.

Diesbezüglich anders verhält es sich bei den ungebundenen Knoten im Graphmuster. Sie müssen über Klassen aus dem Ecore-Modell typisiert werden. Die ungebundenen Knoten sind durch die abstrakte Klasse GPUnboundNode modelliert und dürfen in beliebiger Anzahl im Graphmuster vorkommen. Ungebundene Knoten werden durch die Mustersuche (siehe Abschnitt 6.2.2) gebunden. Sie sind, analog zu den Parametern, in mehrwertige Objekte (Klasse GPMultiObject) und einwertige Objekte (Klasse GPObject) unterteilt. Auch ungebundene Knoten dürfen als optional (Attribut GPNode.optional) gekennzeichnet werden.

Jeder Knoten im Graphmuster, der durch eine von GPNode erbende Klasse modelliert ist, kann als Rückgabewert der Methode deklariert werden, indem das entsprechende Attribut GPNode.outParameter gesetzt wird.

Zudem tragen alle Knoten im Graphmuster Namen und können durch gerichtete Kanten verbunden werden. Dies ist durch die Klasse GTNode und deren Attribut GTNode.name sowie durch die Klasse GTEdge modelliert. Sie stellen den allgemeinen, gerichteten Graphen, auf dem die spezialisierten Klassen zur Darstellung der Knoten und Kanten im Graphmuster aufbauen, dar. Daher erben alle Klassen für Knoten im Graphmuster von GTNode sowie alle Kantenklassen von GTEdge.

Bei der Verbindung von Knoten durch Kanten werden im Graphmuster zwei Arten von Kanten unterschieden: Links und Pfade. Pfade sind durch die Klasse GTPath modelliert und stehen für abgeleitete Referenzen. Sie sind mit einem Pfadausdruck markiert, der auf dem Quellobjekt ausgewertet wird und eine Menge von Zielobjekten zurückliefert. Ein Pfadausdruck ist entweder in OCL, Java oder Xcore (bzw. Xbase) formuliert. Er wird zunächst jedoch als Zeichenkette aufgefasst.[3] Die Zugehörigkeit des Pfades (Aufzählungsdatentyp PathType) setzt fest, ob er Teil eines Graphmuster oder einer negativen Anwendbarkeitsbedingung ist. Pfade innerhalb eines Graphmusters sind immer vom Typ GPPath.[4]

Links (Klasse GPLink) stellen Instanzen von Referenzen im Ecore-Modell dar. Sie referenzieren diese (Referenz exReference). Daher können Links nur zwischen bestimmten Knoten existieren. Diese müssen Instanzen der Klassen repräsentieren, die durch die Referenz, die der Link instanziiert, verbunden werden dürfen. Dabei kann es vorkommen, dass einer der Knoten optional ist. Links, die mit mindestens einem optionalen Knoten verbunden sind, werden ebenfalls als optional angenommen. Unabhängig davon ist jedem Link ein Status (Aufzählungsdatentyp GTStatus) zugeordnet. Ein zu erzeugender Link, der eine mehrwertige Referenz instanziiert, kann geordnet sein. Dies entspricht dem Status ORDERED_CREATE. Bei geordneten Links wird das Einfügen des Zielobjekts an einer bestimmten Position, die durch einen Ausdruck am Link vorgegeben ist, erreicht. Hierbei wird ausgenutzt, dass EMF alle mehrwertigen Referenzen als geordnete Listen umsetzt. Als mögliche Ausdrücke zur Anordnung der Listenelemente sind „first“, „last“, „after <Elementname>“ und „before <Elementname>“ zugelassen, um es als erstes oder letztes Element bzw. vor oder nach einem anderen, das namentlich genannt werden muss,

[3]In welcher Sprache der Ausdruck letztendlich geschrieben ist, wird hier nicht näher angegeben, da diese durch Parsen des Ausdrucks ermittelt werden kann.

[4]Die Zuordnung eines Pfades zu einem Muster ist aus Gründen der Exklusivität nötig.

einzufügen. Standardmäßig werden die Elemente im Übrigen am Ende eingefügt. Damit entspricht das explizite Einfügen am Ende der Liste der Erzeugung eines ungeordneten Links.

Die Knoten des Graphmusters sind, der Vererbungshierarchie folgend, alle Instanzen der Klasse GTNode. Sie dürfen daher Attribute und Felder beinhalten. Attribute haben einen Typ und einen Namen, die beide aus der Referenz exAttribute auf ein EAttribute (strukturelles Modell in Ecore) oder XAttribute (strukturelles Modell in Xcore) ermittelt werden. Ihnen ist ein Attributwert (Attribut GTAttribute.value) zugeteilt, dessen Funktion sich durch den verwendeten Operator unterscheidet. Entweder wird dem Attribut der Wert zugewiesen oder er dient als Bedingung an das Attribut. Operatoren werden durch den Aufzählungsdatentyp GTOperator definiert. Neben der Zuweisung unterstützt dieser den Vergleich von Attributen mit den übergebenen Werten. Dabei können unter anderem Ganzzahlen, Zeichenketten und Wahrheitswerte verglichen werden. Zudem ist es möglich, Werte anderer Attribute auf der rechten Seite der Attributzuweisung zu verwenden. Diese beziehen sich immer auf den Ausgangszustand der Modellinstanz vor der Transformation. Zusätzlich dürfen Knoten im Allgemeinen Felder (GTField) enthalten. Diese haben einen Inhalt, der als Zeichenkette repräsentiert wird und einen Typ (Attribut GTField.kind). Der Typ wird durch den Aufzählungsdatentyp GTFieldKind festgelegt. Dabei kann ein Feld einen Operationsaufruf auf einem Knoten, eine OCL-Bedingung an einen Knoten, ein Xcore-Ausdruck auf dem Knoten oder eine Standardausgabe (zu Testzwecken) repräsentieren.

Diese allgemeine Attribuierung von Knoten und deren Ausstattung mit beliebigen Feldern ist in der Praxis limitiert. So macht es beispielsweise keinen Sinn, Attributzuweisungen auf zu löschenden Knoten zuzulassen. Demzufolge stehen den Knoten im Graphmuster je nach Status verschiedene Möglichkeiten bezüglich ihrer Attribute und Felder zur Wahl. Diese sind als Bedingungen an das Metamodell formuliert und werden daher in Abschnitt 6.1.1 detaillierter besprochen. Die Bedingungen werden mit Hilfe eines Feldes innerhalb des Knotens (Klasse GTField) als OCL- oder Xcore-Bedingung formuliert. Alternativ werden sie als OCL-, Java- oder Xcore-Bedingung an ein Attribut des Knotens (Klasse GTAttribute) gestellt, indem ein in einer der Sprachen formulierter Wert mit dem aktuellen Attributwert verglichen wird. An einem zu erhaltenden oder zu erstellenden Knoten können Änderungen an Attributen vorgenommen werden. Ein Attributwert kann durch einen OCL-, Xcore- oder Java-Ausdruck spezifiziert werden. Hat eine Methode Übergabeparameter, die über einem Datentyp typisiert sind, so können diese für Attributbedingungen und Attributzuweisungen verwendet werden. Zudem besteht die Möglichkeit, Methoden auf einem zu erhaltenden oder zu erstellenden Knoten aufzurufen.

Aufbau der negativen Anwendbarkeitsbedingungen

Die negative Anwendbarkeitsbedingung (NAC), deren Metamodell in Abbildung 6.4 gezeigt ist, modelliert ein Graphmuster, das keinesfalls auftreten darf, sobald die Regel angewendet wird.

Die NAC ist analog zu dem bereits besprochenen Graphmuster aufgebaut, beinhaltet

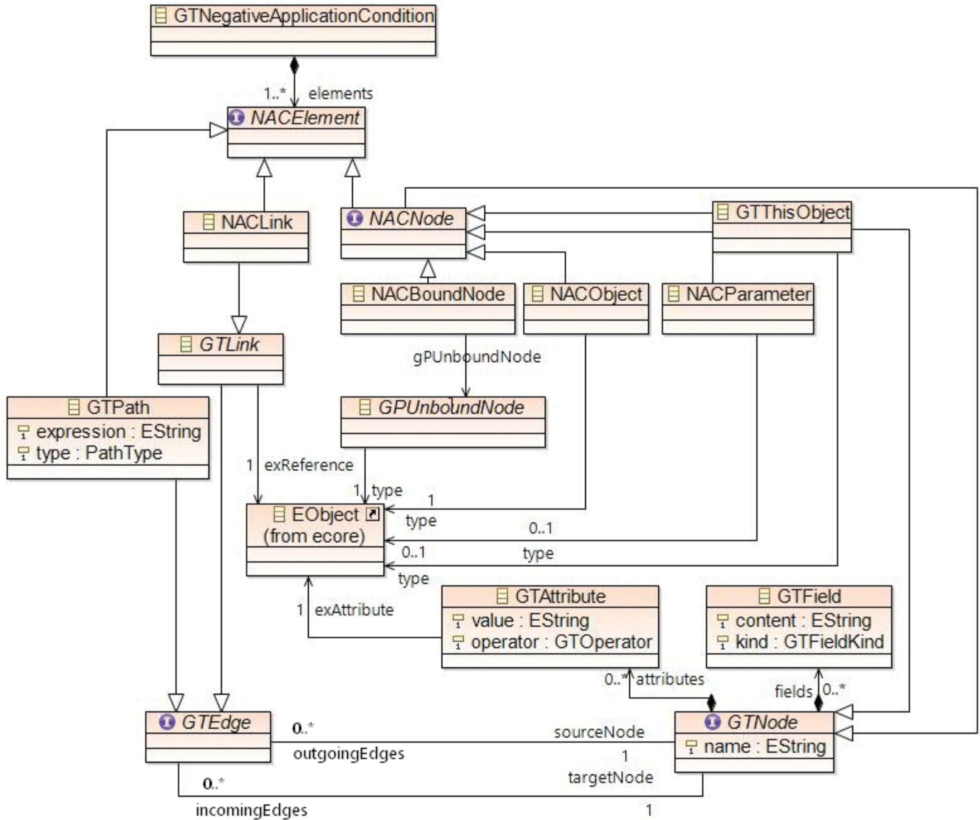


Abbildung 6.4: *Ausschnitt aus dem Metamodell für Graphtransformationen: Aufbau der negativen Anwendbarkeitsbedingung*

jedoch keinerlei Änderungsoperationen. Daher treten nur statuslose Knoten und Kanten als Elemente (KlasseNACElement) innerhalb einer NAC auf. Die Klassen, die diese Knoten und Kanten im Metamodell darstellen, sind ebenfalls Unterklassen von GTNode (Knoten) und GTEdge (Kanten).

Eine NAC beinhaltet ebenfalls mindestens einen gebundenen Knoten. Als gebundene Knoten sind hier das aktuelle Objekt (Klasse GTThisObject) und die nicht-primitiven Übergabeparameter der Methode (Klasse NACParameter) zulässig. Hinzu kommen die gebundenen Knoten der NAC, die durch die Klasse NACBoundNode modelliert werden. Diese stehen für Knoten, die bereits im Graphmuster als ungebundene Knoten vorkommen. Sie referenzieren diese gleichzeitig (Referenz gPUnboundNode). Diese Möglichkeit besteht, da besagte Knoten zum Zeitpunkt des Prüfens der negativen Anwendbarkeitsbedingung bereits gebunden sind, wie in Abschnitt 6.2.2 genauer erklärt werden wird.

Ungebundene, einfache Objekte (Klasse NACObject) sind innerhalb der negativen Anwendbarkeitsbedingung ebenfalls zulässig. NACs lassen, da sie Verbote modellieren, keine mehrwertigen Knoten zu. Grund dafür ist, dass bereits die Verletzung des Verbots durch einen Knoten aus einer Menge zum Erfolg der NAC und somit zur Nicht-

Anwendbarkeit der Regel führt.

Alle Knoten innerhalb einer negativen Anwendbarkeitsbedingung können durch Links (Klasse NACLink) und Pfade (GTPath) verbunden werden. Links stellen Instanzen von Referenzen dar. Weiterhin dürfen Knoten der NAC nicht verändert werden, jedoch dürfen Bedingungen - analog zu den im vorangegangenen Abschnitt beschriebenen Bedingungen - an sie gestellt werden.

Die kontextsensitive Syntax

Die kontextsensitive Syntax der hier erstellten Sprache kommt in der Validierung der Graphtransformationsregeln zum Ausdruck. Dabei werden Bedingungen gestellt, die gegen das Ecore-Metamodell für Graphtransformationen aus dem vorangegangenen Abschnitt definiert werden. Da dieses Graphtransformations-Metamodell das zur Regel gehörende Ecore-Modell referenziert (siehe Abbildung 6.1.1, Referenzen auf EObject), müssen zudem Bedingungen gegen das Ecore-Modell definiert werden. Weiterhin muss die willkürliche Kombination von statusbehafteten Elementen im Graphmuster vermieden werden. Alle gestellten Bedingungen müssen von den einzelnen Graphtransformationsregeln eingehalten werden.

Die Bedingungen gegen das Graphtransformations-Metamodell und das aus der Regel referenzierte Ecore-Modell werden in der Sprache Check verfasst. Beispiele für die konkrete Formulierung solcher Checks finden sich in Anhang A.2. Check ist Teil von OpenArchitectureWare und erlaubt es, deklarativ Bedingungen zu spezifizieren [72]. Diese Vorgaben dürfen - wie in den Abschnitten 7.2 und 7.3.2 gezeigt werden wird - sowohl im Editor als auch während des Build-Prozesses aufgerufen werden. Sie werden im Folgenden aufgelistet und - sofern notwendig - kurz erklärt.

Bedingungen an das Graphtransformations-Metamodell

Für jede Instanz des Graphtransformations-Metamodells gelten die folgenden Vorgaben:

- Vor- und Nachbedingungen, welche in OCL verfasst sind, werden auf ihre syntaktische Korrektheit geprüft. Hierzu wird der OCL-Analysator von Eclipse verwendet (`org.eclipse.ocl.parser.OCLAnalyzer`).
- Jedes Graphmuster und jede NAC müssen mindestens je einen gebundenen Knoten enthalten, der das aktuelle Objekt oder einen nicht-primitiven Parameter der Methode darstellt. Bei NACs sind NACBoundNodes ebenfalls möglich.
- Knoten innerhalb der Regel müssen eindeutig benannt sein.
- Alle ungebundenen Knoten müssen von einem gebundenen Knoten aus erreichbar sein. Daher muss zu jedem ungebundenen Knoten ein Pfad (siehe Abschnitt 2.6) zwischen ihm und einem gebundenen Knoten existieren, der über beliebig viele andere Knoten führen darf. Dabei ist jedoch zu beachten, dass obligatorische Knoten nicht über optionale navigiert werden dürfen.

- Knoten, die in einer Regel einen Parameter repräsentieren dürfen nicht erzeugt werden. Mehrwertige Knoten ebenfalls nicht.

- Knoten die mit „this“ benannt werden, repräsentieren das aktuelle Objekt und sind vom Typ GTThisObject. Kein anderer Knoten darf diesen Namen tragen.

- Attribute dürfen innerhalb eines Knotens beliebig oft als Bedingung und einmal zur Wertzuweisung vorkommen.

- Attributen, die genutzt werden, um Bedingungen an einen Knoten zu formulieren, muss ein gültiger Operator zugewiesen werden. Dabei ist der Typ des Attributs entscheidend:
  - Zahlenwerte erlauben alle Vergleichsoperationen.
  - Attribute, die einen Wahrheitswert enthalten, können nicht kleiner oder größer als der andere Wert sein; gleich bzw. ungleich sind hier möglich.
  - Attribute, die Zeichenketten enthalten werden bei kleiner und größer Vergleichen gemäß der Anzahl ihrer Zeichen verglichen. Zur Äquivalenzprüfung findet ein Objektvergleich durch die `equals`-Methode statt.
  - Alle weiteren Datentypen, die auf Objekte zurückzuführen sind, z.B. EJavaObject oder EDate, erlauben Vergleiche auf (Un-)Gleichheit. Hierzu wird die - in jedem Objekt vorhandene - `equals`-Methode genutzt.

- Felder, die einen Operationsaufruf modellieren, müssen eine Operation der Klasse aufrufen, über die der sie enthaltende Knoten typisiert ist.

- Ein Knoten, der das aktuelle Objekt referenziert, wird unveränderbar mit this benannt.

- Zu löschende Knoten dürfen keine Zuweisungen enthalten.

- An zu erstellende Knoten dürfen keine Bedingungen gestellt werden.

- Knoten innerhalb einer NAC dürfen keinesfalls Zuweisungen enthalten.

- Quell- und Zielknoten einer Kante müssen zum selben Graphmuster / zur selben NAC gehören.

- Kanten dürfen keinesfalls zwei mehrwertige Objekte verbinden, da hier keine eindeutige Zuordnung der Objekte, die durch den Link verbunden werden, gewährleistet werden kann.

- Zu erhaltende Links dürfen nur zwischen zu erhaltenden Knoten (inklusive dem aktuellen Objekt) gezogen werden.

- Zu löschende Links dürfen nicht mit einem zu erzeugenden Knoten in Verbindung stehen.

- Analog hierzu dürfen zu erzeugende Links keinesfalls mit zu löschenden Knoten in Verbindung stehen.
- Attribute innerhalb zu erstellender Knoten werden immer mit dem Zuweisungsoperator initialisiert.
- Attribute innerhalb zu löschender Knoten dürfen nur als Bedingungen an diesen genutzt werden.

Bedingungen im Zusammenhang mit dem referenzierten Ecore-Modell

Zudem werden die folgenden, mit dem referenzierten Ecore-Modell zusammenhängenden Bedingungen geprüft:

- Hat die implementierte Operation einen Rückgabeparameter, der ein Objekt repräsentiert, so muss einer der Knoten entsprechenden Typs (oder einer der Unterklassen des Typs) im Graphmuster als Rückgabewert der Operation gekennzeichnet sein. Gibt die Methode einen Menge an Objekten zurück, so kann dies durch Markierung eines oder mehrerer typkonformer, ein- oder mehrwertiger Objekte geschehen.
- Jeder Link in einem Graphmuster oder einer NAC muss auf eine Referenz im Modell verweisen.
- Ein Link muss über eine Referenz typisiert sein, welche die Klassen bzw. deren Oberklassen verbindet, auf die sein Quell- und Zielknoten verweisen.
- Es dürfen keinesfalls zwei Links vorkommen, die auf dieselbe Referenz verweisen und gleichzeitig dieselben Knoten verbinden.
- Ein Link, der auf eine einwertige Referenz im Modell verweist, darf keinesfalls in einem mehrwertigen Knoten enden.
- Ein geordneter Link muss auf eine mehrwertige Referenz verweisen.
- Die Anzahl der auf die gleiche Referenz zeigenden Links (innerhalb eines Graphmusters oder einer NAC) darf die maximale Multiplizität dieser Referenz nicht überschreiten. Im Graphmuster müssen bei dieser Rechnung allerdings die zu löschenden Links mit den zu erzeugenden Links verrechnet werden.
- Eben genannte Bedingung gilt auch für das Überschreiten der maximalen Multiplizität der entgegengesetzten Referenz.
- Jeder ungebundene Knoten einer Regel muss über eine Klasse im referenzierten Modell typisiert sein. Ist ein Knoten zu erzeugen, darf diese Klasse nicht abstrakt sein. In jedem Fall darf die zur Typisierung verwendete Klasse keinesfalls als Interface deklariert sein.[5]

[5] Letztere Einschränkung ist notwendig, da in Ecore Klassen und Interfaces durch Instanzen der Klasse EClass des Ecore-Metamodells modelliert werden.

- Es gilt das Prinzip der Exklusivität: Jeder Knoten einer Regel darf nur eine eingehende Kante aufweisen, die auf eine Referenz verweist, welche eine Enthaltensbeziehung darstellt. Dabei dürfen Objekte im Graphmuster ihren Behälter wechseln. Dies entspricht einer zu löschenden und einer zu erhaltenden Enthaltenseinsrelation, die jeweils durch einen Link repräsentiert werden.
- Knoten, die in einer Regel einen Parameter repräsentieren, dürfen ausschließlich nicht-primitive Parameter der Methode referenzieren.
- Treten Attribute innerhalb von Knoten auf, muss das zugehörige Attribut im Ecore-Modell in der Klasse, die dem Typ des Knotens entspricht, oder einer ihrer Oberklassen deklariert sein.
- Attributwerte von primitiven Attributen des Typs `String` müssen in Anführungszeichen gesetzt werden.

6.1.2 Die konkrete Syntax

Die konkrete Syntax bildet einen weiteren Bestandteil der Sprachdefinition. Sie wird in diesem Abschnitt anhand von Beispielen erklärt. Hierbei wird auch die lexikalische Syntax besprochen, da sie in den grafischen Elementen enthalten ist.

Die Elemente der konkreten Syntax sind in den Abbildungen 6.5 und 6.6 ihren korrespondierenden Metamodellelementen gegenüber gestellt. Die Beispielelemente der Regeln sind über dem Ecore-Metamodell oder dem Graphtransformations-Metamodell typisiert. Sie finden in einer M1-Anwendung der Regeln Verwendung, da sie Modelle verändern.

Die Graphtransformationsregel entspricht der gesamten Fläche, die zur Spezifikation ihrer Komponenten zur Verfügung steht (1 in Abbildung 6.5). Darauf ist das Graphmuster (5) als graue Box, die mit Graph Pattern beschriftet ist, zu finden. Vor- (2) und Nachbedingungen (3) an die Regel werden durch längliche Boxen, die mit OCL- oder Xcore-Ausdrücken befüllt werden, oberhalb und unterhalb des Graphmusters angebracht. Dabei ist es möglich, mehrere dieser Boxen untereinander anzuordnen. Diese werden als konjunktiv verknüpft angesehen. NACs (16) werden neben dem Graphmuster platziert. Auch hier dürfen beliebig viele als konjunktiv verknüpft anzusehende NACs nebeneinander platziert werden. Der Kommentar (4) wird in ein eigenes Fenster innerhalb der Eclipse-Umgebung ausgelagert. Das aktuelle Objekt (6) wird als weißer Knoten, der Möglichkeiten zur Spezifikation von Änderungen und Bedingungen enthält, dargestellt. Im Graphmuster werden die Elemente nach Status farblich markiert und gekennzeichnet: Knoten, die erhalten werden sollen (7-10, jeweils links), sind grau hinterlegt und tragen keine weitere Markierung, zu löschende Knoten (7-10, jeweils rechts) werden rot hinterlegt und mit $--$ markiert, zu erzeugende Knoten (9, Mitte) grün und mit ++. Links, die erhalten werden, sind markierungslose schwarze Pfeile (11, links), zu erzeugende und zu löschende Links werden analog zu den Knoten grün und mit ++ bzw. rot und mit $--$ markiert (11, Mitte und rechts). Geordnete Links (12) werden als grüne und mit ++ markierte Pfeile eingezeichnet. Sie tragen zudem eine Beschriftung, welche

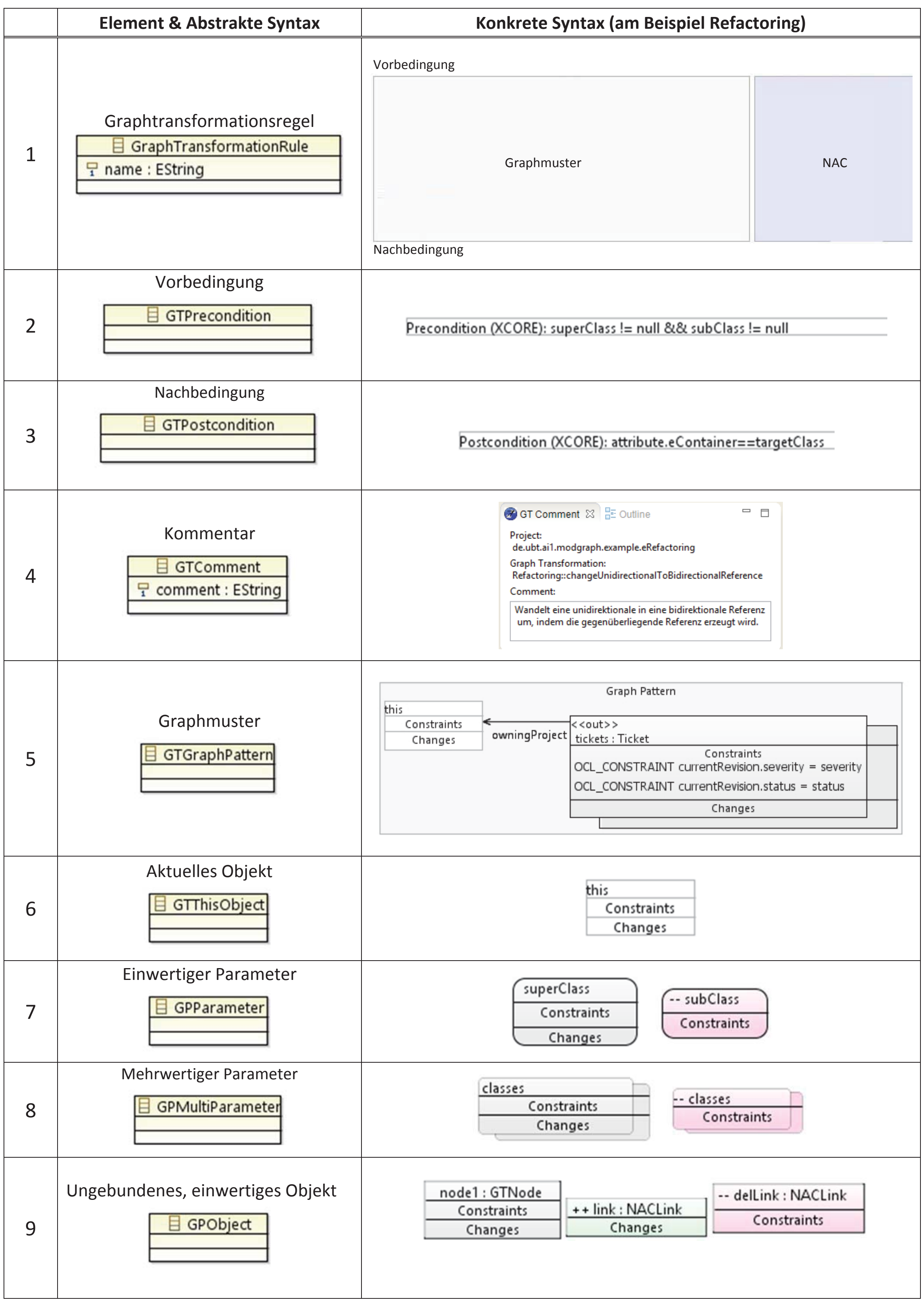
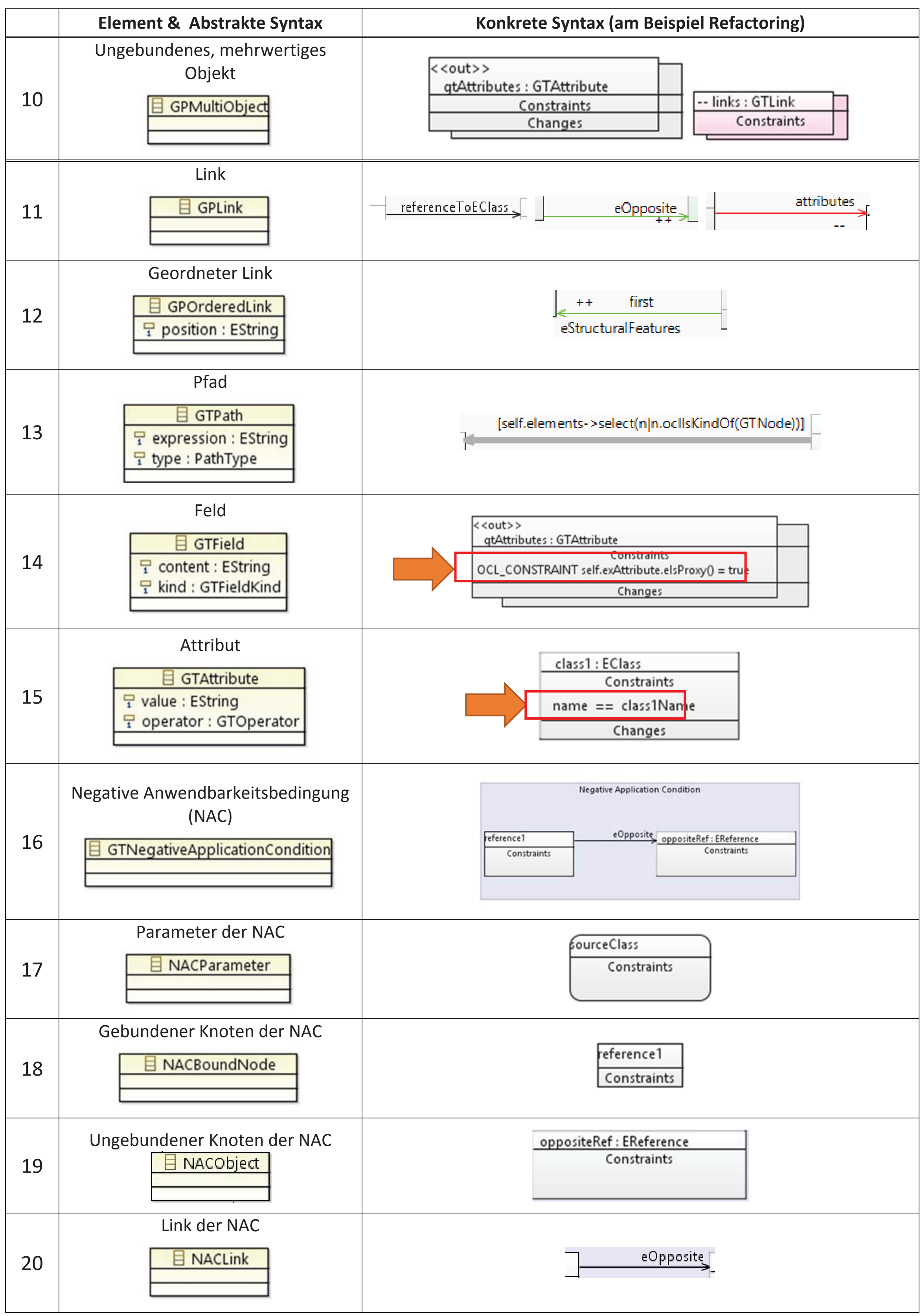


Abbildung 6.5: *Abstrakte Syntax (links) und konkrete Syntax (rechts) auf einen Blick - Teil 1*



	Element & Abstrakte Syntax	Konkrete Syntax (am Beispiel Refactoring)
10	Ungebundenes, mehrwertiges Objekt GPMultiObject	<<out>> gtAttributes : GTAttribute Constraints Changes -- links : GTLink Constraints
11	Link GPLink	referenceToEClass eOpposite ++ attributes --
12	Geordneter Link GPOrderedLink position : EString	++ first eStructuralFeatures
13	Pfad GTPath expression : EString type : PathType	[self.elements->select(n|n.oclIsKindOf(GTNode))]
14	Feld GTField content : EString kind : GTFieldKind	<<out>> gtAttributes : GTAttribute Constraints OCL_CONSTRAINT self.exAttribute.eIsProxy() = true Changes
15	Attribut GTAttribute value : EString operator : GTOperator	class1 : EClass Constraints name == class1Name Changes
16	Negative Anwendbarkeitsbedingung (NAC) GTNegativeApplicationCondition	Negative Application Condition reference1 Constraints eOpposite oppositeRef : EReference Constraints
17	Parameter der NAC NACParameter	sourceClass Constraints
18	Gebundener Knoten der NAC NACBoundNode	reference1 Constraints
19	Ungebundener Knoten der NAC NACObject	oppositeRef : EReference Constraints
20	Link der NAC NACLink	eOpposite

Abbildung 6.6: *Abstrakte Syntax (links) und konkrete Syntax (rechts) auf einen Blick - Teil 2*

die Position des einzufügenden Elements angibt. Alle Links werden darüber hinaus mit dem Namen der von ihnen instanziierten Referenz markiert. Dies gilt auch für Links innerhalb der NAC (20), die schwarz dargestellt werden und wie alle Elemente der NAC keinen Status aufweisen. Pfade (13) werden durch dicke, graue Pfeile repräsentiert, die mit einem in eckigen Klammern stehenden, textuellen Pfadausdruck markiert sind.

Objekte und Parameter werden durch Boxen innerhalb des Graphmusters bzw. einer NAC platziert. Die Parameter der Operation werden durch Boxen mit abgerundeten Ecken gekennzeichnet (7,8,17). Mehrwertige Parameter und Objekte werden durch Boxen mit einer verdoppelten und leicht verschobenen Silhouette grafisch abgebildet (8,10). Statuslose Elemente der NAC werden, wie auch die zu erhaltenden Elemente des Graphmusters, grau hinterlegt (17-19). Möglichkeiten zur Spezifikation von Bedingungen und Änderungen werden, wie in Abschnitt 6.1.1 erläutert, grafisch umgesetzt. Jede Box, die einen Parameter oder ein Objekt repräsentiert, enthält einen Constraints-Abschnitt, in welchem Bedingungen spezifiziert werden, bzw. einen Changes-Abschnitt (7-10, 17-19), der Änderungen definiert. Dabei enthalten alle Elemente der NACs nur Bedingungen (17-19). Bedingungen und Zuweisungen an Attribute werden dabei analog zu Java bzw. Xcore dargestellt (15), Bedingungen, die in OCL oder Xcore spezifiziert sind, werden durch ihre Feldart eingeleitet und beinhalten den entsprechenden Ausdruck (14).

Ihre Beschriftung betreffend, werden Parameter (7,8,17) lediglich mit dem Namen des Parameters gekennzeichnet, Objekte werden mit Namen und Typ, getrennt durch einen Doppelpunkt - wie in der zur in der Modellierung oft verwendeten UML-Notation - markiert. Parameter, die mit einem Typ markiert sind, werden typumgewandelt. Die gebundenen Knoten der NAC (18) sind lediglich mit dem Namen des von ihnen referenzierten Knotens aus dem Graphmuster beschriftet. Die ungebundenen Knoten der NAC (19) werden ebenfalls mit Namen und Typ, getrennt durch einen Doppelpunkt beschriftet.

Entspricht ein Parameter oder ein Objekt im Graphmuster dem Rückgabewert der Operation, wird er mit << *out* >> gekennzeichnet (10, links und 14). Analog kann ein Objekt mit << *optional* >> gekennzeichnet werden.

Damit ist die Syntax der Sprache definiert.

6.2 Dynamische Semantik der Sprache für Graphtransformationsregeln

Die Ausführbarkeit einer Sprache und damit des Modells erfordert die Definition einer dynamischen Semantik. Sie definiert, wie in Abschnitt 2.3 beschrieben, das Verhalten des Modells - und des damit erstellten Programms - zur Laufzeit. Die dynamische Semantik der Sprache für Graphtransformationsregeln wird im Folgenden dargestellt und ist in [21], [78], [80] und [81] veröffentlicht. Dazu wird zunächst ein Überblick der Ausführungslogik einer Regel gegeben, indem klargestellt wird, welche Elemente der Regel in welcher Reihenfolge ausgeführt werden. Ausgehend davon werden die komplexeren, zur Regelausführung nötigen, Schritte näher erläutert.

6.2.1 Ausführungslogik einer Graphtransformationsregel

Jeder Aufruf einer Operation, die durch eine ModGraph-Regel implementiert ist, folgt einem bestimmten Schema. Dieses ist in der Ausführungslogik der Regel begründet und für alle Regeln gleich. Die Ausführungslogik ist unabhängig von der Zielsprache der Regel. Es ist demnach egal, ob die Regel letztendlich in Java-Quelltext übersetzt oder in ein Xcore-Modell eingebettet wird.

Die Ausführungslogik soll hier näher betrachtet werden. Sie ist in einer flussdiagrammartigen Schreibweise in Abbildung 6.7 dargestellt. Anders als bei Flussdiagrammen wurden alle Schritte, die zum Scheitern der Regelausführung führen können, mit einem roten Blitz markiert. Zudem sind die Elemente farblich hinterlegt. Die dadurch markierten Bereiche entsprechen den Nummern der folgenden Aufzählung, welche die Ausführung einer Regel beschreibt.

1. Zuerst werden alle Vorbedingungen überprüft. Schlägt eine Vorbedingung fehl, schlägt die Regelausführung fehl.

2. Sind alle Vorbedingungen erfüllt, findet die Mustersuche - die Suche nach einer Übereinstimmung des Graphmusters der Regel in der betrachteten Modellinstanz - statt. Sie beginnt mit den nicht-primitiven Übergabeparametern der Regel und ihrem aktuellen Objekt. Sie wird in Abschnitt 6.2.2 genauer erklärt. Wird keine Übereinstimmung gefunden, so scheitert die Ausführung der Regel ebenfalls.

3. Ist eine Übereinstimmung gefunden und mindestens eine NAC innerhalb der Regel vorhanden, so werden die NACs geprüft. Dazu wird die Mustersuche auf den NACs ausgeführt. Findet sich eine Übereinstimmung einer NAC in der betrachteten Modellinstanz, beginnt die Mustersuche erneut wie in Punkt 2. beschrieben.

4. Bei erfolgreicher Mustersuche werden alle in der Regel spezifizierten Änderungen an den gefundenen Objekten ausgeführt. Dazu zählt auch die Ausführung der aufzurufenden Operationen (in zufälliger Reihenfolge). Wie dies genau vonstatten geht, wird in Abschnitt 6.2.3 genauer betrachtet.

5. Zuletzt werden die Nachbedingungen geprüft. Schlägt eine Nachbedingung fehl, ist die Ausführung der Regel auch fehlgeschlagen. Ist sie erfolgreich, wird die Ausführung abgeschlossen. Ist der Regel ein Rückgabewert zugeordnet, wird dieser zurückgegeben.

Nach einer erfolgreichen Ausführung der Regel sind alle definierten Änderungen vorgenommen. Die Ausführung des Modells darf wie gewünscht fortgesetzt werden. Schlägt die Regelausführung an einer Stelle fehl, so wird ihre Ausführung abgebrochen und eine Ausnahme ausgelöst. Hierbei macht es einen Unterschied, an welcher Stelle die Regel scheitert. Schlägt das Prüfen einer Vorbedingung oder das Finden des Graphmusters unter Berücksichtigung der vorhandenen NACs fehl, ist die Regel nicht anwendbar und das Modell bleibt unverändert. Schlägt jedoch eine Nachbedingung fehl, sind die Änderungen am Modell bereits vorgenommen worden und das Modell befindet sich in

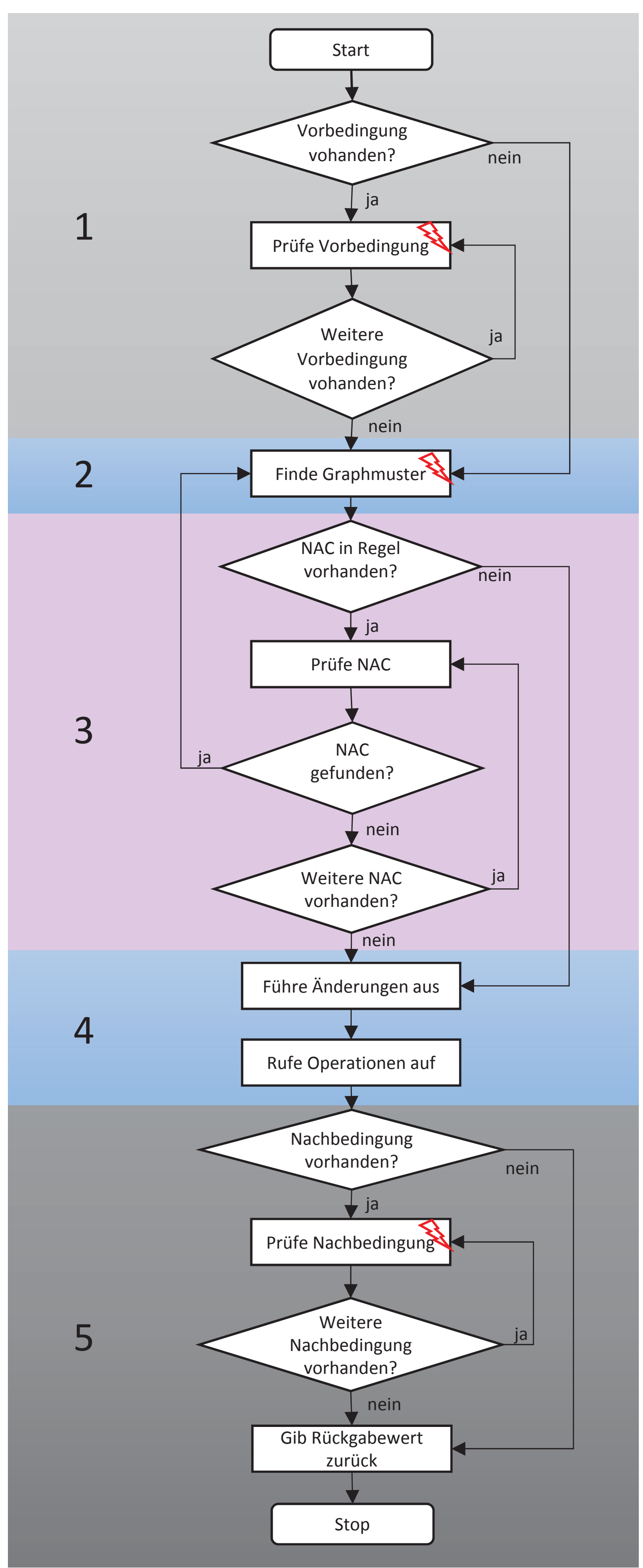


Abbildung 6.7: *Ausführungslogik einer ModGraph-Regel*

einem inkonsistenten Zustand. Dieser muss vom Nutzer rückgängig gemacht werden, da im allgemeinen keine Möglichkeit besteht, eine Regel invers anzuwenden. Hat der Nutzer viele Nachbedingungen, so wird an dieser Stelle geraten, Reparaturregeln zu definieren, die im Falle des Scheiterns einer Regel ausgeführt werden können. Die Steuerung dieses Vorganges übernimmt zum Beispiel der Xcore-Kontrollfluss durch einen geeignete Ausnahmebehandlung.

6.2.2 Mustersuche

Ein wichtiger und komplexer Teil der Ausführungslogik einer Regel ist die Mustersuche. Es handelt sich hierbei um eine lokalisierte Suche. Sie nutzt die durch Knoten in der Regel repräsentierten und zur Laufzeit bekannten Objekte. Dies sind die Parameter der Operation oder das aktuelle Objekt. Sie dienen als Ankerknoten. Die Suche lokalisiert mit ihrer Hilfe die bislang unbekannten Objekte. Sie wird verwendet, um das Graphmuster und die NACs auf die Modellinstanz abzubilden und kommt daher in den Schritten 2 und 3 der im vorangegangenen beschriebenen Ausführungslogik einer Regel zum Einsatz (siehe auch Abbildung 6.7).

Die Algorithmen zur Mustersuche sind in ModGraph, wie in allen regelbasierten Sprachen, bereits implementiert. Der Nutzer spezifiziert lediglich das geforderte Muster sowie die verbotenen Muster als Graphmuster und NACs.[6] Die Mustersuche selbst entspricht einer Teilgraphensuche auf der EMF-Instanz. Sie wird benötigt, um die ungebundenen Knoten der Regel zu binden. Die Teilgraphensuche basiert auf der Navigierbarkeit durch die Graphen der Graphtransformationsregel. Dies bedeutet, dass alle ungebundenen Knoten der Regel (in-)direkt von einem gebundenen Knoten aus über Links erreichbar sein müssen. Die Teilgraphensuche ist in ModGraph als heuristischer Greedy-Algorithmus implementiert. Die durch die Suche implementierte Abbildung der Knoten auf die Objekte ist nicht injektiv. Zwei Knoten gleichen Status könnten auf das selbe Objekt abgebildet werden. Die Abbildung ein und desselben Objekts auf einen zu löschenden und einen zu erhaltenden Knoten wird während der Mustersuche unterbunden. Dies geschieht, indem zur Laufzeit der Mustersuche eine Liste der zu löschenden Objekte verwaltet wird und bei jedem Match eines zu erhaltenden Knotens geprüft wird, ob das Objekt bereits auf einen zu löschenden Knoten abgebildet wurde. Injektivität kann jedoch durch das Setzen entsprechender Bedingungen an die Knoten erreicht werden. Dies ist die übliche und empfohlene Vorgehensweise, um Zweideutigkeiten bei der Regelausführung zu vermeiden. Letzlich kann der Nutzer auf diese Weise selbst entscheiden, ob er injektives Matching erreichen möchte. Ein zusammenhängender Graph ist ebenfalls nicht nötig, jedoch werden Zusammenhangskomponenten mit jeweils mindestens einem gebundenen Knoten gefordert.

Die Ausführung der Mustersuche in ModGraph ist zweigeteilt. Zur Übersetzungszeit wird in einem vorbereitenden Schritt ein Spannwald aus der Regel aufgebaut, der als Suchplan für die eigentliche Suche dient. Auf diesem aufbauend wird die Regel auf eine

[6] Dies verdeutlicht nochmals den Abstraktionsgewinn durch die Nutzung einer regelbasierten Sprache gegenüber einer prozeduralen Sprache.

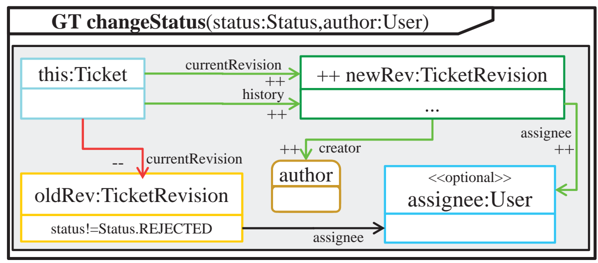


Abbildung 6.8: *Zur Erklärung der Mustersuche farbkodiertes Anwendungsbeispiel* changeStatus *aus dem Bugtacker-Beispiel*

Modellinstanz abgebildet. Der Algorithmus zur Mustersuche wird bei jeder Anwendung einer Regel ausgeführt. Er wird im generierten Code umgesetzt und demnach - je nach Zielsprache bei der Generierung der Regel - durch Java-Methoden oder Xbase-Ausdrücke ausgeführt. Der Aufbau der Spannbäume und der Algorithmus zur Abbildung der Regel auf die Modellinstanz werden im Folgenden erklärt und sind ebenso in [78] und [81] veröffentlicht.

Als Anwendungsbeispiel wird die in Abbildung 6.8 schematisiert dargestellte, bereits aus Kapitel 5.4 bekannte Regel zur Änderung des Status eines Tickets innerhalb eines Bugtrackers näher betrachtet. Zur besseren Verständlichkeit des weiteren Vorgehens sind die Knoten farbig umrandet.

Aufbau der Spannwälder

Ausgangspunkt der Suche sind die bereits gebundenen Knoten der Regel. Betrachtet man zunächst das Graphmuster in Abbildung 6.8, sind dies der Knoten für das aktuelle Objekt (this) und derjenige für den Parameter author, welcher ein nicht-primitiver Übergabeparameter der Operation changeStatus im Bugtracker-Beispiel ist. Die Suche nach den ungebundenen Knoten, im Beispiel-Graphmuster oldRev:Ticketrevision und assignee:User, startet an den gebundenen Knoten. Deswegen ist es nötig, dass alle Knoten der Regel von den gebundenen Knoten aus direkt oder indirekt über Links und Pfade erreichbar sind und damit die nötigen Zusammenhangskomponenten bilden. Dabei ist es bei bidirektionalen Referenzen unwichtig, ob der eingezeichnete gerichtete Link die Hin- oder die Rückrichtung der Referenz instanziiert. Sind die Zusammenhangskomponenten nicht gegeben, ist die Regel nicht ausführbar. Bei der Suche von ungebundenen Knoten, ausgehend von gebundenen, spielt die Reihenfolge in der gesucht wird, eine tragende Rolle. Ein obligatorischer Knoten darf niemals von einem optionalen Knoten aus gesucht werden. Hingegen dürfen ein- und mehrwertige Knoten beliebig in den Spannwald eingefügt werden (unter Beachtung der eben genannten Regel zu optionalen Knoten). Außerdem wird ein Link, der eine einwertige Referenz instanziiert, einem anderen Link, der eine mehrwertige Referenz instanziiert, vorgezogen. Zudem werden Pfade im Graphmuster oder der NAC als sehr teuer angenommen, da zu ihrer Auswertung ein Pfadausdruck evaluiert werden muss. Damit werden kombinatorisch explodierende Kosten für die Teilgraphensuche (soweit möglich) vermieden.

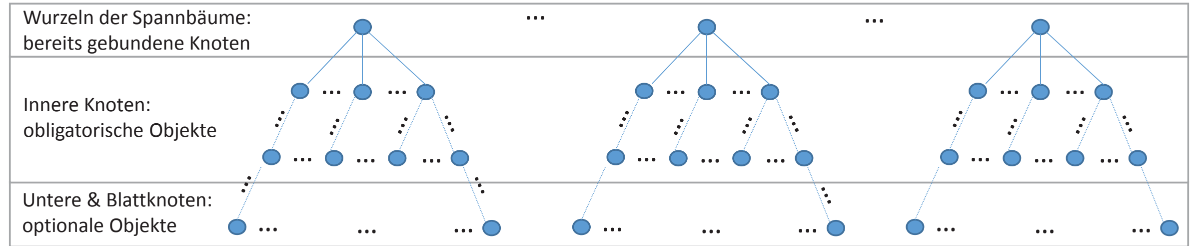


Abbildung 6.9: *Allgemeine Vorgehensweise bei der Mustersuche - Aufbau des Spannwaldes*

Im Beispiel ist der Knoten oldRev:Ticketrevision direkt durch Navigation des Links currentRevision vom Knoten für das aktuelle Objekt (this) navigierbar. Da dieser Link eine einwertige Referenz instanziiert, wird der Knoten mit minimalem Aufwand erreicht. Der optionale Knoten asignee:User steht nur indirekt mit dem das aktuelle Objekt repräsentierenden Knoten this in Verbindung. Er wird durch Navigation des Pfades im Graphen von this ausgehend via currentRevision und assignee erreicht. Der zu erzeugende Knoten newRev:TicketRevision spielt für die Mustersuche keine Rolle.

Das Beispiel bietet nur einen Weg zur Navigation der Knoten. Die Navigationsmöglichkeiten steigen allerdings mit wachsender Komplexität der Regel. Knoten können unter Umständen direkt oder indirekt durch mehrere Links, die für Instanzen von Referenzen unterschiedlicher Wertigkeit stehen, verbunden werden. So kann ein einzelner, ungebundener Knoten über mehrere Pfade von einem gebundenen Knoten aus erreichbar sein. Damit hier keine willkürliche Auswahl des Links oder Pfades stattfindet, die unter Umständen zu enormen Laufzeiten führt, wird ein Verfahren bereitgestellt, die Navigation durch eine Instanz möglichst effizient zu gestalten. Der ModGraph-Algorithmus für die Mustersuche baut zu diesem Zweck Spannwälder aus den in der Regel gegebenen Teilgraphen auf. Dabei entsteht mindestens der Spannwald für das Graphmuster. Beinhaltet die Regel NACs, so wird für jede NAC ein eigener Spannwald erstellt. Ein Beispiel für einen solchen Spannwald ist in Abbildung 6.9 schematisch gezeigt. Er besteht aus Spannbäumen unbeschränkter Ordnung, die wiederum aus Knoten der Regel aufgebaut sind. Jeder Knoten, der in den Spannwald eingefügt wurde, gilt als navigierbar. Damit dürfen von ihm ausgehend weitere Knoten gesucht werden. Der Algorithmus nutzt zunächst alle gebundenen, ein- und mehrwertigen Knoten und betrachtet diese als navigierbar. Sie bilden die Wurzeln der Bäume im Spannwald. Im nächsten Schritt werden die von allen navigierbaren Knoten direkt über Kanten des Graphen erreichbaren, ungebundenen, ein- oder mehrwertigen, obligatorischen Knoten betrachtet. Dabei wird derjenige Knoten in den Spannwald eingefügt, der über die Kante mit den geringsten Kosten erreichbar ist. Der Knoten ist jetzt navigierbar. Muss ein einwertiger Knoten von einem mehrwertigen aus gesucht werden, so werden alle Objekte, die dem mehrwertigen Knoten zugeordnet sind geprüft. Referenzieren alle das gesuchte einwertige Objekt, wird dieses in den Baum eingefügt. Dieses Vorgehen wird zunächst solange wiederholt, bis alle obligatorischen Knoten navigierbar sind. Sind optionale Knoten vorhanden, werden diese im nächsten Schritt nach demselben Vorgehen in den bereits entstandenen Spannwald eingefügt. Sind keine optionalen Knoten vorhanden, sind obligatorische Knoten die

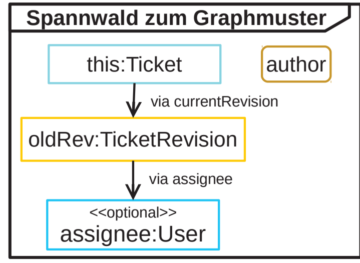


Abbildung 6.10: *Spannwald zum Anwendungsbeispiel*

Blätter des Baumes.

Der entstandene Spannwald zur betrachteten Regel ist in Abbildung 6.10 dargestellt. Die Knoten sind zur besseren Übersicht gleichfarbig markiert. Es handelt sich um einen einfachen Spannwald, der die beiden gebundenen Knoten this und author als Wurzeln nutzt. Die ungebundenen Knoten werden nacheinander in den Spannwald eingefügt. Der Knoten oldRev:TicketRevision wird direkt vom Knoten this erreicht. Der Knoten assignee:User wird sowohl durch die Navigierbarkeit als auch durch seine Optionalität auf Blattebene in den Baum eingefügt.

Abbildung auf die Instanz

Die so entstandenen Spannwälder bilden die Grundlage der zur Mustersuche verwendeten Vorgehensweisen. Sie werden im Folgenden als Suchpläne genutzt. Hier beginnt der zweite Teil der Mustersuche, welcher die Regel möglichst effizient auf die Instanz abbildet. Die Vorgehensweise ist unabhängig von der genauen Implementierung. Sie wird jedoch später durch generierte Java-Methoden bzw. Xbase-Ausdrücke, deren genaue Ausprägung in Abschnitt 7.2 besprochen wird, beschrieben. Die unabhängige Vorgehensweise wird nun erläutert. Dazu wird dieser Teil der Mustersuche auf Beispielinstanzen des Bugtrackers aus Kapitel 5.4 angewendet, indem die betrachtete Regel zur Änderung des Status eines Tickets ausgeführt wird. Ihr eben erstellter Spannwald ist die Grundlage für die weitere Vorgehensweise. Abbildung 6.11 zeigt diesen, die Regel und die im Weiteren betrachteten Instanzen. Zur Veranschaulichung der Suche sind hier alle beteiligten Knoten und Objekte farbig markiert, wobei zusammengehörige Knoten und Objekte dieselbe Farbe tragen.

Im Allgemeinen werden zuerst die Parameterknoten auf die Aktualparameter abgebildet. Im nächsten Schritt wird so lange in der Instanz gesucht, bis ein durch die Umsetzung des Spannwaldes vorgegebenes, erstes, ungebundenes, ein- oder mehrwertiges Objekt gefunden wird. In der Regel ist dieses das auf kürzestem Wege vom aktuellen Objekt aus erreichbare. Die ungebundenen Knoten in den Bäumen werden so sukzessive auf die Instanz abgebildet. Dabei kann es Vorkommen, dass zwei Knoten gleichen Status auf dasselbe Objekt abgebildet werden. Die Ausführung der Änderungen an diesem Objekt und dessen zugehörigen Referenzen folgt dann der Erstellungsreihenfolge der Knoten auf die es abgebildet wird.

Der Regel im Beispiel sind die zwei Übergabeparameter der implementierten Operati-

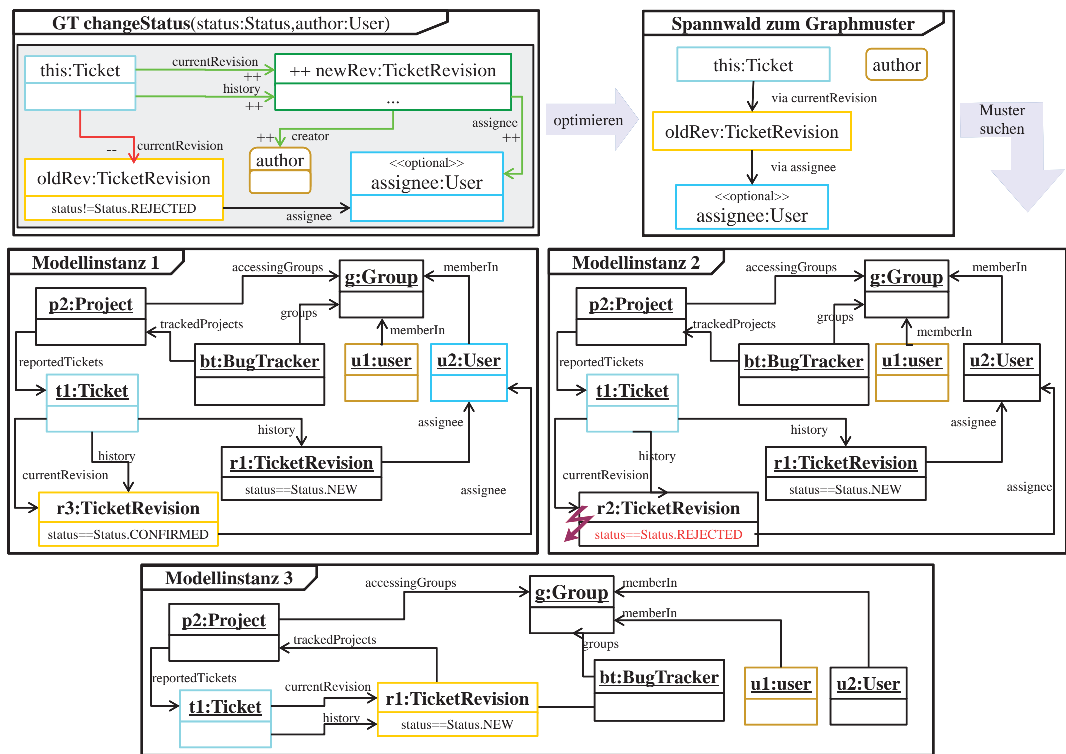


Abbildung 6.11: *Mustersuche in unterschiedlichen Modellinstanzen am Anwendungsbeispiel*

on, der aktualisierte Status und der Autor der Änderung, zugeordnet. Diese sind bereits bei Aufruf der Regel durch die Aktualparameter der Operation bekannt. Die in der Regel enthaltenen Graphmuster werden auf drei verschiedene Modellinstanzen abgebildet, Modellinstanz 1, Modellinstanz 2 und Modellinstanz 3. Die Modellinstanzen zeigen jeweils einen Teil einer Instanz des Bugtracker-Modells, welches in Abschnitt 5.4 erklärt wurde.

Betrachtet man zunächst Modellinstanz 1 in Abbildung 6.11 mitte links, so wird eine Übereinstimmung für das Graphmuster gefunden. Der Parameter-Knoten author der Regel ist bereits an den Benutzer u1 (Aktualparameter der Operation) gebunden. Der Knoten für das aktuelle Objekt ist an die die Operation aufrufende Instanz der Ticket-Klasse t1 gebunden. Damit kann der Knoten oldRev der Revision r3 in Modellinstanz 1 zugeordnet werden, da diese über den Link currentRevision von t1 aus navigierbar ist. Der optionale Knoten assignee:User wird in Modellinstanz 1 auf das Objekt u2 abgebildet, da er über den Link asignee von oldRev erreichbar ist. Modellinstanz 1 zeigt demnach eine erfolgreiche Abbildung der Regel.

Auf Modellinstanz 2 (in Abbildung 6.11 mitte rechts dargestellt) kann die Regel nicht angewendet werden, da die Mustersuche fehlschlägt. Die gebundenen Knoten werden zunächst wie bei Modellinstanz 1 abgebildet. Das Problem entsteht bei der Abbildung des Knotens oldRev. Die aktuelle Revision des Tickets r2 der Instanz wurde zurückgewiesen (Status.REJECTED). Dieser Fall wurde im Graphmuster durch eine Attributbedingung

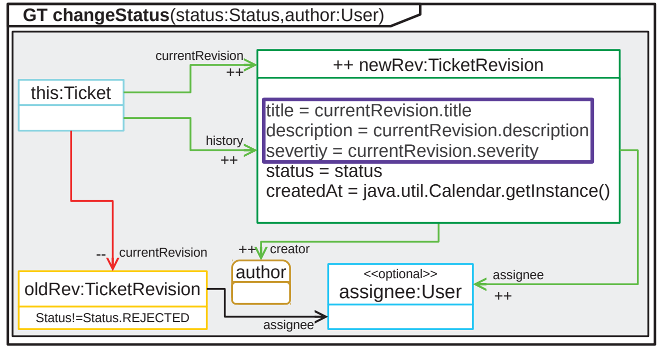


Abbildung 6.12: *Zustandsabhängige Berechnung der Attribute am Anwendungsbeispiel*

ausgeschlossen. Dementsprechend kann der Knoten nicht auf das Objekt abgebildet werden. Es steht auch kein alternatives Objekt zu Verfügung, da die aktuelle Version des Tickets eindeutig durch Navigation des Links currentRevision bestimmt wird, der eine einwertige Referenz instanziiert. Somit scheitert die Regel. Dies ist in Abbildung 6.11 durch eine rote Markierung an der entsprechenden Stelle gekennzeichnet.

Modellinstanz 3, die in Abbildung 6.11 unten dargestellt ist, bietet eine weitere erfolgreiche Abbildung der Regel. Die gebundenen Knoten werden analog zu den vorangegangenen Modellinstanzen abgebildet. Der Knoten oldRev wird durch Navigation des zugehörigen Links currentRevision dem - die Klasse TicketRevision instanziierenden - Objekt r1 zugeordnet. Der optionale Knoten wird hier nicht gefunden. Dieses gänzliche Fehlen eines Objekts für den optionalen Knoten assignee und damit auch der zugehörigen Links in Modellinstanz 3 stellt den bedeutenden Unterschied zu Instanz 1 dar. Dennoch ist in beiden Fällen die Mustersuche erfolgreich. Durch dieses Fehlen der optionalen Elemente ist Modellinstanz 3 eine minimal gültige Übereinstimmung der Regel.

6.2.3 Anwendung der modellierten Änderungen

Nachdem das Graphmuster erfolgreich und eventuell vorhandene NACs nicht erfolgreich auf die Modell-Instanz abgebildet wurden, werden die in der Regel spezifizierten Änderungen vorgenommen. Diese sind in der Ausführungslogik der Graphtransformationsregel in Abschnitt 6.2.1 und in Abbildung 6.7 als Punkt 4 aufgeführt. Dieser bedarf einer ausführlichen Erläuterung, da die Änderungen am Modell nicht in willkürlicher Reihenfolge vorgenommen werden dürfen. Beispielsweise kann sich ein Attributwert aus einem Wert eines anderen Elements, wie eines anderen Attributs, berechnen, das in der Regel ebenfalls verändert wird. Abhängig davon, ob das Element oder das Attribut zuerst geändert wird, basiert der berechnete Wert auf dem alten oder neuen Wert des Elements. Um dies zu umgehen, werden die rechten Seiten der Attributzuweisungen, die Attributwerte, im ersten Schritt der Änderungen berechnet. Attributzuweisungen beziehen sich daher immer auf den Vorzustand des Modells.

Betrachtet man nun die Beispielregel changeStatus, betrifft dies die in Abbildung 6.12 violett markierten Attibutzuweisungen title, description und severity der Regel. Sie

übernehmen die Werte ihrer Vorgängerrevision. Würde hier zuerst der Link currentRevision angepasst, würde er auf die neu erzeugte Revision newRev und ihre nicht zugewiesenen Attribute zeigen, statt auf die Revision oldRev. Damit wären die Attributwerte auf die hier modellierte Art nicht mehr berechenbar. Bei näherer Betrachtung könnte argumentiert werden, dass der Ausweg in diesem Spezialfall die Berechnung der Werte nicht über die Referenz, sondern über das bei der Mustersuche gefundene Objekt oldRev ist. Jedoch würde dies zu einer Einschränkung der Möglichkeiten bei der Erstellung der Graphtransformationsregeln führen. Es wäre außerdem keine allgemein gültige Lösung, da im Allgemeinen sowohl der Link als auch der Knoten, durch die auf den Wert zugegriffen werden kann, als zu löschend markiert sein dürfen. Damit könnten beide bereits bei der Wertberechnung gelöscht sein, so dass diese wiederum scheitert.

Im nächsten Schritt erfolgt die Zuweisung der zuvor berechneten Attributwerte an zu bewahrende Objekte.[7] Die zu löschenden Elemente werden gelöscht. Das Löschen eines Objekts ist dabei nicht trivial. Es muss aus seinem gesamten Kontext entfernt werden, damit keinesfalls sogenannte hängende Kanten auftreten. Dies sind Links ohne Quell- oder Zielknoten, die zu einer nicht mehr validen Modellinstanz führen. Zu löschende Links werden gelöscht. Dabei gibt es einen Sonderfall: Links, die einwertige Referenzen instanziieren und im Zuge der spezifizierten Änderungen neu gesetzt werden, werden nicht explizit gelöscht.

Der zu löschende Link currentRevision in der Regel wird vom ModGraph-Generator im Quelltext als Sonderfall (Instanz einer einwertigen Referenz) erkannt, welcher in derselben Regel durch den zu erzeugenden Link gleichen Namens neu gesetzt wird.

Die zu erzeugenden Elemente werden erstellt. Bei Knoten wird ihr Name als Bezeichner des neu erzeugten Objekts verwendet. Den Objekten werden die zuvor berechneten Attributwerte zugewiesen. Analog werden alle in der Regel aus Abbildung 6.12 als neu zu erstellend markierten Links erzeugt. Links, die in optionalen Knoten enden, werden nur erzeugt, wenn das optionale Objekt bei der Mustersuche gefunden wurde.

Für das Beispiel bedeutet dies die Erzeugung der Revision newRev des Tickets. Die Links history und creator werden erzeugt. Der Link assignee wird nur gesetzt, wenn das zum Knoten passende Objekt gefunden wurde. Für die in Abbildung 6.11 dargestellten Modellinstanzen bedeutet dies, dass der Link für Modellinstanz 1 erzeugt wird, da der optionale Knoten auf das Objekt u2 abgebildet wurde. Für Modellinstanz 3 wird kein Link erzeugt, da hier dem optionalen Knoten kein Objekt zugewiesen wurde. Bei der Erzeugung des Links currentRevision wird der bestehende Link überschrieben. Damit ist auch das in der Regel geforderte Löschen durchgeführt. In einem letzten Änderungsschritt werden alle in der Regel durch Operationsaufrufe auf einem Knoten spezifizierten Operationen aufgerufen. Dabei wird vorausgesetzt, dass diese voneinander unabhängig sind, denn der Aufruf geschieht in beliebiger Reihenfolge. Diese Operationen können entweder weitere Regeln aufrufen oder Hilfsmethoden einbinden. Damit sind die Änderungen an der Modellinstanz abgeschlossen.

[7] Der Terminus „zu bewahrendes/löschendes/erzeugendes Objekt/Link/Element“ steht hierbei immer für das Objekt, das bei der Mustersuche auf den zu bewahrenden/löschenden/erzugenden Knoten oder Link abgebildet wurde.

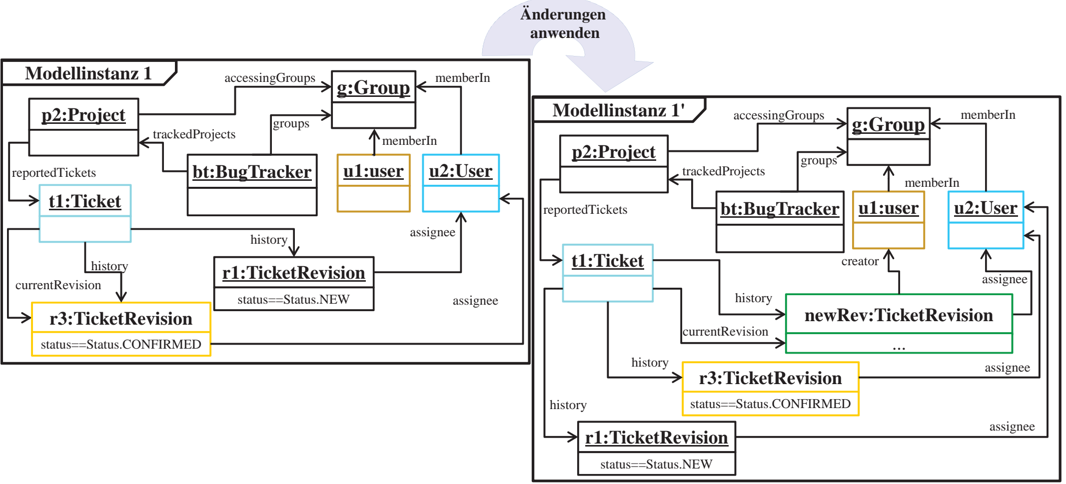


Abbildung 6.13: *Anwendung der in der Regel spezifizierten Änderungen auf* Modellinstanz 1 *des Anwendungsbeispiels*

Die am Beispiel changeStatus an Modellinstanz 1 ausgeführten Änderungen sind in Abbildung 6.13 dargestellt. Die Farbkodierung der Knoten und Objekte folgt der bereits bekannten. Der Zustand der Instanz vor den Änderungen und nach erfolgreicher Mustersuche findet sich links im Bild. Er ist mit Modellinstanz 1 benannt. Die geänderte Instanz ist in der Abbildung rechts gezeigt und mit Modellinstanz 1' benannt. Vergleicht man beide, sind die in der Regel spezifizierten Änderungen korrekt umgesetzt. Die neue Revision newRev wurde erstellt und die zugehörigen Links currentRevision, history, creator und assignee wurden erzeugt. Der Link currentRevision zwischen r3 und dem ticket t1 wurde entfernt.

7 Integration in das EMF-Rahmenwerk

7.1 Konzeptuelle Integration: Wege zur Nutzung von ModGraph

Nachdem nun geklärt ist, wie die Sprache der Graphtransformationsregeln aufgebaut ist und ausgeführt wird, stellt sich im Folgenden die Frage, wie diese Regeln in das EMF-Rahmenwerk eingebunden werden, um die bereits in Kapitel 5 betrachtete Zusammenarbeit der Regeln mit Xcore und Ecore zu verwirklichen.

Historisch gesehen, wurde das Werkzeug zur Erstellung von Graphtransformationsregeln mit ModGraph zunächst nur für Ecore-Modelle entwickelt und später auf Xcore-Modelle ausgeweitet. Der Kontrollfluss wurde zunächst mit Java implementiert und später durch die Möglichkeit Xcore zu nutzen ergänzt. Dabei steht stehts die Trennung von Struktur- und Verhaltensmodellierung im Vordergrund. Die Verhaltensmodellierung ist weitergehend in einen regelbasierten und einen prozeduralen Anteil aufgeteilt, wie zum Beispiel in Abbildung 7.1 zu sehen ist. Grund dafür ist die gewünschte klare Trennung von Kontrollfluss und Graphtransformationsregeln in ModGraph. Diese Trennung wird einerseits durch die Unmöglichkeit, Abläufe direkt in einer Regel abzubilden, andererseits aufgrund der Ergebnisse in Buchmann et. al. [20] vorgenommen. Diese Ergebnisse besagen vereinfacht, dass Graphtransformationsregeln nur sinnvoll sind, wenn sie komplexe Änderungen spezifizieren und weiterhin, dass textuelle Kontrollflüsse den grafischen vorzuziehen sind.

Somit haben sich fünf Wege entwickelt, die Graphtransformationsregeln samt des - zu ihrer Steuerung notwendigen - Kontrollflusses im EMF-Kontext zu verwenden: Der „klassische Weg“, der „neue Weg“, der „Weg zum Neuen“ und der „Weg zurück zu den Ursprüngen“ sowie der „unabhängige Weg“. Die Benennung der Wege ergibt sich aus ihrer Entstehung im ModGraph-Entwicklungsprozess. Jeder Weg nutzt eine Menge ineinandergreifender Modelle und Generatoren. Diese sind mit dem Ziel entstanden, die Modellierungsebene sukzessive auf ein höheres Abstraktionsniveau zu heben. Die Wege und deren Entwicklung wird nun beschrieben. Dabei steht hier zunächst die Vorgehensweise im Mittelpunkt, die genauen Generatormuster werden in Abschnitt 7.2 grundsätzlich betrachtet.

Zusammenarbeit mit Ecore: der „klassische Weg“

Der erste der Wege, der hier als klassischer Weg bezeichnet wird, ist in Abbildung 7.1 dargestellt und in [21] und [78] veröffentlicht. Der Pfeil links in dieser und den folgenden Abbildungen visualisiert das Fallen des Abstraktionsniveaus. Der mit „Start“

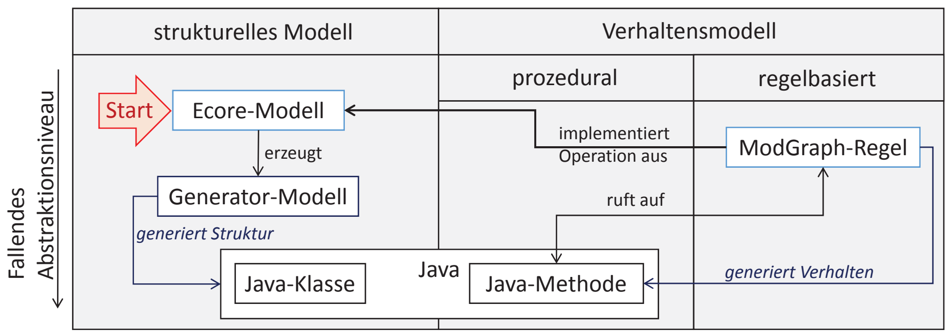


Abbildung 7.1: *Zusammenarbeit von Ecore und ModGraph*

markierte Pfeil zeigt den Einstiegspunkt in die Modellierung: das Ecore-Modell.[1] Die Vorgehensweise dieses Weges ist geradlinig: Zunächst wird das Ecore-Modell erstellt. Aus diesem wird ein Generator-Modell abgeleitet und daraus Quelltext generiert. Dies ist reiner EMF-Quelltext. Er beinhaltet alle modellierten Klassen, Attribute und deren Akzessoren. Zudem beinhaltet er Methodendeklarationen.

Die im Ecore-Modell enthaltenen Operationen können durch Graphtransformationsregeln implementiert werden. Diese Regeln unterstützen OCL- und Java-Ausdrücke, jedoch kein Xbase, da kein zugehöriges Xcore-Modell existiert. Die modellierten Regeln können Java-Methoden aufrufen und aus diesen aufgerufen werden. Der ModGraph-Generator generiert Quelltext. Dieser wird - da ModGraph für und mit EMF entworfen und gebaut wurde - nahtlos in den EMF-generierten Quelltext eingefügt. Dabei bedient sich ModGraph einer Compiler-Lösung, welche die modellierten Regeln in Quelltext übersetzt.

Auf diese Weise wird ausführbarer Quelltext erzeugt, der neben der Erzeugung von Objekten aus Klassen und der Initialisierung dieser die Ausführung modellierter Operationen erlaubt. Der Aufbau dieser Operationen wird im folgenden Unterkapitel 7.2 genauer betrachtet.

Die Kontrollstrukturen zur Steuerung des Ablaufs der Regeln, müssen auf diesem Weg in Java implementiert werden. Damit erweitert dieser Weg EMF um Verhaltensmodellierung. Die Regeln erhöhen den modellgetriebenen Anteil der Entwicklung beträchtlich, jedoch ist hier noch keine totale modellgetriebene Softwareentwicklung möglich.

Zusammenarbeit mit Xcore: der „neue Weg“

Die Entwicklung von Xcore, die parallel zu ModGraph stattfindet, eröffnet eine neue Perspektive auf die modellgetriebene Entwicklung im EMF-Kontext. Xcore repräsentiert Ecore-Modelle textuell und erweitert diese um die Spezifikation von Verhalten. Zunächst wäre Xcore als textuelles Modell auf gleicher Abstraktionsebene anzusehen wie das zugehörige Ecore-Modell. Die Sprache zur Verhaltensspezifikation, Xbase, ist jedoch sehr

[1] Aus Gründen der Übersicht wird die Modellierung vom Ecore-Klassendiagramm ausgehend betrachtet. Natürlich steht dem Nutzer auch die Möglichkeit ein Paketdiagramm zu erstellen und daraus ein Klassendiagramm zu erzeugen zur Verfügung. Da dies jedoch für die Beschreibung der Verhaltensmodelle nicht von Bedeutung ist, wird es hier nicht weiter betrachtet.

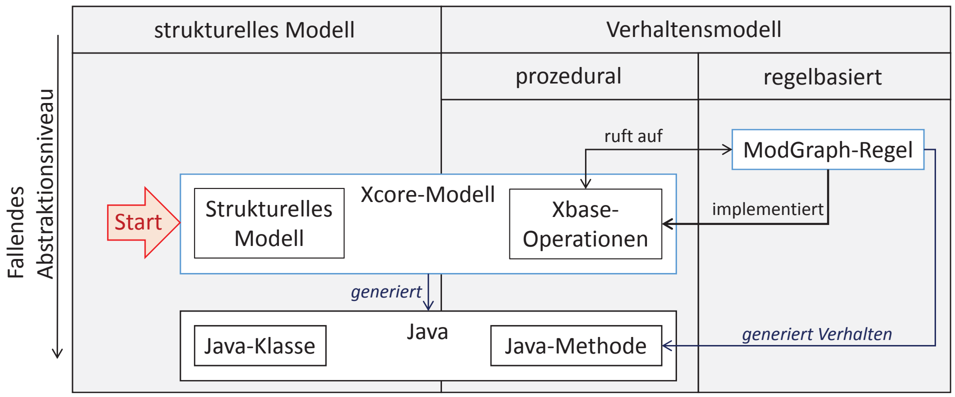


Abbildung 7.2: *Zusammenarbeit von Xcore und ModGraph*

Java-nah. Beispielsweise muss die bei den ModGraph-Regeln gegebene Mustersuche hier explizit angegeben werden. Damit ist Xcore bezüglich des Abstraktionsniveaus etwas unterhalb von Ecore-Modellen anzusiedeln, jedoch deutlich über dem Quelltext.

Abbildung 7.2 zeigt den Ablauf der Modellierung mit Xcore und ModGraph. Beginnend mit einem Xcore-Modell, das wiederum mit einem roten Pfeil gekennzeichnet ist, kann der Modellierer sein Modell textuell gestalten und einfache oder prozedurale Operationen mit Xbase spezifizieren. Komplexe Operationen können mit ModGraph-Regeln implementiert werden. So ist es möglich, Kontrollstrukturen für die Regeln ebenfalls innerhalb des Modells zu spezifizieren. Der Xcore-Generator generiert Quelltext aus dem Xcore-Modell, in welchen der aus der ModGraph-Regel generierte Quelltext ebenso nahtlos eingefügt werden kann. Daher dürfen ModGraph-Regeln Xcore-Operationen aufrufen und von diesen aufgerufen werden.

Auf diesem Weg kann erstmals die Struktur und das komplette Verhalten auf Modellebene spezifiziert werden. Damit ist eine Java-nahe, aber dennoch totale, modellgetriebene Softwareentwicklung mit Xcore und ModGraph möglich.

Einbettung von Kontrollstrukturen: der „Weg zum Neuen“

Werden grafische Notationen bevorzugt, kann totale modellgetriebene Softwareentwicklung in ModGraph auch unter Verwendung von Ecore verwirklicht werden. Abbildung 7.3 zeigt diesen Weg zum Neuen. Dazu wird zunächst ein Ecore-Modell erstellt, dessen komplexe Operationen mit ModGraph-Regeln implementiert werden. Dies ist analog zum klassischen Weg zu sehen. Das Ecore-Modell kann nun über sein Generator-Modell (mittels dessen Export-Rahmenwerks) in ein Xcore-Modell umgewandelt werden. Dieser Umweg ist von Bedeutung, da die Transformation nicht rückgängig gemacht werden kann. Das Xcore-Modell ist unabhängig von dem Ecore-Modell, aus welchem es abgeleitet ist.[2]

[2]Es ist möglich, die Abhängigkeit von Ecore- und Xcore-Modell manuell herzustellen, indem eine Abbildung zwischen dem Ecore- und dem Xcore-Modell definiert wird. Dies ist jedoch erst in Abschnitt 9 von Bedeutung.

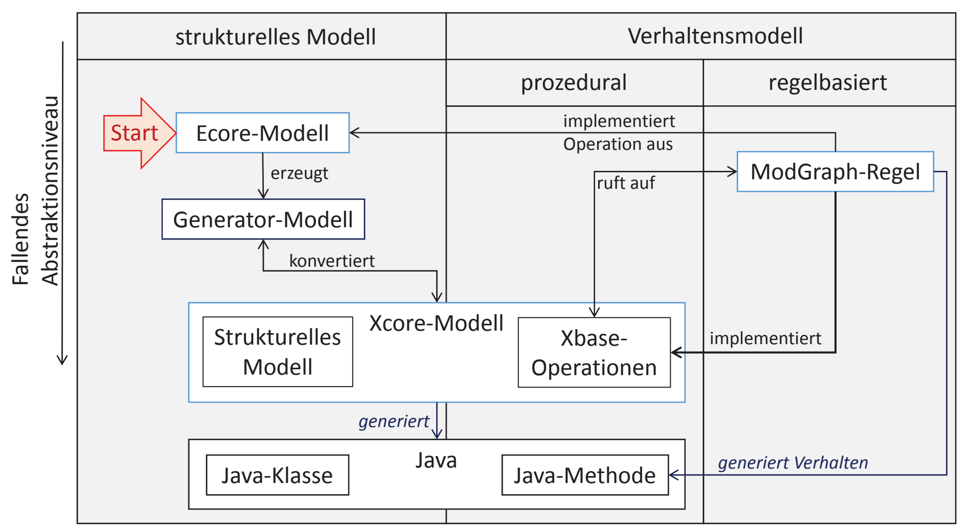


Abbildung 7.3: *Einbettung von Xcore zur Modellierung von Kontrollstrukturen*

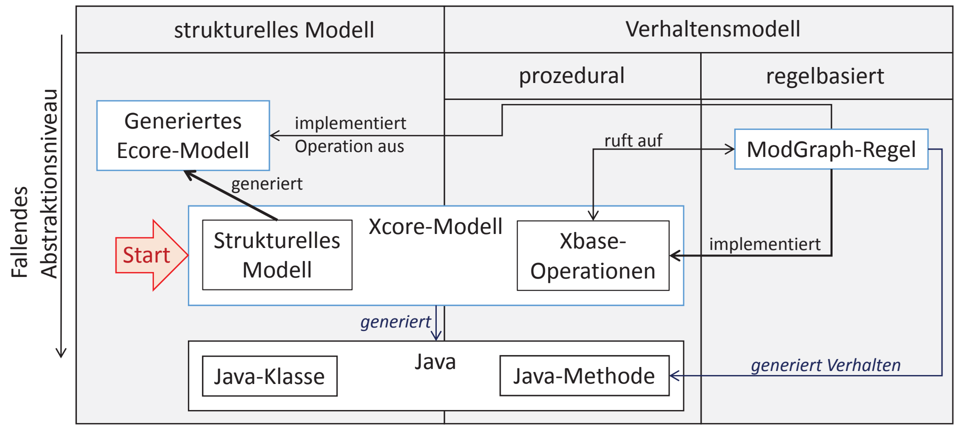


Abbildung 7.4: *Zusammenarbeit von Xcore, dessen generiertem Ecore-Modell und ModGraph*

Diese Modell-Transformation beeinflusst die ModGraph-Regeln nicht. Sie können problemlos wiederverwendet werden, da die referenzierten Operationen aus dem Ecore-Modell auf die korrespondierenden Operationen im Xcore-Modell abgebildet werden. Damit können nun einfache Operationen und prozedurale Operationen, wie Kontrollflüsse für die Regeln als Xcore-Operationen spezifiziert werden. Der ModGraph-Generator fügt den Quelltext, wie im neuen Weg gezeigt, in den Xcore-generierten Quelltext ein. Die Modelle werden auf Quelltext-Niveau zusammengeführt.

Zusammenarbeit mit Xcore und Ecore: der „Weg zurück zu den Ursprüngen“

Der Weg zurück zu den Ursprüngen kann als Variante des Weges zum Neuen verstanden werden. Er ist in Abbildung 7.4 dargestellt. Auch hier wird totale modellgetriebene Softwareentwicklung mit Ecore, Xcore und ModGraph betrieben. Dabei wird, wie

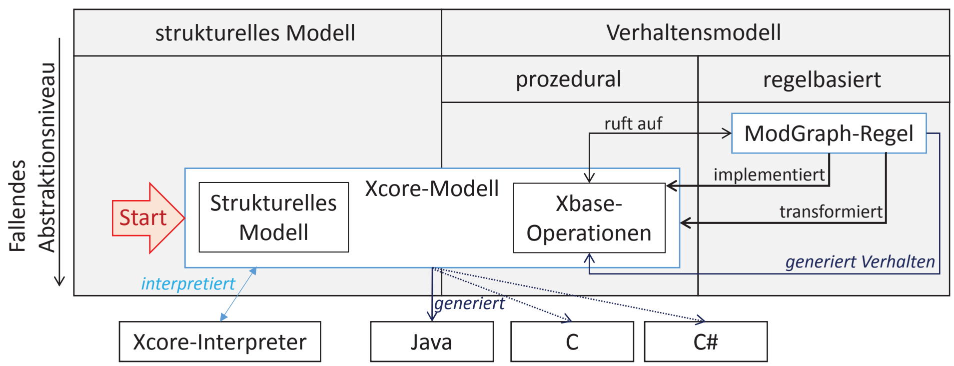


Abbildung 7.5: *Quelltext unabhängige Zusammenarbeit von Xcore und ModGraph*

im neuen Weg, mit der Erstellung eines Xcore-Modells begonnen. Dieses beschreibt die Struktur und die einfachen oder prozeduralen Operationen. Aus dem strukturellen Modell kann nun ein Ecore-Modell generiert werden. Dieses wird hier explizit erstellt. Es kann nun, analog zum klassischen Weg, zur Implementierung von Operationen mit ModGraph-Regeln genutzt werden. Allerdings bleibt die Möglichkeit, den Kontrollfluss in Xcore zu beschreiben. Der ModGraph-generierte Quelltext wird auch hier in den Xcore-generierten Code eingebunden.

Unabhängigkeit von Java: der „unabhängige Weg“

Eine echte Alternative bietet der unabhängige Weg, der in Abbildung 7.5 dargestellt ist. ModGraph-Regeln implementieren hierbei die in einem Xcore-Modell implementierten Operationen. Hierbei ist die Verwendung von Xbase-Ausdrücken in den textuellen Elementen möglich und explizit erwünscht. Alternativ können die Regeln auch, wie im Weg zum Neuen beschrieben, wiederverwendet werden. Sie interagieren mit dem Xcore-Modell, indem sie von Operationen des Xcore-Modells aufgerufen werden und diese aufrufen.

Die entscheidende Neuerung liegt in der Generierung: Die ModGraph-Regeln werden direkt nach Xcore (und Xbase) übersetzt und nahtlos in das Xcore-Modell integriert. Die Modelle werden nicht mehr auf Quelltext-Niveau, sondern auf Modellebene zusammengeführt. Es handelt sich, wie in den vorangegangenen Wegen, um eine Compiler-Lösung. Die Ausführungslogik der Regel bleibt dabei die in Abschnitt 6.2.1 dargestellte. Die Vorgehensweise des Generators ist in Abschnitt 7.2.2 beschrieben.

Im Unterschied zu allen anderen Wegen wird hier die Realisierung einer unidirektionalen Modell-zu-Modell-Transformation einer ModGraph-Regel in eine prozedurale Sprache zur Verhaltensmodellierung umgesetzt. Während die bislang besprochenen Wege die Modelle auf Quelltext-Ebene vereinen, wird dies hier auf Modellebene verwirklicht. Damit bietet der unabhängige Weg eine schrittweise Übersetzung der Regel an und erreicht damit das höchste Abstraktionsniveau im ModGraph-Ansatz. Der generierte Quelltext ist konzise, obwohl die Mustersuche darin berücksichtigt wird, die im Idealfall - wie in den Regeln - implizit gegeben ist. Er ist gut lesbar und einfach, da Xcores Sprachkon-

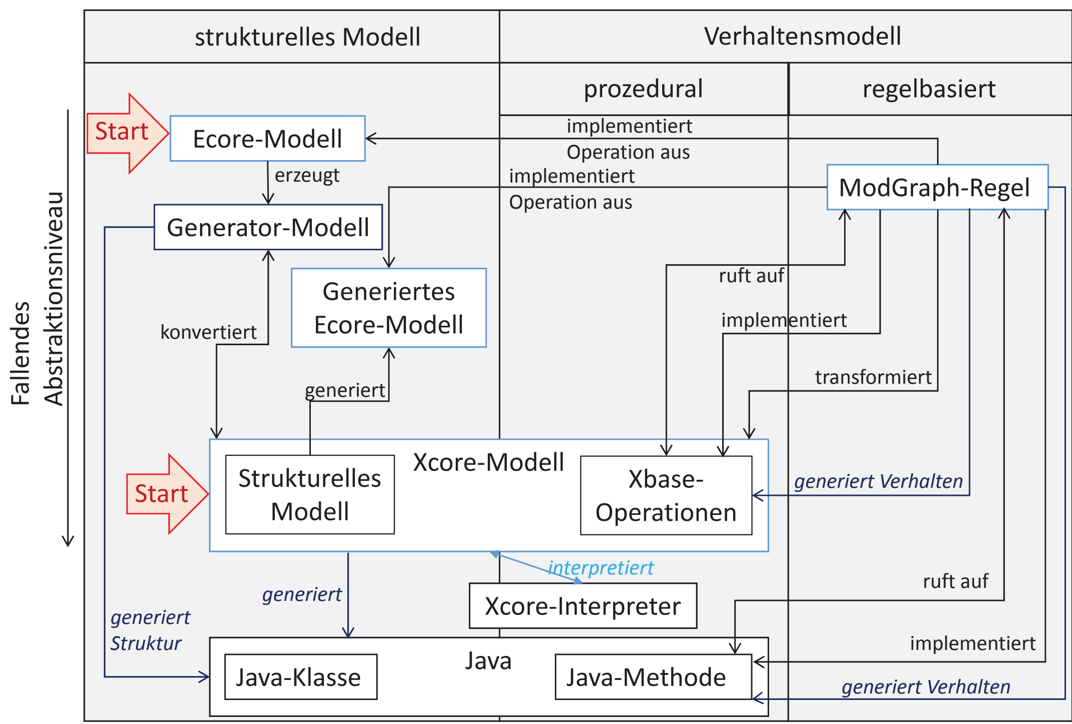


Abbildung 7.6: *Zusammenschau der möglichen Wege*

strukte im Vergleich zu Java ein deutlich höheres Abstraktionsniveau aufweisen. Dies vereinfacht beispielsweise das Debuggen enorm. Des Weiteren ist der Quelltext portabel. Die Übersetzung in mehrere Zielsprachen, nicht nur nach Java, sondern auch nach C, C# oder beliebigen anderen Sprachen wäre nun möglich. Leider bietet Xcore bislang keine Generatoren für C-Dialekte oder andere Sprachen an. Kurz gesagt, die Zielsprache ist auf diesem Weg nicht mehr von Bedeutung. Es besteht sogar die Möglichkeit, überhaupt keinen Quelltext zu generieren. Das Modell ist interpretierbar. Zur Interpretation wird der Xcore-Interpreter genutzt, der die Interpretation von Verhalten auf einer Modellinstanz unterstützt.

Möglichkeiten der Integration in EMF zusammengefasst

Zusammenfassend sind die Wege zur Zusammenarbeit von Ecore, Xcore und ModGraph in Abbildung 7.6 dargestellt. Es handelt sich hier um einen sehr flexiblen und hochgradig integrativen Ansatz. Der Wechsel zwischen Ecore und Xcore ist beliebig erlaubt und die Nutzung der bereits spezifizierten Regeln ist weiterhin problemlos möglich. Das durch diese Zusammenarbeit erstellte Gesamtmodell ist ausführbar. Es kann interpretiert oder in Java-Quelltext übersetzt werden. Die Interpretation erfolgt auf Xcore-Modellinstanzen, die durch die Ausführung der Operationen endogen und überschreibend transformiert werden. Der generierte Quelltext ist nicht nur ausführbar, er kann - da es sich um Java-Quelltext handelt - beliebig weiter verwendet werden. Jede für handgeschriebenen Quelltext gegebene Nutzungsmöglichkeit ist ebenso mit dem hier ge-

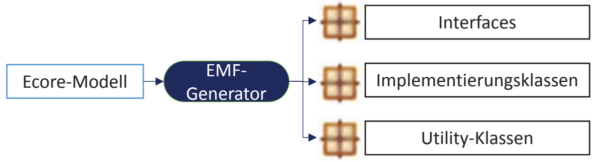


Abbildung 7.7: *Schematische Darstellung des Aufbaus von EMF-generiertem Quelltext*

nerierten Quelltext gegeben. Wie dieser Java-Quelltext bzw. die Xcore-Methoden genau aufgebaut sind, wird im nächsten Abschnitt erläutert.

7.2 Generatoren

Bereits im vorangegangenen Abschnitt 7.1 wurde angesprochen, dass ModGraph-Regeln zu ihrer Ausführung immer in Text übersetzt werden. Dieser Text ist entweder Java-Quelltext oder ein Ausschnitt aus einem textuellen Xcore-Modell, eine Xbase-Operation. Wie diese Texte erzeugt werden wird im Weiteren erläutert. Die Vorgehensweise bei der Erzeugung von Java-Quelltext wird in [21] skizziert und in [78] genauer betrachtet. Diejenige zur Erweiterung eines Xcore-Modells ist in [81] veröffentlicht.

Beide Generatoren sind in Xpand, einer Template-Sprache des oAW[3], verfasst. Mit Xpand lassen sich Modell-zu-Text-Transformationen definieren und damit auch Modell-zu-Modell-Transformationen, wenn für die Zielsprache eine textuelle Syntax definiert ist. Zu ihrem korrekten Aufruf wird die Modeling Workflow Engine (MWE) genutzt, welche ebenfalls Teil von oAW ist. Diese stellt ebenso sicher, dass die Regel vor ihrer Generierung gemäß ihrer kontextsensitiven Syntax (Abschnitt 6.1.1) validiert wird. Zudem wird die Ausdruckssprache von oAW verwendet. Diese Sprache mischt Elemente von Java und OCL und wird in Xpand genutzt um Hilfsmethoden zu definieren. Diese werden zum Beispiel zum Laden von Elementen einer Ressource benötigt.

7.2.1 Generierung von Java-Quelltext mit ModGraph

Die ModGraph-Generierung von Java-Quelltext setzt eine vorangegangene erfolgreiche EMF-Generierung voraus. Der Grund hierfür ist wiederum in der ModGraph-Philosophie zu finden: EMF wird echt erweitert. Dazu wird der von ModGraph generierte Quelltext nahtlos in den EMF-Quelltext eingebunden. Um diesen Vorgang besser verstehen zu können, ist es sinnvoll, zunächst den EMF-generierten Quelltext näher zu betrachten.

Der EMF-generierte Quelltext

Der EMF-Generator erzeugt aus jedem Modell drei Pakete, wie sie schematisch in Abbildung 7.7 dargestellt sind. Das erste Paket enthält Interfaces, das zweite die zugehörigen

[3]Die einzelnen Teilprojekte von oAW sind mittlerweile Teil des Eclipse Modeling Project, dennoch wird hier oAW als Sammelbegriff für die Werkzeuge verwendet.

Implementierungsklassen. Das dritte Paket, im Weiteren Utility-Paket genannt, stellt Utility-Klassen bereit.

Für die zunächst folgenden Betrachtungen ist die Erzeugung von je einem Interface und der zugehörigen Implementierungsklasse für die Klassen des Modells wichtig. Innerhalb dieser bildet der EMF-Generator die modellierte statische Struktur ab. Zudem bietet er Akzessoren für die modellierten Attribute und konsistenzerhaltende Methoden für die Beziehungen zwischen den aus den Klassen instanziierten Objekten. Diese Beziehungen sind Instanzen der Referenzen, genannt Links. Beispielsweise werden Instanzen bidirektionaler Referenzen bei jeder Änderung konsistent für beide Richtungen aktualisiert.

Auflistung 7.1 stellt einen Ausschnitt aus dem generierten Java-Quelltext der Implementierungsklasse zur modellierten Klasse Ticket aus dem Bugtracker-Beispiel dar. Sich wiederholende (Teile der) Kommentare wurden dabei nicht gelistet. Jedes Element ist mit einer @generated-Annotation versehen, wie zum Beispiel in Auflistung 7.1, Zeile 7 gezeigt. Diese kennzeichnet generierte Quelltextelemente. Zudem kann zu jedem Element ein Kommentar, wie in Auflistung 7.1, Zeile 5 exemplarisch gelistet, angegeben werden.

Die gelistete Implementierung definiert zunächst das Attribut number der Klasse Ticket, indem ein Standardwert deklariert (Zeile 10) und dem neu deklarierten Attribut zugewiesen wird (Zeile 13). Außerdem werden Akzessoren für das Attribut erstellt (Zeilen 15-29). Die Methode zur Abfrage der Variablen entspricht dabei einem klassischen Akzessor, der den Wert der Variablen zurückgibt. Die Methode zum Setzen des Wertes der Variablen ist etwas komplexer. Sie setzt nicht nur den Wert, sondern benachrichtigt zudem potenziell an dieser Änderung interessierte Objekte (Zeilen 26-28).

Der generierte Quelltext zur Referenz currentRevision ist ebenfalls in Auflistung 7.1 gezeigt. Die Referenz wird als gleichnamige Variable umgesetzt (Zeile 38), die einerseits über EMFs generische Zugriffsmethoden (`eGet, eSet`, nicht gelistet) verwaltet werden kann, andererseits über die generierten Akzessor-Methoden (Zeilen 40-61). Die Get-Methode (Zeilen 40-52) berücksichtigt, dass das angeforderte Objekt eventuell nicht geladen ist und lädt dieses auf Anfrage nach, indem der Proxy für das Objekt aufgelöst und geprüft wird, ob das geladene Objekt dem erwarteten Objekt entspricht. Ist dies nicht der Fall, werden potenziell interessierte Objekte benachrichtigt. Das Setzen der Referenz (Zeilen 54-61) erfolgt analog zum Setzen des eben betrachteten Attributs number.

Analog zu diesen beiden Beispielen wird Quelltext für alle Attribute und Referenzen erzeugt. Mehrwertige Referenzen werden dabei als Listen, typisiert über die Zielklasse umgesetzt statt als Variablen. Nur einer der Akzessoren, die Get-Methode, wird erzeugt. Das Setzen der Elemente geschieht über Listenoperationen.

Die Umsetzung einer Operation in eine Java-Methode ist am Beispiel von changeStatus(...) in Auflistung 7.2 dargestellt. Die Methodendeklaration (Zeile 6) wird, wie in der modellierten Operation angegeben, korrekt umgesetzt. Der Rumpf der Methode wird mit einer Ausnahme befüllt (Zeile 9) und der Modellierer in einem Kommentar (Zeilen 7 & 8) dazu aufgerufen, die Methode per Hand in Java zu implementieren. In ModGraph wird dieser Aufruf als Anlass, verhaltensbeschreibenden Quelltext an genau diese Stelle zu generieren, betrachtet.

```java
public class TicketImpl extends MinimalEObjectImpl.Container implements Ticket {
...
/**
* The default value of the '{@link #getNumber() <em>Number</em>}' attribute.
* <!-- begin-user-doc --> <!-- end-user-doc -->
* @see #getNumber()
* @generated
* @ordered
*/
protected static final int NUMBER_EDEFAULT = 0;

/** The cached value of the '{@link #getNumber() <em>Number</em>}' attribute. */
protected int number = NUMBER_EDEFAULT;

/**
* <!-- begin-user-doc --> <!-- end-user-doc -->
* @generated
*/
public int getNumber() {
    return number;
}

public void setNumber(int newNumber) {
    int oldNumber = number;
    number = newNumber;
    if (eNotificationRequired())
        eNotify(new ENotificationImpl(this, Notification.SET,
            BugmodelPackage.TICKET__NUMBER, oldNumber, number));
}
...
/**
* The cached value of the '{@link #getCurrentRevision() <em>Current Revision</em>}'
* reference. ...
* @see #getCurrentRevision()
* @generated
* @ordered
*/
protected TicketRevision currentRevision;

public TicketRevision getCurrentRevision() {
    if (currentRevision != null && currentRevision.eIsProxy()){
        InternalEObject oldCurrentRevision = (InternalEObject)currentRevision;
        currentRevision = (TicketRevision)eResolveProxy(oldCurrentRevision);
        if (currentRevision != oldCurrentRevision){
            if (eNotificationRequired())
                eNotify(new ENotificationImpl(this, Notification.RESOLVE,
                    BugmodelPackage.TICKET__CURRENT_REVISION, oldCurrentRevision,
                        currentRevision));
        }
    }
return currentRevision;
}

public void setCurrentRevision(TicketRevision newCurrentRevision) {
    TicketRevision oldCurrentRevision = currentRevision;
    currentRevision = newCurrentRevision;
    if (eNotificationRequired())
        eNotify(new ENotificationImpl(this, Notification.SET,
            BugmodelPackage.TICKET__CURRENT_REVISION, oldCurrentRevision,
                currentRevision));
}
...
```

Auflistung 7.1: *Ausschnitt aus dem EMF-generierten Quelltext für die modellierte Klasse* Ticket *des Bugtracker Ecore-Modells: generierter Quelltext zum modellierten Attribut* number:EInt *und zur modellierten Referenz* currentRevision

```
...
/**
* <!-- begin-user-doc --> <!-- end-user-doc -->
* @generated
*/
public void changeStatus(Status status, User author) {
// TODO: implement this method
// Ensure that you remove @generated or mark it @generated NOT
throw new UnsupportedOperationException();
}
...
```

Auflistung 7.2: *Ausschnitt aus dem EMF-generierten Quelltext für die modellierte Klasse Ticket des Bugtracker Ecore-Modells: generierter Quelltext zur modellierten Operation changeStatus*

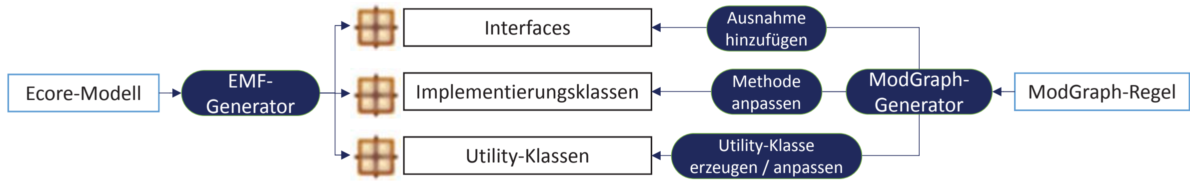


Abbildung 7.8: *Schematische Darstellung der Quelltextgenerierung mit ModGraph und des Eingriffs in den EMF-generierten Quelltext durch den ModGraph-Generator*

Einbettung und Aufbau des ModGraph-generierten Quelltexts

ModGraph bindet den zur Implementierung der Methoden generierten Quelltext nahtlos in den EMF-generierten Quelltext ein. Dazu bietet es eigene Xpand-Templates an. Beispiele dieser Templates finden sich in Anhang A.3. Eine Modifikation der EMF-Templates findet nicht statt. Zur Generierung bedient sich ModGraph einer Compiler-Lösung. Die modellierten Regeln werden in EMF-kompatiblen Quelltext übersetzt, der direkt in den vorhandenen eingebunden wird. Wie der ModGraph-Generator dazu in den EMF-generierten Quelltext eingreift, ist schematisch in Abbildung 7.8 gezeigt. Es wird vorausgesetzt, dass die EMF-Generierung vor der ModGraph-Generierung stattfindet, so dass eine Erweiterung aller EMF-generierten Pakete, die in Abbildung 7.8 mittig dargestellt sind, möglich ist. Dies geschieht einerseits mit Hilfe des abstrakten Syntaxbaums der entsprechenden Java-Klasse des Pakets. Die Interfaces und deren Implementierungsklassen werden auf diese Art und Weise angepasst. Andererseits wird das EMF-generierte Utility-Paket durch Einfügen ModGraph-eigener Utility-Klassen erweitert.

Anpassung des Interfaces

Im ersten Schritt erfolgt die Anpassung des EMF-generierten Interfaces, das die Methodendeklaration enthält. Dies ist notwendig, da alle durch ModGraph implementierten Methoden Ausnahmen auslösen können, zum Beispiel durch eine fehlgeschlagene Nachbedingung. Daher greift der ModGraph-Generator zuerst in das EMF-generierte Interface zur Klasse ein und erweitert, wenn nötig, die Methodendeklaration darin um eine Ausnahmebehandlung. Ob diese Erweiterung bereits besteht oder erfolgen muss,

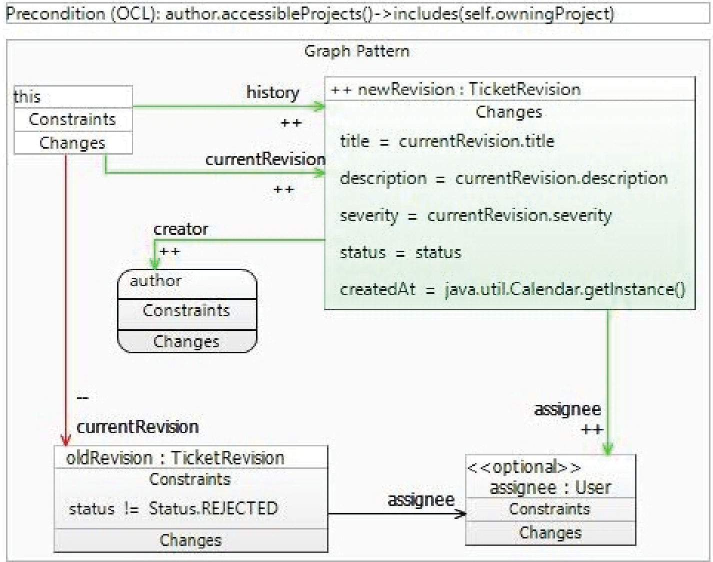


Abbildung 7.9: *Graphtransformationsregel zur Implementierung der Operation* changeStatus *der Klasse* `Ticket` *aus dem Bugtacker Beispiel*

```
1 public interface Ticket extends EObject {
2  ...
3     /**
4     * <!-- begin-user-doc -->Ändert den Status eines Tickets. <!-- end-user-doc -->
5     *   ...
6     * @generated NOT modified by ModGraph
7     */
8     void changeStatus(Status status, User author) throws GTFailure;
9 ...
```

Auflistung 7.3: *Ausschnitt aus dem kombiniert generierten Quelltext für die modellierte Klasse* Ticket *des Bugtracker Ecore-Modells: Anpassung der Methodendeklaration im generierten Interface zur modellierten Operation* changeStatus *(Abbildung 7.9)*

hängt vom Zeitpunkt der EMF-Generierung ab. Bei der Implementierung einer Operation durch eine ModGraph-Regel erfolgt eine Anpassung des zugehörigen Ecore-Modells, indem die Operation um eine Ausnahme, die sie auslösen darf, erweitert wird. Ist dies schon vor der Generierung von Quelltext geschehen, so ist die Ausnahme auch dem Interface bekannt.

Das Ergebnis einer solchen Erweiterung ist exemplarisch für changeStatus in Auflistung 7.3 gezeigt. Die zuvor generierte Annotation `@generated` wird durch ein `NOT modified by ModGraph` ergänzt, so dass der EMF-Generator diese Methode nicht mehr verändert. Zudem wird der EMF-generierte JavaDoc-Kommentar erweitert, indem der Kommentar zur Regel hinzugefügt wird.

Änderungen in der Implementierungsklasse

Die zugehörige Implementierungsklasse wird bezüglich ihrer Ausnahmebehandlung

```java
public class TicketImpl extends MinimalEObjectImpl.Container implements Ticket {
    /**
    * <!-- begin-user-doc --> Ändert den Status eines Tickets. <!-- end-user-doc -->
    * @generated NOT modified by ModGraph
    */
    public void changeStatus(Status status, User author) throws GTFailure {
        OCLHandler.evaluatePrecondition(
            "author.accessibleProjects()->includes(self.owningProject)",
                this, new String[] { "author" }, new Object[] { author });

        GTUtil4TicketchangeStatus util = new GTUtil4TicketchangeStatus();
        util.match(status, author, this);
        TicketRevision oldRevision = (TicketRevision) util.get("oldRevision");
        User assignee = (User) util.get("assignee");

        final java.lang.String newRevisionTitleValue = currentRevision.getTitle();
        final java.lang.String newRevisionDescriptionValue =
            currentRevision.getDescription();
        final Severity newRevisionSeverityValue = currentRevision.getSeverity();
        final Status newRevisionStatusValue = status;
        final java.util.Calendar newRevisionCreatedAtValue =
            java.util.Calendar.getInstance();

        TicketRevision newRevision =
            BugmodelFactory.eINSTANCE.createTicketRevision();

        newRevision.setTitle(newRevisionTitleValue);
        newRevision.setDescription(newRevisionDescriptionValue);
        newRevision.setSeverity(newRevisionSeverityValue);
        newRevision.setStatus(newRevisionStatusValue);
        newRevision.setCreatedAt(newRevisionCreatedAtValue);

        this.setCurrentRevision(newRevision);
        newRevision.setCreator(author);
        this.getHistory().add(newRevision);
        newRevision.setAssignee(assignee);
    }
```

Auflistung 7.4: *Ausschnitt aus dem kombiniert generierten Quelltext für die modellierte Klasse* Ticket *des Bugtracker Ecore-Modells: Implementierung der modellierten Operation* changeStatus *(Abbildung 7.9)*

und des Kommentars analog zu ihrem Interface angepasst. Ein Importmanager ergänzt die fehlenden Importe der Klasse. Fehlende Importe treten insbesondere dann auf, wenn in der ModGraph-Regel ein Objekt einer anderen Klasse, die nicht in direkter Beziehung zu der betrachteten steht, modifiziert oder erstellt wird. Zudem werden ModGraph-Bibliotheken für die Ausnahmebehandlung und die OCL-Unterstützung eingebunden sowie die zur Mustersuche erstellten Klassen im Utility-Paket.

Die Generierung des Methodenrumpfs erfolgt separat. Er wird - unter Zuhilfenahme des abstrakten Syntaxbaums der Java-Klasse - in den Methodenrumpf der zu implementierenden Methode eingefügt. Der EMF-generierte Pseudo-Rumpf wird dabei entfernt. Das Resultat für das Beispiel changeStatus ist in Auflistung 7.4 dargestellt. Um es besser verstehen zu können, ist die zugehörige Regel nochmals in Abbildung 7.9 gezeigt.

Die Ausführungslogik der Methode ist analog zu der einer ModGraph-Regel aufgebaut. Letztere wurde in Kapitel 6.2 durch die dynamische Semantik der Sprache beschrieben. Zuerst werden alle Vorbedingungen überprüft. Bei OCL-Bedingungen geschieht dies mit

Hilfe einer ModGraph-eigenen `OCLHandler`-Klasse. Dieser wird dabei nicht nur der Ausdruck selbst, sondern auch der Kontext des Ausdrucks übergeben. Im Beispiel (Auflistung 7.4) ist die Auswertung der Vorbedingung (Abbildung 7.9 oben) in Zeilen 7-9 dargestellt. Dazu muss der Parameter `author` bekannt sein und wird dementsprechend der `OCLHandler`-Klasse übergeben.

Schlägt eine Vorbedingung fehl, wird eine Ausnahme ausgelöst. Sind alle Vorbedingungen erfüllt, findet die Mustersuche statt. Diese ist in die generierte Utility-Klasse, die später beschrieben wird, ausgelagert. Bei erfolgreicher Mustersuche werden alle Objekte von der Utility-Klasse abgeholt. Im Beispiel sind dies die aktuelle Revision des Tickets, `oldRevision`, und der zuständige Benutzer, `assignee`. Diese sind in der Regel als zu erhaltende ungebundene Knoten gleichen Namens modelliert.

Die rechten Seiten der Attributzuweisungen werden im nächsten Schritt ermittelt. In Auflistung 7.4 werden alle der zu erstellenden Ticketrevision zuzuordnenden Attributwerte berechnet. Dies sind, wie die Regel in Abbildung 7.9 vorgibt, title, description, severity, status und createdAt. Die Werte werden hierbei in finalen Variablen abgelegt, die den Typ des Attributs tragen, dessen zukünftigen Wert sie beinhalten. Sie werden zudem nach diesen Attributen und dem Objekt benannt. Der so erhaltene Name wird durch `Value` ergänzt, um die Wertberechnung an dieser Stelle hervorzuheben. Für den Attributwert des Attributs title ist dies in Zeile 16 dargestellt, die anderen Attribute folgen (Zeilen 17-22).

Für zu bewahrende Objekte erfolgt die Zuweisung der zuvor berechneten Attributwerte. Dabei kann ein zu erhaltendes, optionales Objekt nicht vorhanden sein. Dementsprechend wird vor der Zuweisung durch eine `if`-Abfrage geprüft, ob das Objekt während der Mustersuche gefunden wurde.

Die zu löschenden Elemente werden gelöscht. Das Löschen eines Objekts, das im Beispiel nicht vorkommt, wird an die Utility-Klasse delegiert, da das einfache Löschen des Objekts für sich nicht ausreicht. Es muss aus seinem gesamten Kontext, der normalerweise nicht komplett in einer Regel angegeben ist, entfernt werden. Nur so ist sichergestellt, dass keinesfalls sogenannte hängende Kanten auftreten. Dies sind Links ohne Quell- oder Zielknoten, die zu einer nicht mehr validen Modellinstanz führen. Die Methoden zum Löschen eines Objekts sind in Auflistung 7.5 gezeigt. Die Methode zum Löschen eines einzelnen Objekts (Zeilen 2-4) delegiert dies lediglich an die Klasse `EcoreUtil`, so dass diese auch direkt aus dem generierten Quelltext aufgerufen werden könnte. Der Grund, dies nicht zu tun, liegt in der Verwirklichung einer einheitlichen Methode zum Löschen in ModGraph. Das Löschen von Objekten, die auf einen zu löschenden mehrwertigen Knoten abgebildet wurden, kann nicht durch einen einzigen Aufruf korrekt ausgeführt werden. Diese Objekte werden in EMF als Liste verwaltet, die komplett - Objekt für Objekt - korrekt gelöscht werden muss. Dazu ruft die Methode, die in den Zeilen 5-10 der Auflistung 7.5 gelistet ist, für jedes Objekt einzeln die Methode `EcoreUtil.delete(...)` auf, welche das Objekt aus seinem Kontext entfernt.

Das Löschen eines Links, der eine einwertige Referenz instanziiert, erfolgt durch das `null`-setzen der zugehörigen Variable, es sei denn, er wird in derselben Regel neu gesetzt und damit ersetzt. Instanziiert der zu löschende Link eine mehrwertige Referenz, geschieht das Löschen durch dessen Entnahme aus der in EMF auf Quelltextebene ver-

```
public class GTMatchingUtil { ...
    public void delete(EObject eObject) {
        EcoreUtil.delete(eObject, true);
    }
    public void delete(EList<?> objectList) {
        checkResource();
        for (Object o : objectList) {
            if (o instanceof EObject)
            EcoreUtil.delete((EObject) o, true);
        } ...
    }
```

Auflistung 7.5: *ModGraphs Methoden zum Löschen von Objekten aus ihrem gesamten Kontext*

walteten Liste. Wird ein Link ausgehend aus einem mehrwertigen Knoten gelöscht, so muss dieser mittels einer Schleife von jedem für den Knoten gefundenen Objekt entfernt werden. Soll ein Link, dessen Quell- oder Zielknoten einem optionalen Knoten der Regel entspricht, entfernt werden, so muss zunächst geklärt werden, ob ein passendes Objekt für diesen Knoten vorhanden ist. Wenn ja, so wird der Link gelöscht.

Der zu löschende Link currentRevision in der Regel wird vom ModGraph-Generator als Instanz einer einwertigen Referenz erkannt, die in derselben Regel - durch den zu erzeugenden Link gleichen Namens - neu gesetzt wird. Er wird daher nicht explizit entfernt.

Im nächsten Schritt werden die zu erzeugenden Objekte mittels der EMF-generierten Fabriken erstellt. Ihnen werden die zugehörigen, zuvor berechneten Attributwerte zugewiesen.

Dementsprechend wird auch die als zu erzeugender Knoten newRevision modellierte Ticketrevision erzeugt. Dazu wird die passende Fabrikmethode genutzt (Zeilen 24 & 25 in Auflistung 7.4). Die zuvor berechneten Attributwerte werden, in Auflistung 7.4 Zeilen 27-31, den neu erstellten Objekten zugewiesen. Hierfür werden die passenden Akzessoren aufgerufen.

Analog werden alle in der Regel aus Abbildung 7.9 als neu zu erstellend markierte Links erzeugt. Endet oder beginnt dabei ein Link in einem optionalen Knoten, so muss zunächst geklärt werden, ob ein übereinstimmendes Objekt für diesen Knoten existiert. Ist dies der Fall, wird der Link erzeugt. Das geschieht entweder durch Setzen einer Variable vom Typ des Zielknotens oder durch Hinzufügen des Zielobjekts eines Links in die Liste der referenzierten Objekte. Ein zu erzeugender Link, der eine mehrwertige Referenz instanziiert, kann geordnet sein. Dabei wird das neu einzufügende Objekt in die Liste der referenzierten Objekte genau an der spezifizierten Position eingefügt. Dazu wird die Listenoperation `add(int index, EObject o)` im Falle der Einzelobjekte, mit der gleich parametrisierten Methode `addAll(...)` im Falle mehrerer Objekte, genutzt. Ist an dem Link ein Objektname angegeben, vor oder nach welchem ein oder mehrere Objekte eingefügt werden sollen, so wird dessen Position über den Index der Liste ermittelt.

Beim Setzen des Links currentRevision in Zeile 33 der Auflistung 7.4 wird der bestehende Link überschrieben. Damit ist auch das in der Regel geforderte Löschen durchgeführt. In den folgenden Zeilen werden die weiteren Links erstellt. Der letzte dieser Aufrufe (Zei-

le 36) setzt den Link zum zuständigen Benutzer. Dieser ist in der Regel als optionaler, ungebundener Knoten modelliert, muss demnach nicht in der Instanz enthalten sein und kann damit den Wert `null` tragen. Dies stellt hier kein Problem dar, da Instanzen von einwertigen Referenzen in Variablen des zugehörigen Typs gespeichert werden. Diese können beliebige Werte annehmen und dürfen daher auch nicht gesetzt sein. So wird auch hier - im Fall des Fehlens eines passenden Objekts für den optionalen Knoten - der für das Ticket zuständige Nutzer nicht gesetzt.

Darauffolgend werden die Methoden und die - zu Testzwecken vorgesehenen - Standardausgaben, die in der Regel angegeben sind, in beliebiger Reihenfolge ausgeführt. Dabei werden zuerst die Methodenaufrufe, dann die Ausgaben eines einzelnen Knotens betrachtet. Die Beispielregel sieht keine Aufrufe oder Ausgaben vor.

Analog zur Vorbedingung werden - im Beispiel ebenso nicht vorhandende - Nachbedingungen geprüft, die im Falle des Scheiterns zu einer Ausnahme führen. Danach wird der Rückgabewert der Methode betrachtet. Im Beispiel ist dies `void`. Daher muss hier nichts zurückgegeben werden. Hat die modellierte Operation einen ein- oder mehrwertigen Rückgabeparameter, so muss der zugehörige Knoten in der Regel mit << *out* >> markiert sein. Im einwertigen Fall wird das gefundenene Objekt zum Knoten zurückgegeben, im mehrwertigen Fall eine Liste mit passenden Objekten. Ist der Operation eine Map als Rückgabeparameter zugeordnet, so dürfen mehrere Knoten als Rückgabeparameter mit << *out* >> gekennzeichnet werden, die automatisch in die Map eingefügt und zurückgegeben werden. Dies stellt jedoch, wie sich bei der Nutzung von ModGraph gezeigt hat, einen eher unüblichen Sonderfall dar.

Operationen, die Wahrheitswerte zurückgeben, dürfen ebenfalls mit ModGraph-Regeln implementiert werden. Die generierten Methoden geben den (Miss-)Erfolg der Regel, ausgedrückt durch die Rückgabe von `true` oder `false`, zurück.

Aufbau der Utility-Klasse

Jeder Regel ist eine Utility-Klasse zugeordnet. Sie wird nach dem Schema `GTUtil4-<NameDerKlasse><NameDerMethode>` benannt. Für das Beispiel ist dies die in Auflistung 7.6 gezeigte `GTUtil4TicketchangeStatus`-Klasse. Diese wird ins EMF-generierte Utility-Paket generiert. Sie stellt Methoden zur Mustersuche (und damit auch zur Prüfung der negativen Anwendbarkeitsbedingungen) zur Verfügung und erbt weitere Utility-Methoden, wodurch Abhängigkeiten zu ModGraph-Bibliotheken entstehen.

Der Aufbau der `GTUtil`-Klasse richtet sich nach den während der Mustersuche aufgebauten Spannwäldern (siehe Abschnitt 6.2.2). Sie wird genutzt um die entstandenen Wälder auf die Modellinstanz abzubilden. Um dies zu verwirklichen bietet sie eine Map zur Speicherung der gefundenen Objekte an. Zudem werden Listen verwaltet: eine, die zu löschende Objekte verwaltet und eine, die mögliche implizit entfernte Objekte beinhaltet. Die Liste der zu löschenden Objekte enthält am Ende der Suche alle Objekte, die auf zu löschende Knoten der Regel abgebildet wurden. Die Liste der möglicherweise implizit entfernten Objekte enthält alle Objekte, die auf zu erhaltende Knoten abgebildet werden, die ausschließlich durch zu löschende Links mit anderen Knoten der Regel verbunden sind. Dabei könnte es vorkommen, dass das Objekt aus seinem kompletten

```java
/**
* @generated by ModGraph for changeStatus
*/
public class GTUtil4TicketchangeStatus extends GTMatchingUtil {

    List<TicketRevision> alreadyConsideredObjects4oldRevision = new LinkedList<>();
    List<User> alreadyConsideredObjects4assignee = new LinkedList<>();

    public void match(Status status, User author, Ticket ticket) throws GTFailure {
        if(status == null || autor == null || ticket == null)
            throw new GTFailure("Parameter is null");
        map.put("status", status);
        map.put("author", author);
        map.put("ticket", ticket);
        matchUnboundNodes("oldRevision", ticket);
    }

    private void matchUnboundNodes(String searchedNodeName,
                    Object boundPredecessor) throws GTFailure {
        if (searchedNodeName.equals("oldRevision")) {
            TicketRevision oldRevision =
                getOldRevision4ChangeStatus((Ticket) boundPredecessor);
            if (oldRevision == null)
                throw new GTFailure("oldRevision not found.");
            else {
                map.put("oldRevision", oldRevision);
                implicitDeletionCandidates.add(oldRevision);
                matchUnboundNodes("assignee", (Object) map.get("oldRevision"));
            }
        }
        if (searchedNodeName.equals("assignee")) {
            User assignee =
                getAssignee4ChangeStatus((TicketRevision) boundPredecessor);
            if (assignee != null) {
                map.put("assignee", assignee);
            }
        }
    }

    public TicketRevision getOldRevision4ChangeStatus(Ticket ticket)
                                                    throws GTFailure {
        TicketRevision aTicketRevision =
            (TicketRevision) ticket.getCurrentRevision();
        if (!(alreadyConsideredObjects4oldRevision.contains(aTicketRevision))
                && aTicketRevision instanceof TicketRevision
                && ((TicketRevision) aTicketRevision).getStatus() != Status.REJECTED)
            return (TicketRevision) aTicketRevision;
        else
            return null;
    }

    public User getAssignee4ChangeStatus(TicketRevision oldRevision)
                                                   throws GTFailure {
        User aUser = (User) oldRevision.getAssignee();
    if (!(alreadyConsideredObjects4assignee.contains(aUser))
                && aUser instanceof User)
            return (User) aUser;
        else
            return null;
    }
}
```

Auflistung 7.6: *ModGraph-generierter Quelltext der Utility-Klasse zur Unterstützung der Implementierung der modellierten Operation* changeStatus *der Klasse* Ticket *des Bugtracker Ecore-Modells (Abbildung 7.9)*

Kontext - insbesondere aus seiner in EMF notwendigen Containment-Hierarchie - entnommen wird, damit außerhalb der Ressource und daher verloren ist. Ob dieser Fall eintritt, muss zur Laufzeit geprüft werden. Weitere Listen werden genutzt um die bereits betrachteten, jedoch nicht zum vorgegebenen Teilgraphen passenden Objekte zu verwalten. Dabei wird für jeden ungebundenen Knoten eine Liste angelegt. Diese ist insbesondere notwendig, wenn die Mustersuche teilweise rückgängig gemacht werden muss, um nicht nochmals den selben Pfad im Graphen einzuschlagen. Dabei muss für jeden Knoten eine eigene Liste verwaltet werden, da ein für den einen Knoten potentiell fälschlich gefundenes Objekt durchaus auf einen anderen abgebildet werden kann.

Zur Generierung der Methoden, welche die Mustersuche auf der Modellinstanz übernehmen, werden die Bäume der Spannwälder ebenenweise durchlaufen. In Auflistung 7.6 ist die generierte Utilty-Klasse für die Regel `changeStatus` abgebildet. Im ersten Schritt werden die Listen zur Verwaltung potentiell fälschlich gefundener Objekte erzeugt. Da es im Graphmuster von `changeStatus` nur zwei ungebundene existierende Knoten gibt, werden für diese je eine Liste angelegt (Zeilen 6 & 7). Die `match`-Methode koordiniert die Mustersuche. Sie ist in Zeilen 9-16 gezeigt und trägt dieselben Parameter wie die Methode selbst, erweitert durch einen für das aktuelle Objekt. Darin werden die für gebundene Knoten stehenden Aktualparameter der Methode in die Map eingefügt, sofern keinerlei Bedingungen an sie gestellt sind. Hierbei wird lediglich zuvor geprüft, ob die Parameter nicht `null` sind. Dieser Fall tritt auch im Beispiel auf. Sind Bedingungen an diese gestellt, so werden sie in einer eigenen Methode geprüft. Sie können als Attribut- oder OCL-Bedingungen an das Objekt oder zwischen den gebundenen Objekten durch Links oder Pfade angegeben sein. Dabei kann eine Bedingung von einem ungebundenen Objekt abhängen. Diese kann hier nicht geprüft werden, da das ungebundene Objekt zuerst gefunden werden muss. Folglich werden solche Bedingungen während der Suche nach dem ungebundenen Objekt geprüft. Entspricht ein Objekt den Bedingungen, so wird es in die Map zur Speicherung der gefundenen Objekte eingefügt. Zu löschende Objekte werden der entsprechenden Liste zu ihrer Verwaltung hinzugefügt. Zu erhaltende Objekte werden in die Liste der implizit gelöschten Objekte aufgenommen, wenn sie nur durch zu löschende Links mit ihrer Umgebung verbunden sind. Solche Objekte laufen Gefahr, nach Anwendung der Regel nicht mehr erreichbar zu sein. Dies ist der Fall, wenn der Container des Objekts oder die Enthaltenseinsbeziehung des Objekts zu ihm gelöscht wird. Während das Löschen der Enthaltenseinsbeziehung statisch geprüft werden kann, muss das Löschen eines Containers zur Laufzeit überprüft werden. Dementsprechend wird geprüft, ob sich dieser oder einer seiner Container in der Liste der zu löschenden Objekte findet. Wird der Container nicht in der Liste der zu löschenden Objekt gefunden, so wird auch das Objekt nicht gelöscht. Wird einer der Container gefunden, geht das Objekt bei Anwendung der Regel verloren und es wird eine Ausnahme ausgelöst.

Erfüllen die bereits bekannten Objekte alle zu diesem Zeitpunkt prüfbaren Bedingungen, beginnt die Suche nach den ungebundenen Objekten im Graphmuster. Dazu ist eine weitere koordinierende Methode `matchUnboundNodes` nötig. Sie ist für das Beispiel `changeStatus` in Auflistung 7.6 in den Zeilen 18-38 zu sehen. Die Methode sucht rekursiv ausgehend von gebundenen Objekten diejenigen, die auf die Knoten im Spannwald abgebildet werden. Ausgangspunkt ist hierbei das bekannte Objekt, das auf die Wurzel im

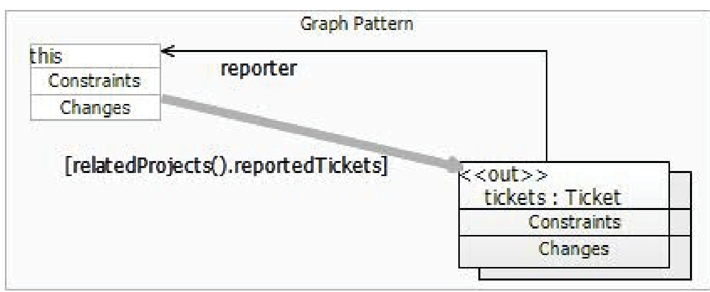


Abbildung 7.10: *Graphtransformationsregel zur Implementierung der Operation* reportedTickets *der Klasse* Ticket *aus dem Bugtacker Beispiel*

ersten Baum abgebildet wird. Es dient als initialer Anker für die Suche. Danach werden Abfragen generiert, die einem ebenenweisen Durchlauf des Waldes (Baum für Baum) entsprechen. Jede dieser Abfragen ruft eine Methode zur Suche nach dem Objekt in der Instanz auf. Diese sucht vom gebundenen Objekt aus eines, das die Bedingungen des zugehörigen Knotens der Regel erfüllt. An dieser Stelle werden auch die übrigen zwischen dem jeweils betrachteten Objekt und gebundenen Knoten bestehenden Bedingungen geprüft. Schlägt dies fehl und entspricht der Elternknoten keinem gebundenen Knoten oder dem aktuellen Objekt im Graphmuster, wird der Elternknoten nochmals gesucht (und unter Umständen auch dessen Elternknoten und wiederum dessen Elternknoten u.s.w.). Da dies in beliebiger Rekursionstiefe möglich ist, kann man hier von Backtracking innerhalb der Mustersuche sprechen. Entspricht der Elternknoten einem gebundenen Knoten im Graphmuster, wird eine Ausnahme ausgelöst. Der gesuchte Kindknoten ist in dieser EMF-Instanz nicht enthalten. Für optionale Knoten werden keine Ausnahmen ausgelöst. Soll von einem optionalen Knoten mindestens ein weiterer gesucht werden und findet dieser keine Übereinstimmung in der Instanz, so kann auch der folgende optionale Knoten (und alle auf ihn folgenden) nicht gefunden werden. Die gefundenen Objekte werden in die Map aufgenommen und - wenn nötig - in einige der Listen. Dabei gelten die bei den gebundenen Objekten dargestellten Kriterien. Zudem werden alle gefundenen Knoten in einer Liste der betrachteten Objekte verwaltet.

Bei der Suche kann es vorkommen, dass ein Knoten nicht über einen Link, sondern über einen Pfadausdruck gesucht werden muss. Dazu wird der Pfadausdruck auf dem Quellknoten ausgewertet.

Im Beispiel wird ausgehend vom bekannten Ticket die - vor den Änderungen aktuelle - Revision gesucht. Dies geschieht aus der `match`-Methode durch Aufruf der rekursiven `matchUnboundNodes`-Methode (Zeile 12). Hier erfolgt der Aufruf der Methode `getOldRevision4ChangeStatus`, die darauf ausgelegt ist, die aktuelle Revision vor den Änderungen unter Beachtung aller Bedingungen zu finden (Zeilen 40-50). Ist dies erfolglos wird eine Ausnahme ausgelöst, da die Suche eindeutig, durch Navigation eines einwertigen Links gescheitert ist. Bei Erfolg, wird das gefundene Objekt in die Map aufgenommen (Zeile 26) und in die Liste der potenziell nicht mehr erreichbaren Objekte (Zeile 27). Daraufhin wird durch direkte Rekursion der Methode der zuständige Nutzer gesucht (Zeile 28). Dazu wird wiederum eine Methode aufgerufen, die den Nutzer - unter

```java
/**
* @generated by ModGraph for reportedTickets
*/
public EList<Ticket> getTickets4ReportedTickets(User user) throws GTFailure {

    List<EObject> allTickets = OCLHandler
        .evaluatePath("relatedProjects().reportedTickets", user);

    if (allTickets == null)
        return null;

    EList<Ticket> tickets = new BasicEList();
    for (EObject _Ticket : allTickets) {
        Ticket aTicket = ((Ticket) _Ticket);

        if (aTicket instanceof Ticket
            && !(alreadyConsideredObjects4tickets.contains(aTicket))
                && ((Ticket) aTicket).getReporter().equals((User) map.get("user"))) {
            tickets.add((Ticket) aTicket);
        }
    }
    if (tickets != null && !(tickets.isEmpty()))
        return (EList<Ticket>) tickets;

    return null;
}
```

Auflistung 7.7: *Utility-Methode zur Suche eines ungebundenen Knotens mit Hilfe eines Pfadausdrucks und anschließendem Filtern des Ergebnisses am Beispiel der Regel* reportedTickets *der Klasse* User *(Abbildung 7.10)*

Beachtung aller Bedingungen - von der aktuellen Version vor den Änderungen aus sucht (Zeilen 52-60). Da dieser durch einen optionalen Knoten modelliert ist, wird er - sollte er gefunden werden - in die Map eingefügt (Zeile 32).

Auflistung 7.7 zeigt einen Ausschnitt der Utility-Klasse zur bereits aus Kapitel 5.4 bekannten Regel reportedTickets (Abbildung 7.10) . Zeilen 6 & 7 bilden den generierten Quelltext für eine Suche über einen Pfadausdruck ab. Das Ergebnis des Pfadausdrucks wird in den folgenden Zeilen der Methode weiter eingeschränkt, indem nur die Objekte berücksichtigt werden, deren Ersteller mit dem gegebenen Benutzer übereinstimmt (Zeilen 13-21).

Sind auf diese Art und Weise mindestens die obligatorischen Knoten der Regel auf die Instanz abgebildet, beginnt die Suche nach den in NACs spezifizierten Teilgraphen. Die Vorgehensweise ist analog zur Mustersuche für das Graphmuster aufgebaut. Für jede NAC übernimmt eine `matchNAC<LaufendeNummerfuerNACs>`-Methode die Koordination der Abbildung des Teilgraphen auf die Instanz. Sie prüft zunächst die gebundenen Parameter und ruft eine rekursive Methode zur Koordination der Abbildung der ungebundenen Knoten auf. Diese ist ähnlich zur `matchUnboundNodes`-Methode aufgebaut. Der bedeutsame Unterschied liegt in der Ausnahmebehandlung. Methoden zur Prüfung der NAC rufen, bei einer erfolgreichen Abbildung der NAC auf die Instanz, nach Möglichkeit die Methoden zur Mustersuche des Graphmusters der Regel auf. Ist eine NAC jedoch alternativlos erfüllt, wird auch hier eine Ausnahme ausgelöst. Alternativlos bedeutet, dass die erneut aufgerufene Mustersuche nicht mehr erfolgreich sein kann. Dies ist beispiels-

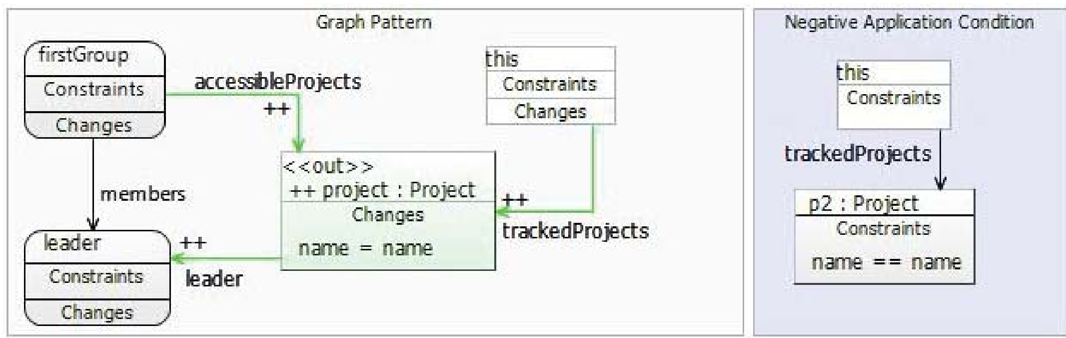


Abbildung 7.11: *Graphtransformationsregel zur Implementierung der Operation* createProject *der Klasse* BugTracker *aus dem Beispiel*

weise der Fall, wenn ein ungebundener Knoten der NAC direkt ausgehend von einem Parameter oder dem aktuellen Objekt gefunden wird. Das Auslösen der Ausnahme aus der NAC ist damit lediglich eine Abkürzung.

Der Quelltext zur Untersuchung der NACs ist am Beispiel der generierten Utility-Methoden der, ebenfalls bereits in Kapitel 5.4 besprochenen, Regel createProject (Abbildung 7.11) in Auflistung 7.8 dargestellt. Diese Utility-Klasse enthält zudem eine Methode zur Prüfung der Bedingung zwischen den beiden Parametern. Zuerst wird hier eine Liste zur Speicherung fälschlich gefundener Objekte zum Knoten *p2* erzeugt (Zeile 6). Die koordinierende `match`-Methode (Zeilen 8-17) legt zunächst alle Parameter ohne Bedingungen in der globalen Map ab (Zeilen 10-14) und ruft die Methode zur Prüfung der bedingten Parameter (Zeile 15) auf. Diese prüft, ob der Gruppenleiter `leader` ein Mitglied der Gruppe ist und fügt ihn, wenn die Bedingung erfüllt ist, in die Map ein (Zeilen 19-25). Ist die Bedingung nicht erfüllt, wird eine Ausnahme ausgelöst. Da im Graphmuster der Regel keine existierenden ungebundenen Knoten auftreten, fehlt in der match-Methode der Aufruf der Methode zum Finden der ungebundenen Knoten, genauso wie die Methoden zur Mustersuche der ungebundenen Knoten selbst. Die match-Methode ruft direkt die Suche nach der NAC auf (Zeile 16), indem die nac1Matched-Methode (Zeilen 27-29) aufgerufen wird. Da in der NAC Bedingungen an gebundene Knoten gestellt werden, wird direkt die rekursive matchUnboundNodesInNAC1-Methode aufgerufen. Diese Methode führt die Mustersuche auf der NAC - analog zu der bereits beschriebenen Suche im Graphmuster, aber mit anderer Ausnahmebehandlung - aus (Zeilen 31-41). Sie nutzt die getP24CreateProject-Methode, um das Projekt im Bugtracker zu finden (Zeilen 43-57). Wird es nicht gefunden, ist die Mustersuche abgeschlossen. Existiert das Projekt jedoch, ist die NAC erfüllt. Da das Bugtracker-Objekt einem gebundenen Knoten entspricht, wird zur Verringerung der Laufzeit nicht die Mustersuche von neuem aufgerufen, sondern die unausweichliche Ausnahme sofort ausgelöst.

Nach der Generierung der Utility-Klasse, ist ausführbarer Quelltext erzeugt, der neben der Erzeugung von Objekten aus Klassen und deren Initialisierung auch die Ausführung modellierter Operationen erlaubt. Er kann beliebig erweitert und getestet werden. Im Falle des Bugtracker-Projekts existieren eine Reihe von JUnit-Tests. Diese können, zu-

```java
/**
* @generated by ModGraph for createProject
*/
public class GTUtil4BugTrackercreateProject extends GTMatchingUtil {

    List<Project> alreadyConsideredObjects4p2 = new LinkedList<>();

    public void match(java.lang.String name, User leader, Group firstGroup,
            BugTracker bugTracker) throws GTFailure {
        if(name == null || leader == null || firstGroup == null ||bugTracker == null)
            throw new GTFailure("Parameter is null");
        map.put("name", name);
        map.put("firstGroup", firstGroup);
        map.put("bugTracker", bugTracker);
        checkBoundNodes(name, leader, firstGroup, bugTracker);
        nac1Matched(bugTracker);
    }

    private void checkBoundNodes(java.lang.String name, User leader,
            Group firstGroup, BugTracker bugTracker) throws GTFailure {
        if (firstGroup.getMembers().contains(leader)) {
            map.put("leader", leader);
        }
        else throw new GTFailure("Parameter constraint violated!");
    }

    public void nac1Matched(BugTracker bugTracker) throws GTFailure {
        matchUnboundNodesInNAC1("p2", (EObject) map.get("bugTracker"));
    }

    private void matchUnboundNodesInNAC1(String searchedNodeName,
            EObject boundPredecessor) throws GTFailure {

        if (searchedNodeName.equals("p2")) {
            Project p2 = getP24CreateProject((BugTracker) boundPredecessor);
            if (p2 != null) {
                alreadyConsideredObjects4p2.add(p2);
                throw new GTFailure("p2  found.");
            }
        }
    }

    public Project getP24CreateProject(BugTracker bugTracker) throws GTFailure {
        EList<Project> trackedProjects = new BasicEList<Project>();
        trackedProjects.addAll(((BugTracker) map.get("bugTracker"))
                .getTrackedProjects());
        for (int i = 0; i < trackedProjects.size(); i++) {
            Project aProject = trackedProjects.get(i);
        if (!(alreadyConsideredObjects4p2.contains(aProject))
                && aProject instanceof Project
                && ((Project) aProject).getName() ==
                        (java.lang.String) map.get("name")) {
                return (Project) aProject;
            }
        }
    return null;
    }
}
```

Auflistung 7.8: *Utility-Methode zur Suche eines Graphmusters, welches mit einer NAC eingeschränkt ist, am Beispiel der Regel* createProject *(Abbildung 7.11)*

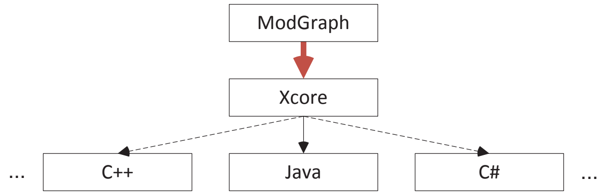


Abbildung 7.12: *Schrittweiser Ansatz zur Generierung von Quelltext*

sammen mit dem ausführbaren Bugtracker, auf der ModGraph-Homepage[4] heruntergeladen werden.

7.2.2 Einbinden der ModGraph-Regeln in ein Xcore-Modell

In Abschnitt 5.4 wurde bereits das Xcore-Modell des Bugtrackers generiert und manuell erweitert. Außerdem können Xcore-Modelle auch durch das automatisierte Einfügen einer Regel erweitert werden. Die Einbindung der Regeln in ein Xcore-Modell erfolgt mit ModGraph's Xcore Compiler. Dieser generiert aus regelbasierten ModGraph-Regeln prozedurale Xcore-Operationen, die ein bestehendes Xcore-Modell direkt und nahtlos erweitern. Er ist in [81] veröffentlicht.

Die Transformation der Regeln nach Xcore stellt eine Modell-zu-Modell-Transformation von Transformationen dar und ist somit eine Transformation höherer Ordnung [75]. Diese könnte alternativ auch mit ATL oder QVT realisiert werden (siehe Kapitel 4.1). Da es sich bei Xcore jedoch um ein textuelles Modell handelt, und ModGraph bereits Java-Quelltext generiert, wurde von dieser Lösung abgesehen. Stattdessen wird eine templatebasierte Modell-zu-Text-Transformation realisiert, indem analog zum Java-Compiler ein Compiler erstellt wird, der die Regeln in Xcore-Code übersetzt, der in das Xcore-Modell eingefügt wird und damit zu einem Modellbestandteil wird. Dieser Ansatz wird genutzt, da er konziser, lesbarer und einfacher zu verwirklichen ist. Statt komplexe Transformationen über der abstrakten Syntax aufzubauen, wird eine Regel direkt in Xcores konkrete Syntax übersetzt.

Die Regeln selbst sind endogene, überschreibende Transformationseinheiten für Xcore-Modellinstanzen. Die Generierung von Quelltext einer Programmiersprache ist dabei Xcore überlassen. Es handelt sich hierbei um einen schrittweisen Übersetzungsprozess. Dieser ist in Abbildung 7.12 dargestellt. Neben der existierenden Java-Generierung ist - sofern Xcore dies zukünftig unterstützt - auch die Erzeugung von Quelltext in einer beliebigen anderen Programmiersprache möglich. Die Übersetzung der Regeln nach Xcore bringt zudem eine neue Ausführungsmöglichkeit mit: Interpretierbarkeit. Xcore-Modelle können interpretiert werden und damit ebenfalls die - aus der Regel generierten - Xcore-Operationen, deren Rumpf mit Xbase spezifiziert wird. Damit ist eine Ausführung auf Modellebene (wie im „unabhängigen Weg“ in Kapitel 7.1 beschrieben) erreicht. Die genaue Arbeitsweise der Kompilierung einer Regel nach Xcore wird im Folgenden betrachtet.

[4] `http://btn1x4.inf.uni-bayreuth.de/modgraph/homepage`

```
package de.ubt.ai1.modgraph.bugmodel

//Importe...

annotation "http://www.eclipse.org/emf/2002/Ecore/OCL/Pivot" as OCL
annotation "http://www.eclipse.org/emf/2002/Ecore" as Ecore
annotation "http://www.eclipse.org/emf/2002/GenModel" as GenModel

@GenModel(loadInitialization="true",operationReflection="true",
modelDirectory="/[PathToProject]/src")
@Ecore(invocationDelegates="http://www.eclipse.org/emf/2002/Ecore/OCL/Pivot",
settingDelegates="http://www.eclipse.org/emf/2002/Ecore/OCL/Pivot",
validationDelegates="http://www.eclipse.org/emf/2002/Ecore/OCL/Pivot")

//Klassen etc. ...
```

Auflistung 7.9: *Definition der Annotationen im Xcore-Modell*

Annotation des Xcore-Modells

Um die ModGraph-Regeln korrekt nach Xcore übersetzen zu können, muss das Modell um OCL-Unterstützung und einige Generator-Modell-Annotationen erweitert werden. Diese sind in Auflistung 7.9 gezeigt. Die Annotationen stellen sicher, dass der Xcore-Generator mit den korrekten Parametern arbeitet. Sie werden genau einmal in den Kopf des Xcore-Modells eingefügt und sind für jedes Modell gleich. Zeilen 5-7 der Auflistung definieren die Annotationen und ihre jeweiligen Xcore-Alias. Der Alias dient der Vereinfachung der Nutzung der Annotationen, indem durch ihn direkt auf die Annotation zugegriffen werden kann. Wie beispielsweise in Zeilen 9 & 10 gezeigt, kann durch Nutzung des Alias, hier `@GenModel`, die Annotation direkt angesprochen werden. Sie wird verwendet um Eigenschaften des Generatormodells zu setzen. Dabei wird unnötiges Nachladen des Modells vermieden und das Zielverzeichnis der Generierung auf den `src`-Ordner des aktuellen Eclipse-Projekts festgelegt. Zudem wird die reflektive Operationsausführung erlaubt. Sie wird benötigt, um die OCL-Unterstützung zu realisieren. Bezüglich OCL werden außerdem die von EMF bereitgestellten Pivot-Evaluatoren eingebunden, siehe Zeilen 5 und 11-13 der Auflistung. Diese wurden aufgrund ihrer strikten Konformität zu den OMG-Standards gewählt.[5]

Annotation der Operation

In diesem Schritt wird zunächst das Xcore-Modell unter Zuhilfenahme seines abstrakten Syntaxbaumes durchsucht, bis die modellierte Operation gefunden ist. Zur Umsetzung der Regel wird diese vom ModGraph-Generator - abhängig von ihrem Inhalt - annotiert. Eine Generator-Modell-Annotation wird immer erzeugt. Diese wird zu Dokumentationszwecken verwendet, indem sie die Operation als mit ModGraph generiert markiert. Zudem wird hier der zur Regel angegebene Kommentar eingefügt. Sind in der Regel Vor- oder Nachbedingungen angegeben die in OCL verfasst wurden, so wird für jede Bedingung eine zusätzliche `OCL`-Annotation benötigt. Betrachtet man die Um-

[5] Näheres zur OMG-Konformität findet sich in der Eclipse Hilfe unter `http://help.eclipse.org/kepler/index.jsp` unter der Rubrik OCL

```
...
class Ticket {
...
    @GenModel(documentation="Generated by ModGraph: Ändert
        den Status eines Tickets.")
    @OCL(pre_pre1="author.accessibleProjects()->includes(self.owningProject)")
    op void changeStatus(Status status , User author) throws GTFailure {
        var TicketRevision oldRevision = null
        var User assignee = null
        var _oldRevision = currentRevision
        var _assignee = oldRevision.assignee
        oldRevision = _oldRevision
        assignee = _assignee
        if(oldRevision == null)
            throw new de.ubt.ai1.modgraph.gt.failure.GTFailure("Object not found")
        val newRevisionTitleValue = currentRevision.title
        val newRevisionDescriptionValue = currentRevision.description
        val newRevisionSeverityValue = currentRevision.severity
        val newRevisionStatusValue = status
        val newRevisionCreatedAtValue = java.util.Calendar.getInstance()

        var newRevision = BugmodelFactory::eINSTANCE.createTicketRevision()

        newRevision.title = newRevisionTitleValue
        newRevision.description = newRevisionDescriptionValue
        newRevision.severity = newRevisionSeverityValue
        newRevision.status = newRevisionStatusValue
        newRevision.createdAt = newRevisionCreatedAtValue

        currentRevision = newRevision
        newRevision.creator = author
        history.add(newRevision)
        newRevision.assignee = assignee
    } ...
}
...
```

Auflistung 7.10: *Ausschnitt aus dem Bugtracker Xcore-Modell: Die generierte Methode zur Regel* changeStatus *(Abbildung 7.9)*

setzung der bekannten Regel changeStatus, so wird diese zunächst mit der Generator-Modell-Annotation versehen, die in Auflistung 7.10 in Zeilen 4 & 5 gezeigt ist. Der zur Regel angegebene Kommentar wird nach der Anmerkung der ModGraph-Generierung der Regel eingefügt. Da die Regel mit einer OCL-Vorbedingung versehen ist, wird diese in eine OCL-Annotation der Operation übersetzt, die in Zeile 6 der Auflistung 7.10 gezeigt wird.

Aufbau des Operationsrumpfes

In den Rumpf der Operation wird die Ausführungslogik der Regel, wie in Kapitel 6.2 beschrieben, eingefügt und dazu mit Xbase spezifiziert. In einem ersten Schritt wird die Methodendeklaration - sofern noch nicht geschehen - um die Ausnahmebehandlung erweitert. Danach werden die in Xcore bzw. Xbase spezifizierten Vorbedingungen der Regel geprüft. Diese sind im Beispiel changeStatus nicht vorhanden. Daher beginnt das Beispiel mit dem nächsten Schritt: Es werden alle zur Ausführung der Regel benötigten Variablen definiert. Dies ist in Zeilen 8 & 9 zu sehen. Im Folgenden werden die Xbase-Operationen

für die Mustersuche generiert. Dazu werden zunächst Hilfsvariablen angelegt, die in Zeilen 10 & 11 definiert sind. Diese dienen der temporären Ablage der über einen Link oder einen Pfad gefundenen Objekte, die unter Umständen durch Bedingungen weiter eingeschränkt werden. Dies ist im Beispiel nicht der Fall. Es zeigt den einfachsten Fall der Mustersuche. Beide Objekte, die durch die ungebundenen Knoten repräsentiert sind, werden durch einen eine einwertige Referenz instanziierenden Link gefunden und enthalten keinerlei weitere Bedingungen. Damit werden die temporären Werte direkt an die tatsächlichen zugewiesen (Zeilen 12 & 13). Im Falle einer komplexeren Mustersuche werden geschachtelte Schleifen generiert. Diese prüfen alle Bedingungen an die Objekte. Die Schachtelung der Schleifen spiegelt dabei den bei der Mustersuche gefundenen Baum wieder. Die erste Schleife sucht - vom gebundenen Knoten aus - die direkt erreichbaren Objekte, von welchen aus durch weitere verschachtelte Schleifen die jeweiligen Kindknoten gefunden werden. Die Variable enthält hierbei den letzten passenden Fund. Beispiele hierzu finden sich in [81], das eine andere Version des Modell-Refacorings betrachtet. In diesem sind die Regeln verändert, so dass sie zu einer komplexen Mustersuche führen. Alternativ können Objekte auch durch die Auswertung eines Pfades gefunden werden. In Auflistung 7.11 werden die Xcore Umsetzungen der Regeln reportedTickets (siehe Abbildung 7.10) und createProject (siehe Abbildung 7.11) dargestellt. Die Zeilen 7 & 8 in Auflistung 7.11 zeigen die Auswertung der Pfades in reportedTickets. Der Pfadausdruck muss hierbei in der Regel durch (OCL) eingeleitet werden, da der ModGraph-Generator bei der Xcore-Generierung zunächst von Xbase-Ausdrücken an den Pfaden ausgeht. Hier ist ebenso eine Schleife mit Bedingung zur Prüfung der - in der Regel durch den Link reporter gegebenen - Zusatzbedingung gezeigt (Zeilen 9-12). Die Prüfung einer NAC wird durch Bedingungen innerhalb der innersten, zur Mustersuche genutzten, Schleife realisiert. Sind keine Schleifen nötig, steht sie gesondert, wie auch in Auflistung 7.11 Zeile 24. Hier findet die Prüfung der NAC direkt nach der Bedingung an die Parameter der Regel statt. In beiden Fällen wird der funktionale Aufruf findFirst verwendet. Zuerst wird geprüft, ob der übergebene Nutzer Mitglied der Gruppe ist. Danach wird sichergestellt, dass bei Anlegen eines Projekts keines gleichen Namens existiert.

Wird ein Objekt nicht gefunden, muss eine Ausnahme des Typs GTFailure ausgelöst werden. Sind alle Objekte gefunden, werden die Attributwerte aus dem Vorzustand des Modells berechnet und in finalen Variablen, die in Xbase durch val eingeleitet werden, gespeichert. Betrachtet man wieder Auflistung 7.10, sind diese in Zeilen 16 - 20 zu sehen. Es werden alle später an das neu erzeugte Objekt des Typs TicketRevision zuzuweisenden Attributwerte berechnet. Ihre Benennung erfolgt wiederum nach dem Objektbezeichner und dem Attributnamen ergänzt durch Value. Darauf werden die zu löschenden Objekte gelöscht, die zu verändernden, erhaltenen Objekte verändert und die zu erstellenden erzeugt. Im Beispiel wird die neue Revision des Tickets durch den Aufruf einer passenden - von Xcore im Hintergrund generierten - Fabrikmethode erstellt (Zeile 22) und deren Attribute initialisiert (Zeilen 24-28). Dazu werden die im Vorangegangenen berechneten Attributwerte genutzt. Im nächsten Schritt werden die Links - wie in der Regel angegeben - gelöscht oder erzeugt. Hierbei gilt dieselbe Sonderregelung wie bei der Java-Generierung: Einwertige Links, die in derselben Regel gelöscht und an anderer Stelle erzeugt sind, werden nicht explizit gelöscht. Im Beispiel sind dies Links, welche die neue

```
...
class User { ...

    @GenModel(documentation="Generated by ModGraph: ")
    op unique Ticket[] reportedTickets() {
        var Ticket[] tickets = null
        var _tickets = OCLHandler.evaluatePath(this,
                "relatedProjects().reportedTickets") as  Ticket[]
        for(ticket : _tickets){
            if(ticket.reporter == this)
                tickets.add(ticket)
        }
        if(tickets.isEmpty()) throw new de.ubt.ai1.modgraph.gt.failure.GTFailure
        return tickets as EList
    }...
}
...
class BugTracker { ...

    @GenModel(documentation="Generated by ModGraph: ")
    op Project createProject(String name , User leader ,
            Group firstGroup) throws GTFailure {
        if(firstGroup.members.findFirst(m| m == leader)!=null){
            if (!(trackedProjects.findFirst(e|e.name == name) != null)) {
                val projectNameValue = name
                var project = BugmodelFactory::eINSTANCE.createProject()
                project.name = projectNameValue
                trackedProjects.add(project)
                project.leader = leader
                firstGroup.accessibleProjects.add(project)
                return project
            } else throw new de.ubt.ai1.modgraph.gt.failure.GTFailure
        } else throw new de.ubt.ai1.modgraph.gt.failure.GTFailure
    }...
}
```

Auflistung 7.11: *Ausschnitt aus dem Bugtracker Xcore-Modell: Die generierten Methoden zu den Regeln* reportedTickets *(Abbildung 7.10) und* createProject *(Abbildung 7.11)*

Ticketrevision in ihren Kontext einbetten (Zeilen 30-33). Danach werden die Operationen aufgerufen und die Xcore-Nachbedingungen geprüft. (Die OCL-Nachbedingungen befinden sich in einer Annotation der Operation.) Zuletzt wird der Rückgabewert der Operation zurückgegeben. Da das Beispiel changeStatus keine Nachbedingung und keinen Rückgabewert hat, endet die Operation nach der Erstellung der Links.

Damit ist die Xcore-Generierung abgeschlossen und ein ausführbares Xcore-Modell wurde erzeugt. Es kann nun interpretiert und zur Generierung von Quelltext verwendet werden. Der von Xcore generierte Quelltext besitzt grundsätzlich die gleiche Struktur wie der EMF-generierte Quelltext. Er ist beispielhaft für die Operation createProject in Auflistung 7.12 gezeigt. Das Beispiel wurde gewählt, da an ihm im Folgenden die Umsetzung des funktionalen Ausdrucks findFirst gezeigt werden kann.

```
public Project createProject(final String name, final User leader,
        final Group firstGroup) throws GTFailure {
    EList<User> _members = firstGroup.getMembers();
    final Function1<User, Boolean> _function = new Function1<User, Boolean>() {
        public Boolean apply(final User m) {
            return Boolean.valueOf(Objects.equal(m, leader));
        }
    };
    User _findFirst = IterableExtensions.<User>findFirst(_members, _function);
    boolean _notEquals = (!Objects.equal(_findFirst, null));
    if (_notEquals) {
            EList<Project> _trackedProjects = this.getTrackedProjects();
        final Function1<Project, Boolean> _function_1 =
                new Function1<Project, Boolean>() {
            public Boolean apply(final Project e) {
                String _name = e.getName();
                return Boolean.valueOf(Objects.equal(_name, name));
            }
        };
        Project _findFirst_1 =
                IterableExtensions.<Project>findFirst(_trackedProjects, _function_1);
        boolean _notEquals_1 = (!Objects.equal(_findFirst_1, null));
        boolean _not = (!_notEquals_1);
        if (_not) {
            final String projectNameValue = name;
            Project project = BugmodelFactory.eINSTANCE.createProject();
            project.setName(projectNameValue);
            EList<Project> _trackedProjects_1 = this.getTrackedProjects();
            _trackedProjects_1.add(project);
            project.setLeader(leader);
            EList<Project> _accessibleProjects = firstGroup.getAccessibleProjects();
            _accessibleProjects.add(project);
            return project;
        } else { throw new GTFailure(); }
    }else { throw new GTFailure(); }
}
```

Auflistung 7.12: *Von Xcore generierter Java-Quelltext zur Implementierung der modellierten Operation* createProject *(Abbildung 7.11) der Klasse* BugTracker *des Bugtracker-Xcore-Modells*

7.2.3 Vergleich der Modelle und Quelltexte

In diesem Abschnitt werden die in ModGraph erstellten Modelle und Quelltexte verglichen. Dazu werden verschiedene Implementierungen der Regeln changeStatus (Abbildung 7.9) und createProject (Abbildung 7.11) betrachtet.

ModGraph-Regel und Xcore

Stellte man der ModGraph-Regel changeStatus (Abbildung 7.9) die Xcore-Operation (Auflistung 7.10) gegenüber, so bleibt die Regel intuitiver als der generierte Quelltext. Sie ist klar strukturiert und weist immer denselben Aufbau auf. Zudem bietet sie durch ihre graphische Darstellung und die Farbkodierung der Knoten und Kanten eine intuitive Visualisierung des Teilgraphen und der Änderungen an diesem.

Das Xcore-Modell ist klar strukturiert, textuell und unabhängig von der Zielsprache. Außerdem ist es konzise und einfach genug, um für den Modellierer gut lesbar zu sein.

Dies macht sich insbesondere bemerkbar, sobald die von Xbase angebotenen funktionalen Ausdrücke zum Einsatz kommen.

Vergleicht man konkret die Regel mit dem generierten Xcore-Modell, so fällt zunächst auf, dass das Xcore-Modell als textuelles Modell gelesen werden muss, um eine Idee seiner Funktionalität zu bekommen, wohingegen die Regel schneller zu erfassen ist. Zudem ist im Xcore-Modell bereits die Mustersuche explizit angegeben (Zeilen 8-15 in Auflistung 7.10) und die Attributwerte werden in den folgenden Zeilen aus dem Vorzustand berechnet. Beides gibt die Regel implizit vor. Zusätzlich sei bemerkt, dass die Xbase-Spezifikation für die Mustersuche - wäre sie per Hand geschrieben - etwas kürzer gefasst werden könnte. Zeilen 8-16 könnten in zwei Zeilen komprimiert werden. Dies ist jedoch nur der Fall, solange keine zusätzlichen Bedingungen an die Knoten gestellt sind: Da der Generator allgemeinen Quelltext generiert, werden Variablendeklarationen definiert, die erst nach erfolgreicher Mustersuche zugewiesen werden. Damit wird zweckmäßig der Erfolg der Mustersuche geprüft. Ist sie erfolglos, bleiben die Variablen, welche die endgültigen Objekte repräsentieren, null. Diese Art der Zuweisung deckt den allgemeinsten Fall ab, in welchem die Mustersuche in (unter Umständen geschachtelten) Schleifen stattfindet. Das ist weder überraschend noch unüblich. Trotzdem bleibt das Xcore-Modell gut lesbar, was für die Fehlersuche durch Debuggen im Vergleich zum generierten Quelltext sehr von Vorteil ist.

Die weiteren Zeilen 22-33 in Auflistung 7.10 der Xcore-Implementierung der Operation bilden die Regel geordnet und relativ direkt ab. Lediglich der Umweg über die berechneten Attributwerte muss in Kauf genommen werden. Für das Attribut title der Regel bedeutet dies, dass die Zuweisung in der Regel sich in zwei Zuweisungen der Xcore-Operation aufspaltet: die Wertberechnung (Zeile 16) und die Zuweisung dieses Wertes (Zeile 24).

Betrachtetet man die Regel createProject (Abbildung7.11) und deren Xcore-Implementierung in Auflistung 7.11, so scheint die Abbildung der Regel auf Xcore nahezu direkt. Zunächst wird der Link zwischen den beiden Parameterknoten durch eine Bedingung ausgedruckt (Zeile 13). In einer weiteren Bedingung folgt die Prüfung der NAC (Zeile 24). Die Erzeugung des Projekts und seines Kontexts erfolgt, wenn beide Bedingungen erfüllt sind. Da hier keine aufwändige Mustersuche notwendig ist, wirkt die generierte Xcore-Operation übersichtlich und gut lesbar, dennoch bleiben die oben genannten Argumente auch hier gültig. So muss sich die Xcore-Operation, im Gegensatz zur Regel, mit der Fehlerbehandlung und der genauen Spezifikation der Fabrikmethode zur Erzeugung des Projekts beschäftigen, was bei Modellierung der Regel implizit geschieht.

Xcore und daraus generierter Java-Quelltext

Vergleicht man den gesamten von Xcore generierten Quelltext zur Operation `createProject(...)` der Klasse `BugTracker` aus Auflistung 7.12 mit dem Xcore-Modell, so erscheint dieser unübersichtlicher und länger, aber dennoch (im unteren Teil, der die Zuweisungen enthält) recht ähnlich. Werden keine Xbase-spezifischen funktionalen Lambda-Ausdrücke verwendet, ist der Quelltext dem Modell umso ähnlicher. Diese stellen also einen echten Abstraktionsgewinn gegenüber Java dar. Der hier betrachtete findFirst-

Aufruf ist ein Einzeiler im Xcore-Modell (Auflistung 7.11, Zeilen 23 oder 24). Jeder dieser Einzeiler sorgt in Java für den Aufruf einer Funktion (Auflistung 7.12, Zeilen 9 bzw. 20 & 21), die eigens zuvor definiert wird (Zeilen 4-8 bzw. 13-19 der Auflistung). Hier wird der Vorteil von Xcore mit Xbase gegenüber Java deutlich.

Zudem werden die - in Xcore üblichen - verkürzten Schreibweisen korrekt in Java-Akzessoren übersetzt. Leider ist die OCL-Unterstützung in Xcore nicht ausgereift.[6] Aufgrund dessen und zu Gunsten einer Verbesserung der Laufzeiten (siehe Kapitel 10.1.1) kann nur geraten werden, die Vor- und Nachbedingungen in Xcore bzw. Xbase zu definieren. Dieser Vorschlag wird im Weiteren ebenso verfolgt und findet in Kapitel 8 Verwendung.

Von ModGraph generierter Java-Quelltext

Der aus einer Regel direkt generierte Quelltext steht nicht in Konkurrenz zum Xcore-Modell. Er stellt eine alternative Verwendung dar.

Vergleicht man den generierten Java Quelltext zur Regel changeStatus (Abbildung 5.8) und den daraus generierten Java-Code (Auflistungen 7.3, 7.4 und 7.6), so ist der Quelltext, bereits durch die Dreiteilung unübersichtlicher und länger als eine Regel. Zudem befindet er sich auf einer deutlich niedrigeren Abstraktionsebene. Die Mustersuche und die Fehlerbehandung müssen beispielsweise explizit erfolgen.

Vergleicht man den ModGraph-generierten Quelltext und das Xcore-Modell (Auflistung 7.10), so findet sich dieser (ebenso wie der direkt aus dem Xcore-Modell generierte) auf einer niedrigeren Abstraktionsebene, da auch er keine abstrakten Sprachkonstrukte wie Lambda-Ausdrücke anbieten kann.

Bei Betrachtung des Xcore-generierten (Auflistung 7.12) und des von ModGraph erstellten Quelltexts, ist der aus Xcore erzeugte zwar auf eine Klasse bzw. eine Operation beschränkt, bleibt für den Nutzer jedoch wesentlich schlechter lesbar. Grund hierfür ist die Nutzung zahlreicher Hilfsvariablen und -funktionen, beispielsweise um Negationen auszudrücken. Der ModGraph-generierte Quelltext ist geradliniger. Er übersetzt eine Regel in Methoden, die größtenteils auf der Java-Standardbibliothek aufbauen. Lediglich im für die Mustersuche generierten Code finden sich wenige Abhängigkeiten zu ModGraph-Bibliotheken. Damit bleibt er recht gut lesbar.

Fazit

Zusammenfassend unterstützen diese Aussagen die Grundideen des ModGraph-Ansatzes. Die Graphtransformationsregel bietet die höchste Abstraktionsebene und sollte zur Spezifikation von komplexen strukturellen Änderungen genutzt werden. Hier bietet sie einen echten Mehrwert. Das Xcore-Modell ist, wie die Regel selbst, konzise, gut lesbar und einfach zu verstehen. Dennoch bietet es eine etwas geringere Abstraktionsebene als eine

[6]Diese Bemerkung bezieht sich auf Eclipse Kepler SR2. Da der Generator lediglich genutzt und nicht im Rahmen von ModGraph entwickelt, wird kann nur gehofft werden, dass die Entwickler von Xcore dies schnellstmöglich beheben.

ModGraph-Regel, da die Mustersuche explizit berücksichtigt werden muss. Das Xcore-Modell ist zudem portabel, da Xcore in beliebige Programmiersprachen übersetzt werden könnte. Der Java-Quelltext, egal ob er von Xcore oder direkt aus der Regel generiert ist, bleibt nach wie vor auf niedrigster Abstraktionsebene. Zwar ist insbesondere der von ModGraph direkt aus der Regel generierte Quelltext für den Programmierer lesbar und verständlich, dennoch bietet er nicht die Übersichtlichkeit der Modelle an. Damit ist der schrittweise Transformationsansatz eine gute Lösung.

7.3 Eclipse-Integration

Die vollständig definierte Syntax der Sprache der ModGraph-Graphtransformationsregeln und deren Einbindung in das EMF-Rahmenwerk - in Verbindung mit ineinandergreifenden Generatoren - erlaubt nun die Anwendung der Sprache zur Spezifikation und Ausführung der Regeln. Dazu bietet ModGraph vorkonfigurierte ModGraph-Projekte, Cheat Sheets und eine eigene Perspektive an, welche die Benutzung des Editors für Graphtransformationsregeln unterstützen. Da ModGraph für und mit EMF arbeitet, sind die Projekte - als Teil einer Reihe von speziell für ModGraph entwickelten Plugins - in Eclipse eingebettet. Die dadurch entstandene Entwicklungsumgebung für Graphtransformationsregeln ist in [21] und [78] veröffentlicht und wird im Folgenden erläutert.

7.3.1 Die ModGraph-Perspektive

Perspektiven in Eclipse sind auf deren Anwendungsbereich zugeschnittene Anordnungen einzelner Editoren und Ansichten innerhalb der Entwicklungsumgebung. Die prominenteste Eclipse-Perspektive ist die Java-Perspektive, welche auf kompakte Weise die zur Java-Entwicklung nötigen Informationen darstellt. Beispielsweise wird hier, neben dem Quelltext, der Aufbau der Klasse als Baumstruktur in einer Ansicht, genannt Outline, angezeigt.

Eine solche Perspektive ist auch für die Entwicklung mit Graphtransformationsregeln sinnvoll. Sie vereinfacht die Arbeit mit den Modellen, indem sie die betrachteten Modelle übersichtlich gegliedert und geordnet nach ihren Zusammenhängen darstellt. Die ModGraph-Perspektive ist durch die Kombination von Ansichten und Editoren für Graphtransformationsregeln in Eclipse umgesetzt. Abbildung 7.13 zeigt deren Aufbau. Neben dem Package-Explorer, der links außen in der Abbildung zu sehen ist und mit 1 markiert wurde, findet sich der mit 2 markierte Editor für Graphtransformationsregeln. Dieser bietet alle in Abschnitt 6.1.2 definierten Elemente der konkreten Syntax zur Modellierung der Regeln an. Er ist gleichzeitig das Hauptfenster der ModGraph-Perspektive. Seine weitere Funktionalität wird im nächsten Abschnitt beschrieben. Rechts neben dem Editor befindet sich ein Kommentarreiter (markiert mit 3), der den Kommentar zur Regel speichert. Dieser teilt sich die Position mit der aus der Java-Perspektive entliehenen Outline. Sie wird genutzt, um die Struktur des Xcore-Modells abzubilden oder auch diejenige der generierten Java-Klassen. Unterhalb der drei genannten Komponenten der Perspektive befindet sich das mit 4 markierte ModGraph-Dashboard. Es stellt eine Beson-

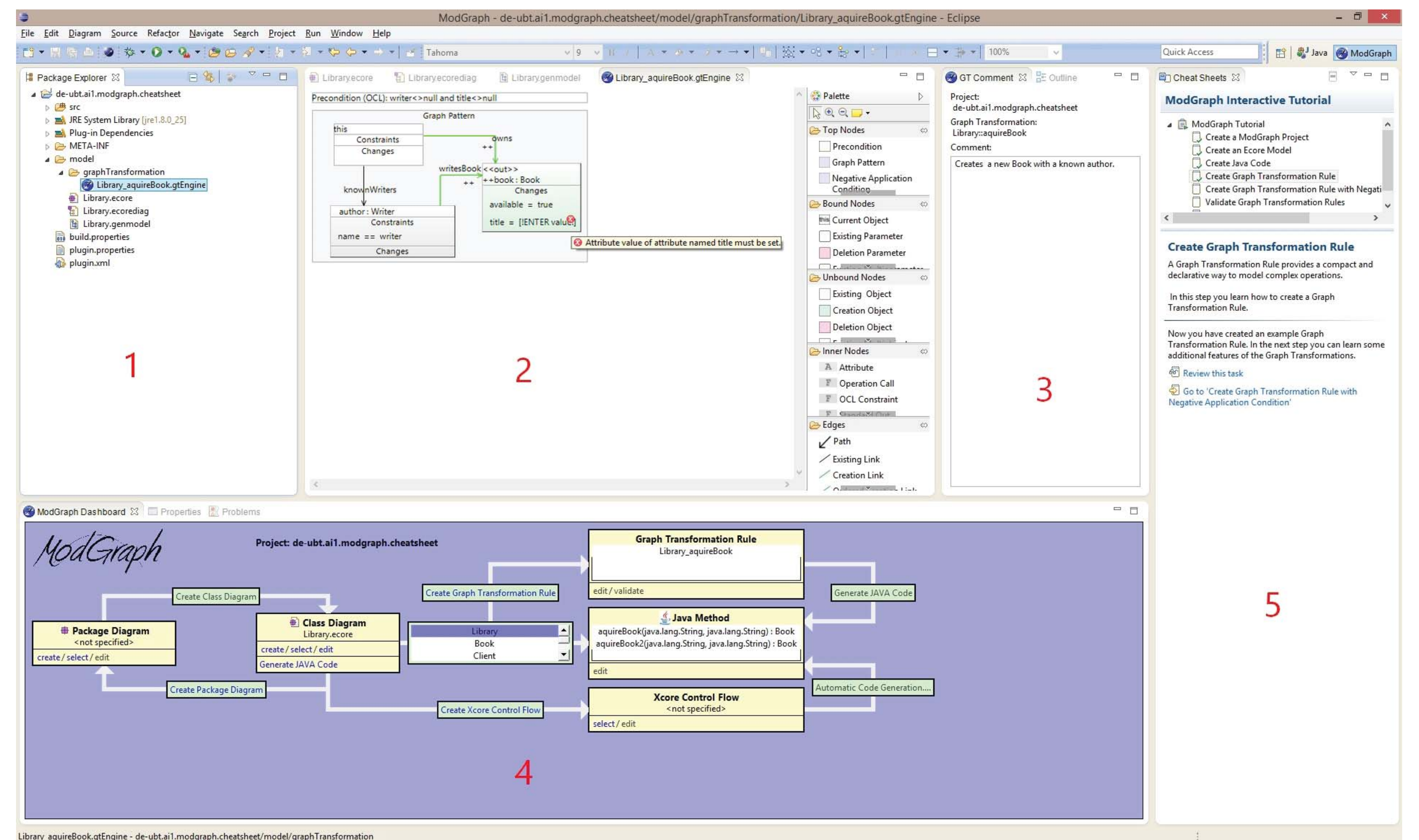


Abbildung 7.13: *Die ModGraph-Perspektive in Eclipse*

derheit der Perspektive dar, indem es einerseits eine Übersicht des Entwicklungsstands und andererseits eine Hilfestellung bei der Entwicklung bietet. Es wird in Abschnitt 7.3.3 näher erläutert. Das Dashboard teilt die Position mit einer - der Ecore-Perspektive entliehenen - Ansicht der Eigenschaften eines Elements und der Eclipse-Ansicht für Probleme im Workspace. Der einzige, die volle Länge des Fensters in Anspruch nehmende, Teil der Abbildung ist optional. Er enthält die mit 5 markierten Cheat-Sheets. Wie in Eclipse üblich, werden sie am rechten Rand des Fensters in die Perspektive eingeblendet. Sie führen neue Benutzer in die Modellierung mit Graphtransformationsregeln ein und werden in Abschnitt 7.3.4 genauer erklärt.

Betrachtet man den Inhalt des Screenshots in Abbildung 7.13, so ist dort eine fehlerhafte Regel des in den Cheat Sheets genutzten Modells abgebildet. Es ist das einfachste und wohl populärste Beispiel der EMF-Welt: die Bibliothek. Die Regel dient der Akquise eines Buches.

7.3.2 Grafischer Editor und Validierung

ModGraph bietet einen grafischen Editor mit Validierung und Codegenerierung. Zur Entwicklung dieses Editors, der in Abbildung 7.13 gezeigt und mit 2 markiert ist, wird das Eclipse Modeling Framework (EMF) und, darauf aufbauend, das Graphical Modeling Framework (GMF) genutzt. Die Vorgehensweise zur Erstellung des Editors entspricht der in Abschnitt 3.4, Abbildung 3.5 beschriebenen. Dabei werden zudem die durch den Editor realisierbaren Teile der Bedingungen an die Elemente einer Regel, wie sie in Abschnitt 6.1.1 besprochen sind, berücksichtigt.

Der Editor ist in Abbildung 7.13 mittig dargestellt und mit 2 markiert. Hier können die Regeln erstellt werden. Ein neu initialisierter Editor enthält ein leeres Graphmuster. Über die Palette, rechts im Editor, können weitere Elemente hinzugefügt werden. Die Palette beinhaltet mehrere Abteile. Das oberste erlaubt die Erstellung von Vor- und Nachbedingungen sowie von NACs. Sollte der Nutzer den Bereich des Graphmusters gelöscht haben, so kann auch dieser hier wiederhergestellt werden. Der Abschnitt darunter bietet Möglichkeiten zur Erstellung von gebundenen Knoten, inklusive dem aktuellen Objekt. Darauf folgen die ungebundenen Knoten. Dabei wird für jeden Knoten in jedem - für ihn zugelassenen - Status ein Eintrag der Palette angeboten. Damit wird vermieden, dass die Knoten Status erhalten, die nicht konsistent sind. Beispielsweise können Parameter nicht erzeugt werden. Sie sind durch die modellierte Operation vorgegeben.

Die in den Knoten beinhalteten Elemente sind im nächsten Abschnitt der Palette zu finden. Es handelt sich hierbei um Attribute, Operationsaufrufe, OCL-Bedingungen und eine zu Testzwecken existente Standardausgabe. Wird ein Attribut aus der Palette in einen Knoten gezogen, so wird je nach Position des Mauszeigers im Knoten eine Attributbedingung (Mauszeiger im constraints-Teil des Knotens) oder eine Attributwertzuweisung (Mauszeiger im changes-Teil des Knotens) erzeugt. Operationsaufrufe und Ausgaben sind nur im changes-Teil zugelassen, während OCL-Bedingungen nur im constraints-Teil des Knotens auftreten dürfen.

Im letzten Abschnitt der Palette werden die zwischen den Knoten möglichen Kanten definiert. Die Links sind in den bekannten Status - zu erzeugen, zu löschen und zu

erhalten - verfügbar. Zudem können hier Pfade erstellt werden. Da die Links keinesfalls willkürlich Knoten verbinden sollen, ist eine Live-Validierung notwendig. Sie ist Teil der dreiteiligen Validierung der ModGraph-Regeln. Durch die Live-Validierung werden, während der Erstellung der Regel, erste Fehler vermieden.

Diese Fehler beziehen sich auf das Zusammenspiel der Elemente im Editor. Sie wurden direkt als Bedingungen in GMF an den Editor formuliert, so dass dieser die Bedingungen während der Erstellung des Diagramms prüft. Das GMF-Abbildungs-Modell regelt hier die korrekte Umsetzung der Bedingungen. Wie in Abschnitt 3.4 gezeigt, vereinigt dieses Modell die grafischen Darstellungen mit den Modellelementen selbst. Dabei werden einem Modellelement verschiedene grafische Darstellungen zugeordnet, die bereits bestimmte Attribute initialisieren. Die genaue grafische Ausprägung eines Knotens bestimmt, welche seiner Eigenschaften im Editor gesetzt werden. Ist ein Knoten beispielsweise Bestandteil einer NAC, so bietet er keine Möglichkeit zur Attributzuweisung an. Des Weiteren werden OCL-Ausdrücke genutzt, um Attribute von Knoten und Kanten bereits bei deren Erzeugung zu initialisieren. Insbesondere wird bei Kanten auf diese Art und Weise sichergestellt, dass die Status der Kanten und der durch sie verbundenen Knoten nicht widersprüchlich sind.

Alle in Abschnitt 6.1.1 beschriebenen Bedingungen, die sich nicht live überprüfen lassen, werden durch eine Batch-Validierung überprüft. Die Prüfung erfolgt gegen das ModGraph-Metamodell für Graphtransformationsregeln und gegen das Ecore-Modell, in dem die implementierte Operation modelliert ist.

Die gefundenen Fehler der Batch-Validierung werden in der Eclipse-Problem-Ansicht angezeigt. Zudem werden in der Graphtransformationsregel Fehlermarkierungen erstellt, die durch rote „X“ rechts oben das fehlerhafte Element markieren. Berührt man sie mit dem Mauszeiger, werden die Fehlermeldungen ebenfalls durch ein Pop-Up angezeigt. So hat der Modellierer der in Abbildung 7.13 modellierten Regel vergessen, den Attributwert des Attributs title zu setzen und erhält nach einer Batch-Validierung die abgebildete Fehlermeldung.

Rechts des Editors befindet sich ein Kommentarreiter, der in Abbildung 7.13 mit 3 markiert ist. Er zeigt den Projektnamen und den Namen der Regel an. Sein eigentlicher Zweck ist jedoch die Erstellung von JavaDoc-Kommentaren zur implementierten Operation. Er stellt eine Erweiterung des GMF-Editors dar, die zur Übersicht beiträgt, da der Kommentar - unabhängig von seiner Länge - den Aufbau der Regel nicht beeinflusst.

7.3.3 Dashboard

Das ModGraph-Dashboard führt den Nutzer strukturiert durch den Entwicklungsprozess. Das Ecore-Modell dient, aufgrund seiner Popularität, als Ausgangspunkt. Das Dashboard fasst die verschiedenen Entwicklungsschritte vom Ecore-Klassendiagramm über die Graphtransformationsregeln bis zum ausführbaren Modell kompakt zusammen. Die Vorgehensweise orientiert sich am in Kapitel 7.1 beschriebenen „Weg zum Neuen“. Es soll den ModGraph-Neuling unterstützen und bietet gleichzeitig einen einfachen Weg, existierende EMF-Projekte mit ModGraph zu erweitern. Das GMF-Dashboard dient ihm

als Vorlage.[7]

Das ModGraph-Dashboard ist in Abbildung 7.13 mit 4 markiert. Neben dem ModGraph-Logo in der linken oberen Ecke ist - mit etwas Abstand - der Name des aktuell im Dashboard abgebildeten ModGraph-Projekts zu sehen. Der Modellierungsprozess wird durch die Boxen und Pfeile dargestellt. Die Elemente werden anhand ihres Namens identifiziert. Die ausführbaren Operationen, wie zum Beispiel das Erstellen eines Ecore-Modells, finden sich jeweils im unteren Teil der Box. Beginnend mit der Architekturmodellierung bietet das Dashboard, von links nach rechts betrachtet, die Möglichkeit ein Paketdiagramm, dessen Name im Dashboard zu sehen ist, zu erstellen(create Eintrag im unteren Teil der Box), zu selektieren (select) oder zu editieren (edit). Alternativ kann es auch aus dem Klassendiagramm abgeleitet werden. Dazu wird der mit Create Package Diagram beschriftete Pfeil genutzt. Genauso kann aus dem Paketdiagramm ein Ecore-Klassendiagramm abgeleitet werden (Pfeil Create Class Diagram). Es ist im Dashboard mittig links dargestellt und wird ebenfalls anhand seines Namens identifiziert.

Auch das Klassendiagramm kann mit Hilfe des Dashboards erstellt, selektiert und editiert werden. Zudem kann die EMF-Quelltextgenerierung für das Diagramm angestoßen werden, indem der Nutzer auf Generate Java Code klickt. Folgt man dem Pfeil Create Xcore Control Flow unten im Bild, so kann das zugehörige Xcore-Modell aus dem Ecore-Klassendiagramm abgeleitet werden. Existiert es bereits, so kann es innerhalb der zugehörigen Box ausgewählt und editiert werden. Direkt rechts neben dem Klassendiagramm im Dashboard befindet sich ein Pfeil, der sich an einer Auswahlbox verzweigt. Die Auswahlbox beinhaltet die modellierten Klassen des Ecore-Modells. Abhängig davon, welche Klasse selektiert ist, werden die übrigen Boxen rechts außen befüllt. Die mittlere Box, auf welche der untere Zweig des Pfeils zeigt, listet die Java-Methoden der Klasse auf. Diese können direkt aus dem Dashboard editiert werden.

Der obere Zweig des Pfeils ist mit einer zusätzlichen Markierung Create Graph Transformation Rule beschriftet, welche die Erzeugung von Regeln zur Implementierung einer Operation der markierten Klasse erlaubt. Die bereits existierenden Regeln sind in der Box rechts oben am Ende des Pfeils gelistet. Sie können direkt aus dem Dashboard editiert und validiert werden.

Zudem kann aus ihnen, durch Nutzung des mit Generate Java Code markierten Pfeils, Quelltext direkt in die zugehörige Methode generiert werden. Die Generierung von Quelltext aus dem Xcore-Modell ist hier ebenfalls durch einen Pfeil eingezeichnet. Dieser dient dem Verständnis, da die Generierung von Java-Quelltext aus Xcore automatisiert erfolgt.

7.3.4 Unterstützung durch Cheat Sheets

Cheat Sheets sind in Eclipse eine verbreitete und gelungene Möglichkeit, den Nutzer sprichwörtlich „an die Hand zu nehmen“. Sie führen ihn Schritt für Schritt, anhand eines einfachen Beispiels, durch den gesamten Entwicklungsprozess. Dabei kann das Cheat Sheet den Nutzer nicht nur durch einen Text anleiten, sondern auch das Öffnen von

[7] Eine Erklärung des GMF-Dashboards findet sich unter https://wiki.eclipse.org/Graphical_Modeling_Framework/Tutorial/Part_4.

Dialogen direkt aus dem Cheat Sheet ermöglichen. Bestehende Cheat Sheets führen den Nutzer beispielsweise in die Plugin-Entwicklung mit Eclipse oder das EMF-Rahmenwerk ein. Selbst das bekannte „Hello World"-Programm kann in Eclipse mit Hilfe eines Cheat Sheets erstellt werden. So liegt es nahe, auch für ModGraph eines anzubieten, das - anhand eines sehr einfachen Beispiels - in die Modellierung mit Graphtransformationsregeln einführt, indem es dem Nutzer die wichtigsten Konzepte erklärt.

Der Nutzer wird dazu direkt aus der EMF-Welt abgeholt, indem ein sehr bekanntes Beispiel aus dieser, die Bibliothek, wiederverwendet wird. Es wird um zwei Methoden erweitert. Beide dienen der Akquise einzelner Bücher. Dabei ist einmal der Autor bekannt, das andere Mal handelt es sich um einen neuen Autor, der ebenso wie das Buch, im System angelegt werden muss. Anhand dieser Beispiele wird ein Einstieg in die Modellierung mit den grundlegenden Elementen einer ModGraph-Regel gegeben.

Das interaktive Cheat Sheet ist in Abbildung 7.14 rechts gezeigt. Es leitet den Nutzer von Anfang an durch den Entwicklungsprozess, indem es im ersten Schritt zur Erzeugung eines neuen ModGraph-Projektes anleitet. Innerhalb des Projekts wird zunächst das Ecore-Modell der Bibliothek erstellt, aus dem Quelltext generiert wird. Die Erstellung des Modells erfolgt Schritt für Schritt, um auch dem nicht EMF-erfahrenen Benutzer einen Einstieg zu bieten. Es ist in Abbildung 7.14 oben links dargestellt. Darauf aufbauend wird die ModGraph-Regel modelliert, indem der Nutzer mit einem Rechtsklick auf die gewünschte Operation klickt und die Implementierung dieser als ModGraph-Regel wählt. Der in Abbildung 7.14 unten dargestellte Editor öffnet sich. Er wird zunächst erklärt und beinhaltet anfangs nur die mit GraphPattern bezeichnete Fläche zur Erstellung des Graphmusters für die Regel. Der Editor wird ebenso kleinschrittig befüllt. Jedes Objekt wird im Cheat Sheet mit einer Erklärung seiner Bedeutung und der korrekten Erstellung eingeführt. Dies ist in Abbildung 7.14 rechts ausschnittsweise für ein Objekt, einen Link und eine Vorbedingung gezeigt. Die sich ergebende Regel ist in der Abbildung links unten zu sehen.

In einer weiteren Regel, welche die zweite modellierte Operation implementiert, wird der Nutzer in die Verwendung von NACs eingeführt. Diese Regel modelliert die Akquise eines Buches, dessen Autor nicht bekannt ist. Das entstehende Graphmuster ist dem abgebildeten ähnlich. Der Autor ist jedoch ebenso durch einen zu erzeugenden Knoten modelliert, dem der übergebene Name zugewiesen wird. Zudem beinhaltet die Regel eine NAC, die verbietet, dass die Bibliothek bereits einen Autor mit diesem Namen kennt.

Neben der Erstellung von Graphtransformationsregeln wird der Nutzer in ihre Validierung und in die Java-Quelltextgenerierung aus einer Regel eingeführt. Dabei ist der Aufruf der Validierung und der Generierung direkt aus dem Cheat Sheet möglich, um den Nutzer anfangs nicht mit der Suche nach Menüeinträgen zu belasten.

Das so erzeugte Beispiel soll den Modellierer dazu animieren, selbstständig die weiteren Möglichkeiten zur Nutzung von ModGraph zu erkunden. So kann er beispielsweise mit Hilfe des Dashboards ein Xcore-Modell erstellen. Dieses kann dazu genutzt werden, mittels eines Xcore-Kontrollflusses eine Abfrage zur Steuerung der beiden Regeln zu erstellen. Alternativ kann auch die Generierung auf Modellebene getestet werden, indem die Regeln selbst in das Xcore-Modell generiert werden.

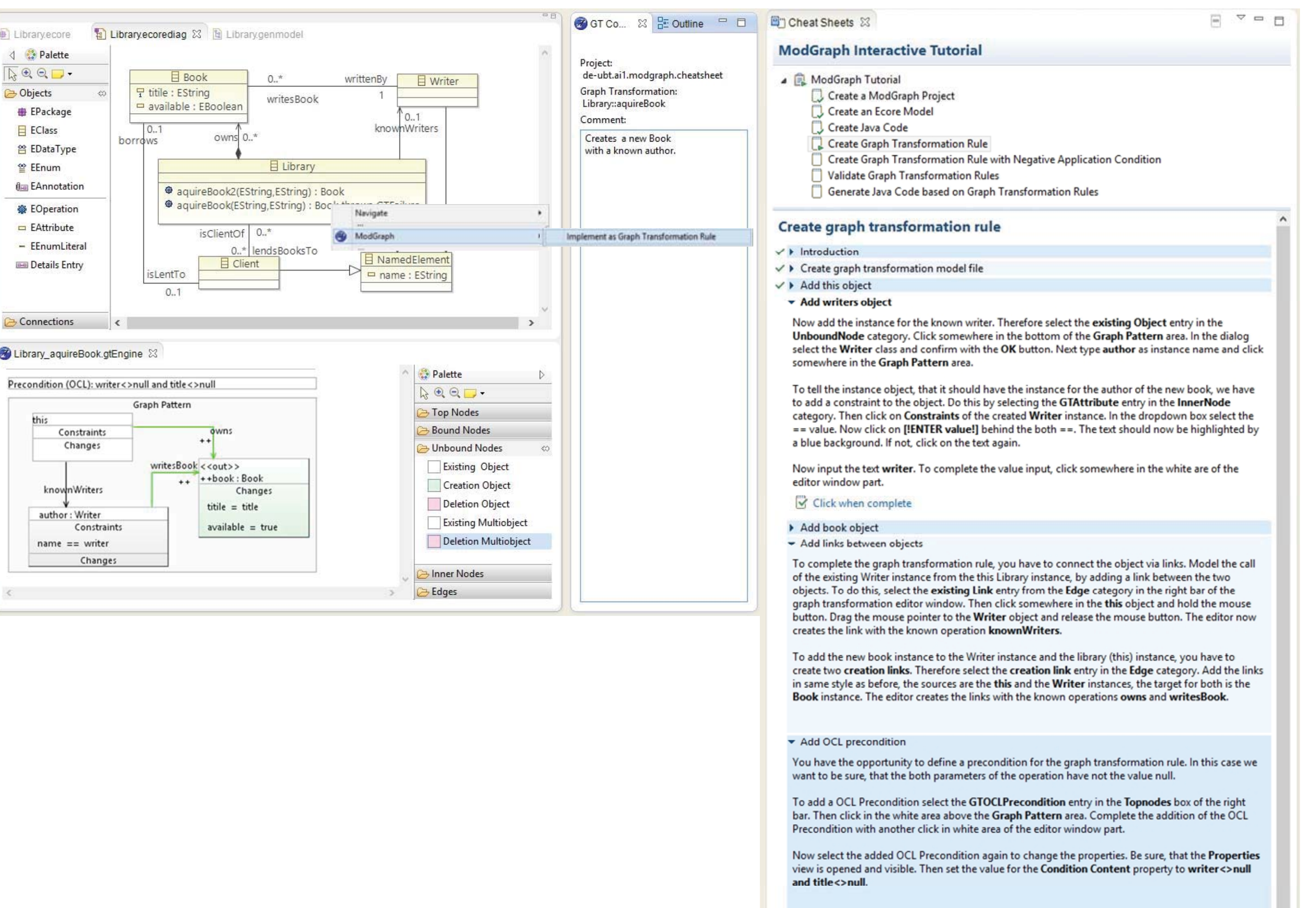


Abbildung 7.14: *Im Cheat Sheet (rechts) und ausgearbeitetes Beispiel (links) mit Ecore-Klassendiagramm (oben) und der Regel zur Implementierung der Operation* aquireBook(...) *(unten)*

Teil IV

Evaluation

„Wenns läuft, dann läufts."
(Sprichwort)

Inhaltsverzeichnis - Teil IV

8 Propagierendes Refactoring 165
8.1 Entstehende Problematik durch abhängige Modelle in ModGraph 166
8.2 Konzeption des propagierenden Refactorings 169
8.3 Implementierung 176
8.3.1 Architektur des propagierenden Refactorings 176
8.3.2 Beispielrefactoring: Ändern einer bidirektionalen Referenz in eine unidirektionale 178
8.3.3 Refactoring 2: Extrahieren einer Klasse 182

9 Integration mit modalen Sequenzdiagrammen 187
9.1 Theoretische Überlegungen 187
9.1.1 Beschreibung typischer Systeme am Beispiel 187
9.1.2 Modale Sequenzdiagramme und Play-out 190
9.1.3 Integration von MSDs mit Graphtransformationsregeln 194
9.2 Prototypische Werkzeugintegration von ModGraph und ScenarioTools . . 197

10 Diskussion und Abgrenzung des Ansatzes 201
10.1 Diskussion des Ansatzes 201
10.1.1 Evaluation der Regeln 201
10.1.2 Laufzeitmessungen 207
10.2 Abgrenzung des ModGraph-Ansatzes zu verwandten Arbeiten 211
10.2.1 EMF und ModGraph 212
10.2.2 ModGraph und Xcore 213
10.2.3 Modelltransformationswerkzeuge und ModGraph 214
10.3 Abgrenzung des propagierenden Refactorings 219
10.4 Abgrenzung der Integration der Regeln mit MSDs 220
10.5 Ergebnis und Ausblick 222

8 Propagierendes Refactoring

Refactoring ist ein gängiger Prozess bei der Entwicklung von Software. Es wird zur Verbesserung des Designs von Programmen verwendet. Ziel dabei ist, diese übersichtlicher zu gestalten, indem unnötige Teile wegfallen oder redundante zusammengefasst werden, um die Wahrscheinlichkeit des Auftretens von Fehlern zu verringern.

Der Definition von Martin Fowler [33] folgend, handelt es sich beim Refactoring um eine Aktivität Software durch Anwendung einer Reihe von Transformationen zu restrukturieren ohne deren nach außen hin sichtbares, beobachtbares Verhalten zu verändern. Die möglichen Transformationen beziehen sich in Fowlers ursprünglicher Fassung auf (objektorienterte) Programme. Das hier gemeinte Verhalten bezieht sich auf das Endprodukt, das an den Nutzer ausgeliefert wird. Für den Entwickler kann sich das interne Verhalten durch Refactoring deutlich ändern - man denke an den einfachen Fall des Hinzufügens oder Entfernens eines Parameters zu einer Methode, der eine Änderung aller Methodenaufrufe mit sich bringt.

Betrachtet man das Refactoring vom Standpunkt der modellgetriebenen Softwareentwicklung, so stellt sich die Frage, inwiefern es auf Modelle angewendet werden kann. Für einzelne strukturelle Modelle ist dieser Vorgang bereits von Biermann [14] und Mens [51, 52] untersucht worden.

Praktisch gesehen sind für sich allein stehende Modelle nicht immer gegeben. Sobald Verhaltensmodellierung zum Softwaredesign genutzt wird, entstehen Abhängigkeiten zwischen Struktur- und Verhaltensmodellen.[1] Dies gilt unabhängig von der genauen Ausprägung der Modelle. Beispielsweise stehen in der UML Aktivitäts- und Sequenzdiagramme in der Designphase immer mit dem zugehörigen Klassendiagramm in Verbindung, indem sie das Verhalten von aus den Klassen instanziierten Objekten darstellen. Ebenso sind die hier vorgestellten Graphtransformationsregeln von dem ihnen zugeordneten Ecore-Modell abhängig, indem sie dessen Elemente referenzieren.

Es entstehen Mengen abhängiger Modelle, welche konsistent gehalten werden müssen. Dazu ist die Propagation aller Änderungen innerhalb der - in sich abhängigen - Modellmenge notwendig. Wie sich dies gestaltet, hängt von der genauen Ausprägung der Modellabhängigkeiten ab.

Im vorliegenden Kapitel wird daher die Konsistenzerhaltung zwischen Graphtransformationsregeln und deren zugehörigem Ecore-Modell bei Anwendung von Refactoringoperationen untersucht. Dazu werden propagierende Refactorings eingeführt. Sie bieten einen - an Fowlers Definition angelehnten - Katalog von Refactorings auf einem Ecore-Modell *und* propagieren die dadurch entstandenen Änderungen, sofern nötig, an

[1] Verhaltensmodellierung kann ebenfalls während der Anforderungsanalyse genutzt werden. Hierbei treten noch keine Abhängigkeiten auf, da unter Umständen noch kein strukturelles Modell existiert.

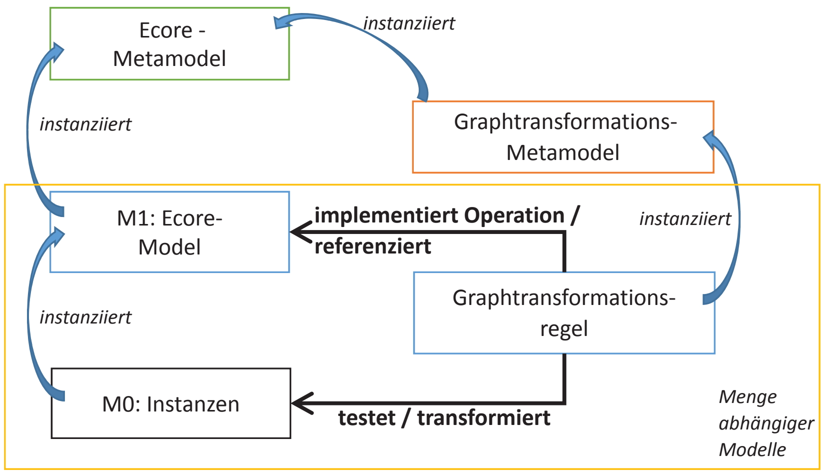


Abbildung 8.1: *Entstehende Abhängigkeiten einer Regel bei M0-Anwendung*

die Graphtransformationsregeln. Diese Refactorings sind in [81] und insbesondere in [82] veröffentlicht. Die erste Version der Modell-Refactorings wurde im Rahmen einer Bachelor-Arbeit [27] erstellt. Sie wurden später erweitert.

Zudem sei an dieser Stelle angemerkt, dass die, unter Umständen vom Nutzer erstellten, Xcore-Operationen zur prozeduralen Verhaltensbeschreibung in ModGraph bislang unberücksichtigt bleiben. Sie müssen manuell angepasst werden.

8.1 Entstehende Problematik durch abhängige Modelle in ModGraph

Um die zu lösende Problematik hinter den Refactorings zu ergründen, werden die bereits in Kapitel 5 vorkommenden Modelle und deren Zusammenhänge weitergehend untersucht. Abbildung 8.1 zeigt die M0-Anwendung einer Regel mit allen zugehörigen Metamodellen und Abhängigkeiten. Dabei entsteht immer eine Abhängigkeit zwischen einer Regel, dem Modell und dessen Instanz. Diese bilden die in Abbildung 8.1 gelb umrandete Menge abhängiger Modelle. Dabei sind insbesondere das Modell und die Regel - durch die vielen in Abschnitt 6.1.1 beschriebenen Referenzen von der Regel zum Modell - stark vernetzt. Zur Erinnerung sei erwähnt, dass jede Regel die Operation, die sie implementiert und auf die diese beinhaltende Klasse verweist. Zudem sind die Knoten und Kanten der Regel über den Elementen des strukturellen Modells typisiert.

Damit liegt es nahe, dass Änderungen am Modell die Regeln beeinflussen und auf diese propagiert werden müssen. Zudem ist eine Propagation der Modelländerungen auf die Instanzen notwendig. Dieser Prozess wird Modellmigration genannt und stellt einen evolutionären Prozess dar, der die Migration der Instanzen als Reaktion auf Typänderungen anstößt. Er wurde im Kontext der modellgetriebenen Softwareentwicklung bereits ausführlich untersucht. Rose et al. stellen einen Katalog von Operationen zur gekoppelten Evolution von Metamodellen und deren Modellen im COPE-Rahmenwerk

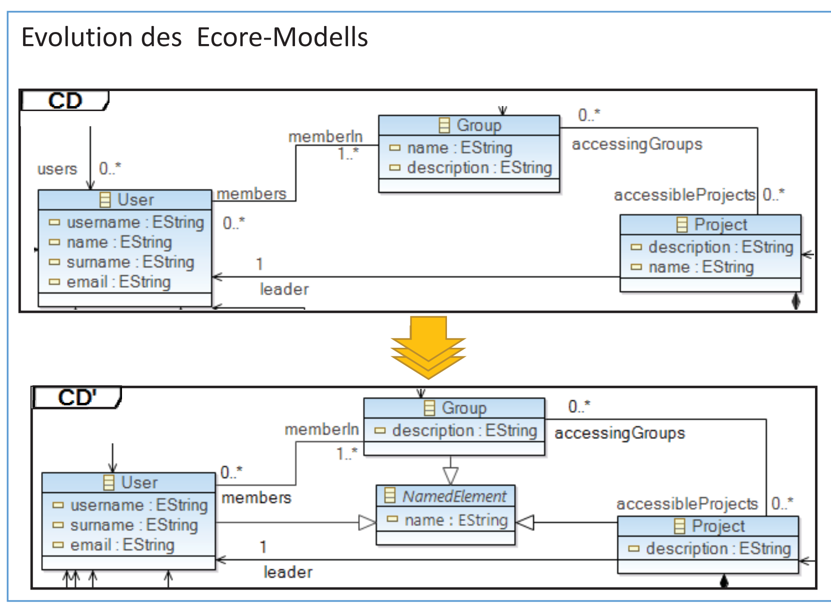


Abbildung 8.2: *Refactoring am Beispiel Bugtracker: Extrahieren einer Oberklasse*

vor [63]. Dabei sind die zu lösenden Probleme im Wesentlichen dieselben, die bei der Schemaevolution von Datenbanken auftreten [11]. Zur Ausführung von Modellmigrationen wird in Rose et al. die Sprache Epsilon Flock eingeführt [64]. Hermannsdörfer betrachet in [41] die Evolution von Metamodellen und deren Modellen im COPE-Rahmenwerk. Unter Berücksichtigung dieser Arbeiten kann die Migration von Instanzen bei Modelländerungen vorgenommen werden. Sie wird hier nicht eingehender betrachtet.

Übrig bleibt somit die Problematik, ob und wie eine Änderung - hier ein Refactoring - am Modell zu Änderungen an den Regeln führt. Die Antwort hängt von der Art der Änderung ab. Dies bedeutet, dass die Propagation für jedes Refactoring einzeln untersucht werden muss. Die Ergebnisse sind im nächsten Abschnitt zusammengefasst. Um diesen Sachverhalt näher betrachten zu können, wurde in [27] ein Katalog von Refactorings für Ecore-Modelle mit ModGraph implementiert. Teile davon sollen nun beispielhaft für den bereits bekannten Bugtracker gezeigt werden. Bei genauerer Betrachtung des Bugtracker-Modells findet man zwei Refactorings, die den Bugtracker positiv verändern können: das Extrahieren einer Oberklasse und das Umwandeln einer bidirektionalen Referenz in eine unidirektionale. Zudem soll die Extraktion einer Klasse zur Verwaltung der zahlreichen Namen eines Nutzers betrachtet werden, da es sich hier um ein komplexes Refactoring handelt.

Betrachtet man den in Abbildung 8.2 oben gezeigten Ausschnitt aus dem Bugtracker-Modell, fällt auf, dass die Klassen User, Group und Project jeweils ein Attribut name besitzen. Dieses soll nun in eine gemeinsame, hier abstrakte, Oberklasse verschoben werden. Abbildung 8.2 zeigt diesen Vorgang, markiert durch einen gelben Dreifach-Pfeil. Oben in der Abbildung ist ein Ausschnitt aus dem Modell vor der Refactoringoperation dargestellt und mit CD beschriftet. Die Klassen tragen jeweils ein gemeinsames Attribut. Der untere Teil der Abbildung, das Klassendiagramm CD', zeigt das Modell nach dem

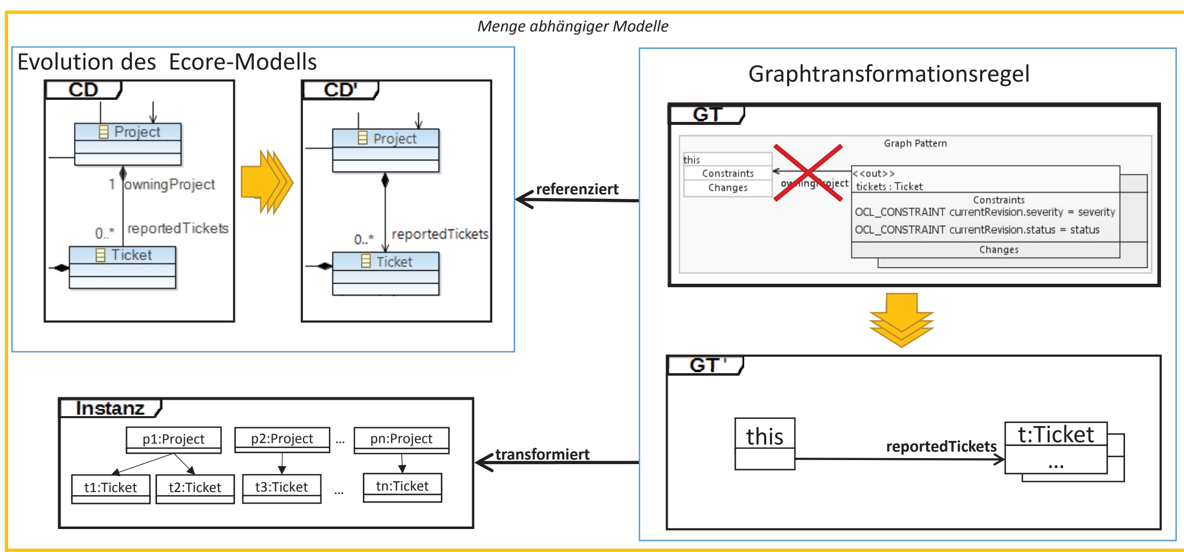


Abbildung 8.3: *Notwendige Anpassungen der Regeln am Beispiel Bugtracker: Umwandeln einer bidirektionalen Referenz in eine unidirektionale*

Refactoring. Die gemeinsame Oberklasse NamedElement beinhaltet nun das Attribut name.[2]

Betrachtet man die Extraktion der Oberklasse hinsichtlich der Propagation auf die Graphtransformationsregeln, so muss zunächst geprüft werden, ob Änderungen an diesen nötig sind. Im Klassendiagramm wurde eine Vererbungshierarchie aufgebaut. Dies beeinflusst die, in den Instanzen verfügbaren, Attribute nicht. Da die Knoten in den Regeln Instanzen repräsentieren, sind auch deren Attribute nicht betroffen. Es wird keine Propagation benötigt.

Dieses Refactoring stellt den einfachsten Fall dar. Die meisten Refactorings ziehen jedoch notwendige Änderungen der Regeln nach sich, die bislang manuell vorgenommen werden müssen, um nicht valide Regeln zu vermeiden. Um diesen Fall auf einer überschaubaren Anzahl von Seiten zu demonstrieren, wird zunächst ein moderat komplexes Refactoring herangezogen: Das Umwandeln einer bidirektionalen Referenz in eine unidirektionale. Auch dieses soll am Beispiel des Bugtrackers untersucht werden.

In Abbildung 8.3 ist links oben die Modellevolution des Bugtrackers anhand eines Ausschnitts aus dem Ecore-Modell dargestellt. Ausgangspunkt ist das mit CD beschriftete Ecore-Modell. Es zeigt die bidirektionale Referenz zwischen einem Projekt und den ihm zugeordneten Tickets. Da es sich um eine Enthaltenseinsbeziehung handelt, muss die Rückrichtung nicht explizit navigierbar sein. Dementsprechend wird auf dieser Referenz nun ein Refactoring ausgeführt, das die bidirektionale Referenz in eine unidirektionale umwandelt (gelber Dreifach-Pfeil). Das Ergebnis ist in der Abbildung mit CD' bezeichnet. Bezüglich des Ecore-Modells ist dies ein gültiges Refactoring. Das Problem entsteht in den Regeln. Enthalten diese Links, welche für Instanzen der nun gelöschten Referenz

[2]Analog könnte nun auch die Beschreibung (description) der Projekte und Gruppen in eine gemeinsame Oberklasse verschoben werden.

stehen, sind sie nach dem Refactoring nicht mehr valide. Nicht valide Regeln dürfen keinesfalls Transformationen auf Instanzen durchführen. Im Beispiel in Abbildung 8.3 ist die rechts dargestellte und mit GT bezeichnete Regel betroffen. Sie stellt die Implementierung der Operation allTicketsWithStatusAndSeverity dar, die alle Tickets eines Projekts mit den vorgegebenen Status und Dringlichkeiten zurückgibt. Der nun nicht mehr valide Link owningProject ist mit einem roten X gekennzeichnet. Der mehrwertige Knoten tickets war vor der Modelländerung implizit durch den Rückwärtslink zu erreichen, der nach dem Refactoring fehlt. Um die Validität der Regel wieder herzustellen, muss sie manuell angepasst werden. Dazu wird der Link vom Nutzer durch einen vormals die entgegengesetzte Richtung der Referenz repräsentierenden Link ersetzt. Für das Beispiel bedeutet dies, dass der Link owningProject durch reportedTickets ersetzt wird, wie es in GT' der Fall ist. Damit ist GT' konsistent mit CD' und es können weitere Transformationen auf Instanzen vorgenommen werden.

Diese Anpassung jeder betroffenen Regel ist für den Nutzer - gerade bei größeren Projekten mit vielen Regeln - eine mühsame Aufgabe. Alle Regeln müssen durchsucht und jede Regel muss einzeln angepasst werden. Dies ist der Ansatzpunkt des propagierenden Refactorings. Wie im Folgenden gezeigt wird, bietet es neben einem Katalog von Refactorings auch eine Propagation auf alle abhängigen Graphtransformationsregeln an. Damit wird auch der in Abbildung 8.3 rechts abgebildete Schritt automatisiert.

8.2 Konzeption des propagierenden Refactorings

Die hier vorgestellten propagierenden Refactorings geben die Änderungen am Ecore-Modell geeignet an die Regeln weiter. Die grundlegende Idee ist in Abbildung 8.4 dargestellt. Dem Nutzer wird ein Katalog von Refactoringoperationen zur Verfügung gestellt, die nicht nur das Modell anpassen, sondern auch - in einem weiteren Schritt - die Änderungen auf die Regeln propagieren. Die entstehenden propagierenden Refactoringoperationen sind unidirektionale, inkrementelle, überschreibende Modelltransformationen. Die Refactoringoperation selbst kann als endogene Transformation des Ecore-Modells aufgefasst werden, die Propagation als exogene Transformation vom Modell auf die Regeln.

Da ModGraph tief in der EMF-Welt verwurzelt ist, liegt es nahe, die Refactorings und deren Propagationen auf die Regeln mit ModGraph zu modellieren. Es entstehen Refactorings *für* und *mit* ModGraph. Zur Implementierung des propagierenden Refactorings werden - in einem ersten Schritt - mit ModGraph umgesetzte Refactorings auf Ecore-Modellen erstellt. Die durch das Refactoring hervorgerufenen Änderungen am Modell werden - in einem zweiten Schritt - auf die Regeln propagiert. Die hierzu notwendigen konsistenzerhaltenden Maßnahmen wurden für und überwiegend mit ModGraph-Regeln gestaltet. Regeln die andere Regeln verändern werden Metaregeln genannt. Dies wird im Abschnitt 8.3 genauer betrachtet. An dieser Stelle sei bemerkt, dass die Regeln zur Definition des propagierenden Refactorings auf sich selbst angewendet werden könnten. Es handelt sich daher um einen reflektierenden Ansatz.

Der Katalog der in ModGraph unterstützten Refactorings für Ecore-Modelle orien-

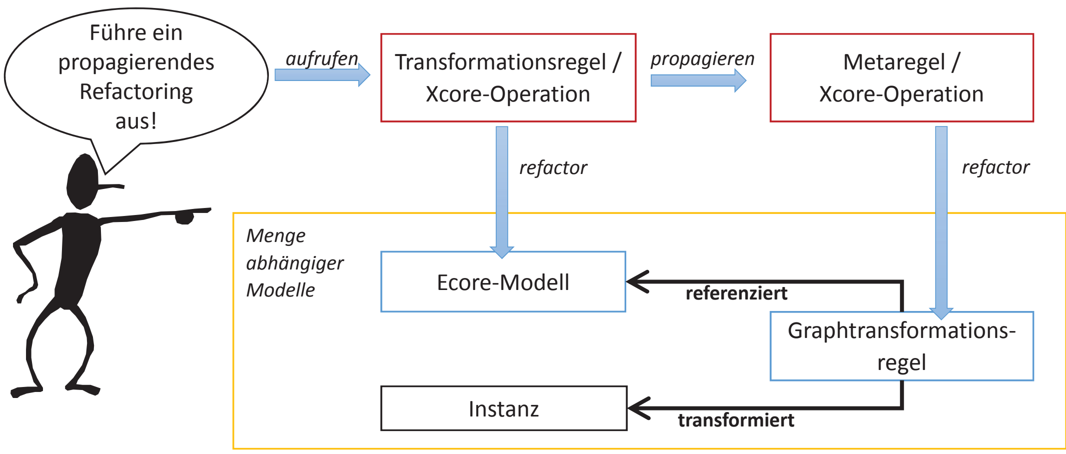


Abbildung 8.4: *Grundlegende Idee des propagierenden Refactorings*

tiert sich an dem von Fowler [33] vorgegebenen. Er bildet einen Ausgangspunkt für die Propagation. Der zweite Startpunkt der Propagation wird durch das Metamodell der Sprache der ModGraph-Regeln vorgegeben. Dieses wird im Bezug auf seine Referenzen zum Ecore-Metamodell analysiert. Tabelle 8.1 zeigt das Ergebnis der Analyse dieser Referenzen. Sie zeigt ein Zwischenergebnis, das zum Design der Propagation verwendet wird. Dazu werden elementare Basisänderungen, die im Rahmen von Refactorings geschehen, auf deren nötige Propagation hin untersucht. Die Tabelle gibt demnach keinen Überblick der gesamten Refactorings, sondern bietet eine Betrachtung, wie die Elemente einer Regel von elementaren Änderungen am Modell betroffen sind. Dazu wird jede Regel in ihre Bestandteile zerlegt und diese werden klassifiziert. Die erste Spalte der Tabelle 8.1 enthält die so erreichte Klassifikation der Elemente der Regel, welche von der Modelländerung betroffen sind: die ganze Regel, Knoten, Attribute, Links und textuelle Anteile. Knoten stehen dabei für die gebundenen und ungebundenen, ein- und mehrwertigen Knoten der Regel. Textuelle Anteile fassen Vor- und Nachbedingungen sowie Bedingungen an Knoten, Pfadausdrücke, Operationsaufrufe und Felder zusammen. Die zweite Spalte der Tabelle führt die elementaren Änderungen am Ecore-Modell auf, welche das Regelelement betreffen können und gleichzeitig bei Refactoringoperationen auftreten. Die dritte Spalte gibt an, welche Anpassungen an einer Regel - als Reaktion auf die Änderung im Modell - vorgenommen werden müssen. Die mit * gekennzeichneten Einträge dieser Spalte sind auf spezifische Eigenschaften von EMF zurückzuführen. Diese sind in der Art der Referenzierung der Elemente begründet. Jedes Element wird in der persistenten XMI-Serialisierung der ModGraph-Regel durch seinen Namen und den Namen seiner Container referenziert. Daher müssen in manchen Fällen Anpassungen vorgenommen werden, die auf rein konzeptueller Ebene nicht notwendig sind. Einträge ohne * sind dagegen auf konzeptueller Ebene erforderlich.

Die Regel als Ganzes referenziert eine Operation und die sie enthaltende Klasse. Änderungen an diesen werden propagiert. Wird eine Klasse umbenannt, muss die Referenz auf diese und auf die in ihr enthaltene Operation, aufgrund von EMF Spezifika, angepasst werden. Falls die Klasse gelöscht wird, kann die Regel einer anderen Klasse

Betroffenes Element der Regel	**Änderung am Ecore-Modell**	**Dadurch notwendige Änderung an der Regel**
ganze Regel	*Klasse* umbenennen löschen abstrakt deklarieren *Operation* umbennennen löschen abstrakt deklarieren verschieben	 *Referenz auf die Klasse und Operation anpassen* löschen - *Referenz auf die Operation anpassen* ganze Regel löschen löschen / anderer Operation zuordnen *Referenz auf die Operation und die Klasse anpassen*
Knoten	*Klasse* umbenennen löschen abstrakt deklarieren *Operation* nicht-primitiven Parameter hinzufügen nicht-primitiven Parameter entfernen	 *neu typisieren* löschen zu erzeugende Knoten löschen/ alle Anderen - - entferne zugehörige Knoten
Attribute	*Attribute einer Klasse* umbenennen löschen neu typisieren verschieben	 *Referenz auf das Attribut anpassen* löschen neu typisieren & neuen Wert zuweisen verschieben, falls unmöglich: löschen
Links	*Referenzen* umbenennen löschen Ändern der Quell-/Zielklasse	 *Referenz auf die Modellreferenz anpassen* löschen Ändern des Quell-/Zielknoten, falls unmöglich: löschen
textuelle Anteile	jede Änderung	textuellen Ausdruck parsen und ändern, falls vom Refactoring betroffen

Tabelle 8.1: *Modelländerungen und deren Effekte auf die Regel*

zugewiesen werden, die eine gleichartige Operation enthält, die stattdessen ausgeführt wird. Damit muss auch die Operation neu zugewiesen werden und die Regel bleibt intakt. Ansonsten wird sie gelöscht. Das Abstrakt-Deklarieren einer Klasse führt zu keinerlei Änderungen, solange nicht auch die Operation abstrakt deklariert wird. Geschieht dies jedoch ebenfalls, so muss die Regel entweder einer anderen, gleich parametrisierten Operation einer Unterklasse zugewiesen werden oder sie wird gelöscht. Dasselbe Vorgehen wird beim Löschen der Operation angewendet. Wird eine Operation umbenannt, muss - analog zur Umbenennung einer Klasse - die Referenz der Regel auf diese neu gesetzt werden.

Jeder Knoten referenziert - implizit oder explizit - eine Klasse. Er muss aktualisiert werden, sobald diese sich ändert. Wird sie beispielsweise umbenannt, so wird der Knoten neu typisiert, um (EMF-spezifisch) weiter auf die Klasse verweisen zu können. Wird eine Klasse gelöscht, so auch der zugehörige Knoten. Wenn die Klasse zur abstrakten Klasse erklärt wird, ist ihre Instanziierung unmöglich. Daher werden alle sie referenzierenden zu erzeugenden Knoten gelöscht. Die anderen Knoten bleiben erhalten.[3] Knoten, die Parameter repräsentieren, können zudem von Änderungen an Operationen betroffen sein. Wird ein zu einem Parameterknoten gehörender Parameter gelöscht, so wird auch der Knoten entfernt. Das ist zur Vollständigkeit tabellarisch dargestellt. Hinzufügen von Parametern beeinflusst die Regeln nicht. Es besteht kein Zwang, jeden Parameter innerhalb der Regel zu verwenden.

Attribute sind ein wichtiger Teil der Regel, da sie Bedingungen und Zuweisungen an Knoten darstellen können. Jedes Attribut referenziert eines im Ecore-Modell und benutzt zu seiner Darstellung dessen Namen. Es muss den Änderungen des Modellattributs folgen. Umbenennungen von Modellattributen führen zu einer Erneuerung der Referenz auf das Attribut. Zudem wird der angezeigte Name des Attributs angepasst. Die Neutypisierung eines Attributs ist noch etwas komplexer: Das Attribut der Regel bekommt zunächst den neuen Typ. Zudem muss der Wert des Attributs vom Benutzer angepasst werden. Beispielsweise muss eine ehemalige Zeichenkette, welche zu einem Wahrheitswert konvertiert wurde, nun einen der Werte `true` oder `false` tragen. Wird ein Attribut im Modell verschoben, so versucht das propagierende Refactoring, es in der Regel ebenfalls zu verschieben. Dies ist möglich, wenn die Regel einen Knoten, der über der Zielklasse der Verschiebung typisiert ist, enthält. Schlägt eine Verschiebung fehl, wird das Attribut gelöscht.

Werden Referenzen im Modell umbenannt, so ändert sich auch der angezeigte Name des Links, da die Referenz des Links auf die Modellreferenz erneuert werden muss. Wird die Quell- oder Zielklasse der Referenz geändert, bedeutet dies, das der Link ebenso Quell- und Zielknoten anderen Typs benötigt. Sind diese in der Regel vorhanden, wird der Link verschoben. Andernfalls wird der Link gelöscht. Dies geschieht ebenfalls, wenn die Modellreferenz gelöscht wird.

Für die textuellen Elemente der Regel bleibt keine andere Möglichkeit als sie nach den

[3] Alternativ wäre es auch möglich, hier Inkonsistenzen zuzulassen, sprich diese Knoten nicht zu löschen. Die korrekte Instanziierung wäre dem Generator überlassen. Dies ist allerdings in der ModGraph-Werkzeugumgebung nicht vorgesehen. Sie erlaubt nur die Instanziierung konkreter Klassen.

Namen von veränderten Elementen zu durchsuchen und diese, im Falle eines Funds, zu ändern. Dies kann entweder automatisiert oder durch Benutzereingabe erfolgen. Wird beispielsweise ein primitiver Parameter der Methode entfernt, so müssen alle textuellen Elemente der Methode auf dessen Vorkommen geprüft und gegebenenfalls angepasst werden.

Die vorangegangenen Vorüberlegungen werden nun mit Fowlers Refactorings zusammengebracht. Daraus entsteht ein Katalog propagierender Refactorings für die betrachtete Menge von abhängigen Modellen. Auffällig hierbei ist, dass manche Refactorings auf Modellebene konzeptionell keine Propagation auf die Regel benötigen. Beispiel hierfür sind das Extrahieren einer Oberklasse im Modell oder das Verschieben eines gemeinsamen Attributs der Unterklassen in die zugehörige Oberklasse. Ersteres entspricht dem Hinzufügen eines Elements zum Modell, Letzteres dem Verschieben eines Elements, jeweils ohne die Funktionalität der anderen, bestehenden Elemente zu beeinflussen. Damit bleiben diese Refactorings propagationsfrei. Der entgegengesetzte Fall tritt beim Umbenennen eines Elements im Modell auf. Dieses für das Modell relativ einfache Refactoring löst mehrere Propagationsschritte aus. Wird zum Beispiel eine Klasse umbenannt, so kann sich dies auf die Regel selbst, die Knoten und alle textuellen Anteile der Regel erstrecken.

Die 19 implementierten Refactorings werden im Folgenden kurz dargestellt. Eine Untersuchung der textuellen Elemente der Regel und deren Validierung finden immer statt, ohne im Folgenden explizit angegeben zu sein. Dabei ist für jedes propagierende Refactoring die Anwendung auf das Modell und deren Propagation auf die Regel angegeben. Hierbei sei angemerkt, dass die - auf dem Modell agierenden - Refactorings für jedes Ecore-Modell anwendbar sind, unabhängig davon, ob es im ModGraph-Kontext steht oder klassisch im EMF-Kontext verwendet wird. Beispiele für diese Regeln finden sich in [81]. Sie wurden im Rahmen einer Bachelor-Arbeit [27] zur vollen Implementierung der Refactorings auf Modellebene erweitert. Die Implementierung der Propagationen ist in [82] dargestellt.

Umbenennen eines Elements:
Benennt ein Element des Ecore-Modells um.
Propagiert diese Änderung auf alle referenzierenden Elemente in den Regeln.

Hinzufügen eines Parameters zu einer Operation:
Fügt einen Übergabeparameter zu einer Operation hinzu.
Propagiert diese Änderung zu den Regeln, die diese Operation implementieren und allen textuellen Feldern, insbesondere solchen, die einen Aufruf dieser Methode darstellen.

Entfernen eines Parameters einer Operation:
Entfernt einen Übergabeparameter aus einer Operation.
Propagiert diese Änderung, indem der Parameter, sofern es sich um einen nicht-primitiven handelt, aus den Regeln entfernt wird. Zudem werden alle textuellen Felder, in welchen die Operation vorkommt, angepasst, insbesondere solche, die einen Aufruf dieser Methode darstellen.

Umwandlung einer bidirektionalen in eine unidirektionale Referenz:
Löscht eine Referenz, zu der eine gegenüberliegende existiert.
Die Propagation erfolgt, indem der Link, der auf die Referenz zeigte, durch einen Link, der auf das ehemaligen Gegenüber zeigt, ersetzt wird. Dabei kann es passieren, dass Knoten innerhalb der Regeln nicht mehr von gebundenen Knoten aus erreichbar sind. Die Regel ist in diesem Fall nicht mehr valide. Daher ist hier die Validierung besonders wichtig.

Umwandlung einer unidirektionalen in eine bidirektionale Referenz:
Fügt dem Modell eine Referenz hinzu und setzt diese als Gegenüber einer bereits existierenden Referenz.
Hier ist keine Propagation nötig, da ein erstelltes Modellelement noch nicht in den Regeln vorkommen kann. Die Validierung der Regeln ist jedoch dringend erforderlich, da eventuell die Multiplizitäten der neuen Referenz verletzt sein könnten.

Extrahieren einer Klasse:
Extrahiert eine Klasse aus einer anderen und setzt beide in Beziehung, indem eine Referenz zwischen ihnen erzeugt wird.
Zur Propagation ist eine Analyse der auf die Klasse zeigenden Knoten innerhalb der Regeln nötig: Benutzt ein Knoten Attribute, Operationen oder Referenzen, die auf die neu erzeugte Klasse transferiert wurden, wird ein neuer Knoten erzeugt, der die neue Klasse referenziert. Die beiden Knoten werden über einen Link, der die neu erzeugte Referenz repräsentiert verbunden. Die Attribute, Operationen oder Referenzen werden in den neuen Knoten verschoben.

Hinzufügen einer Klasse zu einer anderen:
Alle Elemente innerhalb der Klasse werden in die andere Klasse transferiert.
Die Propagation erfolgt, indem alle Knoten einer Regel, die auf die aufgelöste Klasse zeigen, auf die bestehen bleibende Klasse typisiert werden. Sollte die Regel bereits einen Knoten des Typs der bleibenden Klasse besitzen, werden diese fusioniert.

Extrahieren einer Unterklasse:
Extrahiert eine Unterklasse aus einer gegebenen Klasse.
Die Propagation erfolgt folgendermaßen: Falls Operationen, Attribute oder Referenzen verschoben wurden, werden die Knoten, welche die Oberklasse referenzieren und die Operationen, Attribute oder Referenzen beinhalten, neu typisiert. Die Knoten referenzieren nun die neue Unterklasse. Dabei kann es passieren, dass andere im Vorfeld bestehende Unterklassen Eigenschaften verlieren. Es handelt sich um diejenigen, die von der gemeinsamen Oberklasse in die neue Unterklasse verschoben wurden. Diese werden während der Validierung erkannt. Sie müssen manuell angepasst werden.

Extrahieren einer Oberklasse:
Erstellt eine Oberklasse zu einer Menge gegebener Klassen.
Dabei ist, aufgrund der erzeugten Vererbungshierarchie, keine Propagation nötig.

Verschieben eines Attributs oder einer Operation:
Verschiebt ein Attribut oder eine Operation von einer Quell- in eine Zielklasse. Zur Propagation wird geprüft, ob die Regel einen Knoten enthält, der die Quellklasse referenziert und einen, der die Zielklasse referenziert. Wenn ja, verschiebe das Attribut oder die Operation. In allen anderen Fällen wird das Attribut oder die Operation gelöscht.

Nach oben Ziehen eines Attributs oder einer Operation:
Verschiebt ein Attribut oder eine Operation, die in allen Unterklassen vorhanden und identisch definiert ist, in die Oberklasse.
Dabei ist, aufgrund der erzeugten Vererbungshierarchie keine Propagation nötig.

Nach unten Schieben eines Attributs oder einer Operation:
Verschiebt das Attribut oder die Operation in die Unterklassen.
Propagation: Existiert ein Knoten, der über der Oberklasse typisiert ist und gleichzeitig das verschobene Attribut oder die Operation enthält, muss der Nutzer entscheiden, über welche der Unterklassen der Knoten künftig typisiert sein soll. Dies wird ausgeführt.

Ersetzen von Unterklassen durch Felder:
Enthalten die betrachteten Unterklassen keinerlei Attribute, Operationen oder Referenzen, so wird ein Aufzählungsdatentyp, der für jede Unterklasse ein Literal anbietet, und zudem ein Attribut vom Typ des Aufzählungsdatentyps innerhalb der Oberklasse erzeugt. Die Vererbungshierarchie ist damit aufgelöst.
Bei der Propagation werden Knoten, welche über den Subklassen typisiert sind, über die Superklassen typisiert und ihnen das passende Literal des Aufzählungsdatentyps hinzugefügt.

Ersetzen einer Enthaltenseinsbeziehung durch eine einfache Referenz:
Wandelt eine Enthaltenseinsbeziehung in eine einfache Referenz um.
Dabei ist keine Propagation nötig.

Ersetzen einer einfachen Referenz durch eine Enthaltenseinsbeziehung:
Wandelt eine einfache Referenz in eine Enthaltenseinsbeziehung um.
Dabei ist keine eigentliche Propagation nötig, jedoch müssen die Regeln validiert werden, um die Exklusivität der Enthaltenseinsbeziehungen zu überprüfen.

Verstecken des Delegierten:
Entfernt die direkte Referenz zwischen zwei Klassen. Eine indirekte Verbindung über eine andere Klasse bleibt bestehen.
Die Propagation erfolgt durch Löschen der die gelöschte Referenz instanziierenden Links. Dabei kann es vorkommen, dass ein betroffener Knoten nicht mehr erreichbar ist. Dies wird durch die Validierung angezeigt.

Entfernen der Klasse in der Mitte:
Erzeugt eine direkte Referenz zwischen zwei Klassen, die zuvor indirekt verbunden

waren.
Zur Propagation werden die Knoten, welche über diese Klassen typisiert sind, durch einen die neue direkte Referenz instanziierenden Link verbunden.

Ersetzen von Vererbung durch Delegation:
Ersetzt eine Vererbungsbeziehung durch eine Referenz.
Die Propagation verläuft folgendermaßen: Falls eine Regel Knoten beinhaltet, die auf die beiden Klassen verweisen, so werden diese durch einen Link - eine Instanz der neuen Referenz - verbunden. Zudem liefert die Validierung Aufschluss über eventuell genutzte, jedoch nicht mehr verfügbare Eigenschaften. Fowlers Definition folgend, sollte dieses Refactoring nur angewendet werden, wenn die Klasse wenig Funktionalität der Oberklasse nutzt. Die wenigen verwendeten, geerbten Operationen müssen manuell umgeschrieben werden, da sie nun eine komplett andere, delegierende Funktionalität aufweisen.

Ersetzen von Delegation durch Vererbung:
Ersetzt eine Referenz durch eine Vererbungsbeziehung.
Zur Propagation werden Links, welche die ersetzte Referenz instanziieren, gelöscht.

Zusammenfassend betrachet ist die Spannweite der Änderungen beträchtlich: Während bei dem Extrahieren einer Oberklasse gar keine Propagation stattfindet, muss bei dem Umbenennen eines Elements jede Regel komplett untersucht werden. Bei komplexen Refactorings wie dem Extrahieren einer Klasse werden die Knoten und Kanten der Regeln sogar deutlich verändert.

8.3 Implementierung

Nachdem klargestellt ist, welche Refactorings implementiert sind und wie deren Propagation ausgeführt wird, wird hier die Umsetzung mit ModGraph erklärt. Dazu wird zuerst die Architektur des Refactorings im Allgemeinen, danach die Umsetzung mit Graphtransformationsregeln am Beispiel erklärt.

8.3.1 Architektur des propagierenden Refactorings

Die Refactorings sind als M1-Anwendung der Regeln implementiert.[4] Dies bedeutet, dass die definierten Regeln auf einem Modell angewendet werden und Operationen eines Metamodells implementieren. Für das propagierende Refactoring ist dieses ein Wrapper-Metamodell. Es ist eine echte Erweiterung des Ecore-Metamodells einerseits und des Graphtransformationsmetamodells andererseits. Gleichzeitig ist es eine Instanz des Ecore-Metamodells, da es als Xcore-Modell implementiert wird. Abbildung 8.5 zeigt dieses, im Weiteren als Refactoring-Metamodell bezeichnete Modell, im Kontext der an einem propagierenden Refactoring beteiligten Elemente. Die Abbildung stellt eine Variante der M1-Anwendung aus Abbildung 5.3, Kapitel 5 dar.

[4] Bezüglich der Anwendungsmöglichkeiten der Regeln, siehe Kapitel 5.

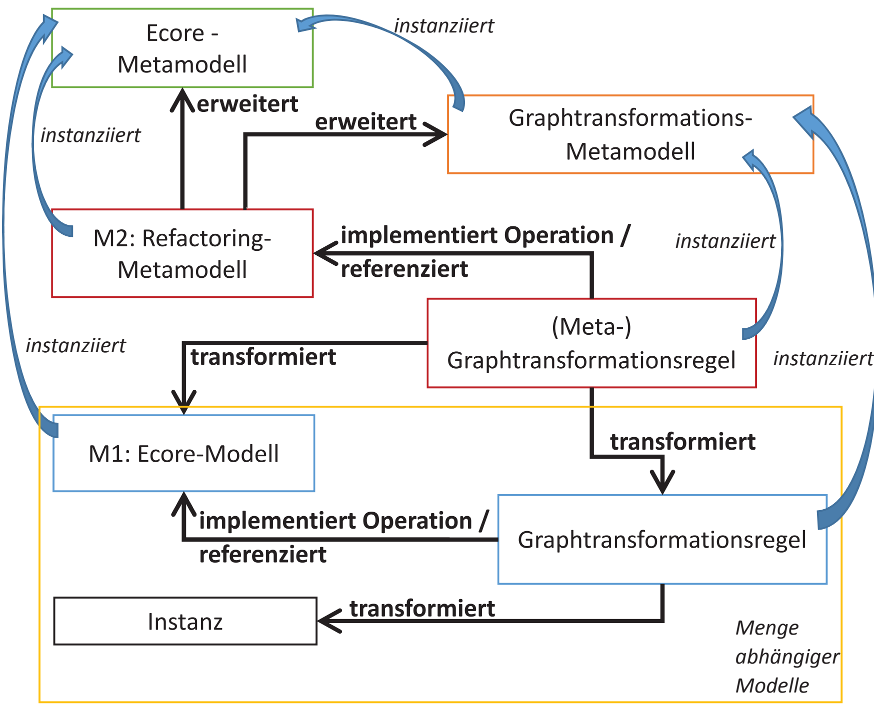


Abbildung 8.5: *Zusammenhang der zum propagierenden Refactoring genutzten Modelle und Regeln*

Das Refactoring-Metamodell erweitert die beiden vorgegebenen Metamodelle im Wesentlichen um zwei Klassen. Eine enthält Operationen zum Refactoring auf einem Ecore-Modell. Die andere definiert solche zur Propagation der Änderungen auf die Regeln.

Zudem wird in Abbildung 8.5 deutlich, dass das Refactoring *für* und *mit* ModGraph erstellt ist. Die Refactorings und Propagationen sind - sofern dies sinnvoll ist - mit Graphtransformationsregeln erstellt. Die nötigen Kontrollflüsse und einfachen Operationen sind in Xcore definiert. Damit verfolgt auch das propagierende Refactoring ModGraphs globales Ziel, eine echte Erweiterung von EMF unter Nutzung des Mehrwerts von Graphtransformationsregeln anzubieten. Dies bedeutet, dass Graphtransformationsregeln Operationen im Refactoring-Modell implementieren und damit das Ecore-Modell transformieren. Zudem gibt es zur Propagation Graphtransformationsregeln, die andere Graphtransformationsregeln transformieren. Erstere werden Metagraphtransformationsregeln genannt, da sie über das Graphtransformations-Metamodell typisierte Knoten nutzen, um Änderungen an Regeln zu modellieren. Der untere Teil der Abbildung 8.5 zeigt die Menge der abhängigen Modelle. Die von außerhalb der Modellmenge in diese eingehenden und mit transformiert markierten Pfeile entsprechen dabei den Eingriffen des propagierenden Refactorings in diese Menge.

Auf diese Art und Weise werden alle in Abschnitt 8.2 aufgelisteten Refactorings implementiert. Sie können auf der ModGraph-Homepage[5] heruntergeladen werden. Aufgrund

[5]Siehe `http://btn1x4.inf.uni-bayreuth.de/modgraph/homepage`

```
class Refactoring{
    ...
    op void changeBidirectionalToUnidirectionalReference(EReference ref) {
        if(ref.EOpposite != null)
            EcoreUtil::remove(ref)
    }
    ...
}
```

Auflistung 8.1: *Xcore Implementierung des Refactorings zur Umwandlung einer bidirektionalen in eine unidirektionale Referenz*

der Menge der Refactorings wird im Folgenden die Implementierung des Refactorings zum Umwandeln einer bidirektionalen in eine unidirektionale Referenz und dessen Propagation beispielhaft vorgestellt.

Um ein propagierendes Refactoring auszuführen, wurden Konnektoren in Java geschrieben, die dessen Aufruf aus dem Ecore-Modell durch einen Rechtsklick auf ein Modellelement erlauben. Dadurch werden nacheinander die Änderung am Modell und an allen zugehörigen Regeln vorgenommen. Dabei ist in manchen Fällen eine Interaktion mit dem Nutzer nötig, da die Propagation bei textuellen Ausdrücken gelegentlich Eingaben fordert. Die Regeln werden nach Abschluss der Änderungen validiert, vor allem um die Navigierbarkeit ihrer Knoten zu prüfen. Ist eine Regel nach einem propagierenden Refactoring nicht mehr valide, so wird sie markiert, um den Nutzer darauf aufmerksam zu machen. Dieser muss nun die Regel ausnahmsweise manuell anpassen.

8.3.2 Beispielrefactoring: Ändern einer bidirektionalen Referenz in eine unidirektionale

Das hier näher betrachtete Refactoring ist das - bereits in der Problemstellung motivierte - Umwandeln einer bidirektionalen Referenz in eine unidirektionale. Betrachtet man zunächst den Sachverhalt und seine Verbesserung nach Fowler:

„Es gibt eine beidseitige Assoziation,
jedoch benötigt eine Klasse Eigenschaften der anderen Klasse nicht mehr.“

„Entferne das unnötige Ende der Assoziation.“
(übersetzt aus [33])

Im EMF-Kontext bedeutet dies, dass die Entgegengesetzte einer Referenz gelöscht werden kann, sofern sie nicht mehr benötigt wird. Dies stellt eine sehr einfache Änderung im Modell, die in Xcore durch den Aufruf der geeigneten Utility-Methode realisiert ist, dar. Auflistung 8.1 zeigt die in Xcore modellierte Methode, welche die Referenz korrekt aus ihrem Kontext löscht. Dazu wird die Referenz mittels einer von EMF gestellten Utility-Klasse gelöscht (Zeile 5).

Die Propagation dieses Refactorings gestaltet sich komplexer. Instanzen dieser Referenzen können sowohl im Graphmuster als auch in der NAC in Form von Links vorkommen. Die Propagation hierfür ist in Auflistung 8.2 gezeigt. Sie ruft die beiden in

```
class Propagation{
...
    op void propagateChangeBidirectionalToUnidirectionalReference(
            EReference formerOpposite , GraphTransformationRule rule) {
        try {
            var pattern =
                    propagateChangeBidirectionalToUnidirectionalReferenceGraphPattern(
                        formerOpposite, rule)
            for (GTNegativeApplicationCondition nac :
                        rule.negativePattern as GTNegativeApplicationCondition[]) {
                this.propagateChangeBidirectionalToUnidirectionalReferenceNAC(
                        formerOpposite, nac)
            }
        } catch (Exception e) {
            e.printStackTrace()
        }
        // Untersuche textuelle Elemente.
    }
...
}
```

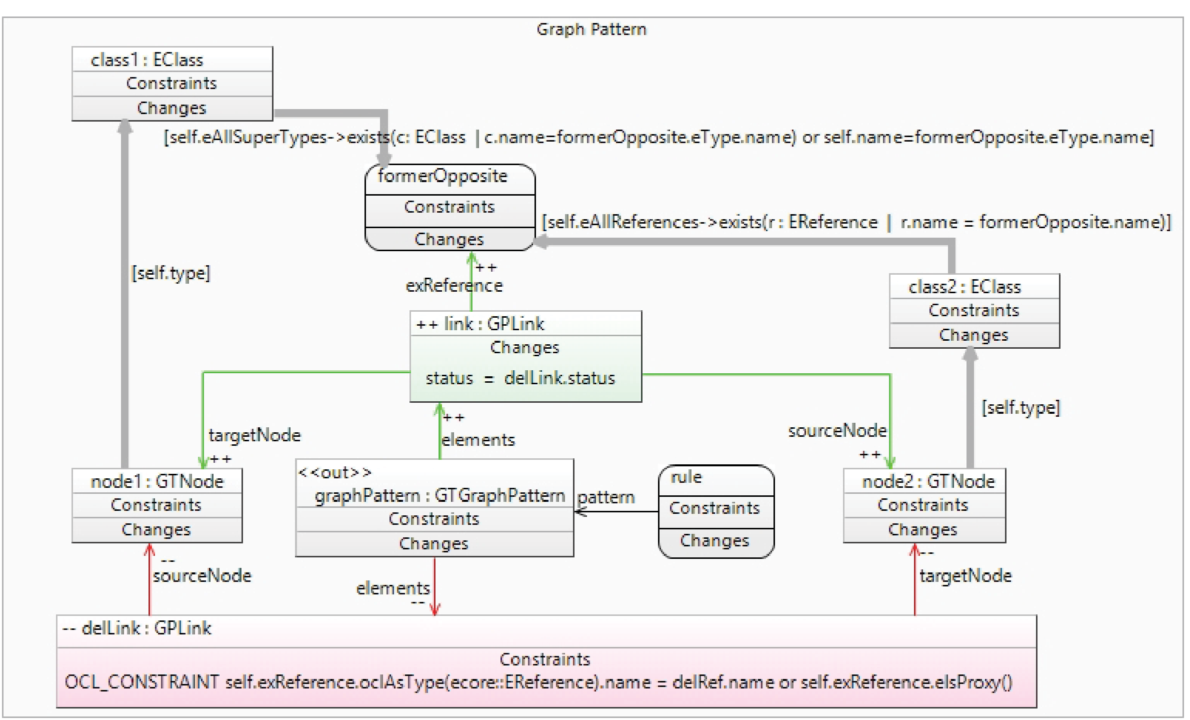
Auflistung 8.2: *Xcore Kontrollfluss: Steuerung der Regeln zur Propagation des Refactorings zur Umwandlung einer bidirektionalen in eine unidirektionale Referenz*

Abbildung 8.6 dargestellten Metaregeln auf. Dabei wird zuerst die Propagation der Regel in das Graphmuster aufgerufen (Zeilen 6 & 7). Danach wird, für jede NAC einzeln, die Regel zu ihrer Änderung aufgerufen (Zeilen 9-13).

Die obere Regel der Abbildung 8.6 zeigt die vorzunehmenden Änderungen im Graphmuster. Der Metaregel werden als Parameter eine Regel rule und die ehemals gegenüberliegende Referenz der gelöschten formerOpposite übergeben. Von der Regel rule ausgehend wird das Graphmuster graphPattern gefunden und von dort aus der zu löschende Link delLink. Die vom Link instanziierte Referenz darf nicht mehr auffindbar sein. Dies wird durch eine OCL Bedingung an den den Link repräsentierenden Knoten der Metaregel ausgedrückt. Dies könnte jedoch auch andere Ursachen als das Refactoring haben. (Zum Beispiel kann der Nutzer die Referenz gelöscht haben.) Daher werden anschließend sein Quellknoten node1 und sein Zielknoten node2 betrachtet. Beide Knoten sind über das Interface GTNode des Graphtransformations-Metamodells typisiert, da es sich jeweils um beliebige im Muster vorkommende Knoten handeln darf. Die Pfade stellen sicher, dass beide Knoten über Klassen typisiert sind, welche durch die Referenz formerOpposite verbunden werden dürfen. Ist dies der Fall, wird der nicht mehr valide Link delLink entfernt und ein neuer Link link erzeugt. Er wird den Elementen des graphPattern hinzugefügt und ist eine Instanz der Referenz formerOpposite. Der Link verbindet die Knoten node1 und node2 nun in entgegengesetzter Richtung.

Die untere Metaregel in Abbildung 8.6 zeigt die Propagation in den NACs. Da diese aus der Xcore-Operation für jede NAC einzeln aufgerufen wird, werden ihr die NAC nac und die ehemalig gegenüberliegende Referenz übergeben. Die weitere Regel ist analog aufgebaut. Der Link, dessen zugehörige Referenz nicht mehr aufgelöst werden kann, wird gelöscht und durch einen entgegengesetzten ersetzt.

Wendet man dieses Refactoring auf den Bugtracker an, um die bidirektionale Referenz zwischen Project und Ticket in eine unidirektionale umzuwandeln, so erhält man das in



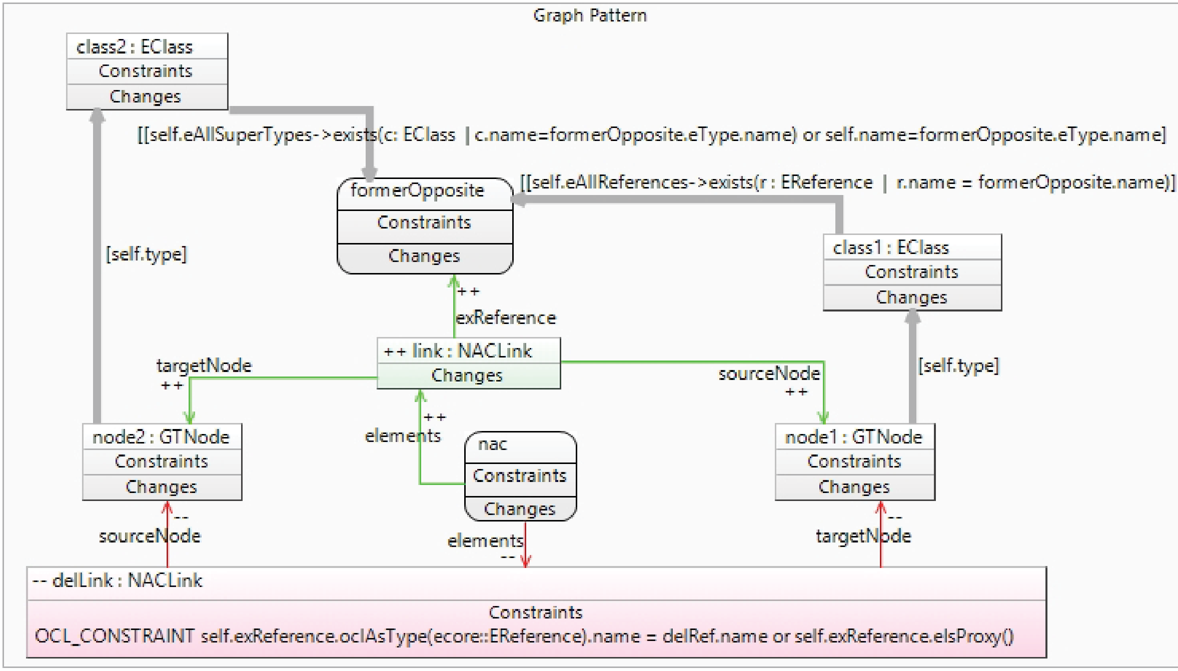


Abbildung 8.6: *ModGraph-Regeln zur Propagation des Ecore-Modell-Refactorings zum Umwandeln einer bidirektionalen Referenz in eine unidirektionale auf betroffene ModGraph-Regeln*

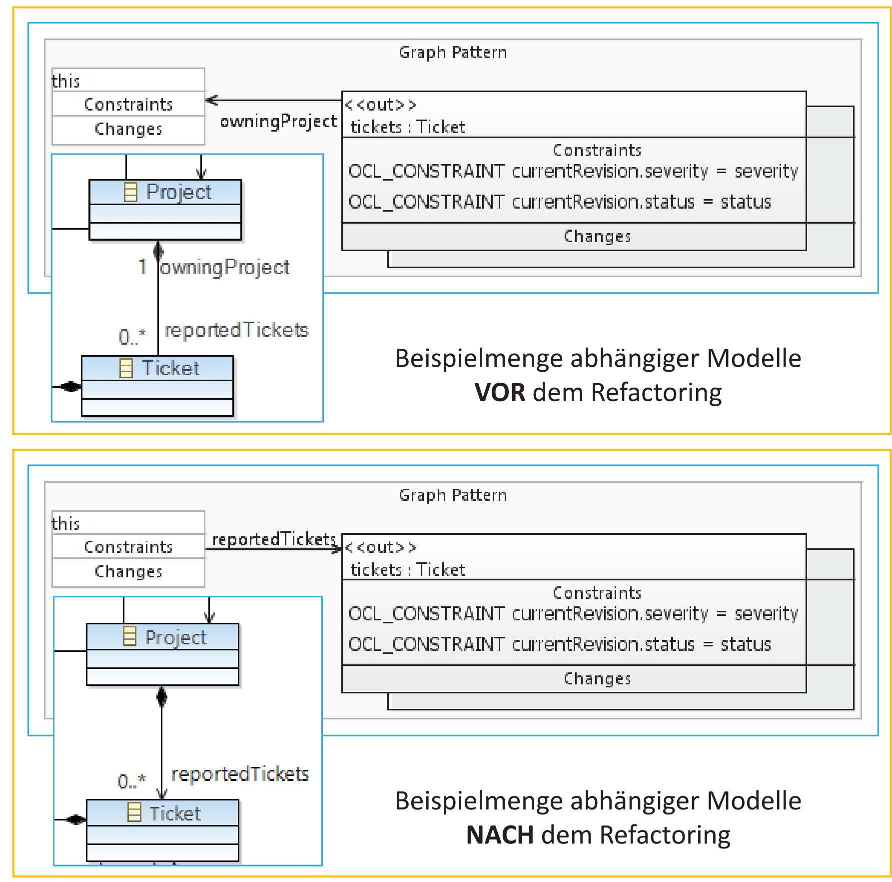


Abbildung 8.7: *Anwendung des propagierenden Refactorings auf das Bugtracker Beispiel: Umwandlung der bidirektionalen Referenz zwischen* Project *und* Ticket *in eine unidirektionale*

Abbildung 8.7 unten gezeigte Ergebnis. Hier werden Screenshots aus der ModGraph-Werkzeugumgebung vor und nach dem propagierenden Refactoring gezeigt. Der obere Teil der Abbildung zeigt einen Ausschnitt aus dem Modell und eine der betroffenen Regeln, getTicketsWithStatusAndSeverity, vor Anwendung des propagierenden Refactorings. Die Referenz zwischen Projekt und Ticket ist bidirektional. Der Link in der Regel zeigt zum aktuellen Objekt des Typs Project. Das Ergebnis nach Ausführung des propagierenden Refactorings, welches die Referenz owningProject ersetzt, ist in Abbildung 8.7 unten dargestellt. Die Referenz zwischen Project und Ticket ist nun eine unidirektionale Referenz. Der Link in der Regel zeigt nun zum mehrwertigen tickets-Objekt.

Eine andere Version dieses Refactorings ist in [82] veröffentlicht. Die hier gezeigte Variante ist eine Weiterentwicklung von [82].[6]

[6]Der explizite Vergleich mit der Veröffentlichung zeigt: Dabei wurde die explizite Verwaltung aller vom Refactoring möglicherweise betroffenen Klassen durch dieses abgeschafft.

8.3.3 Refactoring 2: Extrahieren einer Klasse

Das zweite hier betrachtete Refactoring ist das zur Extraktion einer Klasse. Es bringt einen größeren Eingriff in das Modell und auch die Regeln mit sich. Betrachtet man wiederum Fowlers Definition des Refactorings:

„Es gibt eine Klasse, die Arbeit für zwei verrichtet."

„Erzeuge eine neue Klasse und verschiebe die relevanten Felder und Methoden von der alten in die neue Klasse."
(übersetzt aus [33])

Zusätzlich werden laut Fowler die beiden Klassen über eine Referenz von der alten zur neuen Klasse in Beziehung gesetzt.

Im EMF-Kontext bedeutet dies, dass beim Refactoring des Modells eine Klasse aus einer anderen extrahiert wird und Attribute, Referenzen und Operationen der existierenden Klasse in die neue verschoben werden, sowie dass eine unidirektionale Referenz von der alten zur neuen Klasse erzeugt werden muss. Bei der Propagation werden die Knoten, die für Objekte der Klasse stehen, analysiert. Beinhalten diese verschobene Attribute oder Operationsaufrufe verschobener Operationen, wird ein neuer Knoten erzeugt und diese Elemente werden verschoben. Zudem werden der neue und der alte Knoten über einen Link, welcher die neue Referenz instanziiert, verbunden. Werden Referenzen von der alten zur neuen Klasse verschoben, müssen auch die zugehörigen Links an den neuen Knoten verschoben werden. Dieses propagierende Refactoring hat sowohl prozedurale als auch regelbasierte Anteile.

Die Extraktion der Klasse im Modell erfolgt regelbasiert und ist in Abbildung 8.8 dargestellt. Die Regel wird direkt beim Refactoring aufgerufen. Der Nutzer markiert bei Aufruf des Refactorings die Klasse sowie die Attribute, Referenzen und Operationen, die aus dieser Klasse extrahiert werden sollen. Sie werden der Regel als Parameter übergeben. Die Regel prüft in ihrer Vorbedingung, dass die weiteren Parameter(Klassenname und Referenzname) nicht leer sind.[7] Ist dies der Fall, wird das Paket (aPackage) zur Klasse (existingClass) ermittelt und durch die NAC geprüft, ob dieses Paket bereits eine Klasse des vorgegebenen Namens für die zu extrahierende Klasse enthält (analog für den Referenznamen). Ist dies nicht der Fall, wird die neue Klasse (newClass), sowie eine neue Referenz (newReference) zwischen der existierenden und der neuen Klasse, erzeugt. Die zuvor vom Nutzer angegebenen Attribute und Referenzen (hier zu structuralFeaturesToMove zusammengefasst), sowie die Operationen (operationsToMove) werden nun verschoben. Damit ist das Refactoring auf dem Modell abgeschlossen.

Die Propagation zeigt sich deutlich komplexer. Auflistung 8.3 zeigt einen interessanten Ausschnitt. Hier werden zunächst die Regeln ermittelt, deren Graphmuster Knoten enthalten, die auf die veränderte Klasse zeigen (Zeilen 13-22). Diese Knoten werden in den folgenden Zeilen (23-60) untersucht und Elemente, die auf verschobene Attribute, Felder und Operationen zeigen, ermittelt. In den Zeilen 62 und 63 wird die Regel

[7] Wäre eine leere Zeichenkette zugelassen, bräuchte man keine Vorbedingung, die null-Prüfung erfolgt während der Mustersuche.

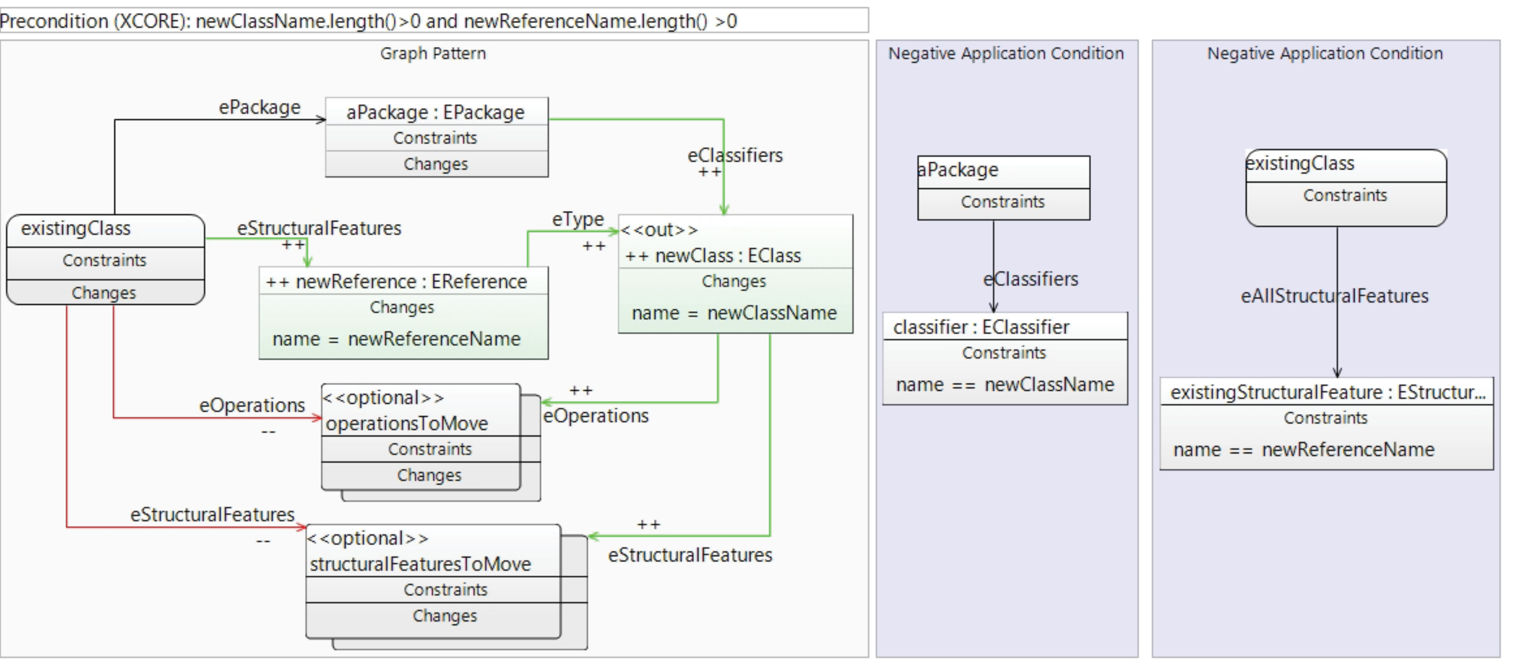


Abbildung 8.8: *ModGraph-Regel zum Extrahieren einer Klasse*

```
op EObject[] propagateExtractClassNEW(EClass extractedClass,
        EClass existingClass, GraphTransformationRule rule) {

    var GTGraphPattern pattern = rule.pattern
    var EList<EObject> objectsToRemove = newBasicEList()
    var GTNode existingNode
    var EList<GTField> gTFields = newBasicEList()
    var EList<GTAttribute> gTAttributes = newBasicEList()
    var EList<GPLink> gTInLinks = newBasicEList()
    var EList<GPLink> gTOutLinks = newBasicEList()

    for (GPElement gpu : pattern.elements) {
        if (gpu instanceof GPUnboundNode && ((gpu as GPUnboundNode).^type as
                    EClass).name.equals(existingClass.name)) {
            existingNode = gpu as GPUnboundNode
        } else if (gpu instanceof GTThisObject &&
                ((gpu as GTThisObject).^type as EClass).name
                    .equals(existingClass.name)) {
            existingNode = gpu as GTThisObject
        } else{existingNode =  gpu as GPBoundNode}
    }
    if (existingNode != null) {
        for (EAttribute ea : extractedClass.getEAttributes()) {
            for (GTAttribute gta : existingNode.attributes) {
                if (gta.exAttribute.eIsProxy() && getNameFromURI(gta.exAttribute)
                    .equals(ea.name)) {
                    gTAttributes.add(gta);
                }
            }
        }
        for (EOperation eo : extractedClass.getEOperations()) {
            for (GTField gtf : existingNode.getFields()) {
                if (gtf.getKind().getName().equals("OPERATION_CALL")) {
                    var String content = gtf.content
                    var String operationName =
                        content.split(Pattern.quote("(")).get(0)
                    if (operationName.equals(eo.name)) {
                        gTFields.add(gtf)
                    }
                }
            }
        }
        for (EReference er : extractedClass.getEReferences()) {
            for (GPLink gtl : existingNode.outgoingEdges
                    .filter(e|e instanceof GPLink) as GPLink[]) {
                if (gtl.exReference.eIsProxy() &&
                        getNameFromURI(gtl.exReference).equals(er.name)) {
                    gTOutLinks.add(gtl)
                }
            }
            for (GPLink gtl : (existingNode.incomingEdges
                    .filter(e| e instanceof GPLink))
                        as  GPLink[]) {
                if (gtl.exReference.eIsProxy()
                        && getNameFromURI(gtl.exReference).equals(er.name)) {
                    gTInLinks.add(gtl)
                }
            }
        }
        try{    if(!gTAttributes.isEmpty() || !gTFields.isEmpty() ||
                    !gTInLinks.isEmpty() ||  !gTOutLinks.isEmpty() )
                propagateExtractClassSF(extractedClass, existingClass, rule,
                        gTAttributes, gTInLinks, gTOutLinks, gTFields)
        }catch(GTFailure f){}
...
// Textuelle Felder parsen und NAC analog behandeln sowie grafisches Update
}
```

Auflistung 8.3: *Xcore Implementierung der Propagation zum Refactoring Extrahieren einer Klasse*

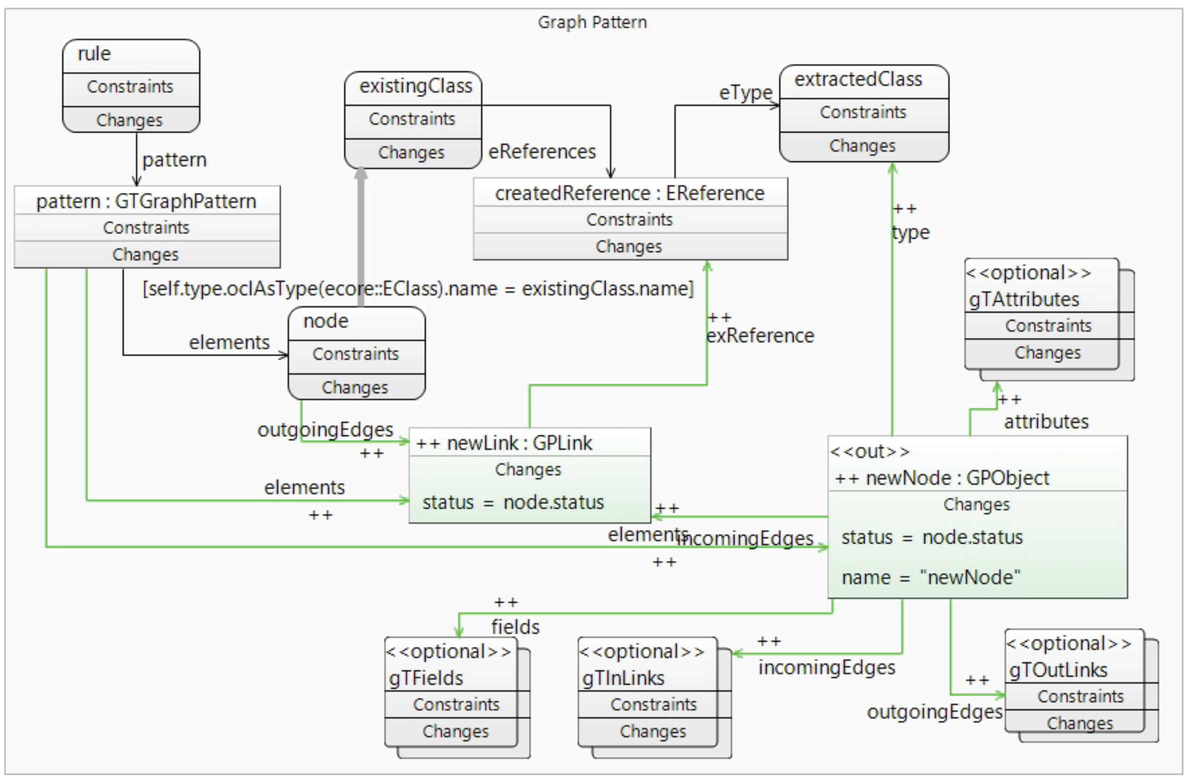


Abbildung 8.9: *ModGraph-Regel zur Propagation der strukturellen Anteile des Refactorings Extrahieren einer Klasse*

zur Transformation der strukturellen Eigenschaften aufgerufen. Sie ist in Abbildung 8.9 dargestellt. Allerdings geschieht dies nur, wenn Elemente der Regel bearbeitet werden müssen, wie die Bedingung direkt darüber zeigt (Zeilen 60 & 61). Danach werden alle anderen textuellen Felder der Regel betrachtet. Dies ist im Wesentlichen durch Parsen von Zeichenketten realisiert und daher nicht abgebildet. Zudem wird ein grafisches Update der Regel ausgeführt, so dass nicht nur der Baumeditor, sondern auch die grafische Darstellung, das GMF-basierte Diagramm, den neuen Zustand der Regel anzeigt.

Die Regel zur Propagation der Änderungen an den Links und Attributen im Graphmuster ist in Abbildung 8.9 dargestellt. Hier werden die Regel (rule), die existierende Klasse (existingClass), die neu extrahierte Klasse (extractedClass), der betroffene Knoten (node) und alle zu verschiebenden Attribute (gTAttributes), Felder (gTFields) und Links übergeben. Die Links werden dabei in eingehende (gTInLinks) und ausgehende (gTOutLinks) Links des Knotens unterschieden.

In der Regel wird ein neuer, einwertiger, ungebundener Knoten erzeugt (newNode), der den Status des übergebenen, betroffenen Knotens zugewiesen bekommt. Ihm werden übergebene Felder, Links und Attribute zugewiesen. Es handelt sich hier um die Zuweisung eines neuen Containers für die Felder, Links und Attribute. Sie werden damit automatisch aus dem Knoten node entfernt (Exklusivität). Des Weiteren wird ein neuer Link (newLink) erzeugt, der eine Instanz der neu erzeugten Referenz im Modell darstellt.

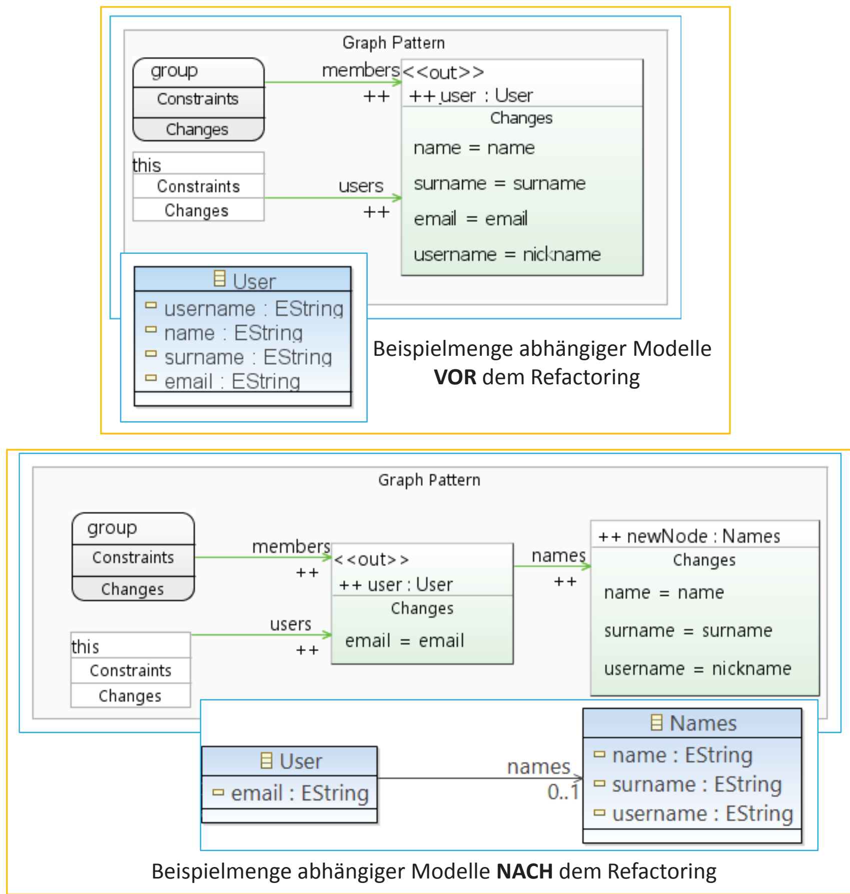


Abbildung 8.10: *Anwendung des propagierenden Refactorings auf das Bugtracker Beispiel: Extraktion der Klasse* Names *zur Verwaltung der Namen des Nutzers*

Er verbindet die beiden betrachteten Knoten und erhält den selben Status wie diese. Die Regel zur Propagation der Änderungen in die NAC ist analog aufgebaut. Als Parameter wird hier wiederum die NAC statt der Regel angegeben.

Betrachtet man nun die Ergebnisse des Refactorings in Abbildung 8.10, wurde das Refactoring korrekt ausgeführt. Oben in der Abbildung ist die Klasse User mit allen Attributen gezeigt, sowie das Graphmuster der Regel zur Implementierung der Operation createUser der Klasse BugTracker. Aus der Klasse User werden die zahlreichen Namen extrahiert. Sie werden in die Klasse Names verschoben, die diese im Weiteren verwaltet. Die Klasse User greift auf sie durch die Referenz names zu. Der Zustand des Modells nach dem Refactoring ist in Abbildung 8.10 unten dargestellt. Die Regel wird gemäß der eben beschriebenen Propagation verändert. Ein Link und ein Objekt kommen hinzu. Diese sind über der neuen Referenz bzw. der extrahierten Klasse typisiert. Der Propagationsregel folgend, wird der neue Knoten mit newNode benannt. Ihm wird, ebenso wie User, der Status zu erzeugen zugeordnet. Die veränderte Regel nach Anwendung der Propagation ist in der Abbildung unten dargestellt. Sie ist wieder konsistent zum Modell und das propagierende Refactoring ist abgeschlossen.

9 Integration von Graphtransformationsregeln und modalen Sequenzdiagrammen mit ModGraph und ScenarioTools

In Zusammenarbeit mit Prof. Dr. Joel Greenyer (derzeit Juniorprofessor im Fachgebiet Software Engineering an der Leibniz Universität Hannover) und Prof. Dr. Matthias Tichy (derzeit Assistant Professor in der Software Engineering Group bei Chalmers | University of Gothenburg in Schweden) wird ModGraph prototypisch mit ScenarioTools, einem Werkzeug zum szenarienbasierten Design von software-intensiven Systemen zusammengefügt. Dabei sollen die Verwendbarkeit sowie der Mehrwert des hier vorgestellten ModGraph-Ansatzes im Kontext verteilter, reaktiver Systeme, die unter Anderem im Automotive-Bereich anzusiedeln sind, untersucht werden.

Zur Umsetzung dieses Vorhabens muss zunächst auf konzeptueller Ebene geklärt werden, wie die im ModGraph-Ansatz verwendeten Graphtransformationsregeln mit den in ScenarioTools genutzten modalen Sequenzdiagrammen interagieren können und warum dies von Vorteil ist. Darüber hinaus muss ein Konsens bezüglich der Werkzeugkompatibilität gefunden werden. Die Ergebnisse dieser Überlegungen sind in Winetzhammer et al. [79] veröffentlicht und werden in diesem Kapitel dargelegt. Dazu folgt eine Beschreibung von modalen Squenzdiagrammen, deren Ausführung mittels Play-out, des Werkzeugs ScenarioTools und der Integration von modalen Sequenzdiagrammen und Graphtransformationsregeln.

9.1 Theoretische Überlegungen

Um die Integration von Graphtransformationsregeln und MSDs beschreiben und deren Vorteile besser verstehen zu können, wird ein autonomes Transportsystem als neues Beispiel eingeführt. Es repräsentiert ein typisches dynamisches, reaktives System.

9.1.1 Beschreibung typischer Systeme am Beispiel

Das betrachtete autonome Transportsystem transportiert mittels Robotern Waren. Dieses bezüglich seiner Struktur hochgradig dynamische System reagiert auf Einflüsse aus der Umwelt und beeinflusst seine Umwelt gleichzeitig. Abbildung 9.1 skizziert ein solches struktur-dynamisches, verteiltes, reaktives System. Arbeiter bestellen Waren an

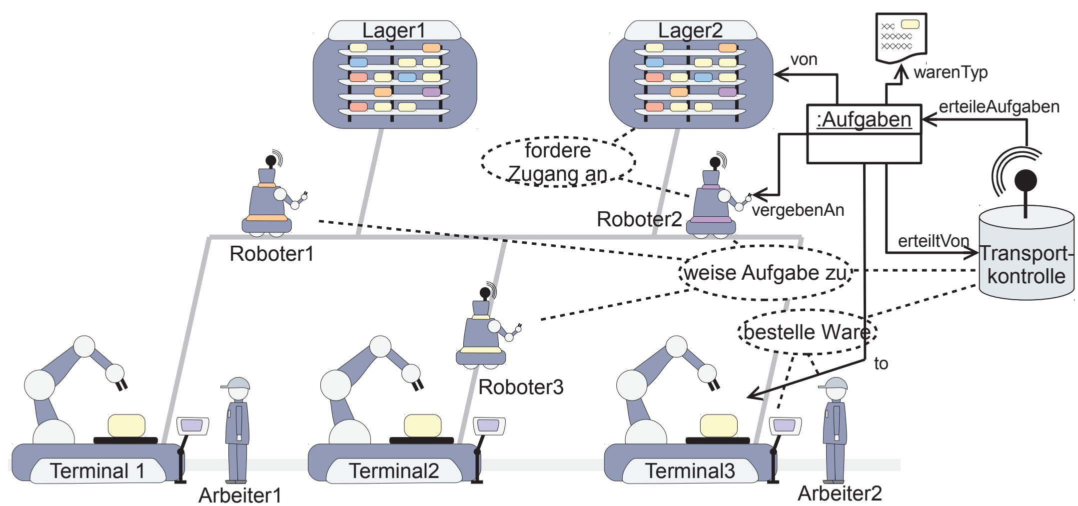


Abbildung 9.1: *Beispiel eines struktur-dynamischen, verteilten, reaktiven Systems: Ein Autonomes Transportsystem*

Terminals, die dorthin gebracht werden. Die korrekte Lieferung wird durch die Transportkontrolle sichergestellt, die Aufgaben an Roboter vergibt. Der beauftragte Roboter holt die Ware aus dem Lager und bringt diese an das Terminal, an welchem sie bestellt wurde. Offensichtlich ist dieses System aus verschiedenen physikalisch verteilten, mechatronischen Komponenten aufgebaut. Diese bestehen aus Hardware und Software, wobei letztere die Funktionaliät des Systems gewährleistet. Die Software verarbeitet Eingaben aus der Umgebung, koordiniert die Komponenten und die Interaktion mit den Arbeitern. Zudem interagiert sie mit der Umwelt durch Aktoren, wie zum Beispiel den Robotern.

Die Modellierung derartiger Systeme ist nicht trivial: Die Anforderungen an das System betreffen oft mehrere Komponenten. Gleichzeitig müssen einzelne Komponenten häufig mehrere Anforderungen erfüllen. Beispielsweise könnte während der Auslieferung einer Ware ein Roboter einen Defekt aufweisen und ein Techniker zu seiner Reparatur benötigt werden. Zudem beziehen sich die Anforderungen oft auf die physikalische oder logische Struktur des Systems, die über mehrere Komponenten verteilt sein kann. Beispielsweise ist die Auswahl des Roboters zur Ausführung eines Jobs von seiner Verfügbarkeit und Position zum Lagerort der Ware abhängig, was für die physikalische Struktur des Systems von Bedeutung ist. Zu welchem Lager ein Roboter Zugang erlangen möchte, hängt vom erhaltenen Auftrag ab. Dies betrifft die logische Struktur des Systems.

Um ein derartiges System in die Software abzubilden, bietet sich zunächst ein Klassendiagramm zur Modellierung seiner statischen Struktur an. Für das betrachtete Beispiel ist das Klassendiagramm im oberen Teil von Abbildung 9.2 dargestellt. Ausgehend von einem zentralen Behälter namens Factory, welcher die gesamte automatisierte Lagerhaltung zu einer Fabrik zusammenfasst, werden hier Klassen für Roboter (Robot), Arbeiter (Worker), Lager (Warehouse), Waren (Item), Warentypen (ItemKind), der Transportkontrolle (TransportSystemControl) und Speichermanagement (StorageManagement) sowie

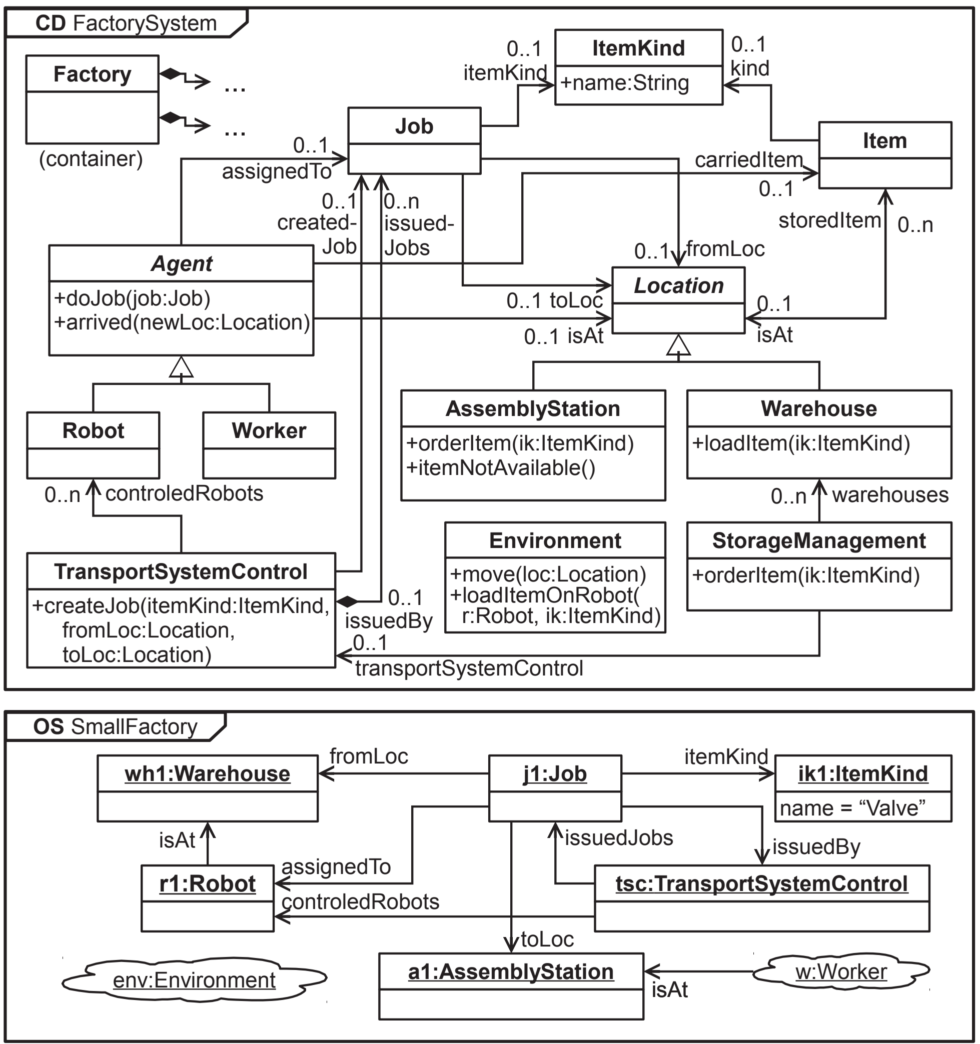


Abbildung 9.2: *Klassendiagramm und beispielhaftes Objektsystem zum autonomen Transportsystem*

für die Umgebung des Systems (Environment) definiert. Außerdem werden alle Klassen attribuiert, passende Operationen definiert und durch Referenzen in Beziehung zueinander gestellt. Weiterhin werden passende Oberklassen eingeführt. Wird dieses Klassendiagramm instanziiert, erhält man ein Objekt-System, das konkrete Objekte des Systems abbildet, die durch Links (Instanzen der Referenzen) miteinander verbunden sein können. Dieses Objekt-System ist gleichwertig zu den bislang betrachteten Instanzen eines Ecore-Modells zu sehen.[1] Ein Beispiel ist im unteren Teil von Abbildung 9.2 gezeigt. Hier werden bereits Systemobjekte und Objekte der Umgebung des Systems unterschieden, letztere werden durch eine Wolke dargestellt und sind nicht direkt durch das System kontrollierbar. Um das Verhalten zwischen den Objekten zu spezifizieren, werden nun

[1]Praktisch gesehen, wird das Objekt-System durch eine Ecore-Modellinstanz simuliert. Dies liegt nahe, da beide Objekte und ihre Beziehungen untereinander darstellen und somit äquivalent sind.

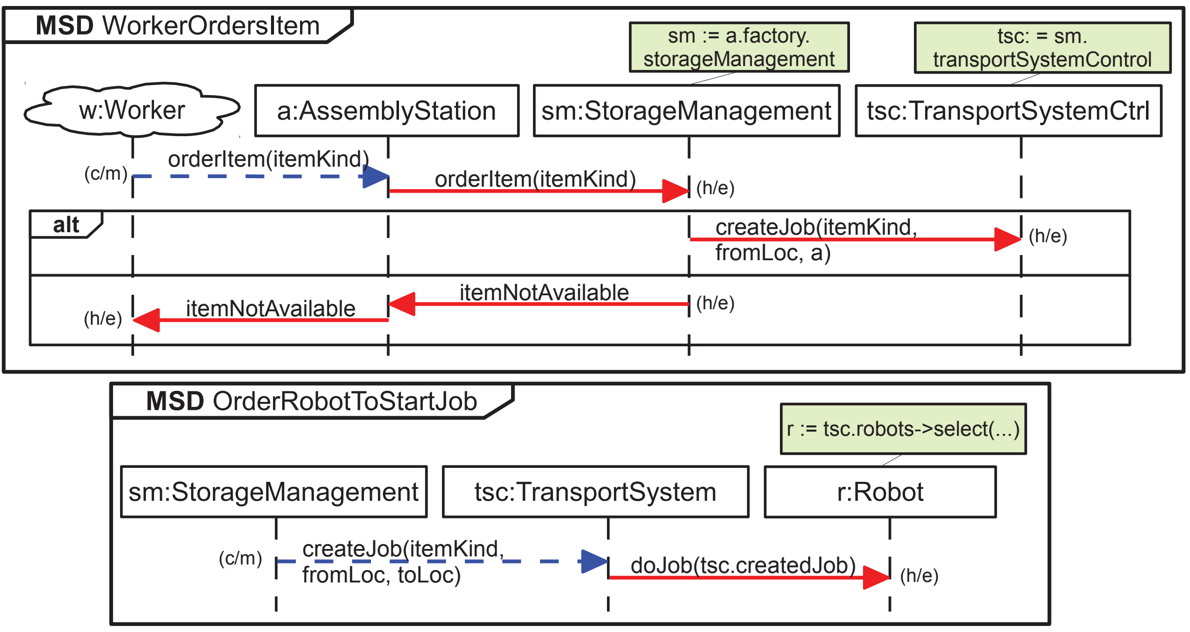


Abbildung 9.3: *Modale Sequenzdiagramme zur Bestellung einer Ware*

modale Sequenzdiagramme, die im nächsten Abschnitt erklärt werden, und die bereits bekannten Graphtransformationsregeln genutzt.

9.1.2 Modale Sequenzdiagramme und Play-out

Modale Sequenzdiagramme (MSDs) [39] stellen eine formale Interpretation von UML-Sequenzdiagrammen, basierend auf Live Sequence Charts [24, 40] dar. Sie wurden von Harel und Maoz definiert und bieten einen szenariobasierten Formalismus, um das Verhalten von - aus Komponenten aufgebauten - Systemen zu beschreiben. Sie erlauben die Spezifikation von Sequenzen von Ereignissen, die in einem System auftreten können, müssen oder keinesfalls auftreten dürfen. Die Komponenten des Systems interagieren durch den Austausch von Nachrichten und reagieren kontinuierlich auf Ereignisse ihrer Umwelt. MSDs können mittels eines - später betrachteten - Play-out Algorithmus ausgeführt werden. Dabei kann ein MSD sowohl Anforderungen an das System als auch Annahmen an seine Umgebung modellieren.

Eine MSD-Spezifikation besteht aus einer Menge von MSDs. MSDs sind existenziell oder universal. Existenziell bedeutet, dass das Diagramm nur Sequenzen enthält, die im System auftreten können. Universal bedeutet, dass das Diagramm die Anforderungen stellt, die von allen Sequenzen von Ereignissen erfüllt werden müssen. Im Folgenden werden universale Sequenzdiagramme betrachtet. Zwei Beispiele solcher MSDs sind in Abbildung 9.3 gezeigt.

Jedes Objekt eines Objekt-Systems wird durch eine Lebenslinie im MSD dargestellt. Dabei werden kontrollierbare Komponenten des Systems und unkontrollierbare Objekte seiner Umgebung unterschieden. Die Systemobjekte (Rechtecke mit Lebenslinie in Abbil-

dung 9.3) werden auch System genannt, die Umgebungsobjekte (Wolken mit Lebenslinie in Abbildung 9.3) Umgebung.

Zwischen den Objekten eines Objekt-Systems findet ein Nachrichtenaustausch statt. Diese Nachrichten entsprechen Methoden, die im Klassendiagramm modelliert sind. Sie haben einen Sender und einen Empfänger und werden je nach der Art ihres Senders in Systemnachrichten und Umgebungsnachrichten unterteilt. Alle Nachrichten sind hier synchron. Dies bedeutet, dass die gesendete und die empfangene Nachricht zusammen ein atomares Ereignis sind. Daher werden sie auch Nachrichten-Ereignis genannt.

Die Nachrichten innerhalb eines MSDs heißen auch Diagramm-Nachrichten und stehen für ein Ereignis innerhalb des Objekt-Systems. Da die Lebenslinien für Objekte des Objekt-Systems stehen, hat jede Diagramm-Nachricht eine sendende und eine empfangende Lebenslinie. Jede Nachricht in einem universalen MSD hat eine Temperatur. Sie ist entweder heiß (durchgehende rote Pfeile mit Markierung h für hot in Abbildung 9.3) oder kalt (blaue gestrichelte Pfeile mit Markierung c für cold in Abbildung 9.3). Des Weiteren ist jeder Nachricht eine Ausführungsart zugeordnet, die entweder ausführend (durchgehender Pfeil mit Markierung e für executed in Abbildung 9.3) oder überwachend (gestrichelter Pfeil mit Markierung m für monitored in Abbildung 9.3) ist. Dabei dürfen Nachrichten, die überwachend sind, auftreten, während auszuführende Nachrichten letztendlich vorkommen müssen. Wird zudem eine heiße Nachricht erwartet, darf keine andere Nachricht auftreten.

Neben der Modellierung von Abläufen im System ist es möglich, MSDs zu spezifizieren, die Annahmen an die Umgebung treffen. Sie werden analog zu den bereits besprochenen (Anforderungs-)MSDs spezifiziert und führen zusätzliche verbotene Nachrichten ein. Diese dürfen keinesfalls im Verlauf des MSDs eintreten, außer die Nachricht ist gerade freigegeben.

Um die Ausführungssemantik eines MSDs im Play-out besser verstehen zu können, müssen weitere Konzepte dargestellt werden. Zunächst wird die Unification von Diagramm-Ereignissen und Nachrichten-Ereignissen besprochen. Diese tritt auf, sobald ein Nachrichten-Ereignis im Objekt-System mit einer Diagramm-Nachricht eines MSDs identifiziert wird, wenn also beide auf die gleiche Operation verweisen und deren Sender und Empfänger durch die entsprechenden Lebenslinien innerhalb des MSDs repräsentiert werden. Ist dies der Fall, wird eine aktive Kopie eines MSDs, genannt aktives MSD, erzeugt. Dabei dürfen mehrere MSDs gleichzeitig aktiv sein, um mehrere Bedingungen an das System stellen zu können. Es wird im Folgenden angenommen, dass das MSD genau mit einer kalten, überwachenden Nachricht beginnt, die vereinheitlicht wird. Treten im weiteren Verlauf Ereignisse auf, die mit den Nachrichten des MSD übereinstimmen, so wird das aktive MSD weiter ausgeführt. Die Ausführung wird durch den sog. Schnitt (Cut) angegeben. Dieser markiert für jede Lebenslinie die anliegenden Enden der Nachrichten, die vereinheitlicht wurden. Diese Punkte werden Position (location) der Lebenslinie genannt. Befindet sich der Schnitt auf der sendenden und empfangenden Lebenslinie direkt vor einer Nachricht, so wird diese Nachricht freigegeben. Der Schnitt nimmt dabei die Temperatur und Ausführungsart der Nachricht an. Eine freigegebene, ausführbare Nachricht heißt aktive Nachricht. Wenn der Schnitt die letzte Position jeder Lebenslinie eines aktiven MSDs passiert hat, wird die aktive Kopie beendet.

Während der Ausführung, sprich des Play-outs können Verstöße auftreten, indem oben beschriebene Ausführungsmechanismen nicht beachtet werden. Ein Sicherheitsverstoß tritt auf, wenn der Schnitt (cut) heiß ist und ein Ereignis stattfindet, das nicht mit einer freigegebenen Nachricht im MSD identifiziert werden kann. Ist der Schnitt bei einem solchen Ereignis kalt, spricht man von einem kalten Verstoß. Sicherheitsverstöße dürfen keinesfalls vorkommen. Sie führen zum sofortigen Anhalten des Systems. Kalte Verstöße ziehen das sofortige Beenden des aktiven MSDs nach sich. Ist die kommende Nachricht und damit der Schnitt zudem ausführend, bedeutet dies, dass das aktive MSD fortfahren muss. Ist dies nicht möglich, entsteht im Play-out ein Lebendigkeitsverstoß.

Bezüglich seiner Konsistenz wird bei der Ausführung eines MSDs pauschal angenommen, dass Ereignisse der Umgebung jederzeit auftreten können und dass das System immer schnell genug ist, um eine endliche Anzahl von Nachrichten zu senden, bevor das nächste Umgebungsereignis stattfindet. Ein Lauf des Systems und seiner Umgebung besteht aus einer unendlichen Anzahl von Nachrichten. Ein Lauf genügt dem System, wenn keine Sicherheits- oder Lebendigkeitsverstöße auftreten und das System keine unbegrenzte Anzahl von Schritten ausführt, sondern immer wieder auf ein Ereignis aus seiner Umgebung wartet. Die MSD-Spezifikation ist konsistent und realisierbar, falls die Systemobjekte auf jede im System mögliche Sequenz von Umgebungsereignissen reagieren können, so dass der Lauf der Spezifikation des MSDs genügt. Diese Systeme werden als valide Implementierungen der MSD-Spezifikation aufgefasst.

Weitere Details zur Ausführung von MSDs sind Harel und Maoz [39] zu entnehmen.

Im Beispiel in Abbildung 9.3 beauftragt ein Arbeiter (w:Worker) an einem Terminal (a:AsssemblyStation) die Lieferung einer Ware eines Warentyps (Parameter itemKind der ersten Nachricht). Das Terminal muss nun eine Nachricht an das Speichermanagement (sm:Storagemanagement) senden. Dieses entscheidet nun, ob die Aufgabe erstellt wird (und letztlich einem Roboter zur Ausführung übergeben) oder die gewünschte Ware nicht vorhanden ist. Hierzu wird das alt-Fragment, das eine Fallunterscheidung kennzeichnet, benötigt. Wird die Aufgabe erstellt, sendet das Speichermanagement eine Nachricht an das Transportsystem (tsc:TransportSystemControl)[2]. Dieser Aufruf aktiviert das untere der in Abbildung 9.3 gezeigten MSDs, das die Aufgabe dem Roboter zuweist.

Am konkreten Beispiel wird zudem deutlich, dass eine Lebenslinie nicht nur für ein bestimmtes Objekt, sondern auch für eine Klasse von Objekten stehen kann. Dazu können beim Auftreten von Nachrichten die Lebenslinien dynamisch an Objekte gebunden werden. Dabei werden die sendende und empfangende Lebenslinie der ersten Nachricht bei der Vereinheitlichung der Nachrichten zur Aktivierung des MSDs gebunden. Die übrigen Lebenslinien werden durch spezielle Bindungsausdrücke, die an der Lebenslinie angebracht werden, gebunden (grüne Kästchen über den Lebenslinien in Abbildung 9.3). Bindungsausdrücke sind in OCL spezifiziert, wobei die Namen der Lebenslinien als Variablen genutzt werden dürfen. Für Details hierzu sei auf Brenner et al. [17] verwiesen.

Eine weitere Eigenschaft der hier betrachteten MSDs ist die Tatsache, dass Nachrichten Parameter anhaften. Diese werden durch die Operation, welche die Nachricht repräsentiert, definiert. Parameter sind über primitive Typen oder Klassen typisiert.

[2] Aus Platz- und Layoutgründen wurde dies zu tsc:TransportSystemCtrl abgekürzt.

Jede Nachricht muss zu ihrer Ausführung die Werte für alle ihr zugeordneten Parameter mitbringen. Dies bedeutet, dass primitive Werte oder Objektreferenzen übergeben werden. Zudem kann jede Diagramm-Nachricht durch die Zuweisung von Konstanten, die Referenzierung von Lebensliniennamen oder anderen Variablen Werte für die Parameter spezifizieren. Beispielsweise wird bei einer createJob-Nachricht als Ziel der Ware das Terminal angegeben, an dem die Ware bestellt wurde. Dies entspricht der Übergabe des Terminals (a) als dritten Parameter an die Nachricht, da dieser in der Methode createJob(...) den Zielort bestimmt.

Eine weitere Eigenschaft von MSDs sind Diagramm-Variablen. Sie werden innerhalb des MSDs erstellt und sind nur hier gültig. Diese Variablen können gebunden oder ungebunden sein, je nachdem, ob ihnen bereits ein Wert zugewiesen ist. In Abbildung 9.3 wird die Variable itemKind durch die Aktivierung des MSDs gebunden. Kann ein Parameter nicht gebunden werden, ist dies, je nach Art der Nachricht, ein Sicherheits- oder kalter Verstoß. Genauere Informationen zu Parametern in Nachrichten finden sich in Harel und Marelly [40] und Brenner et al. [17].

Nachdem der Aufbau des MSDs und die Funktionsweise des Play-out-Algorithmus skizziert wurden, erfolgt hier ein Einblick in die Funktionsweise eines Play-outs: Der Play-out-Algorithmus von Harel und Marelly definert eine Ausführungssemantik für MSDs. Das Prinzip dieses Algorithmus ist, dass bei Auftreten eines Umgebungsereignisses ein oder mehrere MSDs aktiviert oder aktive MSDs mit freigegebenen Nachrichten fortgesetzt werden. Daraus wählt der Algorithmus nicht-deterministisch eine Nachricht zum Senden aus, die aktuell zu keinem Sicherheitsverstoß in einem anderen aktiven MSD führt. Dies wiederholt sich, bis keine aktiven Nachrichten mehr vorhanden sind. Dann wartet der Algorithmus auf das nächste Umgebungsereignis und setzt den Prozess für eben dieses Ereignis fort, solange keine Sicherheitsverstöße vorkommen. Zudem können als Erweiterung des Play-outs MSDs einbezogen werden, die Annahmen an die Umgebung treffen. Damit kann gefordert werden, dass das System den Anforderungen entspricht, solange seine Umgebung den getroffenen Annahmen genügt.

Diese Erweiterung sowie der Play-out sind in ScenarioTools implementiert. ScenarioTools ist ein Werkzeug zum szenariobasierten Design von Software-intensiven Systemen. Es wird von der Software Engineering Group der Leibniz Universität Hannover, der DEEPSE Group des Politecnico di Milano, der Software Engineering Group der Universität Paderborn, und der Software Engineering Division von Chalmers University of Technology and University of Gothenburg entwickelt. ScenarioTools ist im Internet unter `http://scenariotools.org` zu finden. Das Eclipse-basierte Werkzeug unterstützt Nachrichten, die einfache Seiteneffekte auf Objekten innerhalb des Objektsystems zulassen. Beispiele hierfür sind Attributzuweisungen am Empfängerobjekt oder das Ändern von einwertigen Referenzen. Komplexe strukturelle Änderungen, wie das Erstellen eines Objekts, das gleichzeitig in seinen Kontext eingebettet werden muss, sind jedoch nicht vorgesehen. Dies stellt eine Lücke in der Entwicklung mit MSDs dar.

9.1.3 Integration von MSDs mit Graphtransformationsregeln

Zum Schließen der eben aufgezeigten Lücke - dem Mangel an Möglichkeiten zur komplexen strukturellen Änderung des Objektsystems - bieten sich Graphtransformationsregeln an, da sie sich, wie in den vorangegangenen Kapiteln gezeigt, sehr gut zur Beschreibung von komplexen Strukturänderungen von Objektsystemen eignen. Im Weiteren wird die Integration von MSDs und Graphtransformationsregeln durch gegenseitige Einflussnahme beider realisiert.

Dabei wird ein geradliniger Integrationsansatz verfolgt: Die Graphtransformationsregeln implementieren weiterhin die im Klassendiagramm modellierten Methoden. Sie werden als Seiteneffekte von Nachrichten ausgeführt. Wird während eines Laufs des Systems eine Nachricht gesendet, die sich auf eine Operation bezieht, welche durch eine Graphtransformationsregel implementiert ist, so wird die Regel ausgeführt. Nachrichten lösen demnach die Ausführung von Regeln aus. Es handelt sich hierbei um eine synchrone Ausführung. Dies bedeutet, dass das nächste Nachrichtenereignis erst nach Beenden der Regelausführung auftritt. Graphtransformationsregeln modellieren also die komplexen Änderungen an einem System, die eine Nachricht nach sich ziehen kann.

Zudem kann eine Graphtransformationsregel die Ausführung von Aktionen beeinflussen. Sie gibt Anforderungen an den strukturellen Kontext vor, in dem das System bestimmte Änderungen durchführen darf. Außerdem trifft sie Annahmen an die Umgebung. Sie bestimmt den strukturellen Kontext, in welchem gewisse Ereignisse in der Umgebung auftreten dürfen. Ist eine Vorbedingung der Regel nicht erfüllt, sei es, indem das Muster der linken Seite nicht gefunden wird oder eine Negative Anwendbarkeitsbedingung (Negative Application Condition) (NAC) erfüllt ist, darf das zugehörige Ereignis nicht auftreten. Es tritt ein Sicherheitsverstoß bezüglich der Systemanforderungen oder der Annahmen an die Umgebung auf.

Diese Vorgehensweise soll am Beispiel des autonomen Transportsystems verdeutlicht werden. Darin wird die Lieferung einer Ware betrachtet. Die zugehörigen Diagramme finden sich in den Abbildungen 9.4 und 9.5.

Abbildung 9.4, zeigt das bereits aus Abbildung 9.3 bekannte MSD in einer geringfügig angepassten Version sowie zwei Graphtransformationsregeln. Die Regeln implementieren die Operationen createJob(...) der Klasse Transportsystem und doJob(...) der Klasse Agent, der Oberklasse von Robot. Die Implementierung von createJob(...) stellt sich als aufwändiger heraus und wird daher mittels einer Graphtransformationsregel modelliert, die eine konzise, visuelle Modellierungsmöglichkeit zur Erstellung der Aufgabe (Job) und deren Einbettung in den Kontext des Systems anbietet. Die Regel zur Implementierung von createJob(...) erstellt dazu ein neues Objekt der Klasse Job und bettet dieses in seinen Kontext ein, indem Links auf die bereits existierenden Objekte erzeugt werden. Dabei fällt auf, dass keine Verbindung zum item - das die angeforderte Ware repräsentiert - hergestellt wird. Dieses dient als Bedingung an die Ausführbarkeit der Regel. Das neue Objekt job: Job dient als Rückgabeparameter der Regel. Er wird der Diagramm-Variablen job im aufrufenden MSD zugeordnet. Damit wird im Übrigen die Referenz createdJob im Klassendiagramm (Abbildung 9.2) überflüssig und kann entfernt werden.

Die Operation doJob(...) könnte auch durch eine Nachricht im Diagramm modelliert

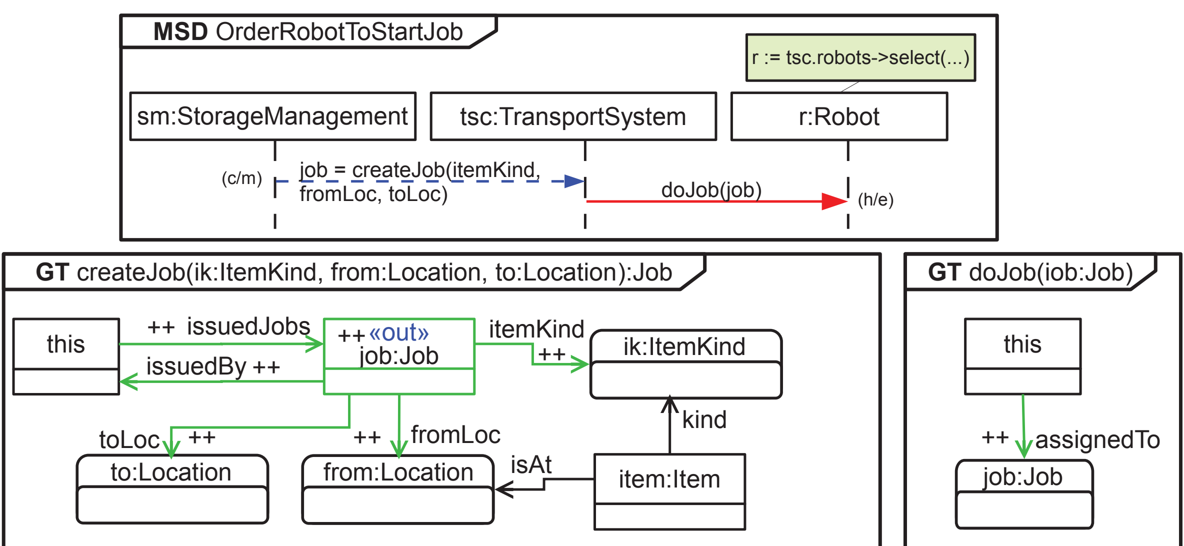


Abbildung 9.4: *Integration von Graphtransformationsregeln und modalen Sequenzdiagrammen am Beispiel - Teil 1*

werden. Dies zeigt, dass es zu Überschneidungen in der Ausdrucksfähigkeit zwischen MSDs und Graphtransformationsregeln kommt. Die Operation erzeugt einen Link zwischen einem Agenten (hier ein Roboter) und dem ihm ab diesem Zeitpunkt zugeordneten Auftrag.

Die Ausführung dieses Auftrags ist in Abbildung 9.5 gezeigt. Zunächst wird das MSD RobotMoveToPickUpLocation betrachtet. Hier bewegt sich der Roboter (r:Robot) zu der durch den Auftrag (Nachrichtenparameter job) vorgegebenen Lagerstelle der Ware. Dies ist als Nachricht an die Umgebung modelliert, da der Roboter sich hier physisch zur angegebenen Position bewegen muss. Die Nachricht, die aus der Umgebung zum Roboter zurück kommt, simuliert die Ankunft des Roboters (z.B. indem die Sensoren des Roboters dies melden). Diese Stelle wird nun mit dem Lager identifiziert, das die Ware - identifiziert durch ihren Warentyp - enthält und in welchem der Roboter beladen wird. Dabei wird angenommen, dass es sich um einen automatisierten Vorgang handelt, der nicht näher betrachtet wird. Der Roboter transportiert die Ware nun zur ebenfalls durch den Auftrag vorgegebenen Stelle und lädt diese dort ab. Das Abladen der Ware ist in Abbildung 9.5 aus Gründen der Übersichtlichkeit nicht eingezeichnet. Es ist in einem anderen, hier nicht näher betrachteten, MSD modelliert.

Dieses MSD berücksichtigt jedoch nicht alle Faktoren des Prozesses. Bereits das Ankommen des Roboters an einer vorgegebenen Stelle kann zu Problemen führen, falls diese nicht frei ist. Um diese Problematik zu berücksichtigen, wird die Operation arrived(...) durch eine Graphtransformationsregel implementiert (unten in Abbildung 9.5), die mittels einer NAC sicherstellt, dass eine Stelle nur dann betreten werden kann, wenn sie frei ist, sprich, wenn sich kein anderer Agent dort befindet. Zusätzlich ist die Nachricht arrived(...) eine Umgebungsnachricht. Sie muss die linke der beiden Annahmen an die Umgebung (Abbildung 9.5, Mitte) einhalten. Diese besagt, dass sich der Roboter an

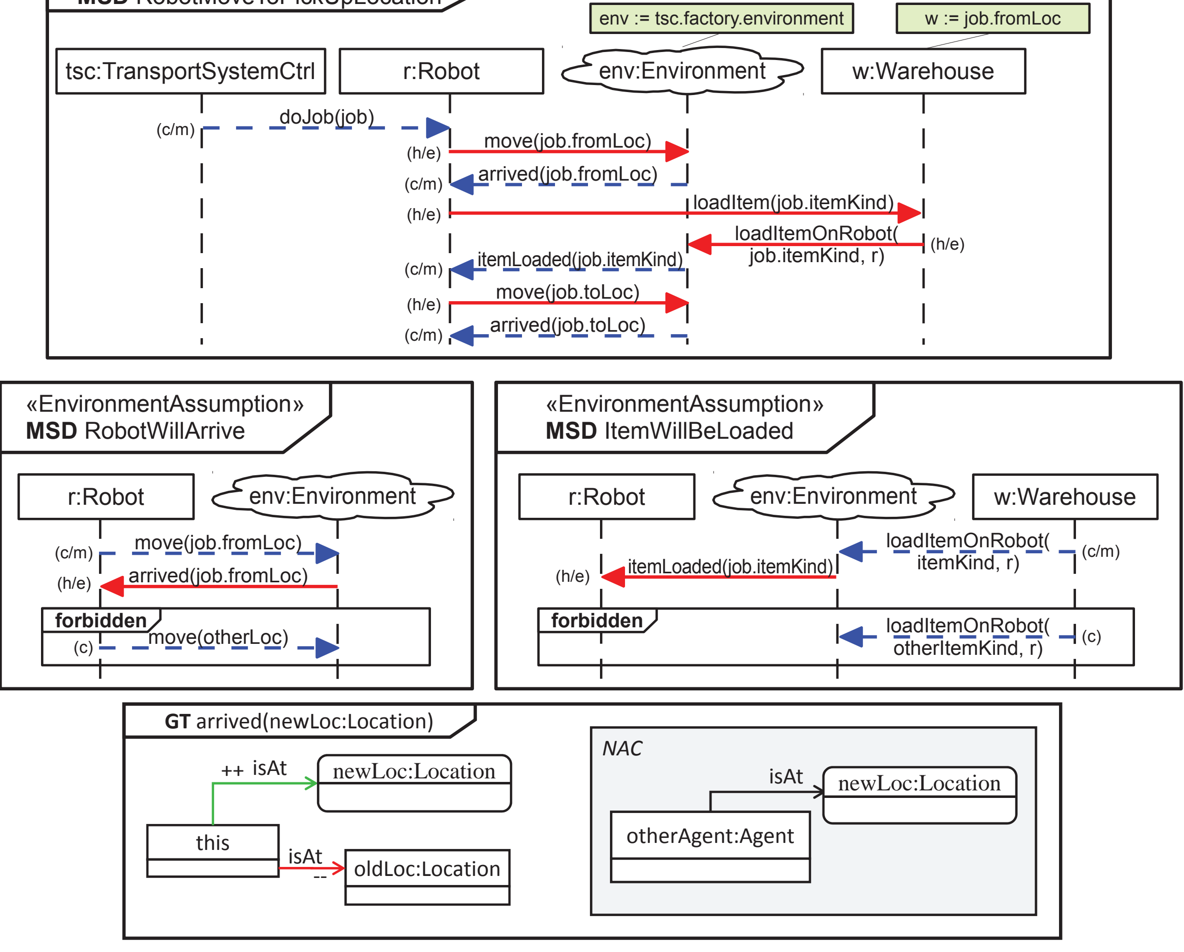


Abbildung 9.5: *Integration von Graphtransformationsregeln und modalen Sequenzdiagrammen am Beispiel - Teil 2*

keine andere Stelle bewegen darf, solange er nicht an der vorgegebenen angekommen ist. Ein Fehlschlagen der Regel würde also zu einem Sicherheitsverstoß bezüglich der Umgebung führen. Die rechte Annahme an die Umgebung betrifft das Beladen des Roboters und stellt sicher, dass der Roboter korrekt nur und ausschließlich mit der angeforderten Ware beladen wird.

Zur Ausführung dieser Integration muss auch der Play-out-Algorithmus angepasst werden, indem er - zusätzlich zur bereits beschriebenen Ausführungssemantik - prüft, ob das zur Ausführung vorgesehene Ereignis keine Vorbedingungen einer Regel verletzt.

Die Ausführung des ersten Teils des laufenden Beispiels (Abbildung 9.4) ist in der zweiseitigen Abbildung 9.6 dargestellt.

Hier wurde zunächst das MSD orderRobotToStartJob aktiviert (Schritt 1 in Abbildung 9.6). Dies wird durch den Schnitt (cut) im Diagramm verdeutlicht, der bei der ersten Nachricht im Diagramm steht, die mit einem Ereignis vereinheitlicht wurde. Sie ruft im nächsten Schritt die Graphtransformationsregel auf (Schritt 2). Für die folgenden

Schritte muss eine Fallunterscheidung getroffen werden.

Ist die Vorbedingung der Regel verletzt, findet sich hier keine Übereinstimmung des Musters im Objektsystem, schlägt die Regel fehl (Schritt 3) und führt zu einem Sicherheitsverstoß und dem sofortigen Beenden des Play-outs (Schritt 4).

Wird jedoch nach dem Aufruf eine Übereinstimmung des Musters der Graphtransformationsregel mit dem Objektsystem gefunden, so wird die neue Aufgabe erstellt, ein Objekt der Klasse Job (Schritt 5), und zurückgegeben (Schritt 6). Das zurückgegebene Objekt wird in der Diagrammvariablen job gespeichert. Der Schnitt (cut) aktiviert die nächste Nachricht. Diese hat im Beispiel bei ihrer Ausführung den Aufruf der Regel doJob zur Folge (Schritt 7). Wird diese korrekt abgearbeitet (wovon hier ausgegangen wird), hat der Schnitt (cut) die letzte aktive Nachricht des MSDs passiert (Schritt 8) und das MSD wird terminiert. Das Play-out wird in weiteren (aktiven) MSDs fortgesetzt (Schritt 9).

Zusammenfassend wurde in diesem Abschnitt dargestellt, wie MSDs und Graphtransformationsregeln sich gegenseitig unterstützen und aufeinander Einfluss nehmen können. Beide stützen und erweitern das Konzept und die Mächtigkeit des jeweils anderen Konzepts bzw. Werkzeugs. Damit ist auf konzeptueller Ebene gezeigt, dass Graphtransformationsregeln auch im Automotive- und Embedded-Bereich eine Bedeutung erlangen können.

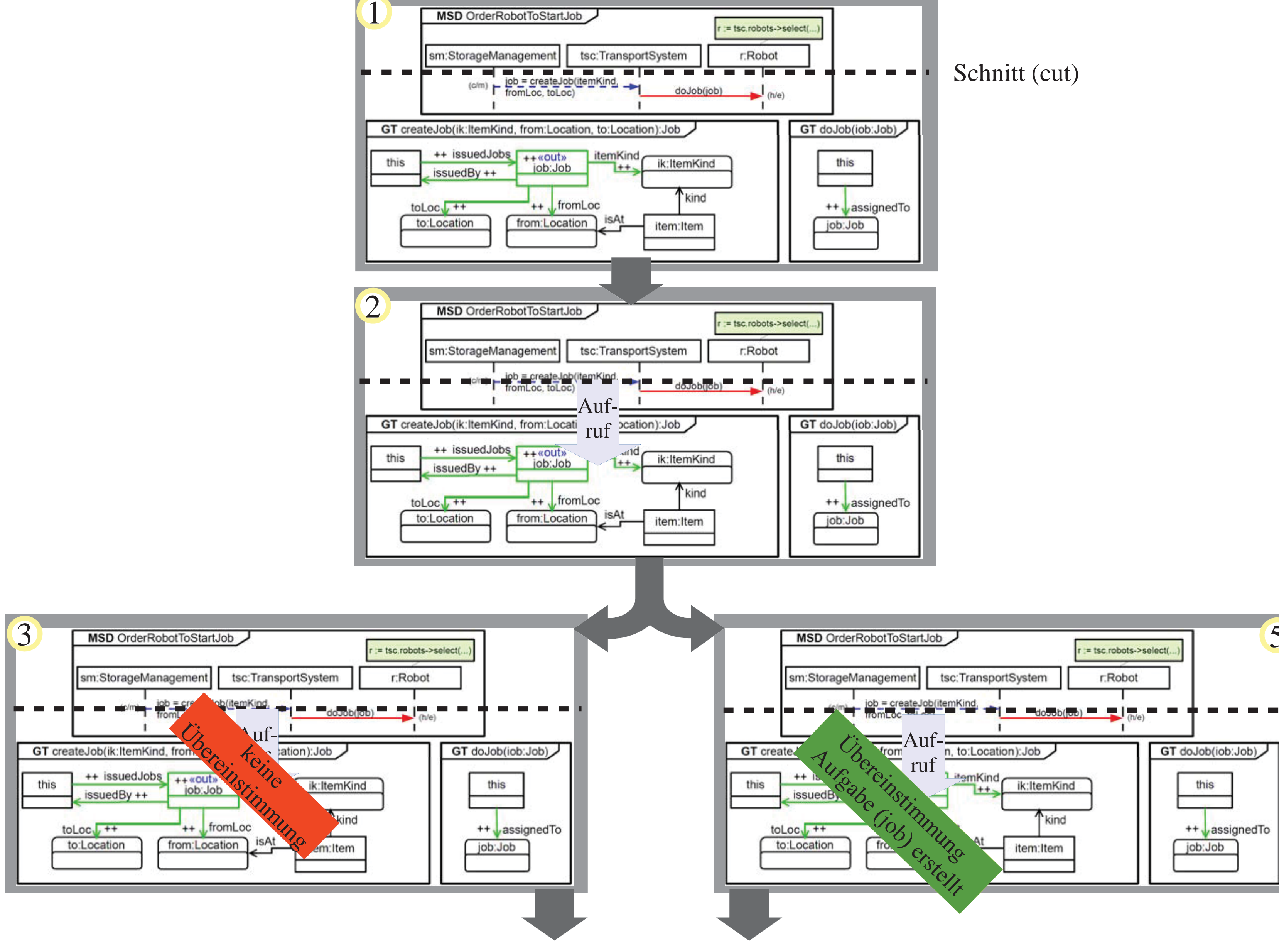
9.2 Prototypische Werkzeugintegration von ModGraph und ScenarioTools

Damit die eben erläuterte Integration von MSDs und Graphtransformationsregeln kein reines Konzept bleibt, wurde eine prototypische Integration der beiden Werkzeuge ScenarioTools und ModGraph vorgenommen. Die Interaktion der beiden Werkzeuge ist in Abbildung 9.7 dargestellt.

Beginnend mit ScenarioTools werden die MSDs zunächst in UML unter Benutzung des Papyrus-Editors[3] modelliert (Schritt 1 in Abbildung 9.7). Dazu gehört auch die Modellierung des UML-Klassendiagramms. Der Editor wurde durch Profile angepasst, um Modalitäten zu den Sequenzdiagrammen bereitzustellen. Das UML-Klassendiagramm wird im Folgenden in ein Ecore-Klassendiagramm transformiert (Schritt 2). Basierend auf diesem Ecore-Modell können nun Graphtransformationsregeln mit ModGraph erstellt werden, die einzelne Operationen aus dem Ecore-Modell implementieren (Schritt 3). Zudem wird das Ecore-Modell in ein Xcore-Modell übersetzt und die Implementierung der fertigen Regeln in das Xcore-Modell generiert (Schritt 4).

Zur Ausführung erzeugt ScenarioTools eine Instanz des Ecore-Modells, die als Objekt-System dient (Schritt 5). Auf dieser wird der ScenarioTools Play-out ausgeführt (Schritt 6). Während des Play-outs werden die Xcore-Implementierungen der Regeln aufgerufen sobald ein Nachrichten-Ereignis auftritt, das einer Operation entspricht, die mittels einer Regel implementiert ist.

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


Schnitt (cut)
1
MSD OrderRobotToStartJob
r := tsc.robots->select(...)
sm:StorageManagement
tsc:TransportSystem
r:Robot
(c/m)
job = createJob(itemKind, fromLoc, toLoc)
doJob(job)
(h/e)
GT createJob(ik:ItemKind, from:Location, to:Location):Job
this
++ issuedJobs
issuedBy ++
++ «out» job:Job
itemKind ++
ik:ItemKind
kind
toLoc ++
to:Location
++ fromLoc
from:Location
isAt
item:Item
GT doJob(iob:Job)
this
++ assignedTo
job:Job
2
Auf-ruf
3
keine Übereinstimmung
5
Übereinstimmung Aufgabe (job) erstellt

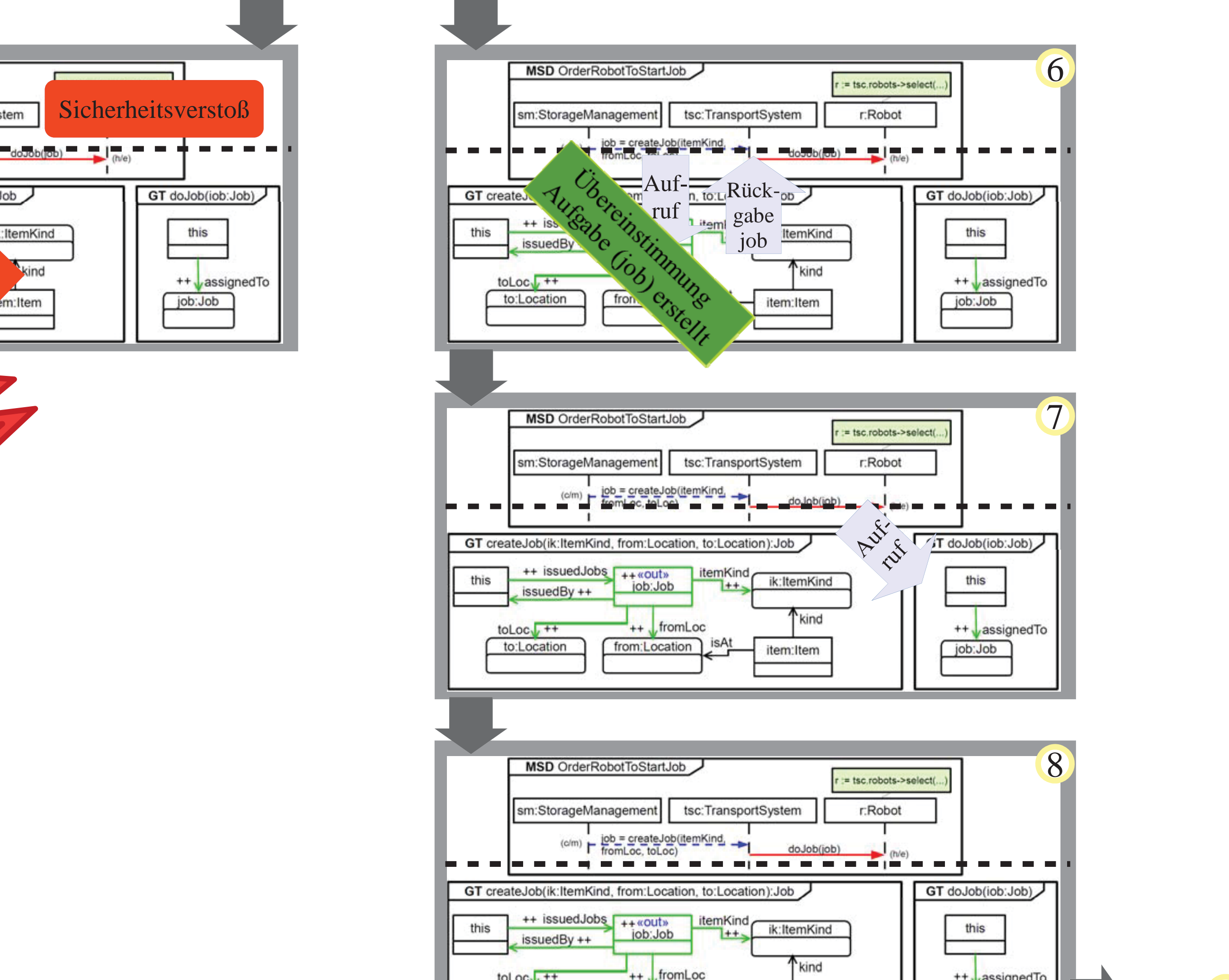


Abbildung 9.6: *Ablauf des Play-outs nach der Integration*

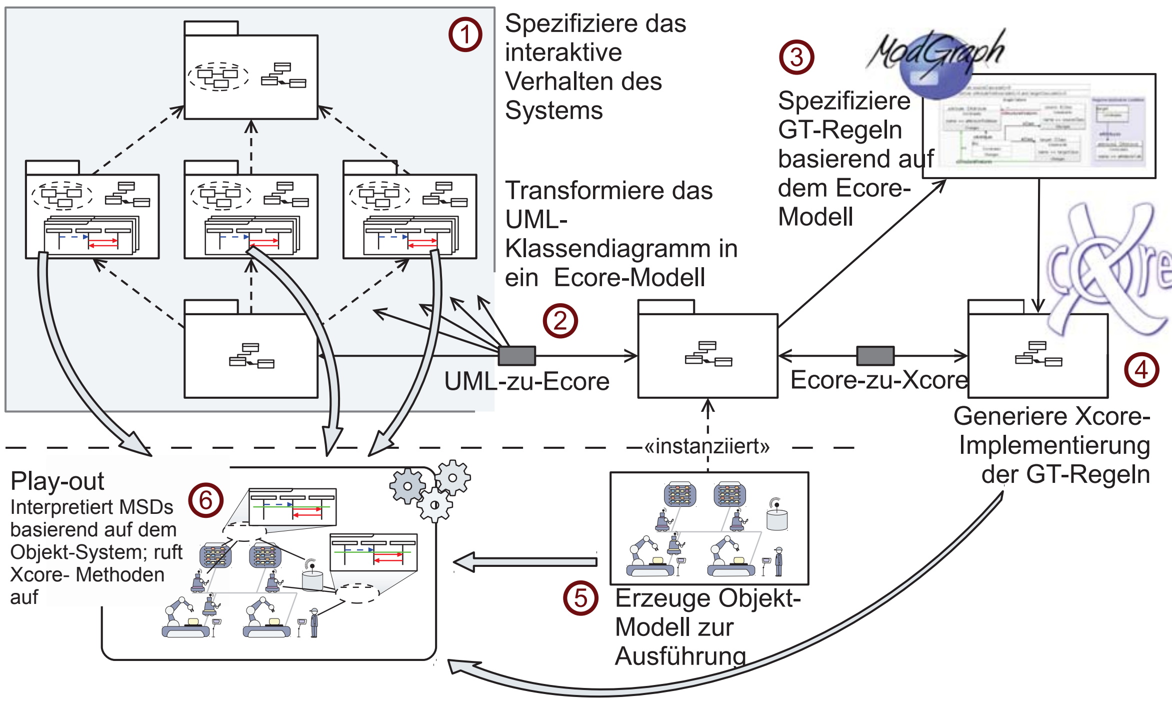


Abbildung 9.7: *Zusammenarbeit von ModGraph und* SCENARIOTOOLS

Da bei Aufruf einer Regel zunächst geklärt werden muss, ob ihre Vorbedingungen erfüllt sind, werden bei der Generierung der Graphtransformationsregeln nach Xcore eine prüfende Operation - zum Test der Vorbedingungen der Regel - und eine ausführende Operation - zur Ausführung der Regel - erzeugt. Der in Abschnitt 7.2.2 betrachtete Generator für Xcore-Operationen wird dahingehend modifiziert. Er generiert nun - analog zum Java-Generator (Abschnitt 7.2.1) - eine Hilfsmethode, die die Mustersuche übernimmt. Selektiert der Play-out während seiner Ausführung eine Nachricht, die zur Ausführung einer Regel führt, so wird zuerst die prüfende Methode aufgerufen, danach - bei Erfolg dieser - die ausführende. Schlägt die prüfende Methode fehl, tritt ein Sicherheitsverstoß auf.

Dieser im allgemeinen schlüssige Ansatz unterliegt leider einigen Limitierungen. SCENARIOTOOLS unterstützt lediglich Nachrichten mit einem Parameter. Dies erschwert die Umsetzung des oben beschriebenen Ansatzes. Eine Erweiterung zur Unterstützung von Nachrichten mit mehreren Parametern ist bereits in Planung. Um das Beispiel dennoch umsetzen zu können, werden mehrere Nachrichten genutzt, um die Parameter zu übergeben, was das System etwas verkompliziert.

ModGraph bietet keine direkte Interpretation seiner Regeln an. Daher ist der Umweg über Xcore notwendig, denn ModGraph nutzt den Xcore-Interpreter zur indirekten Interpretation der Regeln. Hier wäre ein Interpreter oder dynamisches Laden von Quelltext erforderlich.

10 Diskussion und Abgrenzung des Ansatzes

10.1 Diskussion des Ansatzes

Dieser Abschnitt widmet sich der Statistik und einigen Vergleichen. Zunächst folgt eine Untersuchung der Regeln an sich. Daraufhin werden Laufzeitmessungen durchgeführt. Dazu werden die beiden - aus dieser Arbeit bereits bekannten - Projekte Bugtracker und Refactoring untersucht.

10.1.1 Evaluation der Regeln

Zur Evaluation der Regeln wird das Auftreten der Elemente in 59 Regeln gezählt. Sie gibt einen ersten Eindruck der Nutzung von Graphtransformationsregeln in ModGraph. Um einen Überblick über das durchschnittliche Auftreten eines Elements innerhalb einer Regel zu erlangen, wird das arithmetische Mittel aus den durch die Zählung erhaltenen Werten nach der üblichen Formel gebildet, wobei n für die Anzahl der Werte steht:

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

Außerdem wird die minimale und maximale Anzahl der in einer Regel vorkommenden Elemente des jeweiligen Typs ermittelt.

Erfassung

Beide untersuchten Projekte wurden mit ModGraph modelliert. Der Bugtracker stellt eine frühe Anwendung dar, die möglichst modellgetrieben entwickelt ist. Die Grundfunktionalität wird durch Ecore und ModGraph-Regeln dargestellt. Lediglich die Methode `getNextNumber()` ist in Java bzw. in Xcore implementiert. Der Bugtracker besteht aus sieben Klassen und drei Aufzählungsdatentypen. Erstere bieten insgesamt 25 Operationen an, und enthält 24 Regeln. Davon sind 13 Graphabfragen, neun Transformationen und zwei Tests.

Das Projekt zum propagierenden Refactoring nutzt Xcore und Graphtransformationsregeln nicht ganz zu gleichen Teilen: 35 der 71 Operationen, die im Xcore-Modell des propagierenden Refactorings implementiert sind, nutzen ModGraph-Regeln. Diese sind auf drei Klassen verteilt. Das Refactoring importiert Klassen aus dem Ecore-Metamodell und dem Graphtransformations-Metamodell. Die Regeln setzen sich aus fünf Abfragen und 30 Transformationen zusammen.

Hauptkomponenten	*Anzahl*	*Elemente/Regel*	*Max. Auftreten*	*Min. Auftreten*
Vorbedingung	19	0,32	2	0
Nachbedingung	4	0,07	1	0
Graphmuster	59	1	1	1
NAC	9	0,15	2	0

Tabelle 10.1: *Vorkommen der Hauptkomponenten einer Regel*

Graphmuster	*Anzahl*	*Elemente/Regel*	*Max. Auftreten*	*Min. Auftreten*
geb. Knoten	112	1,90	8	1
ungeb. Knoten	147	2,49	7	1
Knoten gesamt	259	4,39	12	2
Links	246	4,17	14	0
Pfade	38	0,64	4	0
Kanten gesamt	284	4,81	15	1

Tabelle 10.2: *Vorkommen der Elemente im Graphmuster*

Insgesamt werden 59 Regeln betrachtet. Dabei wird für jede Regel vermerkt, aus welchen Komponenten sie aufgebaut ist. Die Ergebnisse finden sich in den folgenden Tabellen. Zunächst werden die Hauptkomponenten der Regel, deren Ergebnisse in Tabelle 10.1 gezeigt sind, betrachtet. Jede Regel besitzt ein Graphmuster. Das ist ein zu erwartendes Ergebnis, da ModGraphs Metamodell für Graphtransformationsregeln es fordert. Die anderen Hauptkomponenten, Vor- und Nachbedingungen sowie NACs, sind optionale Komponenten einer Regel und dürfen daher wegfallen. Dies deckt sich mit dem jeweiligen minimalen Auftreten, das null entspricht. Das maximale Auftreten der Hauptkomponenten ist immer größer null. Es ist damit bestätigt, dass alle Hauptkomponenten der Regel Verwendung finden. Wie oft dies der Fall ist, kann aus dem arithmentischen Mittel abgelesen werden. Vorbedingungen stellen dabei die häufigste Form der Bedingung eines Graphmusters dar. Sie treten in einem Drittel der Fälle auf. Nachbedingungen und NACs werden eher selten genutzt. Dabei werden NACs öfter als die Nachbedingungen verwendet.

Die einzelnen Elemente im Graphmuster und in den NACs sowie die Elemente innerhalb aller Knoten wurden ebenfalls untersucht. Betrachtet man die Knoten im Graphmuster, ergeben sich die in Tabelle 10.2 gezeigten Resultate. Jedes enthält mindestens einen gebundenen Knoten. Dies ist entweder ein Parameterknoten oder der für das aktuelle Objekt stehende Knoten. Oft sind es sogar mehrere, wie aus dem arithmetischen Mittel abzulesen ist, das einer gebundenen Knotenzahl von durchschnittlich 1,90 pro Regel entspricht. Parameter werden gegenüber dem aktuellen Objekt minimal bevorzugt. Interessanterweise treten mehrwertige Parameter eher selten auf. Weiterhin können bis zu acht gebundene Knoten gemeinsam in einem Graphmuster zu finden sein. Tabelle 10.3 fasst dies zusammen.

Die Ergebnisse zeigen, dass die betrachteten Graphmuster auch mindestens einen un-

Graphmuster	*Anzahl*	*Elemente/Regel*	*Max. Auftreten*	*Min. Auftreten*
Knoten für aktuelles Objekt	42	0,71	1	0
einwertige geb. Knoten	60	1,02	4	0
mehrwertige geb. Knoten	10	0,17	4	0

Tabelle 10.3: *Vorkommen der gebundenen Knoten im Graphmuster*

Graphmuster	*Anzahl*	*Elemente/Regel*	*Max. Auftreten*	*Min. Auftreten*
einwertige ungeb. Knoten	107 (3)	1,81 (0,05)	7 (1)	0 (0)
mehrwertige ungeb. Knoten	40 (7)	0,68 (0,12)	4 (3)	0 (0)

Tabelle 10.4: *Vorkommen der ungebundenen Knoten im Graphmuster (angegeben als gesamt (optional))*

gebundenen Knoten, im Durchschnitt sogar 2,49, aufweisen (Tabelle 10.2). Mehr als sieben treten bislang nicht gemeinsam in einem Graphmuster auf. Die ungebundenen Knoten werden anhand von Tabelle 10.4 genauer untersucht. Die angegebenen Werte entsprechen den Vorkommen von obligatorischen und optionalen Knoten. Zur besseren Einschätzung sind die Werte für die optionalen Knoten nochmals in Klammern angegeben. Es stellt sich heraus, dass die Verwendung obligatorischer Knoten überwiegt. Dies war zu erwarten, denn die optionalen Knoten stellen einen Sonderfall dar. Einwertige Knoten kommen mehr als doppelt so oft vor wie mehrwertige. Dabei treten mehrwertige optionale Knoten tendenziell öfter auf als einwertige optionale.

Betrachtet man nochmals die Werte für das Graphmuster in Tabelle 10.2, enthält es im Schnitt etwas mehr als vier Knoten. Graphmuster mit weniger als zwei und mehr als zwölf Knoten kommen nicht vor.

Zur Verbindung der Knoten werden Kanten benötigt. Jedes Graphmuster enthält mindestens eine Kante, im Durchschnitt fast fünf (4,81). Das Graphmuster mit den meisten Kanten beinhaltet 15 davon. Dabei werden häufiger Links als Pfade eingesetzt. Die betrachteten Graphmuster enthalten bis zu 14 Links und bis zu vier Pfade. Durchschnittlich sind 4,17 Links und 0,64 Pfade zu finden.

Da nur neun NACs vorkommen, sind die Ergebnisse hier nicht so repräsentativ wie die der 59 Graphmuster, sollen jedoch trotzdem in Tabelle 10.5 gezeigt werden. Jede der NACs hat mindestens einen gebundenen Knoten. Dies kann entweder ein das aktuelle Objekt repräsentierender (drei mal) oder ein Parameterknoten (einmal) sein. Alternativ kann ein bereits im Graphmuster gebundener Knoten in der NAC wiederverwendet werden (fünf mal), was vor allem beim Refactoring auftritt. Jede NAC enthält im Durchschnitt etwas mehr als einen ungebundenen Knoten und eine Kante, die in allen

NAC	*Anzahl*	*Elemente/Regel*	*Max. Auftreten*	*Min. Auftreten*
geb. Knoten	9 (3/1/5)	1,00	1	1
ungeb. Knoten	10	1,11	2	1
Kanten	10 (10/0)	1,11	2	1

Tabelle 10.5: *Durchschnittliches Vorkommen der Elemente in der NAC*

Elemente in Knoten	*Anzahl*	*Elemente/Regel*	*Max. Auftreten*	*Min. Auftreten*
OCL / Xcore Bedingung	14	0,24	2	0
Attribut-bedingung	44	0,76	4	0
Attribut-zuweisung	40	0,69	7	0
Operations-aufruf	4	0,07	3	0
Rückgabe-parameter	29	0,5	1	0

Tabelle 10.6: *Durchschnittliches Vorkommen von Elementen in den Knoten*

betrachteten Fällen ein Link ist (zehnmal, null Pfade).

Im Folgenden werden die Elemente innerhalb der Knoten untersucht. Tabelle 10.6 zeigt, dass in etwa einem Viertel aller Regeln OCL- oder Xcore-Bedingungen an Knoten gestellt werden. Bedingungen und Wertzuweisungen an Attribute kommen deutlich öfter vor. Zwei aus drei Regeln enthalten eine Attributbedingung, maximal treten vier gemeinsam auf. Mehr als zwei von drei Regeln enthält eine Zuweisung. Die Regel mit den meisten Zuweisungen enthält sieben. Jedoch existieren auch Regeln ohne Bedingungen oder Wertzuweisungen an Attribute. Operationsaufrufe kommen eher selten vor. Der Grund hierfür ist in der Kontrollflussmodellierung mit Xcore im Refactoring Projekt zu suchen. Durch die externe Modellierung der prozeduralen Anteile, werden auch die gegenseitigen Aufrufe der Regeln untereinander tendenziell weniger. Die Hälfte aller Regeln gibt einen durch einen Knoten repräsentierten Wert zurück.(Die Graphtests, die boolsche Werte zurückliefern, sind hier nicht erfasst.)

Diskussion

Nach eingehender Betrachtung der Fakten bleibt die Frage: Untermauern diese den Mehrwert, den Graphtransformationsregeln bieten können? Um diese Frage zu beantworten, wird ModGraph bezüglich der in [20] gefundenen, generellen Beobachtungen bei Nutzung von Graphtransformationsregeln untersucht. Diese lauten zugespitzt formuliert:

1. Das Verhaltensmodell ist hochgradig prozedural.

2. Die Ausdrucksmächtigkeit der Regeln wird kaum genutzt.

Ausgehend vom ersten Punkt wird gefordert, dass textuelle, prozedurale Elemente zur Steuerung der Regeln und zur Programmierung der prozeduralen Anteile vorhanden sein sollten, da grafische Elemente zur Steuerung keine Vorteile mit sich bringen. Dies ist im ModGraph-Ansatz gegeben, da ModGraph Xcore zur Steuerung der Regeln nutzt. Zudem wird dem Nutzer die geforderte Wahlfreiheit zwischen der prozeduralen und der regelbasierten Implementierung gewährt. Durch die nahtlose Integration der Regeln in Xcore (siehe Abschnitt 7.1) steht es ihm frei, selbst zu wählen, in welchen Anteilen ModGraph-Regeln, Ecore und Xcore zur Umsetzung des aktuellen Problems verwendet werden. Dabei wird ihm geraten, sich auf den Mehrwert, den die Regeln bieten, die einfache Modellierung komplexer struktureller Veränderungen, zu konzentrieren: Der Modellierer wird demnach nicht gezwungen, Regeln zu nutzen. Diese Wahlfreiheit ist in den meisten anderen Werkzeugen nicht gegeben, wie in Abschnitt 10.2 gezeigt werden wird. Es entsteht eine strikte Trennung von prozeduraler und regelbasierter Verhaltensmodellierung, die zur Folge hat, dass prozedurale Elemente in den Regeln großteils vermieden werden. „Prozedurale Einzeiler", wie Operationsaufrufe, sind trotzdem möglich.

Betrachtet man die Verteilung in den beiden untersuchten Beispielprojekten, so ist im propagierenden Refactoring etwa die Hälfte des modellierten Verhaltens durch Regeln dargestellt. Im Bugtracker ist das Verhalten bis auf eine Ausnahme ausschließlich durch Regeln modelliert.[1] Allerdings kann daraus nicht gefolgert werden, dass die Regeln im Allgemeinen den prozeduralen Operationen vorzuziehen sind, sondern lediglich, dass der Modellierer des Bugtrackers gerne Regeln verwendet. Der Bugtracker ist zudem ein kleines Projekt, das keine Schichten aufweist. Die darin modellierten Operationen bilden auf elementarer Ebene die Funktionsweise eines Bugtrackers ab. Diese lässt sich recht gut durch Regeln ausdrücken. Komplexere Operationen auf dem BugTracker würden ebenso zur Nutzung prozeduraler Anteile im Modell führen. Ein Beispiel hierfür findet sich in Abschnitt 5.4, Auflistung 5.4. Darin wird eine Methode erstellt, die zwei Regeln komponiert.

Die in [20] getroffene Aussage, dass das Verhaltensmodell (gemeint ist hier eine Menge von Regeln) sehr viele prozedurale Elemente enthält, kann trotzdem teilweise entkräftet werden. Nutzt man die Regeln nur, wenn sie einen Mehrwert bieten, geht die Verwendung der prozeduralen Anteile innerhalb der Regeln (nicht global) zurück.

Zur Diskussion des zweiten Punkts aus [20], der die Nutzung der Ausdrucksmächtigkeit der Regeln anzweifelt, werden die zuvor erstellten Statistiken über die Regeln betrachtet. In einem Graphmuster befinden sich im Durchschnitt 4,39 Knoten (1,90 gebundene und 2,49 ungebundene Knoten) und 4,81 Kanten (4,17 Links und 0,64 Pfade). Um diese Zahlen vergleichen zu können, muss zunächst geklärt werden, welche der Ergebnisse aus [20] dazu herangezogen werden können. Zudem sei an dieser Stelle angemerkt, dass hier nur insgesamt 259 Knoten und 284 Kanten in 59 Regeln evaluiert wurden, während in [20] 988 Storymuster (die hier als Analogon zu einer Regel genutzt werden) mit über

[1] Dies ist kein Einzelfall. Auch das Verhalten des mit ModGraph realisierten Kalenderprojekts ist nahezu nur mit Regeln modelliert. Es kann unter `http://btn1x4.inf.uni-bayreuth.de/modgraph/homepage` heruntergeladen werden.

Vergleich	*Ergebnis aus [20]*	*ModGraph*
Objekte / einw. Knoten	1,71	3,54
Multiobjekte / mehrw. Knoten	0,03	0,85
Links	0,73	4,17
Pfade	0,01	0,64

Tabelle 10.7: *Vergleich der Ergebnisse mit [20]*

1600 Objekten und über 700 Kanten betrachtet werden. Ein Trend sollte sich dennoch ablesen lassen. Tabelle 10.7 stellt die Ergebnisse gegenüber. Die Objekte der Storymuster werden mit den einwertigen (gebundenen und ungebundenen) Knoten im Graphmuster verglichen, die Multiobjekte mit den mehrwertigen (gebundenen und ungebundenen) Knoten . Die Links und Pfade entsprechen in ihrer Funktion einander und können daher direkt verglichen werden. Das in [20] genutzte Fujaba bietet negative Knoten statt der in ModGraph vorhandenen negativen Anwendbarkeitsbedingungen an. Zwar könnten viele der in den evaluierten Beispielen genutzten NACs durch negative Knoten ausgedrückt werden, jedoch handelt es sich hierbei um zwei unterschiedliche Konzepte, die nicht direkt zueinander in Relation gestellt werden können.

Die ModGraph-Regeln schneiden in der Tabelle deutlich besser ab. Die durchschnittliche Anzahl der Objekte bzw. einwertigen Knoten verdoppelt sich, die der Multiobjekte bzw. mehrwertigen Knoten vervielfacht sich. Die Anzahl der Links steigt etwa um den Faktor fünf, Pfade werden sehr viel öfter verwendet. Dieser Anstieg der Elemente pro Regel weist auf eine gestiegene Komplexität dieser hin. Damit kann davon ausgegangen werden, dass die Ausdrucksmächtigkeit der Regeln zumindest besser ausgeschöpft wird. Wie sich diese im Einzelfall verbessert, hängt vor allem von den Vorlieben und Fähigkeiten des Modellierers ab. Insbesondere das Refactoring-Projekt bringt, durch die Ausgliederung der prozeduralen Anteile, die erwartete Steigerung der Elemente pro Regel mit sich.

Abschließend soll an dieser Stelle die Eingangsfrage dieses Vergleichs, ob der durch die Regeln gebotene Mehrwert durch die Fakten untermauert wird, beantwortet werden. Dazu können folgende Argumente angeführt werden:

- Die fortgeschrittenen Konzepte (Mengenknoten, NACs und optionale Knoten) werden eher selten genutzt. Die Abschaffung dieser Konzepte, die zu deren Simulation im Kontrollfluss führen würde, würde allerdings den Programmieranteil erhöhen. So müsste beispielsweise zur Simulation jedes mehrwertigen Knotens eine Schleife programmiert werden. Damit wäre auch der Mehrwert der Regeln in Frage gestellt.

- Hinsichtlich der Größe der Regeln gibt es eine große Bandbreite. Die ModGraph-Regeln im Bugtracker sind größer, da es sich um ein kleines Projekt ohne Schichten handelt. Die Regeln im Refactoring werden durch (prozedurale) Vorbereitungsaktionen unterstützt. Diese finden vor Aufruf einer komplexen Regel statt.

- Das Hauptargument für die Regeln bleibt weiterhin die Abstraktion der Regel vom Algorithmus. Der Modellierer muss sich beispielsweise nicht mit der Mustersuche beschäftigen. Die anschauliche grafische Notation der Regeln ist nicht zu vernachlässigen.

10.1.2 Laufzeitmessungen

Zur weiteren Analyse des Ansatzes wurden Laufzeitmessungen durchgeführt, welche die Anwendung verschiedener Refactorings auf dem Bugtracker betrachten. Sieben Refactoringoperationen und deren Propagationen sind zu diesem Zweck sowohl in Java als auch in ModGraph implementiert und verglichen worden: das Verschieben eines Attributs bzw. einer Operation, das Umwandeln einer bidirektionalen in eine unidirektionale Referenz, das Verstecken des Delegierten, das Entfernen der Klasse in der Mitte, das Ersetzen einer Enthaltenseinsbeziehung durch eine einfache Referenz und deren Umkehrung. Im Folgenden werden die Laufzeiten für die ersten drei Refactorings und deren Propagation besprochen. Es handelt sich dabei um aussagekräftige und dennoch überschaubare Refactorings. Die Laufzeitmessungen wurden auf einem Intel Core i5-2400 mit 3.1 GHz (2 Kerne mit Hyperthreading) und 8GB RAM ausgeführt. Da sie innerhalb der Laufzeitumgebung von Eclipse durchgeführt werden, ist es sinnvoll, ebenso deren Parameter anzugeben: Die minimale Speichergröße auf dem Heap der JVM beträgt 512MB (-Xms512m) und die maximale 1024MB (-Xmx1024m).

Um die Laufzeiten zu ermitteln, werden pro Refactoring 200 Messungen durchgeführt, die sich auf 100 Messungen mit der modellierten und 100 Messungen mit der in Java geschriebenen Methode verteilen. Aufgrund ihrer Ausführung in der Laufzeitumgebung ist der Speicher etwas knapper bemessen, so dass sie in Blöcken von 16-18 Messungen ausgeführt werden. Um die entstehende Menge an Messungen in endlicher Zeit zu bewältigen, werden diese automatisiert, ohne die Ergebnisse zu verfälschen. Die Testumgebung besteht jeweils aus einer minimalen Menge abhängiger Modelle: einem Ecore-Modell und einer zugehörigen ModGraph-Regel. Die Benutzerinteraktion wurde programmatisch vorgegeben und die Validierung deaktiviert, da sie eine zusätzliche Zeitkonstante auf allen Messungen darstellt. Die Validierung ein und derselben Regel dauert immer gleich lang. Zudem wird das Testprojekt nach jeder Messung in seine Ausgangskonfiguration zurückgesetzt.

Zur Auswertung der so erhaltenen Rohdaten wird der Median aus diesen berechnet. Er bietet einen stabilen Mittelwert der erhaltenen Werte. Sortiert man alle Messwerte aufsteigend, stellt er, bildlich gesprochen, den in der Mitte stehenden Wert dar. Gibt es keine Mitte, weil die Anzahl der Werte gerade ist, wird der Mittelwert der beiden in der Mitte stehenden Messwerte genutzt. Die Berechnung für die hier vorgenommenen Messungen folgt daher der Formel für die Berechnung des Medians bei geradzahliger Anzahl n der betrachteten Werte:

$$\bar{x} = \frac{x_{\frac{n}{2}} + x_{\frac{n}{2}+1}}{2}$$

Das erste betrachtete Refactoring ist das Verschieben eines Attributs. Die betrach-

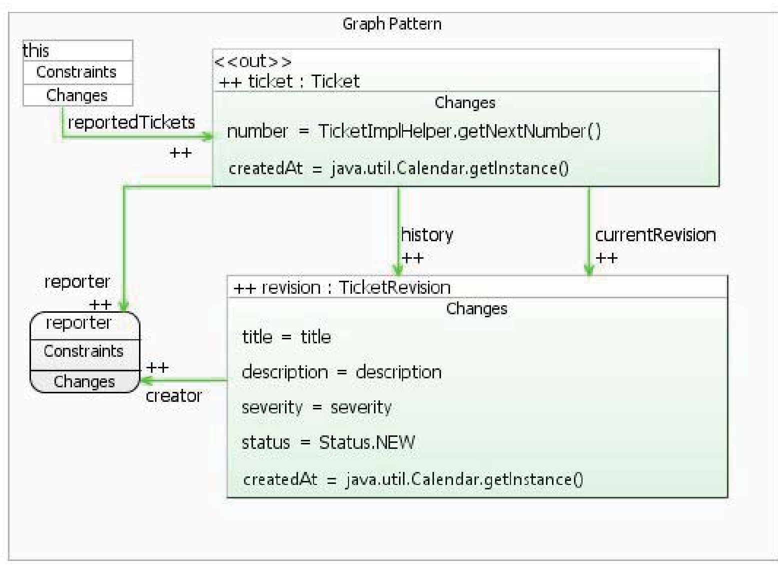


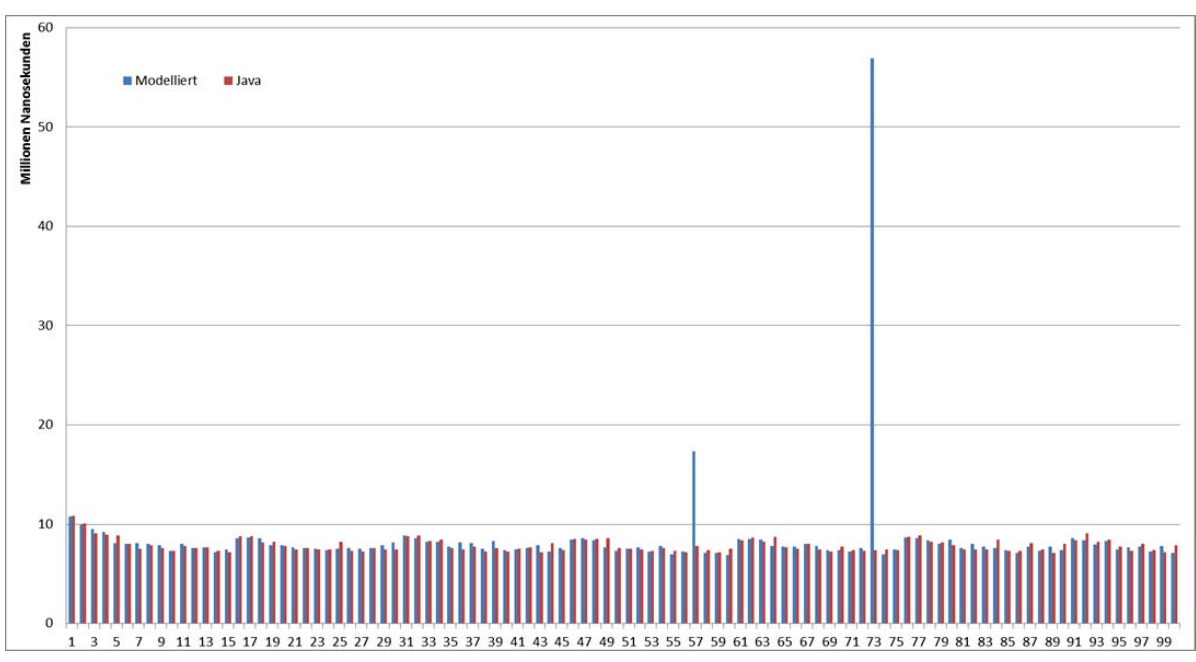
Abbildung 10.1: *Regel zur Implementierung der Operation* createTicket(...) *des Bugtracker-Beispiels*

tete Regel createTicket ist in Abbildung 10.1 zu sehen. Sie entspricht der bereits bekannten Regel, jedoch ohne die OCL-Bedingungen. Diese wurden entfernt, da sie zu Benutzerinteraktionen führen würden. Zudem liegt der Schwerpunkt des Testens auf der Veränderung der Teilgraphen. Das Attribut title soll aus der Klasse TicketRevision in die Klasse Ticket verschoben werden.

Das Refactoring wurde mit ModGraph modelliert, aus dem Xcore-Code erzeugt wurde. Die Propagation wurde mit ModGraph, unter Verwendung von OCL, erstellt und direkt in den Xcore-generierten Quelltext eingefügt.

Diagramme zu den einzelnen Messungen sind in Abbildung 10.2 dargestellt. Betrachtet man das obere der beiden Diagramme, ist klar ersichtlich, dass die programmierten und die modellierten Elemente, bis auf wenige Ausnahmen, gleiche Zeiten benötigen. Die Ausnahmen sind auf einen Überlauf des zugeordneten Speichers der Virtuellen Maschine zurückzuführen, da sie immer (nicht nur in dieser Messreihe) am Ende des Messintervalls von 16-18 Messungen auftreten.

Anders verhält es sich im unteren Diagramm der Abbildung. Dieses zeigt die Laufzeiten der Propagation. Die modellierte Propagation benötigt annähernd die doppelte Zeit der programmierten. Der Grund dafür ist in der Nutzung der OCL-Ausdrücke zu finden, die während der Ausführung des Refactorings interpretiert werden müssen. Da eine Interpretation normalerweise langsamer als die Ausführung von generiertem Quelltext abläuft, ist hier auch die modellierte Propagation langsamer. Die berechneten Mediane sind in Tabelle 10.8 gezeigt. Damit wird folgende Annahme getroffen: Eine nach Xcore generierte Regel, die ausschließlich Xcore zur Spezifikation textuellen Elemente nutzt, ist annähernd genauso schnell auszuführen, wie eine direkt in Xcore spezifizierte Operation. Dahingegen sind Regel n, die OCL verwenden, deutlich langsamer sind. Dies schlägt sich ebenso auf die Gesamtlaufzeiten nieder.



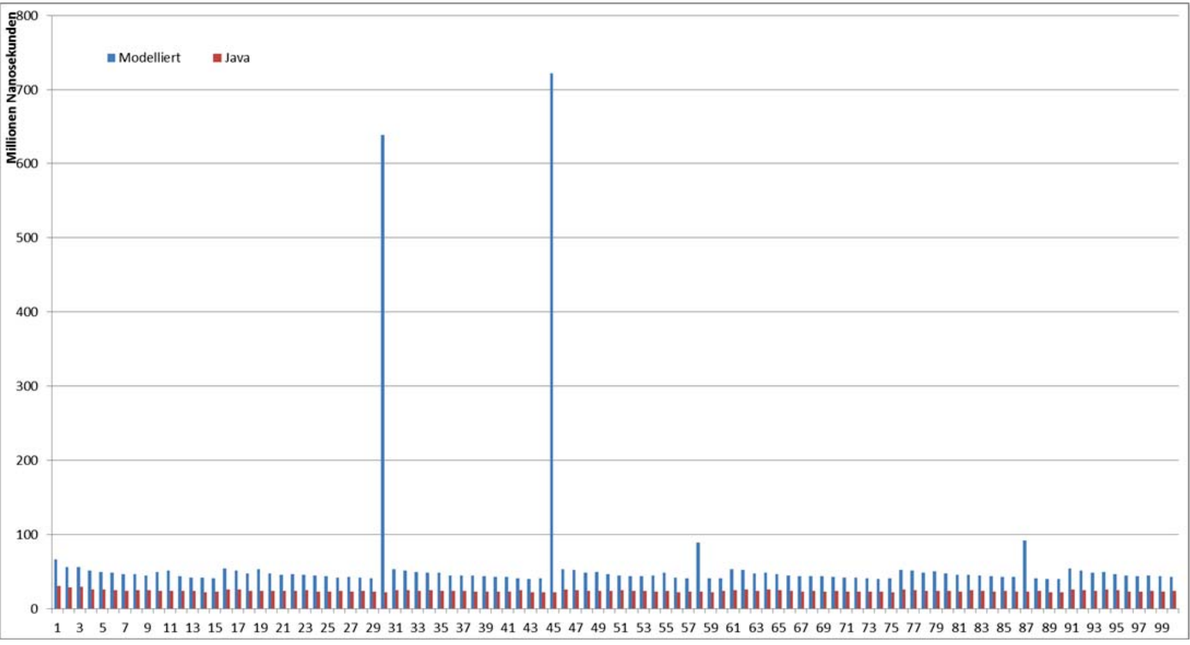


Abbildung 10.2: *Diagramme aus den Rohdaten der Laufzeitmessungen für das Refactoring (oben) und die Propagation (unten) bei der Verschiebung des Attributs*

Implementierung	*Refactoring [ms]*	*Propagation [ms]*	*Gesamtlaufzeit [ms]*
ModGraph	7,75	45,30	53,05
Java	7,64	23,94	31,58

Tabelle 10.8: *Durchschnittliche Laufzeiten für das Verschieben eines Attributs*

Implementierung	*Refactoring [ms]*	*Propagation [ms]*	*Gesamtlaufzeit [ms]*
ModGraph	9,98	24,22	34,20
Java	9,52	23,47	32,99

Tabelle 10.9: *Durchschnittliche Laufzeiten für das Verschieben einer Operation*

Die Messungen zum Verschieben einer Operation untermauern diese Annahme. Hier wird die Operation createTicket(...) von der Klasse Project in die Klasse BugTracker verschoben. Die betrachtete Regel ist nochmals createTicket. Sie implementiert diese Methode. Da die Regel keine Operationsaufrufe beinhaltet, müssen lediglich die Referenzen auf die Klasse und Operation neu gesetzt werden. Das Refactoring ist mit ModGraph, unter ausschließlicher Verwendung von Xcore für textuelle Ausdrücke, modelliert. Aus der Regel wird Xcore-Code erzeugt. Die Propagation ist - aufgrund ihrer prozeduralen Natur - mit Xcore und Xbase erstellt. Zur Ausführung wird demnach Xcore-generierter Java-Quelltext genutzt. Tabelle 10.9 zeigt die Ergebnisse. Die Laufzeiten der modellierten und der programmierten propagierenden Refactorings sind annähernd gleich. Die etwas höhere Laufzeit der modellierten Variante lässt sich durch den aus Abschnitt 7.2.3 bekannten Fakt erklären, dass handgeschriebener Quelltext in der Regel etwas sparsamer bezüglich der verwendeten Konstrukte ist.

Zur weiteren Untermauerung der bislang dargestellten Annahmen wird abschließend das Refactoring zum Umwandeln einer bidirektionalen in eine unidirektionale Referenz betrachtet.

Diese Tests werden an der Regel isActive, welche die gleichnamige Operation der Klasse Group implementiert, durchgeführt. Sie beinhaltet einen Link, der die bidirektionale Referenz zwischen der Gruppe und den zugeordneten Nutzern instanziiert. Während des Refactorings wird die Referenz members gelöscht. Dies führt zu einer Ersetzung des Links members durch den entgegengesetzten Link memberIn.

Das Refactoring wird in ModGraph modelliert und nach Xcore generiert. Bei der Propagation wird der Quelltext aus der ModGraph-Regel in den Xcore-generierten Java Quelltext eingefügt. Wie in Tabelle 10.10 zu sehen, ist die programmierte Lösung schneller als die modellierte. Der Grund hierfür ist wiederum in der Verwendung von OCL bei der Propagation zu finden.

Alle folgenden Untersuchungen zur Laufzeit untermauern die hier aufgestellten Annahmen und Begründungen zu den Laufzeiten. Die Ergebnisse dieser Untersuchungen sind in Anhang A.5 zu finden. Daher können die Annahmen als zutreffend bezeichnet werden. Die Laufzeiten OCL-freier Refactorings sind nahezu identisch mit den programmierten. OCL-behaftete Regeln werden ausschließlich wegen der zu ihrer Ausführung nötigen

Implementierung	*Refactoring [ms]*	*Propagation [ms]*	*Gesamtlaufzeit [ms]*
ModGraph	9,45	47,41	56,86
Java	8,06	30,87	38,93

Tabelle 10.10: *Durchschnittliche Laufzeiten für das Umwandeln einer bidirektionalen in eine unidirektionale Referenz*

OCL-Interpretation langsamer ausgeführt. Dies spricht für die ModGraph- Generatoren.[2] Zudem zeichnet sich ab, dass eine Propagation immer mehr Zeit in Anspruch nimmt als ein Refactoring. Dies ist durch den größeren Aufwand der Propagation zu erklären, die - im Regelfall - jede beteiligte Graphtransformationsregel elementweise durchsucht.

10.2 Abgrenzung des ModGraph-Ansatzes zu verwandten Arbeiten

Bereits in Teil II der Arbeit wurden in Verbindung stehende Ansätze und Werkzeuge besprochen. Dabei wurde zunächst auf das Eclipse Modeling Framework (EMF) eingegangen, das eine solide und in Industrie und Forschung anerkannte Möglichkeit bietet, Software teilweise modellgetrieben zu entwickeln. Dazu werden Ecore-Modelle verwendet. Diese stellen eine ausgereifte und mächtige Möglichkeit zur Strukturmodellierung bei überschaubarer Anzahl an Konstrukten dar. EMFs Modell-zu-Text-Transformationsmöglichkeiten bieten zudem eine Quelltextgenerierung aus diesen statischen Diagrammen an. Die Spezifikation des Verhaltens auf Modellebene bleibt jedoch unberücksichtigt.

Ein weiterer Abschnitt beschreibt Modell-zu-Modell-Transformationsansätze. Dabei bietet Xcore, neben einer textuellen Syntax für Ecore, eine Möglichkeit, dem Modell Verhalten hinzuzufügen, indem es die Java-nahe Sprache Xbase integriert. Dies erlaubt nicht nur die Generierung von lauffähigem Quelltext, sondern auch die Interpretation von Verhalten auf Ecore-Modellen. Damit ist es möglich, endogene, überschreibende Transformationen auf Modellinstanzen durchzuführen. Im weiteren Verlauf dieses Kapitels wurden ATL und QVT skizziert, die ebenfalls Modelle, darunter auch EMF-Modelle, transformieren.

Den Abschluss der Betrachtungen in Teil II bilden Werkzeuge, die Graphtransformationen anwenden. Bei einigen Werkzeugen, wie AGG, liegt der Fokus auf der theoretischen Fundierung, andere Werkzeuge wie PROGRES, Fujaba und Viatra2 sind eher praxisorientiert. Werkzeuge wie eMoflon und MDELab nutzen das EMF-Rahmenwerk. Die Art der Steuerung der Ausführungsreihenfolge der Transformationen divergiert. Nur einige Werkzeuge erlauben eine gezielte Steuerung durch die explizite Angabe von Kontrollflüssen. Viele bevorzugen grafische Elemente zur Darstellung der Transformation. Die Regeln werden entweder interpretiert oder zu ihrer Ausführung in eine Programmiersprache übersetzt. All diese Werkzeuge werden in diesem Abschnitt mit ModGraph

[2] An dieser Stelle wäre ein Compiler von OCL nach Java angebracht, der meines Wissens nach nicht existiert.

verglichen, wozu verschiedene Kriterien maßgeblich sind.

Des Weiteren wird in diesem Kapitel das propagierende Refactoring aus Kapitel 8 gegen existierende Ansätze abgegrenzt. Selbiges gilt für die Integration von Graphtransformationsregeln mit modalen Sequenzdiagrammen.

10.2.1 EMF und ModGraph

EMF und ModGraph arbeiten Hand in Hand. Ohne ModGraph bietet EMF lediglich Strukturmodellierung in Form von Ecore-Klassendiagrammen an. Zudem kann im reinen EMF Java-Quelltext für diese erzeugt werden. Er stellt neben der Abbildung der Klassen, Attribute und Referenzen auch elementare Operationen zur Verfügung, die Werte setzen oder sie zurückliefern. Für im Modell definierte Operationen werden nur die Signaturen generiert. EMF unterstützt demnach nur teilweise modellgetriebene Softwareentwicklung.[73]

Der ModGraph-Ansatz verwendet das EMF-Rahmenwerk zur Strukturmodellierung und erweitert es nahtlos und inkrementell um die fehlende Verhaltensmodellierung. Es wird ein evolutionärer Ansatz geboten, der den EMF-Nutzer von der Java-Welt in die Modellierung leitet. Der ModGraph-Ansatz nutzt Ecore zur Strukturmodellierung und bindet Graphtransformationsregeln nahtlos in EMF ein. Dabei soll jedoch nur der Mehrwert, den Graphtransformationsregeln bieten, berücksichtigt werden. Dieser Mehrwert entsteht durch deren Fähigkeit, komplexe, überschreibende Modelltransformationen auf einer hohen Abstraktionsebene zu modellieren. Diese wird einerseits durch die, hinter den Regeln liegenden, Graphen ermöglicht, andererseits durch die übersichtliche grafische und farbkodierte Darstellung der Regeln. Damit kann jedes EMF-Projekt, durch Integration in den ModGraph-Ansatz, um Graphtransformationsregeln erweitert werden.

ModGraph nutzt ebenfalls EMF-Generatoren zur Erzeugung von strukturellem Quelltext und injiziert den eigenen Quelltext zur Verhaltensbeschreibung in diesen. Die EMF-Generatoren werden nicht verändert. ModGraph nutzt eigene Templates zur Generierung. Auf diese Weise kann jedes bestehende EMF-Projekt durch ModGraph-Regeln ergänzt werden. Es bietet demnach eine echte Erweiterung bezüglich der Generierung von Quelltext. Dies erweitert die Fähigkeiten zur modellgetriebenen Softwareentwicklung in EMF beträchtlich, da nun das Verhalten mit Regeln nicht nur spezifiziert, sondern auch ausgeführt werden kann. Viele Operationen können nun modellgetrieben entwickelt werden, indem deren Verhalten durch eine Regel modelliert wird.

Dennoch konzentriert sich ModGraph auf den Mehrwert, den es EMF bieten kann, die Graphtransformationsregeln. ModGraph ist keine große eigene Modellierungsumgebung. Es integriert und ergänzt vorhandene Technologien. Da EMF eine in Industrie und Forschung anerkannte und oft genutzte Möglichkeit der Strukturmodellierung ist, muss das sprichwörtliche „Rad“ an dieser Stelle nicht neu erfunden werden. Daher ist auch der grafische Editor für Graphtransformationsregeln im EMF-Kontext unter Einsatz von GMF entwickelt worden.

Die tiefe Verwurzelung von ModGraph in der EMF-Welt zeigt sich zudem in der Nutzung des Ecore-Metamodells. Es dient als Meta-Metamodell für die Graphtransformationsregeln. Damit sind die Regeln nicht nur *für*, sondern auch *mit* EMF erstellt.

Das EMF-Rahmenwerk wird demnach komplett wiederverwendet und erweitert. Dadurch wird ein Ziel des ModGraph-Ansatzes erreicht: Es wird eine echte und gleichzeitig leichtgewichtige Erweiterung angeboten, die Verhaltensmodellierung auf EMF-Basis anbietet.

10.2.2 ModGraph und Xcore

Die eben beschriebene Verhaltensmodellierung für EMF ermöglicht noch keine totale modellgetriebene Softwareentwicklung. Einfache oder stark prozedurale Methoden werden in Java geschrieben. Um auch diese auf die Modellierungsebene zu heben, wird Xcore in den ModGraph-Ansatz eingebunden. Xcore bietet eine textuelle Syntax für Ecore und erweitert diese um eine Verhaltensbeschreibung, indem die Java-nahe Sprache Xbase eingebunden wird. Die Entscheidung für Xcore fußt auf den in [20] gezeigten Gründen und ModGraphs Grundidee, existierende Technologien zu integrieren, anstatt bestehende Konzepte zu reimplementieren. Zudem ist Xcore wie auch ModGraph darauf bedacht, den Programmierer in der Java-Welt abzuholen und ihm Modelle näher zu bringen, so dass er sich ausschließlich mit diesen, anstatt mit Quelltext beschäftigen kann.

Durch diese Integration wird totale modellgetriebene Softwareentwicklung im ModGraph-Ansatz erreicht. Der Nutzer kann Ecore oder Xcore zur Strukturmodellierung nutzen. Die Graphtransformationsregeln werden weiterhin zur Modellierung von komplexen strukturellen Änderungen genutzt und Kontrollstrukturen oder einfache bzw. prozedurale Operationen können nun in Xcore auf Modellebene spezifiziert werden. Zudem können Xcore-Operationen zur Steuerung der Ausführung der Regeln, also als Kontrollfluss, verwendet werden. Es entsteht eine strikte Trennung von regelbasierter und prozeduraler Verhaltensmodellierung, die sich - wie im vorangegangenen Abschnitt gezeigt - positiv auf die Regeln auswirkt.

ModGraph-Regeln stellen gegenüber Xcore-Operationen einen Abstraktionsgewinn dar. Ihre grafische Darstellung, ihre einheitliche Struktur und die implizit gegebene Mustersuche machen eine Regel leichter verständlich als die textuellen Xcore-Operationen. Zudem vereinfachen ModGraph-Regeln die Modellierung. So kann ein Objekt durch einen Knoten erzeugt werden, ohne dass sich der Nutzer mit den dahinter liegenden Fabrikmethoden beschäftigen muss, wie es in Xcore der Fall wäre. Auch die Erzeugung von Links geht wesentlich einfacher von statten. Man zieht eine Kante zwischen zwei Knoten und wählt, sofern mehrere Möglichkeiten bestehen, die entsprechende Referenz aus. Besteht nur eine Möglichkeit zur Verbindung der Knoten, wird diese automatisch gesetzt. Dabei spielt es bei der Modellierung einer Regel keine Rolle, ob der Link optionale oder obligatorische Knoten verbindet. Dies müsste in Xcore explizit geprüft werden. Zudem erfordert das Setzen eines Links in Xcore die Ausführung von Listenoperationen auf den, die Links verwaltenden, Listen. Ein weiterer Vorteil der Regel ist der Einsatz von mehrwertigen Knoten. Sie entsprechen Listen von Objekten in Xcore, die immer durch Iteration einer Schleife betrachtet werden müssen.

Zur Ausführung des Modells kann, wie in Kapitel 7.1 beschrieben, einerseits die Generierung von ausführbarem Java-Quelltext angestoßen werden, andererseits können die Regeln in Xcore-Operationen übersetzt werden. ModGraph und Xcore können sowohl

auf Modellebene als auch auf Quelltextebene interagieren. Die Regeln können in ihre Xcore-Darstellung übersetzt und so interpretiert werden. Damit ist eine Ausführung auf Modellebene durch Interpretation möglich. Dies ist eine echte Alternative zur Implementierung eines Interpreters für die Regeln und bringt Vorteile bezüglich des Debuggens und Testens mit sich. Die Generierung nach Xcore führt zu einem schrittweisen Übersetzungsprozess, der zu einer Unabhängigkeit der ModGraph-Regeln von der letztendlich generierten Zielsprache führt. Damit wäre es möglich, nicht nur ausführbaren Java-Quelltext sondern auch ausführbaren Quelltext einer anderen Programmiersprache zu erzeugen (sofern Xcore zukünftig einen entsprechenden Generator anbieten wird).

Auf diese Art und Weise wird ein einzigartiger Ansatz angeboten, der es erlaubt, die Regeln in eine ausführbare Modellierungssprache statt in eine konventionelle Programmiersprache zu übersetzen und dabei den Mehrwert, den Graphtransformationsregeln bieten, nutzt. Dies zeigt sich insbesondere durch komplexere Regeln (siehe Abschnitt 10.1.1). Deren Aufruf kann durch prozedurale Operationen vorbereitet werden. Ein Beispiel hierfür ist die in Kapitel 8 betrachtete Propagation des Refactorings zum Extrahieren einer Klasse. Hier werden die aufgrund des Modellrefactorings zu verschiebenden Elemente einer Regel zunächst vorsortiert und der Metaregel als Parameter übergeben, so dass sie die strukturellen Änderungen an der Regel durchführen kann. Keines der in Teil II dargestellten Werkzeuge ist dazu in der Lage.

10.2.3 Modelltransformationswerkzeuge und ModGraph

ModGraph im Vergleich zu ATL und QVT

ATL [46] und QVT [57] teilen sich ihren operationalen Kontext und werden daher gemeinsam betrachtet. Beide beschäftigen sich in erster Linie mit Ein-Ausgabe-Transformationen von Modell zu Modell, leiten also ein Zielmodell aus einem Quellmodell ab. Die Transformation kann endogen oder exogen sein. Überschreibende Transformationen sind zwar möglich, werden jedoch in beiden Fällen auf Ein-Ausgabe-Transformationen zurückgeführt und sind in ihrer Ausdrucksmächtigkeit limitiert. ModGraph hingegen unterstützt von Anfang an endogene überschreibende Transformationen.

Desweiteren definieren die Transformationsregeln in ATL und QVT meist globale Transformationen. Dies bedeutet, dass sie an jeder möglichen Stelle angewendet werden. ModGraph bietet im Gegensatz dazu parametrisierte Transformationen an, welche die Anwendungsstelle durch die Parameter festlegen. Die Anwendung einer ModGraph-Regel kann zudem über Kontrollflüsse gesteuert werden.

Abgrenzung von anderen Graphtransformationswerkzeugen

Graphtransformationen und deren Nutzung, unter Beachtung ihres Mehrwerts, bilden das Kernkonzept im ModGraph-Ansatz. Damit reiht sich ModGraph in eine Reihe von Ansätzen und Werkzeugen zur Graphtransformation ein, die in den letzten Jahrzehnten entwickelt wurden und in Ehrig et al. [31] erläutert sind. Dennoch sticht ModGraph gleichzeitig durch spezifische Eigenschaften aus dieser Menge heraus. Sie werden

im Folgenden konkretisiert und ModGraph wird somit von existierenden Werkzeugen abgegrenzt. Hierzu werden die bereits in Teil II betrachteten Ansätze und Werkzeuge verglichen. Sie werden bezüglich der Darstellung und Mächtigkeit von Regeln und Kontrollflüssen sowie ihrer Ausführbarkeit in Relation zu ModGraph gesetzt. An dieser Stelle sei nochmals bemerkt, dass hier nur die großen Ansätze und Werkzeuge betrachtet werden, die endogene Modelltransformationen, teils auf EMF-Basis, durchführen.

Die objektorientierte Modellierungsumgebung Fujaba [83] bietet Klassendiagramme zur Modellierung der statischen Struktur, sowie Storydiagramme zur Verhaltensmodellierung. Ein Storydiagramm realisiert hierbei genau eine Methode aus dem Klassendiagramm und stellt ein Kontrollflussdiagramm dar. Es enthält Statement-Aktivitäten (dies sind reine Java-Quelltextfragmente) sowie Story-Muster, die Graphtransformationsregeln darstellen. Fujaba bietet demnach, im Gegensatz zu ModGraph, eine rein grafische Notation von Regeln und Kontrollfluss, die textuelle Elemente erlaubt. Diese wird von allen weiteren Story-basierten Werkzeugen genutzt. Zu ihrer Ausführung werden die Storydiagramme in Java-Quelltext übersetzt oder interpretiert. Fujaba wurde - im Gegensatz zu ModGraph - unabhängig von EMF entwickelt. Die Idee der Storydiagramme wurde aber zumindest partiell in EMF reimplementiert [36]. Das entsprechende Werkzeug MDELab verwendet Ecore-Klassendiagramme zur Beschreibung der statischen Struktur und bietet einen grafischen Editor und einen Interpreter für Storydiagramme. Dies bedeutet, dass der Endanwender auf den Interpreter beschränkt ist, um mit Modellinstanzen zu arbeiten, während im ModGraph-Ansatz EMF-kompatibler Quelltext erzeugt wird, der auf alle möglichen Arten verwendet werden kann.

eMoflon [6] nutzt Fujabas Storydiagramme. Es baut auf dessen Komponenten auf und erweitert diese deutlich, indem es eine Umgebung anbietet, die nicht nur Storydiagramme und Klassendiagramme, sondern auch Tripelgraphgrammatiken unterstützt. eMoflon erlaubt die Nutzung von Ecore zur struktuellen Modellierung sowie die Generierung von EMF-kompatiblem Quelltext. Da Storydiagramme zum Einsatz kommen, findet auch in eMoflon die komplette Verhaltensmodellierung auf grafischer Basis statt. Regeln und Kontrollfluss werden wie in Story-basierten Ansätzen üblich in einem Diagramm dargestellt. ModGraph hingegen separiert diese strikt. Zur Quelltext-Generierung nutzt eMoflon das Moca-Framework, das vor der eigenlichen Generierung zunächst einen komplexen Baum aufbaut. Zudem wird ein separater Parser zur Erzeugung von Quelltext benötigt. ModGraph verwendet hier Xpand-Templates, die einen zuvor generierten einfachen Suchplan in Form eines Spannwaldes zu jeder Regel nutzen.

Die Ausdrucksmächtigkeit von Story-Mustern ist im Allgemeinen geringer als die von Graphtransformationsregeln in ModGraph, da nur zweitere Vor- und Nachbedingungen, komplexe negative Anwendbarkeitsbedingungen (diese sind verbotene Muster im Graphen) sowie eine OCL-Integration bieten[3].

Wie ModGraph verwendet auch EMF TIGER [15] Ecore zur strukturellen Modellierung. Allerdings bietet EMF TIGER nur Unterstützung für Graphtransformationsregeln, die viele der fortgeschrittenen Konstrukte, die ModGraph zur Verfügung stellt (Men-

[3]Bezüglich der OCL-Integration gab es einen prototypischen Ansatz für Fujaba, der es aber nicht bis zum Release-Stadium geschafft hat [74].

genknoten, Vor- und Nachbedingungen, Pfade, Methodenaufrufe und OCL-Integration), vermissen lassen. Der Nachfolger namens Henshin [8] erweitert die Graphtransformationsregeln von EMF TIGER mit Mengenknoten, Kontrollstrukturen und Vor- bzw. Nachbedingungen. Die in ModGraph genutzten Pfade, Methodenaufrufe und die Möglichkeit, eigene OCL-Constraints anzugeben, fehlen bislang. Wie ModGraph bieten auch EMF TIGER und sein Nachfolger Henshin die Möglichkeit, negative Anwendbarkeitsbedingungen zu spezifizieren.

Graphtransformationsregeln werden in EMF TIGER in eigene Java Klassen anstatt in Methoden übersetzt. Möchte nun ein Entwickler eine benutzerdefinierte Operation, die im Ecore Modell spezifiziert wurde, implementieren, so muss dieser zunächst Code schreiben, um die der Regel zugeordnete Klasse zu instanziieren. Anschließend müssen der Instanz die notwendigen Parameter übergeben werden, bevor schließlich die Regel ausgeführt werden kann. Dies bedeutet aber, dass mehrere Schritte notwendig sind, um eine Regel aufzurufen, was zu hohem Programmieraufwand und hohen Laufzeiten führt.

In ModGraph hingegen ist eine Graphtransformationsregel einer benutzerdefinierten Operation zugeordnet. Weiterhin wird durch den ModGraph-Compiler eine gewöhnliche Java-Methode generiert. Aus der Sicht eines Anwendungsentwicklers bietet ModGraph somit eine nahtlose und effiziente Integration des aus den Graphtransformationsregeln generierten Java-Quelltextes was EMF TIGER nicht bietet. Bis jetzt bietet Henshin nur einen Interpreter, was die Einbettung von Graphtransformationsregeln in eigene Anwendungen erschwert.

Ein Ansatz, der statt der grafischen Notation - wie in ModGraph - auf eine textuelle Repräsentation der Graphtransformationsregeln setzt, ist VIATRA2 [76]. VIATRA2 wurde *mit* Hilfe des EMF-Rahmenwerks gebaut, indem das eigene Metamodell mit Ecore beschrieben wurde. Im Gegensatz dazu wurde ModGraph *für* das EMF-Rahmenwerk gebaut und nahtlos integriert. Insbesondere verwendet ModGraph Ecore oder Xcore für das strukturelle Modell.

GReAT [5] ist eines der großen, EMF-unabhängigen Systeme, die hier aufgrund ihrer Popularität betrachtet werden. Es stellt drei grafische Sprachen zur Spezifikation von Graphtransformationen zur Verfügung: eine zur Erstellung der Muster, eine zur Erzeugung der Transformationen und die dritte zur Definition von Kontrollflüssen. Zudem werden UML-Klassendiagramme als Metamodelle der zu transformierenden Modelle genutzt. ModGraph ist durch seine Verankerung in EMF etwas einfacher aufgebaut. Zwar werden auch in ModGraph Kontrollflüsse in einer anderen Sprache als Graphtransformationsregeln spezifiziert, die Muster und Transformationen lassen sich jedoch gemeinsam spezifizieren. GReAT generiert C++-Quelltext zur Ausführung der Spezifikationen, während ModGraph entweder Java-Quelltext oder Xcore-Modellbestandteile generiert.

GrGen.NET [43] ist ebenfalls EMF-unabhängig und bietet neben anderen Komponenten eine textuelle DSL zur Spezifikation von Graphtransformationsregeln an, die zu ihrer Ausführung in C#-Quelltext übersetzt werden. Dies gilt ebenso für die durch logische oder reguläre Ausdrucke definierten Anweisungen zur Steuerung der Regeln. ModGraph und GrGen.NET verbindet lediglich die Nutzung des Konzepts der Graphtransformation. ModGraph nutzt grafische statt textueller Regeln, die mit textuellen Elementen ergänzt werden und durch prozedurale, imperative Programmierkonstrukte, statt logischer oder

regulärer Ausdrücke, gesteuert werden. Auch die generierten Zielspachen unterscheiden sich.

PROGRES [70] basiert ebenfalls nicht auf EMF, es wird - als Urahn der Graphtransformationswerkzeuge - dennoch verglichen. Es nutzt getypte, attribuierte Graphen zur Definition von Graphtransformationsregeln. PROGRES bietet eine hybride Sprache zur Spezifikation von Graphtransformationsregeln und deren Steuerung. Es werden, wie in ModGraph, Texte und grafische Anteile angeboten. Dabei ist in PROGRES jedem grafischen Element eine textuelle Darstellung zugeordnet. PROGRES integriert prozedurale und regelbasierte Anteile in einer Sprache, während ModGraph diese strikt separiert. Alle diese Konstrukte sind innerhalb einer Spezifikation untereinander angeordnet. Zudem ist in PROGRES die explizite Deklaration von linker und rechter Regelseite nötig, während ModGraph eine zusammengefügte Ansicht beider Seiten nutzt.

Tabellen 10.11 und 10.12 geben einen Überblick über die Werkzeuge und deren Gemeinsamkeiten und Unterschiede. Betrachtet man zunächst Tabelle 10.11, listet diese die Werkzeuge in einer alphabetisch geordneten Weise auf und nennt deren jeweilige Darstellung von Regeln und Kontrollflüssen. Zudem wird ein Überblick zur EMF-Kompatibilität gegeben. Eine Sprache bzw. ein Werkzeug wird hier als EMF-kompatibel bezeichnet, wenn das Ecore-Metamodell zur strukturellen Modellierung genutzt wird. Aus dieser Tabelle ist deutlich ersichtlich, dass die meisten Ansätze, wie auch ModGraph, eine grafische einer textuellen Repräsentation der Graphtransformationsregeln vorziehen. Im Gegensatz dazu herrscht bei den Kontrollflüssen nahezu ein Gleichgewicht zwischen der Verwendung textueller und grafischer Repräsentationen. Im ModGraph-Ansatz wird ein textueller Kontrollfluss genutzt. Die Entscheidung für diesen basiert auf den Ergebnissen aus [20] und den unter Anderem dort niedergeschriebenen, am Lehrstuhl vorhandenen Erfahrungen mit Fujaba, das Kontrollflüsse grafisch darstellt. Unter allen betrachteten Ansätzen bietet nur PROGRES eine ähnliche Vorgehensweise, indem es seine native Kontrollflusssprache nutzt. ModGraph nutzt Xcore hierfür.

Weiterhin sind die meisten der verglichenen Sprachen in Schichten aufgebaut, die den Kontrollfluss oberhalb der regelbasierten Sprache zur Spezifikation der Graphtransformationen ansiedeln. Beispielsweise ist in PROGRES der elementarste Bestandteil die Regelspezifikation. Nur ModGraph und Fujaba weichen von diesem schichtweisen Aufbau ab. In ModGraph können einfache Operationen mit Xcore spezifiziert werden, Fujaba lässt Java-Quelltext in Form von Statement-Aktivitäten zu, die direkt in den generierten Quelltext übernommen werden. Damit entstehen in Fujaba direkte Abhängigkeiten zur Programmiersprache Java, die als Zielsprache zur Ausführung der Regeln verwendet wird. So bricht Fujaba die Separation von Modellierung und Programmierung. In ModGraph werden hingegen Operationen auf Modellebene genutzt. Damit bleibt das Modell strikt getrennt vom Quelltext und unabhängig von der zur Ausführung genutzten Zielsprache.

Wie die Regeln der betrachteten Ansätze und Werkzeuge ausgeführt werden, ist in Tabelle 10.12 dargestellt. Wird Quelltext in einer Programmiersprache generiert, so ist auch die Zielsprache angegeben. Henshin und MDELab bieten jeweils einen eigenen Interpreter zur Ausführung der Regeln an. Sie generieren keinen Quelltext. Damit sind die Möglichkeiten der Ausführung der Regel auf die Fähigkeiten des Interpreters beschränkt.

Sprache/Werkzeug	*GT Regeln*		*Kontrollfluss*		*EMF-kompatibel*
	textuell	*grafisch*	*textuell*	*grafisch*	
eMOFLON [6]	-	x	-	x	x
Fujaba [83]	-	x	-	x	-
GReAT [5]	-	x	-	x	-
GrGen.NET [43]	x	-	x	-	-
Henshin [7]	-	x	-	x	x
MDELab [36]	-	x	-	x	x
ModGraph	-	x	x	-	x
PROGRES [70]	x	x	x	-	-
EMF TIGER [15]	-	x	-	x	x
VIATRA2 [76]	x	-	x	-	-

Tabelle 10.11: *Graphtransformationssprachen und Werkzeuge*

Sprache/Werkzeug	*Interpreter*	*Compiler*	*Zielsprache(n) (falls kompiliert)*
eMOFLON [6]	-	x	Java
Fujaba [55]	x	x	Java
GReAT [5]	x	x	C++
GrGen.NET [43]	-	x	C#
Henshin [8]	x	-	-
MDELab [36]	x	-	-
ModGraph	(x)	x	*Xcore* oder Java
PROGRES [70]	x	x	C oder Java
EMF TIGER [15]	-	x	Java
VIATRA2 [76]	-	x	Java

Tabelle 10.12: *Ausführbarkeit von Graphtransformationssprachen*

EMF Tiger, eMoflin und GrGen.NET sind die einzigen betrachteten Werkzeuge, die keine Möglichkeit zur Interpretation der Regeln bieten. Aus einer GrGen-Spezifikation wird immer ausführbarer C#-Quelltext generiert. EMF Tiger generiert Java-Klassen aus den Regeln.

Alle anderen Werkzeuge sind mit einem Interpreter und einem Compiler für die Regeln ausgestattet. Fujaba, PROGRES und VIATRA2 generieren, wie ModGraph, ausführbaren Java-Quelltext. PROGRES ermöglicht zudem die Nutzung von C als Zielsprache der Generierung. ModGraph nimmt hier eine Sonderstellung ein. Es erlaubt die Kompilierung von Regeln in ein textuelles Xcore-Modell. Die Generierung auf Modellebene ist einzigartig. Alle anderen Werkzeuge generieren Quelltext einer Programmiersprache. Durch diese Übersetzung der Regeln nach Xcore werden die ModGraph-Regeln interpretierbar. Die Markierung der Spalte *Interpreter* in der ModGraph Zeile der Tabelle ist daher zu Recht eingeklammert. Die Regeln sind nur über den Umweg nach Xcore interpretierbar. Dies stellt eine kostengünstige Alternative zur Implementierung

eines Interpreters für die Regeln dar, was insbesondere im vorangegangenen Kapitel 9 deutlich wird. Hier werden die nach Xcore übersetzten Regeln zum Play-out verwendet.

Der Literaturvergleich zeigt, dass es bereits einige EMF-basierte Systeme für die Erstellung und die Ausführung von Graphtransformationsregeln gibt. ModGraph zeichnet sich jedoch durch eine nahtlose Integration mit dem EMF-Rahmenwerk und den Compiler-Ansatz, der Regeln auf Modell- und Quelltextebene kompilieren kann, aus. Es ist das einzige System, das aus Graphtransformationsregeln Code erzeugt, der den von EMF bzw. Xcore generierten Code erweitert. Zudem bietet es eine (bislang im Graphtransformationskontext nicht verfügbare) Übersetzung der Regeln direkt in das Xcore-Modell an. Außerdem zeichnet es sich durch eine einzigartige, strikte Trennung von regelbasierter und prozeduraler Verhaltensmodellierung aus. ModGraph forciert zudem den Mehrwert, den Graphtransformationsregeln für EMF bieten können und lässt dem Nutzer die Freiheit, Regeln nur in Fällen anzuwenden, in welchen dieser Mehrwert ausgeschöpft werden kann. Dass dies zu einer positiven Entwicklung der Regelnutzung führt, ist in Abschnitt 10.1 gezeigt.

10.3 Abgrenzung des propagierenden Refactorings

Die in Kapitel 8 eingeführten propagierenden Refactorings wurden *für* und *mit* ModGraph erstellt. Dennoch bilden sie eine eigene Anwendung, die hier explizit einzeln abgegrenzt werden soll.

Mens et Tourwé geben in [53] einen Überblick zum Thema Softwarerefactoring. Fowlers Buch zum Thema [33] ist eine der meist genannten Referenzen auf diesem Gebiet. Allerdings beziehen sich die im Buch genannten Refactorings auf objektorientierte Programme. Sie werden demnach auf Quelltext, statt auf Modelle angewendet. Zudem werden die Refactorings im Buch nur informell definiert. Eine Formalisierung durch Programmgraphen und Graphtransformationsregeln wird von Mens et al. in [51] gegeben. Die bedeutende Einschränkung bei dieser Arbeit ist die Betrachtung einzelner Regeln, die normalerweise nicht ausreicht. Man benötigt programmierte Graphtransformationsregeln, um Refactoringtransformationen zu spezifizieren. Diese Problemstellung wurde im GraBaTs-Workshop 2008[4] näher betrachtet. Dazu wurde die Aufgabe, drei Refactorings (Verschieben und nach oben Ziehen einer Methode sowie Kapseln eines Felds) auf Programmgraphen mit Graphtransformationswerkzeugen zu implementieren, gestellt. Eine der Lösungen findet sich in Geiger [34]. Dabei wird Fujaba zur Umsetzung genutzt.

Die Forschung auf dem Gebiet des Refactorings beschränkt sich jedoch auf strukturelle Modelle, während im propagierenden Refactoring Struktur- und Verhaltensmodelle, die Graphtransformationsregeln, betrachtet werden. Die bislang in der Forschung untersuchten Refactoringtransformationen werden in der Regel auf Klassendiagrammen ausgeführt. Beispielsweise bieten Biermann et al. [14] ein Refactoring für Ecore-Modelle, dessen Transformationen mit Regeln in AGG spezifiziert sind. Zudem zeigt Mens in

[4]Siehe `http://fots.ua.ac.be/events/grabats2008/`

[50], wie Refactorings auf UML-Klassendiagramme angewendet werden können. Dabei betrachtet er sowohl eine Implementierung der Refactorings in AGG als auch eine in Fujaba und kommt zu dem Schluss, dass beide Graphtransformationssprachen verwendet werden können.

Der Ansatz von Bottoni [16] geht einen Schritt weiter. Er bietet integrierte Refactoringtransformationen an, die auf ein UML-Klassendiagramm und auf den daraus generierten Quelltext gleichermaßen angewendet werden. Dies unterscheidet sich von dem hier gezeigten propagierenden Refactoring, da im propagierenden Refactoring zusätzlich zum strukturellen Modell Verhaltensmodelle betrachtet werden, Bottoni sich jedoch mit Quelltext befasst.

Bislang gibt es nur wenige Ansätze, die sich mit der Propagation von Modelländerungen auf das Verhaltensmodell beschäftigen. Rosner und Bauer zeigen in [65] eine Möglichkeit, Modelltransformationen als Reaktion auf Metamodelländerungen zu aktualisieren. Dazu wird eine ontologische Abbildung zwischen den Metamodellversionen benötigt und diese genutzt, um QVT-R-Transformationen anzupassen. Diese Evolution von Modelltransformationen entspricht einer Transformation höherer Ordnung [75].

Der einzige auffindbare Ansatz, der sich mit der Propagation von Änderungen auf Graphtransformationsregeln beschäftigt, ist in Levendovszky et al. [49] zu finden. Er beschäftigt sich mit der Propagation auf Regeln in GReAT [5]. Dieser Ansatz unterliegt jedoch einigen Limitierungen. Es werden nur elementare Änderungen am Metamodell in Betracht gezogen, wie zum Beispiel das Umbenennen einer Klasse. Diese werden weiterhin nur halbautomatisch propagiert und die entstehenden Regeln können nach der Propagation syntaktische und semantische Fehler aufweisen. Das hier vorgestellte propagierende Refactoring beschäftigt sich mit komplexen Refactorings, automatisiert viele Abläufe und vermeidet bei elementaren Änderungen syntaktische Fehler in den Regeln. (Bei komplexen Änderungen können diese nicht vollends ausgeschlossen werden.)

Zusammenfassend betrachtet ist das propagierende Refactoring einzigartig. Es gibt bislang keine Möglichkeit, Refactoring *für* und *mit* Graphtransformationsregeln durchzuführen. Das Refactoring wird hierbei integrativ durchgeführt, da Modell und Regeln in einem Schritt restrukturiert werden, um die Konsistenz zwischen den beiden zu erhalten.

10.4 Abgrenzung der Integration der Regeln mit modalen Sequenzdiagrammen

Die in Kapitel 9 besprochene Integration von Graphtransformationsregeln und modalen Sequenzdiagrammen erweitert das Spektrum der verwandten Arbeiten. MSD-ähnliche Ansätze müssen zunächst berücksichtigt werden, als auch Ansätze, die Graphtransformationsregeln mit UML-Diagrammen vereinen. Die einzelnen Werkzeuge und Ansätze werden hier lediglich abgegrenzt. Zur genaueren Information über diese Werkzeuge sei auf die im weiteren Verlauf angegebene Literatur verwiesen.

Unter den Scenario-basierten Werkzeugen und Ansätzen finden sich einige Spezifikations- und Analyse-Ansätze. Manche davon basieren auf MSDs oder Live Sequence

Charts. Dennoch finden sich bislang keine Ansätze, die eine Rekonfiguration der Objekte zur Laufzeit, wie sie in Kapitel 9 gezeigt wurde, derart rigoros unterstützt. Dennoch sollen zwei Ansätze zu dem vorgestellten Ansatz in Relation gesetzt werden.

MechatronicUML [12] ist eine Designmethode für selbst-adaptive mechatronische Systeme. Diese Methode besteht aus einer Sprachfamilie, die die Modellierung von Echtzeitverhalten der Systemkomponenten und die Rekonfiguration der Architektur unterstützt [62]. Das Echtzeitverhalten der Komponenten wird mittels Zustandsautomaten, die durch Echtzeit-Annotationen ergänzt werden, realisiert. Die Rekonfiguration der Architektur wird mittels Graphtransformationsregeln, die auf der Struktur der Komponenten agieren, erreicht. Genau wie im hier präsentierten Ansatz werden die Graphtransformationsregeln dazu genutzt, die Struktur aktiver Komponenten und deren Verhalten zu verändern. Jedoch wird in MechatronicUML die nachrichtenbasierte Interaktion durch Zustandsautomaten innerhalb der Komponenten beschrieben, während im hier präsentierten Ansatz komponentenübergreifende Szenarien betrachtet werden. Letztere sind in der frühen Designphase deutlich verständlicher.

Diethelm et al. [26] zeigen einen komplementären Ansatz zur Kombination von Szenarien und Graphtransformationsregeln. Sie nutzen eine Menge einfacher - durch Graphtransformationsregeln dargestellter - Szenarien als Eingabe und bauen daraus Storydiagramme auf, wie sie bereits in 4.2.3 erklärt wurden. Die Idee dahinter ist, Szenarien zu identifizieren, die gleiche Aktionen beinhalten, um diese in eine Aktivität des Storydiagramms zu überführen. Zudem wird nicht berücksichtigt, dass Graphtransformationsregeln bei ihrer Ausführung die Struktur der Objekte verändern, was eine Änderung der möglichen ausführbaren Szenarien hervorruft.

Story-basierte Ansätze sind zudem die einzigen unter den Graphtransformationsansätzen, die die Vereinigung von Graphtransformationsregeln und UML-Verhaltensdiagrammen anbieten. UML-Aktivitätsdiagramme werden mit Graphtransformationsregeln durch direkte Einbettung der Regeln in die Aktivitäten des Diagramms vereint. Augenscheinlich nutzen beide Ansätze zunächst Klassendiagramme zur Strukturmodellierung. Zur Verhaltensmodellierung nutzen die Story-basierten Ansätze jedoch Aktivitätsdiagramme statt modaler Sequenzdiagramme. Zudem sind die Graphtransformationsregeln in dem in Kapitel 9 vorgestellten Ansatz nicht direkt in das UML-Diagramm integriert, sondern werden gesondert aufgerufen.

MDELab [36], einziger Vertreter der Story-basierten Ansätze, der Interpretation unterstützt, ist am ehesten mit dem in Kapitel 9 vorgestellten Ansatz zu vergleichen. MDELab bietet einen Interpreter für Storydiagramme an, der jedoch explizit aufgerufen werden muss, um ein System zu interpretieren. Dies bedeutet, dass jede Regel einzeln interpretiert werden muss. Trotz seines visuellen Debuggers, der eine schrittweise Ausführung der Storydiagramme ermöglicht, ist MDELab nicht zur Simulation von komplexen Szenarien geeignet. Im Gegensatz dazu bietet der szenarienbasierte Ansatz eine Simulationsumgebung zur Interpretation mittels Play-out, innerhalb dessen zusammenhängende Szenarien betrachtet werden können, ohne diesen expliziten Aufruf durchführen zu müssen.

10.5 Ergebnis und Ausblick

In dieser Arbeit wurde der ModGraph-Ansatz und die daraus resultierende Werkzeugumgebung entwickelt und dargestellt. Dazu wurden im ersten Teil der Arbeit einige Grundlagen erörtert. Im zweiten Teil wurden bestehende Ansätze und Werkzeuge, die im ModGraph-Ansatz genutzt werden oder mit diesem in Verbindung stehen, untersucht. Im dritten Teil der Arbeit wurde der ModGraph-Ansatz und die daraus resultierende Werkzeugumgebung von der Konzeption bis zur fertigen Modellierungsumgebung - die totale modellgetriebene Softwareentwicklung mit EMF ermöglicht - eingehend betrachtet. Die entstandene Umgebung wurde im vierten Teil evaluiert. ModGraph bietet einen integrativen Ansatz, der eine echte Erweiterung *für* und *mit* EMF darstellt, die EMFs Lücke, den Mangel an Verhaltensmodellierung, schließt. Dazu wird zunächst die Ecore-Modellinstanz als Graph aufgefasst und durch Verhaltensmodellierung mit parametrisierten, programmierten Graphregeln ergänzt. Des Weiteren wird ein Paketdiagrammeditor eingebunden, der die Modellierung im Großen ermöglicht. Da dieser zu Beginn des Projekts bereits vorhanden war, liegt der hier betrachtete Schwerpunkt auf der Verhaltensmodellierung.

Die Sprache der verwendeten Regeln wird *für* und *mit* EMF erstellt, so dass die Regeln nahtlos in EMF integriert werden können. Um dies zu erreichen, wird jede Regel als Implementierung einer im Ecore-Modell definierten Operation betrachtet. Zur Modellierung ihres Verhaltens bietet ModGraph Graphtests, Graphabfragen und Graphtransformationsregeln an. Diese werden in einem kompakten, grafischen Modell dargestellt, das es erlaubt, sowohl Graphmuster als auch Änderungen an diesen in einer übersichtlichen und intuitiv farbkodierten Darstellung zu spezifizieren. Die Muster dürfen zudem mit Bedingungen versehen werden. Es stehen textuelle Vor- und Nachbedingungen sowie durch Graphen modellierte NACs zur Verfügung.

Bezüglich des Einsatzes konzentriert sich ModGraph auf den Mehrwert, den Graphtransformationsregeln bieten. Ihre große Stärke liegt in der übersichtlichen und kompakten Art, komplexe strukturelle Veränderungen darzustellen. Allerdings bietet die Integration von Regeln noch keine Möglichkeit, Kontrollflüsse, die jedes Programm benötigt, zu modellieren. Hierzu wird im ModGraph-Ansatz Xcore verwendet. Damit ist die Ausführung der Transformationsregeln durch Xcore-Operationen steuerbar. Zudem erlaubt die Xcore-Integration die Spezifikation einfacher oder prozeduraler Operationen. Die Regeln interagieren mit den Xcore-Operationen, indem sie sich gegenseitig aufrufen. Damit erhält man nicht nur mit Xcore programmierte, parametrisierte Regeln, sondern auch die Möglichkeit, einfache Operationen, direkt aus der Regel aufzurufen. Die Verhaltensmodellierung bleibt dabei in zwei hochgradig interagierende Anteile, einen regelbasierten und einen prozeduralen, getrennt.

Dieses Zusammenspiel von Ecore, ModGraph-Regeln und Xcore ermöglicht totale modellgetriebene Softwareentwicklung, die speziell für und außerdem mit EMF entwickelt wurde. Es handelt sich hierbei um einen integrativen, leichtgewichtigen Ansatz, der EMF erweitert, ohne „das Rad neu zu erfinden“, indem existierende Technologien (Ecore-Modelle und Xcore) wiederverwendet werden. Gleichzeitig wird der Mehrwert genutzt,

den Graphtransformationsregeln zur Verhaltensmodellierung bieten.

Die Ausführung des Modells kann auf Quelltextebene oder auf Modellebene stattfinden. Soll das Modell auf Quelltextebene ausgeführt werden, generiert zunächst der EMF- oder Xcore-Generator Java-Quelltext. Danach wird Quelltext aus den Regeln generiert, der nahtlos in den EMF oder Xcore-generierten Quelltext eingebunden wird, so dass das ausführbare Modell entsteht. Andererseits kann die Ausführung auf Modellebene stattfinden. Hierzu werden die Regeln nach Xcore kompiliert. Dieses kann nun interpretiert oder zu Java-Quelltext kompiliert werden.

Zudem werden Refactoringoperationen für und mit den Graphtransformationsregeln angeboten. Diese wurden in und für ModGraph erstellt und propagieren Änderungen am Ecore-Modell auf die Regeln.

Zuletzt wurden Integrationsmöglichkeiten der Regeln in die szenarienbasierte Entwicklung untersucht, die Echtzeitsysteme simuliert. Dazu wurden die Regeln mit modalen Sequenzdiagrammen integriert und genutzt, um komplexe Seiteneffekte im System zu modellieren und mittels Play-out zu simulieren.

Zusammengefasst zeichnet sich der ModGraph-Ansatz durch folgende Punkte aus:

- Gesamtkonzept zur totalen modellgetriebenen Softwareentwicklung in EMF
- Konzentration auf den Mehrwert der Graphtransformationsregeln
- Integration teils bestehender Einzelkonzepte
- Integration von Paketdiagrammen zur Modellierung im Großen
- Leichtgewichtige, nahtlos integrierte Lösung, die *für* und *mit* EMF erstellt wurde
- Strikte Trennung von prozeduraler und regelbasierter Verhaltensmodellierung
- Intuitive grafische Darstellung der Graphtransformationsregeln
- Steuerung der Regeln mit Xcore-Kontrollflüssen
- Erstellung einfacher Operationen in Xcore
- Nahtlose Interaktion von Xcore und Graphtransformationsregeln
- Werkzeugunterstützung in Eclipse mit Dashboard und Cheat-Sheets zur Anleitung und Einarbeitung des Nutzers
- Compiler-Lösung auf Modell- und Quelltextebene
- Refactoringoperationen *für* und *mit* Graphtransformationsregeln
- Einen Ansatz zur Integration der Regeln in Szenarien zur Simulation von Echtzeitsystemen
- Dokumentation und Beispiele auf der ModGraph-Homepage

Aufbauend auf diesen umfassenden Modellierungsmöglichkeiten wäre es denkbar, den Ansatz zu erweitern. Zunächst wäre, zusätzlich zur aktuell bestehenden konjunktiven Verknüpfung der Bedingungen an eine Regel, das Zulassen einer disjunktiven zu überlegen. Damit könnten beispielsweise NACs spezifiziert werden, die wahlweise nicht auftreten dürfen. Zudem wäre es denkbar, ModGraph mit einen konfigurierbaren Generator auszustatten. Dieser wurde bereits in Abschnitt 5.2.1 erwähnt. Er könnte die, unter Umständen bei großen Modellen durch Einhaltung der Konsistenzbedingungen an die Instanz entstehenden, höheren Laufzeiten positiv beeinflussen. Dabei wäre es dem Modellierer erlaubt, die Anzahl der einzuhaltenden Bedingungen vor der Code-Generierung zu bestimmen.

Als Weiterentwicklung des Ansatzes wäre eine textuelle Darstellung der Regeln denkbar. Diese kann komplett eigenständig oder im Xcore-Kontext erstellt werden. Die Xcore Syntax kann um die Fähigkeit Graphtransformationsregeln textuell im Xcore-Modell als Operationsrümpfe zu spezifizieren erweitert werden. Damit müssten die Regeln nicht mehr in Xcore- bzw. Xbase-Code übersetzt werden. Grafische Regeln würden lediglich in ihrer textuellen Form im Modell notiert. Das Ergebnis wäre ein erweitertes Xcore, das direkt die Spezifikation textueller Regeln erlaubt. Dabei wäre es von Vorteil, wenn eine textuelle Regel in ihre grafische Repräsentation und andersherum überführt werden kann.

Eine andere Alternative, die wahrscheinlich komplexer umzusetzen ist, wäre Xcore, wie es ist, zu verwenden und zu versuchen, ein Mapping zwischen den Xcore-Elementen und den Elementen einer Graphtransformationsregel zu definieren. Der große Vorteil dieses Ansatzes ist das entstehende Round-Trip-Engineering zwischen Xcore und ModGraph. Damit müsste allerdings die aktuelle Modell-zu-Text-Transformation zwischen der Regel und dem textuellen Modell durch eine - im Idealfall bidirektionale - Modell-zu-Modell-Transformation zwischen Xcore und ModGraph-Regeln ersetzt werden. Auf diese Weise würde ein Regelelement wohl auf mehrere Xcore-Elemente abgebildet werden, was die Rückrichtung der Transformation erschwert. Dies bliebe bei der Umsetzung zu untersuchen. Das Mapping könnte beispielsweise mittels Tripelgraphgrammatiken oder QVT erstellt werden.

Des Weiteren wäre die nächste Entwicklungsstufe des propagierenden Refactorings anzudenken. Änderungen am Ecore-Modell sollten hierbei nicht nur auf die Regeln, sondern auch auf die in Xcore spezifizierten Kontrollflüsse propagiert werden. Dies kann erreicht werden, indem weitere (Meta-)regeln und Transformationsoperationen definiert werden, die auf Elementen des Xcore-Modells operieren.

Zudem wäre eine tiefer gehende Untersuchung der Anwendbarkeit der Regeln im Kontext der szenarienbasierten Entwicklung interessant. Hierbei könnten nicht nur interpretierte, sondern auch kompilierte Lösungen untersucht werden.

Außerdem kann über eine tiefer gehende Evaluation des Ansatzes durch die Auswertung künftiger mittelgroßer Projekte nachgedacht werden.

Zusammengefasst bietet ModGraph einen fundierten Ansatz und ein darauf aufbauendes Werkzeug zur totalen modellgetriebenen Softwareentwicklung im EMF-Kontext, der umfangreiche Möglichkeiten bietet, indem er den Mehrwert von programmierten Graphtransformationsregeln in EMF nutzt und gleichzeitig Spielraum für eine Weiterentwicklung lässt.

Literaturverzeichnis

[1] EMF Online-Dokumentation:
http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html, 2014.

[2] OCLinEcore Online-Dokumentation:
http://wiki.eclipse.org/OCL/OCLinEcore, 2014.

[3] Xcore Online-Dokumentation:
http://wiki.eclipse.org/Xcore, 2014.

[4] Xtext Online-Dokumentation:
http://www.eclipse.org/Xtext/documentation.html, 2014.

[5] Aditya Agrawal, Gabor Karsai, Sandeep Neema, Feng Shi, and Attila Vizhanyo. The design of a language for model transformations. *Software and System Modeling*, 5(3):261–288, 2006.

[6] A. Anjorin, M. Lauder, S. Patzina, and A. Schürr. eMoflon: Leveraging EMF and Professional CASE Tools. In *INFORMATIK 2011*, Band 192 aus *Lecture Notes in Informatics*, page 281, Bonn, October 2011. Gesellschaft für Informatik, Gesellschaft für Informatik. extended abstract.

[7] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced concepts and tools for in-place EMF model transformations. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Proceedings 13th International Conference on Model Driven Engineering Languages and Systems (MODELS 2010), Part I*, Band 6394, Seiten 121–135, Oslo, Norway, October 2010.

[8] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced concepts and tools for in-place EMF model transformations. In *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems: Part I*, MODELS'10, Seiten 121–135, Berlin, Heidelberg, 2010. Springer-Verlag.

[9] Daniel Balasubramanian, Anantha Narayanan, Christopher P. van Buskirk, and Gabor Karsai. The graph rewriting and transformation language: Great. *ECEASST*, 1, 2006.

[10] Andras Balogh and Daniel Varro. Advanced model transformation language constructs in the viatra2 framework. In Hisham Haddad, editor, *SAC*, Seiten 1280–1287. ACM, 2006.

[11] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data (SIGMOD 1987)*, Seiten 311–322, San Franciso, CA, 1987.

[12] Steffen Becker, Stefan Dziwok, Christopher Gerking, Wilhelm Schäfer, Christian Heinzemann, Sebastian Thiele, Matthias Meyer, Claudia Priesterjahn, Uwe Pohlmann, and Matthias Tichy. The MechatronicUML design method – process and language for platform-independent modeling. Technical Report tr-ri-14-337, Heinz Nixdorf Institute, University of Paderborn, March 2014. Version 0.4.

[13] Jean Bézivin. On the unification power of models. *Software and System Modeling*, 4(2):171–188, 2005.

[14] Enrico Biermann, Karsten Ehrig, Christian Köhler, Günter Kuhns, Gabriele Taentzer, and Eduard Weiss. EMF model refactoring based on graph transformation concepts. In Jean-Marie Favre, Reiko Heckel, and Tom Mens, editors, *Proceedings of the Third Workshop on Software Evolution Through Transformations: Embracing the Change*, Band 3 aus *Electronic Communications of the EASST*, Natal, Rio Grande del Norte, Brazil, September 2006. 16 p.

[15] Enrico Biermann, Karsten Ehrig, Christian Köhler, Günter Kuhns, Gabriele Taentzer, and Eduard Weisss. Graphical definition of in-place transformations in the eclipse modeling framework. In *Proceedings of Model Driven Engineering Languages and Systems (MoDELS'06)*, Band 4199 aus *LNCS*, Seiten 425–439, 2006.

[16] Paolo Bottoni, Francesco Parisi-Presicce, and Gabriele Taentzer. Specifying integrated refactoring with distributed graph transformations. Seiten 220–235.

[17] Christian Brenner, Joel Greenyer, and Valerio Panzica La Manna. The ScenarioTools play-out of modal sequence diagram specifications with environment assumptions. In *Proc. 12th Int. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2013)*, Band 58. EASST, 2013.

[18] Thomas Buchmann, Alexander Dotor, and Martin Klinke. Supporting modeling in the large in fujaba. In Pieter van Gorp, editor, *Proceedings of the 7th International Fujaba Days*, Seiten 59–63, Eindhoven, The Netherlands, November 2009.

[19] Thomas Buchmann, Bernhard Westfechtel, and Sabine Winetzhammer. MODGRAPH - A Transformation Engine for EMF Model Transformations. In *Proceedings of the 6th International Conference on Software and Data Technologies*, Seiten 212 – 219, 2011.

[20] Thomas Buchmann, Bernhard Westfechtel, and Sabine Winetzhammer. The added value of programmed graph transformations - a case study from software configuration management. In Andy Schürr, Dániel Varró, and Gergely Varró, editors, *Applications of Graph Transformations with Industrial Relevance*, Band 7233 aus *Lecture Notes in Computer Science*, Seiten 198–209. Springer Berlin / Heidelberg, 2012.

[21] Thomas Buchmann, Bernhard Westfechtel, and Sabine Winetzhammer. Modgraph: Graphtransformationen für EMF. In Elmar J. Sinz and Andy Schürr, editors, *Modellierung 2012*, Band 201 aus *Lecture Notes in Informatics*, Seiten 107–122, Bamberg, Deutschland, March 2012. GI.

[22] Per Cederqvist, Roland Pesch, et al. Version management with cvs. *Available online with the CVS package. Signum Support AB*, 1992.

[23] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006.

[24] Werner Damm and David Harel. LSCs: Breathing life into message sequence charts. In *Formal Methods in System Design*, Band 19, Seiten 45–80. Kluwer Academic, 2001.

[25] Reinhard Diestel. *Graphentheorie*. Springer Verlag, 2006.

[26] Ira Diethelm, Leif Geiger, Thomas Maier, and Albert Zündorf. Turning collaboration diagram strips into storycharts. Florida, Orlando, USA, 2002.

[27] Nikita Dümmel. Refactoring mit Graphtransformationsregeln. Master's thesis, University of Bayreuth, Bayreuth, Germany, 2013.

[28] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Wilhelm Hasselbring, Robert von Massow, and Michael Hanus. Xbase: Implementing domain-specific languages for java. In *GPCE '12 Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, Seiten 112–121. ACM, New York, NY, USA, 2012.

[29] Sven Efftinge and Markus Völter. oaw xtext: A framework for textual dsls. In *Eclipsecon Summit Europe 2006*, 2006.

[30] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, Berlin, illustrated edition edition, 2006.

[31] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools*, Band 2. World Scientific, Singapore, 1999.

[32] Claudia Ermel, Michael Rudolf, and Gabriele Taentzer. The agg approach: Language and environment. In *Handbook of graph grammars and computing by graph transformation*, Seiten 551–603. World Scientific Publishing Co., Inc., 1999.

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

[34] Leif Geiger. Graph transformation-based refactorings using Fujaba. In Arend Rensink and Pieter van Gorp, editors, *4th International Workshop on Graph-Based Tools: The Contest*, Leicester, UK, 2008.

[35] Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack, and Adam Szalkowski. Grgen: A fast spo-based graph rewriting tool. In *Graph Transformations, Third International Conference, ICGT 2006, Natal, Rio Grande do Norte, Brazil, September 17-23, 2006, Proceedings*, Seiten 383–397, 2006.

[36] Holger Giese, Stephan Hildebrandt, and Andreas Seibel. Improved flexibility and scalability by interpreting story diagrams. In Artur Boronat and Reiko Heckel, editors, *Proceedings of the 8th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2009)*, Band 18 aus *Electronic Communications of the EASST*, York, UK, March 2009. 12 p.

[37] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. The Eclipse Series. Boston, MA, 1st edition, 2009.

[38] Hans Jürgen Schneider. Graph grammars. In Marek Karpinski, editor, *Proceedings of the International Conference on Fundamentals of Computation Theory (FCT 1977)*, Band 56 aus *Lectures Notes in Computer Science*, Seiten 314–331, Berlin, 1977. Springer-Verlag.

[39] David Harel and Shahar Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software and Systems Modeling (SoSyM)*, 7(2):237–252, May 2008.

[40] David Harel and Rami Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, August 2003.

[41] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens. Cope-automating coupled evolution of metamodels and models. *ECOOP 2009–Object-Oriented Programming*, Seiten 52–76, 2009.

[42] Martin Hitz and Gerti Kappel. *UML @ Work - Objektorientierte Modellierung mit UML 2*. Dpunkt Verlag, 69115 Heidelberg, 3. auflage edition, 2005.

[43] Edgar Jakumeit, Sebastian Buchwald, and Moritz Kroll. Grgen.net — the expressive, convenient and fast graph rewrite system. *International Journal on Software Tools for Technology Transfer*, 12:263–271, 2010.

[44] F. Jouault and I. Kurtev. On the architectural alignment of atl and qvt. In *Proceedings of the 2006 ACM symposium on Applied computing*, Seiten 1188–1195, New York, April 2006. ACM Press.

[45] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Sci. Comput. Program.*, 72(1-2):31–39, 2008.

[46] Frédéric Jouault and Ivan Kurtev. Transforming models with ATL. In Jean-Michel Bruel, editor, *MoDELS Satellite Events*, Band 3844, Seiten 128–138, 2005.

[47] Thomas Kühne. Matters of (meta-)modeling. *Software and System Modeling*, 5(4):369–385, 2006.

[48] E. Leblebici, A. Anjorin, and A. Schürr. Developing eMoflon with eMoflon. In D. Ruscio and D. Varro, editors, *ICMT 2014*, Band 8568 aus *Lecture Notes in Computer Science (LNCS)*, Seiten 138–145, Heidelberg, 2014. Springer Verlag, Springer Verlag.

[49] Tihamer Levendovszky, Daniel Balasubramanian, Anantha Narayanan, and Gabor Karsai. A novel approach to semi-automated evolution of DSML model transformation. In Mark van den Brand, Dragan Gasevic, and Jeff Gray, editors, *Proceedings of the Second International Conference on Software Language Engineering (SLE 2009)*, Band 5969, Seiten 23–41, Denver, CO, 2009.

[50] Tom Mens. On the use of graph transformations for model refactoring. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *International Summer School on Generative Techniques in Software Engineering (GTTSE 2005)*, Band 4143 aus *LNCS*, Seiten 219–257, Braga, Portugal, July 2005.

[51] Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing refactorings with graph transformations. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, 2005.

[52] Tom Mens, Gabriele Taentzer, and Olga Runge. Analysing refactoring dependencies using graph transformation. *Software and Systems Modeling*, 6(3):269–285, 2007.

[53] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, February 2004.

[54] Siegfried Nolte. *QVT - Relations Language: Modellierung mit der Query Views Transformation (Xpert.press) (German Edition)*. Springer, 2009.

[55] Ulrich Norbisrath, Albert Zündorf, and Ruben Jubeh. *Story Driven Modeling*. CreateSpace Independent Publishing Platform, 2013. ISBN-10: 1483949257.

[56] OMG. *MOF QVT Final Adopted Specification*. Object Modeling Group, June 2005.

[57] OMG. *Meta Object Facility (MOF) 2.0 Query/View/Transformation, v1.1.* OMG, January 2011.

[58] OMG. OMG Unified Modeling Language (OMG UML), Infrastructure, Version 2.4.1, August 2011.

[59] OMG. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1, August 2011.

[60] OMG. *Meta Object Facility (MOF) Version 2.4.2.* Object Modeling Group, April 2014.

[61] John L. Pfaltz and Azriel Rosenfeld. Web grammars. In *Int. Joint Confenence on Artificial Intelligence*, Seiten 609–619, 1969.

[62] Claudia Priesterjahn, Dominik Steenken, and Matthias Tichy. Timed hazard analysis of self-healing systems. In Rogério de Lemos Javier Camara, Carlo Ghezzi, and Antonia Lopes, editors, *Assurances for Self-Adaptive Systems*, Band 7740 aus *Lecture Notes in Computer Science*, Seiten 112–151. Springer Berlin Heidelberg, 2013.

[63] Louis M. Rose, Markus Herrmannsdoerfer, James R. Williams, Dimitris Kolovos, Kelly Garcés, Richard F. Paige, and Fiona A.C. Pollack. A comparison of model migration tools. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Proceedings 13th International Conference on Model Driven Engineering Languages and Systems (MODELS 2010), Part I*, LNCS 6394, Seiten 61–75, 2010.

[64] Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. Model migration with epsilon flock. In *Proceedings of the Third International Conference on Theory and Practice of Model Transformations*, ICMT'10, Seiten 184–198, Berlin, Heidelberg, 2010. Springer-Verlag.

[65] Stephan Roser and Bernhard Bauer. Automatic generation and evolution of model transformations using ontology engineering space. In Stefan Spaccapietra, Jeff Z. Pan, Philippe Thiran, Terry Halpin, Steffen Staab, Vojtech Svatek, Pavel Shvaiko, and John Roddick, editors, *Journal of Data Semantics XI*, Band 5383, Seiten 32–64. 2008.

[66] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation: Band I. Foundations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[67] Uwe Schöning. *Theoretische Informatik - kurzgefasst*. Spektrum Akademischer Verlag, 4. a. (korrig. nachdruck 2003) edition, 2003.

[68] Andreas Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemem*. Deutscher Universitäts Verlag, 1991.

[69] Andy Schürr and Bernhard Westfechtel. Graphgrammatiken und graphersetzungssysteme. Technical Report Nr. 92-15, RWTH Aachen Fachgruppe Informatik, Fachgruppe Informatik der RWTH, Ahornstr. 55, 52062 Aachen, 1992.

[70] Andy Schürr, Andreas J Winter, and Albert Zündorf. The progres approach: Language and environment. In *Handbook of graph grammars and computing by graph transformation*, Seiten 487–550. World Scientific Publishing Co., Inc., 1999.

[71] Herbert Stachowiak. *Allgemeine Modelltheorie*. 1973.

[72] Thomas Stahl, Markus Völter, Sven Efftinge, and Arno Haase. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. dpunkt, Heidelberg, 2 edition, 2007.

[73] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, Boston, MA, 2 edition, 2009.

[74] Mirko Stölzel, Steffen Zschaler, and Leif Geiger. Integrating OCL and model transformations in Fujaba. In Dan Chiorean, Birgit Demuth, Martin Gogolla, and Jos Warmer, editors, *Proceedings of the 6th OCL Workshop OCL for (Meta-)Models in Multiple Application Domains (OCLApps 2006)*, Band 5 aus *Electronic Communications of the EASST*, Genova, Italy, October 2006. 16 p.

[75] Massimo Tisi, Frédérix Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the use of higher-order model transformations. In Richard F. Paige, Alan Hartman, and Arend Rensink, editors, *ECMDA-FA 2009*, Band 5562, Seiten 18–33, 2009.

[76] Dániel Varró and Andras Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, 68(3):214–234, 2007.

[77] Dániel Varró and András Pataricza. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. *Journal of Software and Systems Modeling*, 2(3):187–210, October 2003.

[78] Sabine Winetzhammer. Modgraph - generating executable EMF models. *ECEASST*, 54, 2012.

[79] Sabine Winetzhammer, Joel Greenyer, and Matthias Tichy. Integrating graph transformations and modal sequence diagrams for specifying structurally dynamic reactive systems. In Gunther Mussbacher Daniel Amyot, Pau Fonseca i Casas, editor, *System Analysis and Modeling: Models and Reusability - 8th International Conference, SAM 2014, Valencia, Spain, September 29-30, 2014, Proceedings*, Band 8769 aus *Lecture Notes in Computer Science*, Seiten 126–141. Springer Berlin / Heidelberg, 2014.

[80] Sabine Winetzhammer and Bernhard Westfechtel. Modgraph meets xcore: Combining rule-based and procedural behavioral modeling for EMF. *ECEASST*, 58, 2013.

[81] Sabine Winetzhammer and Bernhard Westfechtel. Compiling graph transformation rules into a procedural language for behavioral modeling. In Luis Ferreira Pires, Slimane Hammoudi, Joaquim Filipe, and Rui C'esar das Neves, editors, *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2014)*, Seiten 415–424, Lisbon, Portugal, 2014. SCITEPRESS Science and Technology Publications, Portugal.

[82] Sabine Winetzhammer and Bernhard Westfechtel. Propagating model refactorings to graph transformation rules. In *ICSOFT-PT 2014 - Proceedings of the 9th International Conference on Software Paradigm Trends, Vienna, Austria, 29-31 August, 2014*, Seiten 17–28, 2014.

[83] Albert Zündorf. Rigorous object oriented software development. Technical report, University of Paderborn, Germany, 2001.

Abkürzungsverzeichnis

AMMA	Atlas Model Management Architecture
ATL	Atlas Transformation Language
DSL	domänenspezifische Sprache
EMF	Eclipse Modeling Framework
GEF	Graphical Editing Framework
eMOF	essential Meta Object Facility
GMF	Graphical Modeling Framework
MOF	Meta Object Facility
MWE	Modeling Workflow Engine
NAC	Negative Anwendbarkeitsbedingung (Negative Application Condition)
oAW	openAchitectureWare
OCL	Object Constraint Language
OMG	Object Management Group
QVT	Query View Transformation
UML	Unified Modeling Language
VTCL	Viatra Textual Command Language
VTML	Viatra Textual Metamodeling Language

Abbildungsverzeichnis

1.1 Der ModGraph-Ansatz: Ein Ansatz, um mit bestehenden und neuen Werkzeugen totale modellgetriebene Softwareentwicklung für EMF zu erreichen 6
1.2 Übersicht über das entstandene Werkzeug ModGraph 7

2.1 Klassifikation von Modellen in der modellgetriebenen Softwareentwicklung 15
2.2 Metaebenen der OMG (analog zu [72]) 16
2.3 Definition einer Sprache und deren Einordnung in die Modellierung . . . 18
2.4 Klassifikation von Modelltransformationen 22
2.5 Allgemeines Transformationsmuster der Modell-zu-Modell-Transformation (analog zu [45]) 23
2.6 Zwei Arten der Darstellung einer Graphtransformationsregel 28
2.7 Der klassische algebraische double-pushout Ansatz 30

3.1 EMF vereinigt UML, Java und XML (Zeichnung analog zu [73]) 35
3.2 Metaebenen in Ecore 36
3.3 Das Ecore-Metamodell (aus [1]) 37
3.4 Das Refactoring Ecore-Modell 39
3.5 Zusammenarbeit von EMF und GMF 43

5.1 Überblick über die Architektur des ModGraph-Ansatzes 72
5.2 Verwendete Modellierungskonzepte und deren Nutzung im ModGraph-Ansatz 73
5.3 M0-Anwendung (links) und M1-Anwendung (rechts) von Graphtransformationsregeln mit EMF 80
5.4 Schematischer Aufbau einer Graphtransformationsregel in ModGraph am Beispiel moveAttribute(EAttribute ma, EClass target) 82
5.5 Ecore-Modell eines einfachen Bugtrackers 87
5.6 ModGraph-Regel zur Erzeugung eines Tickets innerhalb eines Projekts im Bugtracker; Implementierung der Operation createTicket((severity:Severity, title:EString, description:EString, reporter:User):Ticket der Klasse Project . 89
5.7 ModGraph-Regel zur Erzeugung eines Projekts im Bugtracker; Implementierung der Operation createProject(name:EString, leader:User, firstGroup:Group):Project der Klasse BugTracker 91
5.8 ModGraph-Regel zur Änderung des Status eines Tickets im Bugtracker; Implementierung der Operation changeStatus(status:Status, author:User) der Klasse Ticket 92

5.9 Ausschnitt aus der alternativen Modellierung der ModGraph-Regel zur Implementierung der Operation changeStatus(status:Status, author:User) der Klasse Ticket . . . 92
5.10 ModGraph-Regel zur Zuweisung eines Tickets an einen Benutzer im Bugtracker; Implementierung der Operation reportedTickets():Ticket[0..*] der Klasse User . . . 93

6.1 Überblick - Das gesamte ModGraph Metamodell für Graphtransformationen 103
6.2 Ausschnitt aus dem ModGraph-Metamodell für Graphtransformationen: Die Hauptkomponenten einer hierarchisch aufgebauten ModGraph-Regel 104
6.3 Ausschnitt aus dem Metamodell für Graphtransformationen: Aufbau des Graphmusters . . . 105
6.4 Ausschnitt aus dem Metamodell für Graphtransformationen: Aufbau der negativen Anwendbarkeitsbedingung . . . 108
6.5 Abstrakte Syntax (links) und konkrete Syntax (rechts) auf einen Blick - Teil 1 . . . 113
6.6 Abstrakte Syntax (links) und konkrete Syntax (rechts) auf einen Blick - Teil 2 . . . 114
6.7 Ausführungslogik einer ModGraph-Regel . . . 117
6.8 Zur Erklärung der Mustersuche farbkodiertes Anwendungsbeispiel changeStatus aus dem Bugtacker-Beispiel . . . 119
6.9 Allgemeine Vorgehensweise bei der Mustersuche - Aufbau des Spannwaldes 120
6.10 Spannwald zum Anwendungsbeispiel . . . 121
6.11 Mustersuche in unterschiedlichen Modellinstanzen am Anwendungsbeispiel 122
6.12 Zustandsabhängige Berechnung der Attribute am Anwendungsbeispiel . . 123
6.13 Anwendung der in der Regel spezifizierten Änderungen auf Modellinstanz 1 des Anwendungsbeispiels . . . 125

7.1 Zusammenarbeit von Ecore und ModGraph . . . 128
7.2 Zusammenarbeit von Xcore und ModGraph . . . 129
7.3 Einbettung von Xcore zur Modellierung von Kontrollstrukturen . . . 130
7.4 Zusammenarbeit von Xcore, dessen generiertem Ecore-Modell und ModGraph . . . 130
7.5 Quelltext unabhängige Zusammenarbeit von Xcore und ModGraph . . . 131
7.6 Zusammenschau der möglichen Wege . . . 132
7.7 Schematische Darstellung des Aufbaus von EMF-generiertem Quelltext . 133
7.8 Schematische Darstellung der Quelltextgenerierung mit ModGraph und des Eingriffs in den EMF-generierten Quelltext durch den ModGraph-Generator . . . 136
7.9 Graphtransformationsregel zur Implementierung der Operation changeStatus der Klasse Ticket aus dem Bugtacker Beispiel . . . 137
7.10 Graphtransformationsregel zur Implementierung der Operation reportedTickets der Klasse Ticket aus dem Bugtacker Beispiel . . . 144

7.11 Graphtransformationsregel zur Implementierung der Operation createProject der Klasse BugTracker aus dem Beispiel 146
7.12 Schrittweiser Ansatz zur Generierung von Quelltext 148
7.13 Die ModGraph-Perspektive in Eclipse 157
7.14 Im Cheat Sheet (rechts) und ausgearbeitetes Beispiel (links) mit Ecore-Klassendiagramm (oben) und der Regel zur Implementierung der Operation aquireBook(...) (unten) 162

8.1 Entstehende Abhängigkeiten einer Regel bei M0-Anwendung 166
8.2 Refactoring am Beispiel Bugtracker: Extrahieren einer Oberklasse 167
8.3 Notwendige Anpassungen der Regeln am Beispiel Bugtracker: Umwandeln einer bidirektionalen Referenz in eine unidirektionale 168
8.4 Grundlegende Idee des propagierenden Refactorings 170
8.5 Zusammenhang der zum propagierenden Refactoring genutzten Modelle und Regeln 177
8.6 ModGraph-Regeln zur Propagation des Ecore-Modell-Refactorings zum Umwandeln einer bidirektionalen Referenz in eine unidirektionale auf betroffene ModGraph-Regeln 180
8.7 Anwendung des propagierenden Refactorings auf das Bugtracker Beispiel: Umwandlung der bidirektionalen Referenz zwischen Project und Ticket in eine unidirektionale 181
8.8 ModGraph-Regel zum Extrahieren einer Klasse 183
8.9 ModGraph-Regel zur Propagation der strukturellen Anteile des Refactorings Extrahieren einer Klasse 185
8.10 Anwendung des propagierenden Refactorings auf das Bugtracker Beispiel: Extraktion der Klasse Names zur Verwaltung der Namen des Nutzers . . 186

9.1 Beispiel eines struktur-dynamischen, verteilten, reaktiven Systems: Ein Autonomes Transportsystem 188
9.2 Klassendiagramm und beispielhaftes Objektsystem zum autonomen Transportsystem 189
9.3 Modale Sequenzdiagramme zur Bestellung einer Ware 190
9.4 Integration von Graphtransformationsregeln und modalen Sequenzdiagrammen am Beispiel - Teil 1 195
9.5 Integration von Graphtransformationsregeln und modalen Sequenzdiagrammen am Beispiel - Teil 2 196
9.6 Ablauf des Play-outs nach der Integration 199
9.7 Zusammenarbeit von ModGraph und ScenarioTools 200

10.1 Regel zur Implementierung der Operation createTicket(...) des Bugtracker-Beispiels 208
10.2 Diagramme aus den Rohdaten der Laufzeitmessungen für das Refactoring (oben) und die Propagation (unten) bei der Verschiebung des Attributs . 209

Tabellenverzeichnis

2.1 Vergleich der theoretischen Ansätze zur Graphtransformation 31

8.1 Modelländerungen und deren Effekte auf die Regel 171

10.1 Vorkommen der Hauptkomponenten einer Regel 202
10.2 Vorkommen der Elemente im Graphmuster 202
10.3 Vorkommen der gebundenen Knoten im Graphmuster 203
10.4 Vorkommen der ungebundenen Knoten im Graphmuster (angegeben als gesamt (optional)) 203
10.5 Durchschnittliches Vorkommen der Elemente in der NAC 204
10.6 Durchschnittliches Vorkommen von Elementen in den Knoten 204
10.7 Vergleich der Ergebnisse mit [20] 206
10.8 Durchschnittliche Laufzeiten für das Verschieben eines Attributs 210
10.9 Durchschnittliche Laufzeiten für das Verschieben einer Operation 210
10.10 Durchschnittliche Laufzeiten für das Umwandeln einer bidirektionalen in eine unidirektionale Referenz 211
10.11 Graphtransformationssprachen und Werkzeuge 218
10.12 Ausführbarkeit von Graphtransformationssprachen 218

A.1 Durchschnittliche Laufzeiten für das Verstecken des Delegierten 249
A.2 Durchschnittliche Laufzeiten für das Entfernen der mittleren Klasse . . . 249
A.3 Durchschnittliche Laufzeiten für das Ersetzen der Delegation durch Vererbung 249
A.4 Durchschnittliche Laufzeiten für das Ersetzen der Vererbung durch Delegation 249

Auflistungsverzeichnis

4.1 Ausschnitt aus der Xcore Grammatik 46
4.2 Ausschnitt aus dem erRefactoring Xcore-Modell 47
4.3 Einfache Xbase Implementierung der Methode addEParameter 47

5.1 EMFs Lücke: Verhalten muss in reinem EMF in Java beschrieben werden; Ausschnitt aus dem EMF-generierten Quelltext für die modellierte Klasse Project des Bugtracker Ecore-Modells 88
5.2 Java Implementierung der Methode `getNextNumber()` der Klasse BugTracker 94
5.3 Ausschnitt aus dem generierten Xcore-Modell der Bugtrackers: Klassen Group und BugTracker mit Ergänzung um die Methoden zur Berechnung der laufenden Nummer für Tickets 95
5.4 Xcore als Kontrollflussprache zur korrekten Kombination von Projekt und Ticketerstellung 97

7.1 Ausschnitt aus dem EMF-generierten Quelltext für die modellierte Klasse Ticket des Bugtracker Ecore-Modells: generierter Quelltext zum modellierten Attribut number:EInt und zur modellierten Referenz currentRevision 135
7.2 Ausschnitt aus dem EMF-generierten Quelltext für die modellierte Klasse Ticket des Bugtracker Ecore-Modells: generierter Quelltext zur modellierten Operation changeStatus 136
7.3 Ausschnitt aus dem kombiniert generierten Quelltext für die modellierte Klasse Ticket des Bugtracker Ecore-Modells: Anpassung der Methodendeklaration im generierten Interface zur modellierten Operation changeStatus (Abbildung 7.9) 137
7.4 Ausschnitt aus dem kombiniert generierten Quelltext für die modellierte Klasse Ticket des Bugtracker Ecore-Modells: Implementierung der modellierten Operation changeStatus (Abbildung 7.9) 138
7.5 ModGraphs Methoden zum Löschen von Objekten aus ihrem gesamten Kontext 140
7.6 ModGraph-generierter Quelltext der Utility-Klasse zur Unterstützung der Implementierung der modellierten Operation changeStatus der Klasse Ticket des Bugtracker Ecore-Modells (Abbildung 7.9) 142
7.7 Utility-Methode zur Suche eines ungebundenen Knotens mit Hilfe eines Pfadausdrucks und anschließendem Filtern des Ergebnisses am Beispiel der Regel reportedTickets der Klasse User (Abbildung 7.10) 145

7.8 Utility-Methode zur Suche eines Graphmusters, welches mit einer NAC eingeschränkt ist, am Beispiel der Regel createProject (Abbildung 7.11) . 147
7.9 Definition der Annotationen im Xcore-Modell 149
7.10 Ausschnitt aus dem Bugtracker Xcore-Modell: Die generierte Methode zur Regel changeStatus (Abbildung 7.9) 150
7.11 Ausschnitt aus dem Bugtracker Xcore-Modell: Die generierten Methoden zu den Regeln reportedTickets (Abbildung 7.10) und createProject (Abbildung 7.11) 152
7.12 Von Xcore generierter Java-Quelltext zur Implementierung der modellierten Operation createProject (Abbildung 7.11) der Klasse BugTracker des Bugtracker-Xcore-Modells 153

8.1 Xcore Implementierung des Refactorings zur Umwandlung einer bidirektionalen in eine unidirektionale Referenz 178
8.2 Xcore Kontrollfluss: Steuerung der Regeln zur Propagation des Refactorings zur Umwandlung einer bidirektionalen in eine unidirektionale Referenz 179
8.3 Xcore Implementierung der Propagation zum Refactoring Extrahieren einer Klasse 184

A.1 Auszug aus den in ModGraph genutzten Bedingungen, die in der Sprache Check definiert sind 242
A.2 Auszug aus den in ModGraph zur Java Generierung genutzten Templates 243
A.3 Das Bugtracker Xcore Modell 245
A.4 Die Xtext Grammatik zu Xcore 250
A.5 Die XText Grammatik zu Xbase 253

A Anhang

A.1 Metamodell für Graphtransformationsregeln als Baum

- platform:/resource/de.ubt.ai1.modgraph.gt/model/GT.ecore
  - gtEngine
    - GraphTransformationRule<EXOperation extends EObject, EXClass extends EObject>
      - EXOperation extends EObject
      - EXClass extends EObject
      - implementedOperation : EXOperation
      - correspondingClass : EXClass
      - thrownException : GTFailure
      - conditions : GTCondition
      - negativePattern : GTNegativeApplicationCondition
      - pattern : GTGraphPattern
      - comment : GTComment
      - name : EString
    - GPElementWithStatus -> GPElement
    - GTNode
    - GTEdge
    - NACParameter<EXClass extends EObject> -> NACNode
    - NACObject<EXClass extends EObject> -> NACNode
    - GTStatus
    - GTAttribute<EXAttribute extends EObject>
    - GTOperator
    - GTField
    - GTFieldKind
    - GTFailure [de.ubt.ai1.modgraph.gt.failure.GTFailure]
    - GTNegativeApplicationCondition
    - GTCondition
    - GTPrecondition -> GTCondition
    - GTPostcondition -> GTCondition
    - GTPath -> GTEdge, NACElement, GPElement
    - GTGraphPattern
    - GTLink<EXReference extends EObject> -> GTEdge
    - NACElement
    - GPNode -> GPElementWithStatus, GTNode
    - GPElement
    - GPLink<EXReference extends EObject> -> GTLink<EXReference>, GPElementWithStatus
    - GPUnboundNode<EXClass extends EObject> -> GPNode
    - GPObject<EXClass extends EObject> -> GPUnboundNode<EXClass>
    - GPMultiObject<EXClass extends EObject> -> GPUnboundNode<EXClass>
    - GPBoundNode<EXClass extends EObject> -> GPNode
    - GTThisObject<EXClass extends EObject> -> GPElement, GTNode, NACNode
    - NACNode -> NACElement, GTNode
    - NACLink<EXReference extends EObject> -> GTLink<EXReference>, NACElement
    - PathType
    - NACBoundNode<EXClass extends EObject> -> NACNode
    - GPOrderedLink<EXReference extends EObject> -> GPLink<EXReference>
    - GTComment
    - GPParameter<EXClass extends EObject> -> GPBoundNode<EXClass>
    - GPMultiParameter<EXClass extends EObject> -> GPBoundNode<EXClass>
    - ConditionKind
- platform:/resource/org.eclipse.emf.ecore/model/Ecore.ecore

A.2 Nutzung der Validierungssprache Check in ModGraph

Check ist eine Sprache zur Definition von Constraints und dient zur Validierung von Modellen. Sie wird in ModGraph genutzt um die kontextsensitive Syntax zu definieren und damit um die Graphtransformationsregeln zu validieren. Ein Ausschnitt aus den definieren Checks ist in folgender Auflistung gezeigt:

```
/* Author: S. Winetzhammer */
import ecore;
import gtEngine;

extension de::ubt::ai1::modgraph::validation::checks::Extensions;
extension de::ubt::ai1::modgraph::validation::checks::GlobalVar;

// Jede Regel muss die Operation referenzieren, die sie implementiert.
context gtEngine::GraphTransformationRule ERROR
    "The rule must implement an EOperation!":
    this.implementedOperation.metaType == EOperation;
...
// Im Graphmuster muss mind. ein gebundener Knoten existieren.(Analog für die NAC)
context  gtEngine::GTGraphPattern ERROR
    "Bound element missing in Graphpattern":
    existsBoundElem(this.elements.typeSelect(GPNode)
        .select(e| e.status.toString() != "CREATE")
            .addAll(elements.typeSelect(GTThisObject)));
...
//Objekte sind in  genau einem Container. (Analog für die NAC)
context gtEngine::GPObject ERROR
    "Please check containment of " + name + ". ":
    if(incomingEdges.size>=1) then
        noSecondContainment(ecoreModel().eAllContents
            .typeSelect(ecore::EReference).select(e|e.containment == true).name,
                incomingEdges.typeSelect(gtEngine::GTLink))
    else true;
...
// Pfadausdruck darf nicht leer sein.
context gtEngine::GTPath ERROR
"Each path needs a pathexpression":
this.expression != null && this.expression !="";
...
//GPLink muss eine Status haben
context gtEngine::GPLink ERROR
    "Each link in a graphpattern needs a status":
        this.status != null;
...
//Syntaxanalyse der Vorbedingung  (Nachbedingung analog)
context GTPrecondition if(kind == ConditionKind::OCL)ERROR
    "Condition content is no valid OCL-expression!":
        parseOCL("pre: " + this.condition.toString());
```

Auflistung A.1: *Auszug aus den in ModGraph genutzten Bedingungen, die in der Sprache Check definiert sind*

A.3 Nutzung der Templatesprache Xpand in ModGraph

XPand Templates werden in ModGraph zur Generierung von Java- und Xcore-Code verwendet. Die folgende Auflistung zeigt Auszüge aus den zur Java-Generierung genutzten Templates. Die Xcore Templates sind analog aufgebaut.

```
1  << IMPORT gtEngine>>
2  << IMPORT ecore>>
3
4  << EXTENSION de::ubt::ai1::modgraph::generator::extensions::GlobalVar>>
5  << EXTENSION de::ubt::ai1::modgraph::generator::extensions::Extensions>>
6
7  << REM >> An Konten und Kanten spezifizierten Änderungen << ENDREM >>
8  << DEFINE gPManager FOR gtEngine::GTGraphPattern>>
9      << REM >>zu löschende Knoten<< ENDREM >>
10     << EXPAND ModifyNodes::deleteNode FOREACH
11     elements.typeSelect(GPNode).select(o|o.status.toString() == "DELETE")->>
12     << REM >> zu löschende Kanten zwischen zu erhaltenden Knoten << ENDREM >>
13     << EXPAND ModifyEdges::deleteEdge FOREACH
14         elements.typeSelect(GPLink).select(e|e.status.toString() ==  "DELETE" &&
15             (e.sourceNode.metaType == GTThisObject
16              ||((GPNode)e.sourceNode).status.toString()=="PRESERVE") &&
17             (e.targetNode.metaType == GTThisObject
18              ||((GPNode)e.targetNode).status.toString()=="PRESERVE") &&
19             !(((EReference)load(e.exReference)).upperBound ==
20              ((EReference)load(e.exReference)).lowerBound == 1))->>
21     << REM >>zu erzeugende Knoten << ENDREM >>
22     << EXPAND ModifyNodes::createNode FOREACH
23         elements.typeSelect(GPObject).select(o|o.status.toString() == "CREATE" )>>
24     << REM >>zu erzeugende Links << ENDREM >>
25     << EXPAND createEdge FOREACH
26         elements.typeSelect(GPLink) .select(e|e.status.toString() ==  "CREATE")
27             .addAll(elements.typeSelect(GPOrderedLink))->>
28 << ENDDEFINE >>
29
30 << REM >> Erzeugen eines Links<< ENDREM >>
31 << DEFINE createEdge FOR gtEngine::GPLink>>
32 << IF ((GPNode)this.targetNode).optional == true &&
33     ((EReference)load(this.exReference)).upperBound!=1>>try{ << ENDIF >>
34 << IF this.metaType == GPOrderedLink &&
35         ((EReference)load(this.exReference)).upperBound != 1>>
36     << IF ((GPOrderedLink)this).position == "last"
37         ||((GPOrderedLink)this).position ==""
38             || ((GPOrderedLink)this).position == null >>
39         << EXPAND normalLink>>
40     << ELSEIF ((GPOrderedLink)this).position == "first">>
41         << EXPAND createRefN(((EReference)load(this.exReference)),0)>>
42     << ELSE >>
43         << LET ((EReference)load(exReference)) AS modelRef>>
44                 EList<< <modelRef.eType.name>>> << modelRef.name>> =
45                 << sourceNode.name>>.get<< modelRef.name.toFirstUpper()>>();
46         << IF ((GPOrderedLink)this).position.split(" ").get(0) == "after">>
47         int pos = << modelRef.name>>.indexOf(<<modelRef.eType.name
48             .toFirstLower()>>)+1;
49         << ELSE >>
50         int pos = << modelRef.name>>.indexOf(<<modelRef.eType.name
51             .toFirstLower()>>)-1; << ENDIF >>
52         << IF targetNode.metaType.name == "gtEngine::GPMultiObject"
53             || targetNode.metaType.name == "gtEngine::GPMultiParameter">>
54         << sourceNode.name>>.get<< modelRef.name.toFirstUpper()>>()
55             .addAll(pos, << targetNode.name>>);
56     << ELSEIF sourceNode.metaType.name == "gtEngine::GPMultiObject"
57         || sourceNode.metaType.name == "gtEngine::GPMultiParameter">>
58     for(<< ((ENamedElement)((EReference)load(this.exReference)).eContainer).name>>
```

```
59      << ((ENamedElement)((EReference)load(this.exReference))
60          .eContainer).name.toFirstLower()>> :
61              << sourceNode.name>>){
62          << ((ENamedElement)((EReference)load(this.exReference)).eContainer).name
63              .toFirstLower()>>.set<< ((ENamedElement)load(this.exReference)).name
64                  .toFirstUpper()>>(pos, << targetNode.name>>);
65      } << ELSE >> << sourceNode.name>>.get<< modelRef.name.toFirstUpper()>>()
66              .add(pos, << targetNode.name>>);
67      << ENDIF >> << ENDLET >><< ENDIF >>
68  << ELSE >>  << EXPAND normalLink>> << ENDIF >>
69  << IF ((GPNode)this.targetNode).optional == true &&
70      ((EReference)load(this.exReference)).upperBound==-1>>}
71      catch(Exception e){}
72  << ENDIF>>
73  << ENDDEFINE >>
74
75  << DEFINE normalLink FOR GPLink>>
76  << LET ((EReference)load(exReference)) AS modelRef>>
77  << IF (modelRef.lowerBound==0 || modelRef.lowerBound==1) && modelRef.upperBound==1>>
78  << EXPAND createRef01(modelRef) FOR this>>
79  << ELSEIF modelRef.upperBound==-1>>
80  << EXPAND createRefN(modelRef) FOR this>>
81  << ENDIF >> << ENDLET >>
82  << ENDDEFINE >>
83
84  << DEFINE createRef01(ecore::EReference modelRef) FOR gtEngine::GPLink>>
85      << IF sourceNode.metaType != gtEngine::GPMultiObject>>
86          << sourceNode.name>>.set<< modelRef.name.toFirstUpper()>>
87              (<< targetNode.name>>);
88      << ELSE >> << EXPAND multiCreate>> << ENDIF >>
89  << ENDDEFINE >>
90
91  << DEFINE createRefN(ecore::EReference modelRef) FOR gtEngine::GPLink>>
92      << IF targetNode.metaType.name == "gtEngine::GPMultiObject"
93              || targetNode.metaType.name == "gtEngine::GPMultiParameter">>
94          << sourceNode.name>>.get<< modelRef.name.toFirstUpper()>>()
95              .addAll(<< targetNode.name>>);
96      << ELSEIF (sourceNode.metaType.name == "gtEngine::GPMultiObject"
97          || sourceNode.metaType.name == "gtEngine::GPMultiParameter") &&
98              ((ecore::EReference)load(modelRef)).eOpposite !=null>>
99          << targetNode.name>>.get<< modelRef.eOpposite.name.toFirstUpper()>>()
100             .addAll(<< sourceNode.name>>);
101     << ELSEIF (sourceNode.metaType.name == "gtEngine::GPMultiObject"
102         || sourceNode.metaType.name == "gtEngine::GPMultiParameter") &&
103             ((ecore::EReference)load(modelRef)).eOpposite == null>>
104         << EXPAND multiCreate>>
105     << ELSE >>
106         << sourceNode.name>>.get<< modelRef.name.toFirstUpper()>>()
107             .add(<< targetNode.name>>);
108     << ENDIF >>
109 << ENDDEFINE >>
```

Auflistung A.2: *Auszug aus den in ModGraph zur Java Generierung genutzten Templates*

A.4 Vollständiges Bugtracker Xcore-Modell

```
package de.ubt.ai1.modgraph.bugmodel

import org.eclipse.emf.ecore.EBoolean
import org.eclipse.emf.ecore.EInt
import org.eclipse.emf.ecore.EString
import org.eclipse.emf.common.util.EList
import de.ubt.ai1.modgraph.gt.ecoreFromXcore.generator.ocl.OCLHandler

annotation "http://www.eclipse.org/emf/2002/Ecore/OCL/Pivot" as OCL
annotation "http://www.eclipse.org/emf/2002/Ecore" as Ecore
annotation "http://www.eclipse.org/emf/2002/GenModel" as GenModel

@Ecore(invocationDelegates="http://www.eclipse.org/emf/2002/Ecore/OCL/Pivot",
settingDelegates="http://www.eclipse.org/emf/2002/Ecore/OCL/Pivot",
validationDelegates="http://www.eclipse.org/emf/2002/Ecore/OCL/Pivot",
nsPrefix="de.ubt.ai1.modGraph",
nsURI="http://de/ubt/ai1/modgraph/BugModel.ecore")
@GenModel(loadInitialization="true",operationReflection="true",
modelDirectory="/Bugtracker/src", modelName="BugModel",
importerID="org.eclipse.emf.importer.ecore", complianceLevel="6.0")
    class User {
        String username
        String name
        String surname
        String email
        refers Group[+] memberIn opposite members

        op unique Project[] relatedProjects()
        @GenModel(documentation="Generated by ModGraph: ")
        op unique Ticket[] reportedTickets() {
            var Ticket[] tickets = null
            //the Element is ONLY reachable via a path!
            var _tickets = OCLHandler.
                evaluatePath(this, "relatedProjects().reportedTickets") as Ticket[]
            for (ticket : _tickets) {
            if (ticket.reporter == this)
                tickets.add(ticket)
            }
            if(tickets.isEmpty()) throw new de.ubt.ai1.modgraph.gt.failure.GTFailure
                return tickets as EList
        }
        op unique Ticket[] assignedTickets()
        op unique Project[] accessibleProjects() throws GTFailure
        op unique Project[] leaderAt() throws GTFailure
    }

    class Group {
        String name
        String description
        refers User[] members opposite memberIn
        refers Project[] accessibleProjects opposite accessingGroups

        op EBoolean isActive() throws GTFailure
    }

    class Project {
        String description
        String name
        contains Ticket[] reportedTickets opposite owningProject
        refers User[1] leader
        refers Group[] accessingGroups opposite accessibleProjects

        op unique Ticket[] allTicketsWithStatus(Status status)
```

```
64          op unique Ticket[] allTicketsWithSeverity(Severity severity)
65          op unique Ticket[] allBugsWithStatusAndSeverity(Status status ,
66                  Severity severity) throws GTFailure
67      @GenModel(documentation="Generated by ModGraph: ")
68      @OCL(pre_pre1 = "reporter.accessibleProjects()->includes(self)")
69      @OCL(pre_pre2 = "title.size() > 0 and description.size() > 0")
70      @OCL(post_post1 = "reportedTickets->isUnique(t : Ticket | t.number)")
71      op Ticket createTicket(Severity severity , EString title ,
72              EString description , User reporter) throws GTFailure{
73          val ticketNumberValue = ((eContainer().eContainer())
74                  as BugTracker).getNextNumber()
75          val ticketCreatedAtValue = java.util.Calendar.getInstance()
76          val revisionTitleValue = title
77          val revisionDescriptionValue = description
78          val revisionSeverityValue = severity
79          val revisionStatusValue = Status.NEW
80          val revisionCreatedAtValue = java.util.Calendar.getInstance()
81          var  ticket = BugmodelFactory::eINSTANCE.createTicket()
82          ticket.number = ticketNumberValue
83          ticket.createdAt = ticketCreatedAtValue
84          var  revision = BugmodelFactory::eINSTANCE.createTicketRevision()
85          revision.title = revisionTitleValue
86          revision.description = revisionDescriptionValue
87          revision.severity = revisionSeverityValue
88          revision.status = revisionStatusValue
89          revision.createdAt = revisionCreatedAtValue
90          reportedTickets.add(ticket)
91          ticket.reporter = reporter
92          revision.creator = reporter
93          ticket.history.add(revision)
94          ticket.currentRevision = revision
95          return  ticket
96      }
97          op boolean hasOpenTickets() throws GTFailure
98      }
99
100     class TicketRevision {
101         String title
102         String description
103         Severity severity
104         Status status
105         Calendar createdAt
106         refers User[1] creator
107         refers User assignee
108
109         op EBoolean copyAssignee(TicketRevision oldRevision) throws GTFailure
110     }
111
112     class BugTracker {
113         contains User[] users
114         contains Project[] trackedProjects
115         contains Group[] groups
116
117         op User createUser(String nickname , String surname ,
118                 String name , String email , Group group) throws GTFailure
119         op Group createGroup(String name) throws GTFailure
120         @GenModel(documentation="Generated by ModGraph: ")
121         op Project createProject(String name , User leader ,
122                 Group firstGroup) throws GTFailure {
123             if (firstGroup.members.findFirst(m|m == leader)!=null) {
124                 if (!(trackedProjects.findFirst(e|e.name == name) != null)) {
125                     val projectNameValue = name
126                     var project = BugmodelFactory::eINSTANCE.createProject()
127                     project.name = projectNameValue
128                     trackedProjects.add(project)
129                     project.leader = leader
```

```
130                     firstGroup.accessibleProjects.add(project)
131                     return project
132                 } else
133                     throw  new de.ubt.ai1.modgraph.gt.failure.GTFailure
134             } else
135                 throw new de.ubt.ai1.modgraph.gt.failure.GTFailure
136         }
137         op User loginUser(String nickname) throws GTFailure
138         op unique Ticket[] allOpenTickets() throws GTFailure
139         op unique Project[] allCleanProjects() throws GTFailure
140         op unique Project[] allBrokenProjects() throws GTFailure
141
142     //ergänzter prozeduraler Quelltext
143         int runningNo
144         op void init() throws GTFailure {
145             runningNo = 0
146             var group = createGroup("inititators")
147             var user = createUser("firstUser", "someone", "", "", group)
148             createProject("Ein Projekt", user, group)
149         }
150         op int getNextNumber() {
151             runningNo = runningNo + 1
152             return runningNo
153         }
154         op boolean createProjectAndTickets(String projectName ,
155                 User projectLeaderAndReporter,  Group projectsfirstGroup,
156                     Severity[] ticketSeverity,  EString[] ticketTitle,
157                         EString[] ticketDescription) {
158             var Project project = null
159             try {
160                 project = createProject(projectName,
161                         projectLeaderAndReporter, projectsfirstGroup)
162             } catch (de.ubt.ai1.modgraph.gt.failure.GTFailure failure) {
163                 project = this.trackedProjects
164                         .findFirst[Project p|p.name == projectName]
165                 if (project == null)
166                     throw new de.ubt.ai1.modgraph.gt.failure
167                         .GTFailure("Object not found")
168                 else {
169                     /*Notify user, that a project with the given name already exists
170                     * and will be used if he doesn't cancel the operation */
171                 }
172             }
173             var int i = 0
174             if (ticketSeverity.size == ticketTitle.size == ticketDescription.size) {
175             try{
176                 while (i < ticketSeverity.size) {
177                     project.createTicket(ticketSeverity.get(i), ticketTitle.get(i),
178                         ticketDescription.get(i),   projectLeaderAndReporter)
179                     i = i + 1
180                 }
181                 }catch (de.ubt.ai1.modgraph.gt.failure.GTFailure failure) {
182                     throw new de.ubt.ai1.modgraph.gt.failure
183                         .GTFailure("Ticket creation failed")
184                 }
185             }
186             return true
187     }
188
189     class Ticket {
190         EInt number
191         Calendar createdAt
192         contains TicketRevision[+] history
193         refers TicketRevision[1] currentRevision
194         refers User[1] reporter
195         contains Comment[] comments
```

```
196         container Project[1] owningProject opposite reportedTickets
197
198         op Comment addComment(EString content , EString title , User author)
199         op void changeTicket(Severity severity , EString title ,
200                 EString description , User user)
201         op void assignTo(User assignee , User author)
202         op User getAssignee() throws GTFailure
203         @GenModel(documentation=
204                 "Generated by ModGraph: Ändert  den Status eines Tickets.")
205         @OCL(pre_pre1="author.accessibleProjects()->includes(self.owningProject)")
206         op void changeStatus(Status status , User author) throws GTFailure {
207             var TicketRevision oldRevision = null
208             var User assignee = null
209             var _oldRevision = currentRevision
210             var _assignee = oldRevision.assignee
211             oldRevision = _oldRevision
212             assignee = _assignee
213             if(oldRevision == null)
214                 throw new de.ubt.ai1.modgraph.gt.failure
215                         .GTFailure("Object not found")
216             val newRevisionTitleValue = currentRevision.title
217             val newRevisionDescriptionValue = currentRevision.description
218             val newRevisionSeverityValue = currentRevision.severity
219             val newRevisionStatusValue = status
220             val newRevisionCreatedAtValue = java.util.Calendar.getInstance()
221             var newRevision = BugmodelFactory::eINSTANCE.createTicketRevision()
222             newRevision.title = newRevisionTitleValue
223             newRevision.description = newRevisionDescriptionValue
224             newRevision.severity = newRevisionSeverityValue
225             newRevision.status = newRevisionStatusValue
226             newRevision.createdAt = newRevisionCreatedAtValue
227             currentRevision = newRevision
228             newRevision.creator = author
229             history.add(newRevision)
230             newRevision.assignee = assignee
231         }
232     }
233
234     class Comment {
235         String title
236         String content
237         Calendar createdAt
238         refers User[1] author
239     }
240
241     enum Severity {
242         LOW
243         NORMAL = 1
244         HIGH = 2
245     }
246     enum Status {
247         NEW
248         CONFIRMED = 1
249         REJECTED = 2
250         DONE = 3
251         FIXED = 4
252         IN_PROGRESS = 5
253     }
254
255 type Calendar wraps java.util.Calendar
256 type GTFailure wraps de.ubt.ai1.modgraph.gt.failure.GTFailure
257 type OCLHandler wraps de.ubt.ai1.modgraph.gt.ecoreFromXcore.generator.ocl.OCLHandler
```

Auflistung A.3: *Das Bugtracker Xcore Modell*

A.5 Laufzeittest

	Refactoring (ms)	Propagation (ms)	Gesamtlaufzeit (ms)
Modelliert	5,73	46,09	51,82
Java	5,28	22,78	28,06

Tabelle A.1: *Durchschnittliche Laufzeiten für das Verstecken des Delegierten*

	Refactoring (ms)	Propagation (ms)	Gesamtlaufzeit (ms)
Modelliert	5,80	33,57	39,37
Java	5,63	22,87	28,50

Tabelle A.2: *Durchschnittliche Laufzeiten für das Entfernen der mittleren Klasse*

	Refactoring (ms)	Propagation (ms)	Gesamtlaufzeit (ms)
Modelliert	7,71	44,06	51,77
Java	7,37	24,00	31,37

Tabelle A.3: *Durchschnittliche Laufzeiten für das Ersetzen der Delegation durch Vererbung*

	Refactoring (ms)	Propagation (ms)	Gesamtlaufzeit (ms)
Modelliert	6,17	23,44	29,61
Java	5,91	23,49	29,40

Tabelle A.4: *Durchschnittliche Laufzeiten für das Ersetzen der Vererbung durch Delegation*

A.6 Xtext-Grammatiken

A.6.1 Die Xcore-Grammatik

```
grammar org.eclipse.emf.ecore.xcore.Xcore
with org.eclipse.xtext.xbase.Xbase
import "http://www.eclipse.org/emf/2002/Ecore"
import "http://www.eclipse.org/emf/2002/GenModel" as genmodel
import "http://www.eclipse.org/xtext/xbase/Xbase" as xbase
import "http://www.eclipse.org/xtext/common/JavaVMTypes" as javaVMTypes
import "http://www.eclipse.org/xtext/xbase/Xtype" as xtype
import "http://www.eclipse.org/emf/2011/Xcore"
XPackage returns XPackage:
{XPackage}
(annotations+=XAnnotation)*
'package'
name = QualifiedName
(importDirectives += XImportDirective)*
(annotationDirectives += XAnnotationDirective)*
(classifiers += XClassifier)*
;
XAnnotation:
'@' source=[XAnnotationDirective|ValidID]
('(' details+=XStringToStringMapEntry (','
details+=XStringToStringMapEntry)* ')')?
;
XStringToStringMapEntry:
key=QualifiedName
'='
value=STRING
;
XImportDirective:
'import' (importedNamespace=QualifiedNameWithWildcard |
importedObject=[EObject|QualifiedName])
;
QualifiedNameWithWildcard:
QualifiedName '.*'
;
XAnnotationDirective:
'annotation' sourceURI=STRING 'as' name=ValidID
;
XClassifier:
XClass |
XDataType |
XEnum
;
XDataType:
(annotations+=XAnnotation)*
'type' name = ID
('<' typeParameters+=XTypeParameter (',' typeParameters+=XTypeParameter)* '>')?
'wraps' instanceType=JvmTypeReference
(
/*In scope for create should be what's visible in XyzFactoryImpl
* and 'this' will denote the literal value.
* The block expression must yield null or an instance of the wrapped type.
*/
(serializable?='create' createBody=XBlockExpression)? &
/*In scope for create should be what's visible in XyzFactoryImpl
* and 'this' will denote an instance of the wrapped type.
* The block expression must yield a java.lang.String.
*/
('convert' convertBody=XBlockExpression)?
)
;
```

```
61  XEnum:
62  (annotations+=XAnnotation)*
63  'enum' name = ID
64  '{'
65  (literals+=XEnumLiteral ((',')? literals+=XEnumLiteral)*)?
66  '}'
67  ;
68  XEnumLiteral:
69  (annotations+=XAnnotation)*
70  name=ID
71  ('as' literal=STRING)?
72  ('=' value=INT)?
73  ;
74  XClass:
75  {XClass}
76  (annotations+=XAnnotation)*
77  ((abstract?='abstract'? 'class') | interface?= 'interface') name = ID
78  ('<' typeParameters+=XTypeParameter
79  (',' typeParameters+=XTypeParameter)* '>')?
80  ('extends' superTypes+=XGenericType (',' superTypes+=XGenericType)*)?
81  ('wraps' instanceType=JvmTypeReference) ?
82  '{'
83  (members+=XMember)*
84  '}'
85  ;
86  XMember:
87  XOperation |
88  XReference |
89  XAttribute
90  ;
91  XAttribute:
92  (annotations+=XAnnotation)*
93  (
94  (unordered?='unordered')? &
95  (unique?='unique')? &
96  (readonly?='readonly')? &
97  (transient?='transient')? &
98  (volatile?='volatile')? &
99  (unsettable?='unsettable')? &
100 (derived?='derived')? &
101 (iD?='id')?
102 )
103 (type=XGenericType multiplicity=XMultiplicity? | 'void')
104 name=ID
105 ('=' defaultValueLiteral=STRING)?
106 /*
107 * In scope for getBody should be what's visible in AbcImpl
108 * and 'this' will denote an instance of the feature's type.
109 * The block expression must yield a value of the feature's type.
110 */
111 (('get' getBody=XBlockExpression)? &
112 ('set' setBody=XBlockExpression)? &
113 ('isSet' isSetBody=XBlockExpression)? &
114 ('unset' unsetBody=XBlockExpression)?)
115 ;
116 XReference:
117 (annotations+=XAnnotation)*
118 ((resolveProxies?='resolving'? & (containment?='contains' |
119 container?='container')) | (local?='local'? & 'refers'))
120 (
121 (unordered?='unordered')? &
122 (unique?='unique')? &
123 (readonly?='readonly')? &
124 (transient?='transient')? &
125 (volatile?='volatile')? &
126 (unsettable?='unsettable')? &
```

```
127 (derived?='derived')?
128 )
129 type=XGenericType
130 multiplicity=XMultiplicity?
131 name=ID
132 (
133 'opposite' opposite=[genmodel::GenFeature|ValidID]
134 )?
135 (
136 'keys' keys+=[genmodel::GenFeature|ValidID]
137 (',' keys+=[genmodel::GenFeature|ValidID])*
138 )?
139 /*
140 * In scope for getBody should be what's visible in AbcImpl
141 * and 'this' will denote an instance of the feature's type.
142 * The block expression must yield a value of the feature's type.
143 */
144 (('get' getBody=XBlockExpression)? &
145 ('set' setBody=XBlockExpression)? &
146 ('isSet' isSetBody=XBlockExpression)? &
147 ('unset' unsetBody=XBlockExpression)?)
148 ;
149 XOperation:
150 (annotations+=XAnnotation)*
151 'op'
152 (
153 unordered?='unordered' unique?='unique'? |
154 unique?='unique' unordered?='unordered'?
155 )?
156 ('<' typeParameters+=XTypeParameter
157 (',' typeParameters+=XTypeParameter)* '>')?
158 (type=XGenericType | 'void')
159 multiplicity=XMultiplicity?
160 name=ID
161 '(' (parameters+=XParameter (',' parameters+=XParameter)*)? ')'
162 ('throws' exceptions+=XGenericType (',' exceptions+=XGenericType)*)?
163 /*
164 * This is the logic for the operation.
165 * How are we going to resolve all references
166 * that are in scope for Xbase language?
167 * Will things like variables that are actually there in generated
168 * in the Impl class be accessible directly?
169 */
170 (body=XBlockExpression)?
171 ;
172 XParameter:
173 (annotations+=XAnnotation)*
174 (
175 unordered?='unordered' unique?='unique'? |
176 unique?='unique' unordered?='unordered'?
177 )?
178 type=XGenericType
179 multiplicity=XMultiplicity?
180 name=ID
181 ;
182 XTypeParameter :
183 (annotations+=XAnnotation)*
184 name=ID ('extends' bounds+=XGenericType ('&' bounds+=XGenericType)*)?
185 ;
186 XMultiplicity returns XMultiplicity:
187 '['
188 ('?' | '*' | '+' | (INT ('..' (INT | '?' | '*'))?))?
189 ']'
190 ;
191 XBlockExpression returns xbase::XBlockExpression:
192 {xbase::XBlockExpression}
```

```
193 '{'
194 (expressions+=XExpressionInsideBlock ';'?)*
195 '}'
196 ;
197 XGenericType returns XGenericType:
198 type=[genmodel::GenBase|XQualifiedName] (=>'<'
199 typeArguments+=XGenericTypeArgument
200 (',' typeArguments+=XGenericTypeArgument)* '>')?
201 ;
202 XGenericTypeArgument returns XGenericType :
203 XGenericType |
204 XGenericWildcardTypeArgument
205 ;
206 XGenericWildcardTypeArgument returns XGenericType :
207 {XGenericType}
208 '?' ('extends' upperBound=XGenericType |
209 'super' lowerBound=XGenericType)?
210 ;
211 XQualifiedName:
212 XID ('.' XID)*
213 ;
214 XID:
215 ID | 'get' | 'set' | 'isUnSet' | 'isSet'
216 ;
217 ValidID:
218 XID | 'void'
219 ;
```

Auflistung A.4: *Die Xtext Grammatik zu Xcore*

A.6.2 Die Xbase-Grammatik

```
1
2 /*******************************************************************************
3 * Copyright (c) 2010 itemis AG (http://www.itemis.eu) and others.
4 * All rights reserved. This program and the accompanying materials
5 * are made available under the terms of the Eclipse Public License v1.0
6 * which accompanies this distribution, and is available at
7 * http://www.eclipse.org/legal/epl-v10.html
8 *******************************************************************************/
9 grammar org.eclipse.xtext.xbase.Xbase with org.eclipse.xtext.xbase.Xtype
10 import "http://www.eclipse.org/xtext/xbase/Xbase"
11 import "http://www.eclipse.org/xtext/common/JavaVMTypes" as types
12 import "http://www.eclipse.org/emf/2002/Ecore" as ecore
13 XExpression returns XExpression :
14 XAssignment;
15 XAssignment returns XExpression :
16 {XAssignment} feature=[types::JvmIdentifiableElement|FeatureCallID]
17 OpSingleAssign value=XAssignment |
18 XOrExpression (
19 =>({XBinaryOperation.leftOperand=current}
20 feature=[types::JvmIdentifiableElement|OpMultiAssign]) rightOperand=XAssignment
21 )?;
22 OpSingleAssign:
23 '='
24 ;
25 OpMultiAssign:
26 '+=' | '-=' | '*=' | '/=' | '%=' |'<' '<' '=' |'>' '>'? '>=';
27 XOrExpression returns XExpression:
28 XAndExpression (=>({XBinaryOperation.leftOperand=current}
29 feature=[types::JvmIdentifiableElement|OpOr])
30 rightOperand=XAndExpression)*;
31 OpOr:
```

```
32  '||';
33  XAndExpression returns XExpression:
34  XEqualityExpression (=>({XBinaryOperation.leftOperand=current}
35  feature=[types::JvmIdentifiableElement|OpAnd])
36  rightOperand=XEqualityExpression)*;
37  OpAnd:
38  '&&';
39  XEqualityExpression returns XExpression:
40  XRelationalExpression (=>({XBinaryOperation.leftOperand=current}
41  feature=[types::JvmIdentifiableElement|OpEquality])
42  rightOperand=XRelationalExpression)*;
43  OpEquality:
44  '==' | '!=' | '===' | '!==';
45  XRelationalExpression returns XExpression:
46  XOtherOperatorExpression
47  (=>({XInstanceOfExpression.expression=current}
48   'instanceof') type=JvmTypeReference |
49  =>({XBinaryOperation.leftOperand=current}
50   feature=[types::JvmIdentifiableElement|OpCompare])
51    rightOperand=XOtherOperatorExpression)*;
52  OpCompare:
53  '>=' | '<' '=' | '>' | '<' ;
54  XOtherOperatorExpression returns XExpression:
55  XAdditiveExpression (=>({XBinaryOperation.leftOperand=current}
56   feature=[types::JvmIdentifiableElement|OpOther])
57  rightOperand=XAdditiveExpression)*;
58  OpOther:
59  '->'
60  | '..<'| '>' '..'| '..'| '=>'| '>' (=>('>' '>') | '>')
61  | '<' (=>('<' '<') | '<' | '=>')| '<>'| '?:';
62  XAdditiveExpression returns XExpression:
63  XMultiplicativeExpression (=>({XBinaryOperation.leftOperand=current}
64   feature=[types::JvmIdentifiableElement|OpAdd])
65  rightOperand=XMultiplicativeExpression)*;
66  OpAdd:
67  '+' | '-';
68  XMultiplicativeExpression returns XExpression:
69  XUnaryOperation (=>({XBinaryOperation.leftOperand=current}
70   feature=[types::JvmIdentifiableElement|OpMulti]) rightOperand=XUnaryOperation)*;
71  OpMulti:
72  '*' | '**' | '/' | '%';
73  XUnaryOperation returns XExpression:
74  {XUnaryOperation} feature=[types::JvmIdentifiableElement|OpUnary]
75  operand=XUnaryOperation
76  | XCastedExpression;
77  OpUnary:
78  "!" | "-" | "+";
79  XCastedExpression returns XExpression:
80  XPostfixOperation (=>({XCastedExpression.target=current}
81   'as') type=JvmTypeReference)*
82  ;
83  XPostfixOperation returns XExpression:
84  XMemberFeatureCall =>({XPostfixOperation.operand=current}
85   feature=[types::JvmIdentifiableElement|OpPostfix])?
86  ;
87  OpPostfix:
88  "++" | "--";
89  XMemberFeatureCall returns XExpression:
90  XPrimaryExpression
91  (=>({XAssignment.assignable=current} ('.'|explicitStatic?="::")
92   feature=[types::JvmIdentifiableElement|FeatureCallID] OpSingleAssign)
93    value=XAssignment
94  |=>({XMemberFeatureCall.memberCallTarget=current}
95  ("."|nullSafe?="?."|explicitStatic?="::"))
96  ('<' typeArguments+=JvmArgumentTypeReference
97   (',' typeArguments+=JvmArgumentTypeReference)* '>')?
```

```
98  feature=[types::JvmIdentifiableElement|IdOrSuper] (
99  =>explicitOperationCall?='('
100 (
101 memberCallArguments+=XShortClosure
102 | memberCallArguments+=XExpression (',' memberCallArguments+=XExpression)*
103 )?
104 ')')?
105 memberCallArguments+=XClosure?
106 )*;
107 XPrimaryExpression returns XExpression:
108 XConstructorCall |XBlockExpression | XSwitchExpression |
109 XSynchronizedExpression | XFeatureCall | XLiteral |
110 XIfExpression | XForLoopExpression | XBasicForLoopExpression |
111 XWhileExpression | XDoWhileExpression | XThrowExpression |
112 XReturnExpression | XTryCatchFinallyExpression | XParenthesizedExpression;
113 XLiteral returns XExpression:
114 XCollectionLiteral | XClosure | XBooleanLiteral |
115 XNumberLiteral | XNullLiteral | XStringLiteral |
116 XTypeLiteral
117 ;
118 XCollectionLiteral:
119 XSetLiteral | XListLiteral
120 ;
121 XSetLiteral:
122 {XSetLiteral} '#' '{' (elements+=XExpression
123  (',' elements+=XExpression )*)? '}'
124 ;
125 XListLiteral:
126 {XListLiteral} '#' '[' (elements+=XExpression
127  (',' elements+=XExpression )*)? ']'
128 ;
129 XClosure returns XExpression:
130 =>({XClosure}
131 '[')
132 =>((declaredFormalParameters+=JvmFormalParameter
133  (',' declaredFormalParameters+=JvmFormalParameter)*)?
134  explicitSyntax?='|')?
135 expression=XExpressionInClosure
136 ']';
137 XExpressionInClosure returns XExpression:
138 {XBlockExpression}
139 (expressions+=XExpressionOrVarDeclaration ';'?)*
140 ;
141 XShortClosure returns XExpression:
142 =>({XClosure} (declaredFormalParameters+=JvmFormalParameter
143  (',' declaredFormalParameters+=JvmFormalParameter)*)?
144  explicitSyntax?='|') expression=XExpression;
145 XParenthesizedExpression returns XExpression:
146 '(' XExpression ')';
147 XIfExpression returns XExpression:
148 {XIfExpression}
149 'if' '(' if=XExpression ')'
150 then=XExpression
151 (=>'else' else=XExpression)?;
152 XSwitchExpression returns XExpression:
153 {XSwitchExpression}
154 'switch' (=>('(' declaredParam=JvmFormalParameter ':') switch=XExpression ')'
155 | =>(declaredParam=JvmFormalParameter ':')? switch=XExpression) '{'
156 (cases+=XCasePart)*
157 ('default' ':' default=XExpression )?
158 '}';
159 XCasePart:
160 {XCasePart}
161 typeGuard=JvmTypeReference? ('case' case=XExpression)?
162 (':' then=XExpression | fallThrough?=',') ;
163 XForLoopExpression returns XExpression:
```

```
164 =>({XForLoopExpression}
165 'for' '(' declaredParam=JvmFormalParameter ':') forExpression=XExpression ')'
166 eachExpression=XExpression;
167 XBasicForLoopExpression returns XExpression:
168 {XBasicForLoopExpression}
169 'for' '('(initExpressions+=XExpressionOrVarDeclaration
170 (',' initExpressions+=XExpressionOrVarDeclaration)*)? ';'
171 expression=XExpression? ';'
172 (updateExpressions+=XExpression (',' updateExpressions+=XExpression)*)? ')'
173 eachExpression=XExpression;
174 XWhileExpression returns XExpression:
175 {XWhileExpression}
176 'while' '(' predicate=XExpression ')'
177 body=XExpression;
178 XDoWhileExpression returns XExpression:
179 {XDoWhileExpression}
180 'do'
181 body=XExpression
182 'while' '(' predicate=XExpression ')';
183 XBlockExpression returns XExpression:
184 {XBlockExpression}
185 '{'
186 (expressions+=XExpressionOrVarDeclaration ';'?)*
187 '}';
188 XExpressionOrVarDeclaration returns XExpression:
189 XVariableDeclaration | XExpression;
190 XVariableDeclaration returns XExpression:
191 {XVariableDeclaration}
192 (writeable?='var'|'val')
193 (=>(type=JvmTypeReference name=ValidID)| name=ValidID)
194  ('=' right=XExpression)?;
195 JvmFormalParameter returns types::JvmFormalParameter:
196 (parameterType=JvmTypeReference)? name=ValidID;
197 FullJvmFormalParameter returns types::JvmFormalParameter:
198 parameterType=JvmTypeReference name=ValidID;
199 XFeatureCall returns XExpression:
200 {XFeatureCall}
201 ('<' typeArguments+=JvmArgumentTypeReference
202 (',' typeArguments+=JvmArgumentTypeReference)* '>')?
203 feature=[types::JvmIdentifiableElement|IdOrSuper]
204 (=>explicitOperationCall?='('
205 (
206 featureCallArguments+=XShortClosure
207 | featureCallArguments+=XExpression
208 (',' featureCallArguments+=XExpression)*
209 )?
210 ')')?
211 featureCallArguments+=XClosure?;
212 FeatureCallID:
213 ValidID | 'extends' | 'static' | 'import' | 'extension'
214 ;
215 IdOrSuper :
216 FeatureCallID | 'super'
217 ;
218 XConstructorCall returns XExpression:
219 {XConstructorCall}
220 'new' constructor=[types::JvmConstructor|QualifiedName]
221 (=>'<' typeArguments+=JvmArgumentTypeReference
222 (',' typeArguments+=JvmArgumentTypeReference)* '>')?
223 (=>explicitConstructorCall?='('
224 (
225 arguments+=XShortClosure
226 | arguments+=XExpression (',' arguments+=XExpression)*
227 )?
228 ')')?
229 arguments+=XClosure?;
```

```
230 XBooleanLiteral returns XExpression :
231 {XBooleanLiteral} ('false' | isTrue?='true');
232 XNullLiteral returns XExpression :
233 {XNullLiteral} 'null';
234 XNumberLiteral returns XExpression :
235 {XNumberLiteral} value=Number;
236 XStringLiteral returns XExpression:
237 {XStringLiteral} value=STRING;
238 XTypeLiteral returns XExpression :
239 {XTypeLiteral} 'typeof' '(' type=[types::JvmType|QualifiedName]
240  (arrayDimensions+=ArrayBrackets)* ')'
241 ;
242 XThrowExpression returns XExpression :
243 {XThrowExpression} 'throw' expression=XExpression;
244 XReturnExpression returns XExpression :
245 {XReturnExpression} 'return' (->expression=XExpression)?;
246 XTryCatchFinallyExpression returns XExpression:
247 {XTryCatchFinallyExpression}
248 'try'
249 expression=XExpression
250 (
251 catchClauses+=XCatchClause+
252 (=>'finally' finallyExpression=XExpression)?
253 | 'finally' finallyExpression=XExpression
254 );
255 XSynchronizedExpression returns XExpression:
256 =>({XSynchronizedExpression}
257 'synchronized' '(') param=XExpression ')' expression=XExpression;
258 XCatchClause :
259 =>'catch' '(' declaredParam=FullJvmFormalParameter ')' expression=XExpression;
260 QualifiedName:
261 ValidID (=>'.' ValidID)*;
262 Number hidden():
263 HEX | (INT | DECIMAL) ('.' (INT | DECIMAL))?;
264 /**
265 * Dummy rule, for "better" downwards compatibility,
266 * since GrammarAccess generates non-static inner classes,
267 * which makes downstream grammars break on classloading,
268 * when a rule is removed.
269 */
270 StaticQualifier:
271 (ValidID '::')+
272 ;
273 terminal HEX:
274 ('0x'|'0X') ('0'..'9'|'a'..'f'|'A'..'F'|'_')+
275 ('#' (('b'|'B')('i'|'I') | ('l'|'L')))?;
276 terminal INT returns ecore::EInt:
277 '0'..'9' ('0'..'9'|'_')*;
278 terminal DECIMAL:
279 INT
280 (('e'|'E') ('+'|'-')? INT)?
281 (('b'|'B')('i'|'I'|'d'|'D') | ('l'|'L'|'d'|'D'|'f'|'F'))?;
```

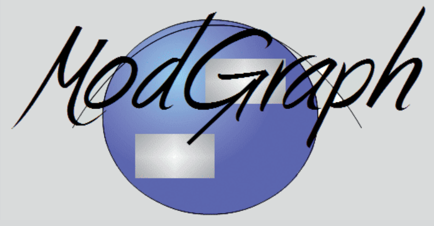
Auflistung A.5: *Die XText Grammatik zu Xbase*

Dank

Zuletzt möchte ich hier kurz meinen herzlichen Dank an alle Beteiligten ausdrücken. Ich danke allen wissenschaftlichen und nicht wissenschaftlichen Mitarbeitern des Lehrstuhls für angewandte Informatik 1, meinen studentischen Helfern und insbesondere meinem Doktorvater, Prof. Dr. Westfechtel. Vielen Dank für die hilfreichen Anregungen und Rückmeldungen! Mein Dank gilt ebenso meinem Zweitkorrektor, Prof. Dr. Schürr und den Mitgliedern der Promotionskommission sowie den Mitgliedern meiner Prüfungskommission.

Zudem danke ich meinen Eltern und Großeltern, die mir meine Ausbildung und somit den Start dieser Arbeit ermöglicht haben und mich im Laufe derer unterstützt haben. Mein Dank gilt ebenso meinen Freuden für das unermütliche Korrekturlesen und Essen kochen. Schön, dass es euch gibt!

Danke!



Kurzfassung

Das im Rahmen dieser Dissertation vorgestellte Projekt ModGraph bietet einen ganzheitlichen, leichtgewichtigen und hochgradig integrativen Ansatz zur totalen modellgetriebenen Softwareentwicklung. Der ModGraph-Ansatz bietet eine echte Erweiterung des EMF-Rahmenwerks, einem in Industrie und Forschung etablierten Werkzeug zur Strukturmodellierung (und anschließender Quelltextgenerierung) mit Ecore-Klassendiagrammen. Dabei steht die Vervollständigung von EMF um die - bislang fehlende - Verhaltensmodellierung im Vordergrund. Diese wird durch einen hybriden Ansatz aus regelbasierter und prozeduraler Verhaltensmodellierung erreicht, der programmierte Graphtransformationsregeln zur Spezikation von Verhalten anbietet und gleichzeitig eine strikte Trennung der Regeln und der prozeduralen Elemente fordert. Dazu setzt sich ModGraph zum Ziel, bestehende Konzepte zu erweitern und zu nutzen, statt diese nochmals zu reimplementieren. ModGraph konzentriert sich dabei auf den Mehrwert der Graphtransformationsregeln im Kontext der Verhaltens-modellierung. Die Regeln stellen komplexe Änderungen an Objektstrukturen deklarativ dar, indem lediglich der Zustand der Objektstruktur vor und nach ihrer Ausführung angegeben wird.

Die entstandene Modellierungssprache wird in dieser Dissertation beschrieben und kann in einem ebenfalls hier vorgestellten, intuitiv zu bedienenden grafischen Editor in Eclipse genutzt werden. Abschließend werden zwei Fallstudien zur Evaluation von ModGraph betrachtet: propagierende Refactorings für und mit ModGraph sowie die Anwendung der Graphtrans-formationsregeln im Kontext der szenarienbasierten Modellierung zur Simulation von Echtzeitsystemen.