Generierung domänenspezifischer Transformationssprachen

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Dipl.-Inform. Ingo Weisemöller
aus Osnabrück

Berichter: Universitätsprofessor Dr. rer. nat. Bernhard Rumpe
Universitätsprofessor Dr. rer. nat. Andy Schürr

Tag der mündlichen Prüfung: 11. Mai 2012

[Wei12] I. Weisemöller
Generierung domänenspezifischer Transformationssprachen
Shaker Verlag, ISBN 978-3-8440-1191-3.
Aachener Informatik-Berichte, Software Engineering, Band 12. 2012.
www.se-rwth.de/publications

Kurzfassung

Die Bedeutung domänenspezifischer Sprachen (DSLs) im Software-Engineering ist in den letzten Jahren deutlich gestiegen. Diese Sprachen erlauben es Domänenexperten, Teile des zu entwickelnden Systems in einer problemorientierten Notation zu beschreiben, die in der jeweiligen Domäne bekannt ist. Sie können so die Einbindung der Domänenexperten in Softwareentwicklungsprozesse wesentlich verbessern.

Mit der Einführung einer neuen Programmier- oder Modellierungssprache entsteht gleichzeitig der Bedarf, Modelle oder Programme in dieser Sprache zu transformieren. Häufig verwendete Transformationen werden dabei in Transformationsregeln formalisiert, sodass sie computerunterstützt oder sogar vollautomatisch durchgeführt werden können.

Die Beschreibung von Transformationsregeln ist jedoch nicht nur für Sprachentwickler, sondern auch für Sprachnutzer relevant. Beispielsweise können sie so spezifische Analyse- und Editieroperationen oder Folgen solcher Operationen in Makros beschreiben und bei Bedarf erneut ausführen.

Eines der in der vorliegenden Arbeit vorgestellten Ergebnisse ist eine Transformationssprache, welche die konkrete Syntax einer gegebenen textuellen Modellierungssprache (Basissprache) wiederverwendet. Dadurch werden Domänenexperten in die Lage versetzt, sowohl Modelle als auch Modelltransformationen in einer ihnen vertrauten Notation zu beschreiben. Die Benutzung und Entwicklung der Transformationssprache werden am nichttrivialen Beispiel der Vereinfachung hierarchischer Statecharts beschrieben, einer semantikerhaltenden Transformation, die als ein Schritt innerhalb eines Codegenerierungsprozesses eingesetzt werden kann.

Die domänenspezifische Transformationssprache und die Transformationsengine bestehen aus drei Kernkomponenten, die jeweils in einem separaten Kapitel dieser Arbeit vorgestellt werde. Erstens enthält die Transformationssprache eine Regelsprache mit domänenspezifischer Notation. Diese Transformationsregeln können in eine andere, generische Regelsprache übersetzt werden; diese zweite Sprache verwendet eine auf UML-Objektdiagrammen basierende Syntax, aus der die Regeln wiederum nach Java übersetzt werden können. Drittens erlaubt es eine Kontrollflusssprache, Regeln aus den zuvor genannten Sprachen zu komplexen Transformationsmodulen zusammenzusetzen.

Der wissenschaftlich neue und einzigartige Beitrag dieser Arbeit besteht in der Möglichkeit, die domänenspezifische Transformationssprache automatisch aus der textuellen Basissprache abzuleiten. Für den Sprachentwickler bedeutet die Einführung einer solchen Transformationssprache daher kaum zusätzlichen Aufwand, da diese automatisch aus ohnehin existierenden Artefakten generiert wird.

Somit stellt die Generierung domänenspezifischer Transformationssprachen einen erheblichen Mehrwert für den Sprachnutzer dar, dem nur unwesentlicher Mehraufwand auf Seiten des Sprachentwicklers gegenübersteht.

Abstract

Domain specific languages (DSLs) have recently become increasingly important in software engineering. These languages allow domain experts to model parts of the system under development in a problem-oriented notation that is well-known in the respective domain. Thus, they can substantially improve the involvement of domain experts into software engineering processes.

The introduction of a DSL is accompanied by the need to transform models or programs in this language. In doing so, frequently used transformations are formalized as transformation rules for computer aided or even automatic repeated execution.

Describing transformation rules is of interest not only for language developers, but also for language users. For instance, they can describe specific analysis and editing operations or sequences of such operations in macros, and execute them again as needed.

The results presented in this thesis include a transformation language that reuses the concrete syntax of a given textual modeling language (base language), which allows experts for the base language to describe both models and model transformations in a notation familiar to them. The usage and development of the transformation language are illustrated based on the rather complex example of simplifying hierarchical statecharts, a semantics preserving transformation that can be used as a step of a code generation process.

The domain specific transformation language and the transformation engine consist of three main components, each of which is presented in a separate chapter of this thesis. First, the language contains rule language with the domain specific notation mentioned above. These rules can be translated to rules in a different, generic transformation language; this second language uses a notation based on UML object diagrams, which can in turn be translated to Java. Third, a control flow language allows to combine rules in either of these languages to complex transformation modules.

As a unique and novel innovation, the results presented here allow to automatically derive the domain specific transformation language from the description of the base textual DSLs. From the language developers' perspective, introducing such a language therefore causes very little effort, because it is generated automatically from previously existing artifacts.

Hence, domain specific transformation languages are substantially valuable for the users of domain specific modeling languages, and they comes with negligible additional efforts for the language developer.

Danksagung

Viele Wissenschaftler und Kollegen, Verwandte, Freunde und Bekannte haben zum Gelingen dieser Arbeit beigetragen. Ihnen allen möchte ich an dieser Stelle meinen besonders herzlichen Dank aussprechen. All das aufzulisten, was ich bei der Anfertigung dieser Dissertationsschrift und während der Promotion an Unterstützung erfahren habe, würde viele Seiten füllen, und so versuche ich, mich auf das Allerwichtigste zu beschränken.

Mein ganz besonderer Dank gilt Prof. Dr. Bernhard Rumpe für die Betreuung meiner Promotion. Die zahlreichen, stets zielgerichteten, niemals langweiligen und unterhaltsamen Diskussionen mit Bernhard spiegeln sich in dieser Arbeit in zahlreichen Facetten wieder – angefangen von der Auswahl des Themas über das Finden eines geeigneten Beispiels bis hin zur Feinabstimmung des Sprachdesigns. Ferner bedanke ich mich bei Bernhard (und bei vielen anderen, aber dazu später), für die wunderbare Zeit am Lehrstuhl Software Engineering der RWTH Aachen, zahlreiche Kickermatches und Skatpartien, an die ich sehr gerne zurückdenke.

An Prof. Dr. Andy Schürr geht ebenfalls ein besonders herzliches Dankeschön. Andy hat es nicht nur geschafft, mich für die wissenschaftlichen Fragen der Modellierung und der Transformationen während meiner Zeit an der TU Darmstadt zu begeistern, sondern wir blieben auch in der anschließenden Aachener Zeit stets in sehr gutem Kontakt zueinander. Ganz besonders freut es mich, dass Andy das Zweitsutachten für diese Arbeit übernommen hat, und dass er mir geholfen hat, dem Abschlussvortrag den letzten Schliff zu verleihen. [1]

Bei Prof. Dr.-Ing. Stefan Kowalewski und Priv.-Doz. Dr. Thomas Noll möchte ich mich sehr herzlich dafür bedanken, dass sie sich als Mitglieder der Prüfungskommission zu meinem Promotionsvorhaben zur Verfügung gestellt haben.

Eine ganz besonders lieber Dank geht an meine Freundin Lise Voigt. Sie hat mich nicht nur in den arbeitsreichen Phasen der Promotion liebevoll unterstützt, sondern auch geduldig sowohl das Fachchinesisch als auch die Folgen zwischenzeitlicher Rückschläge („Die ... geht nicht!“) ertragen, als in ihrer Wohnung im Sommer 2011 ein Großteil der Implementierung entstanden ist.

Eine erstaunliche Begeisterung für das eben genannte Fachchinesisch kann ich auch immer wieder bei meiner Familie beobachten, allen voran bei meinen Eltern, die mich seit meiner Kindheit jederzeit phänomenal und auf jede erdenkliche Art und Weise unterstützt haben, und die mir so ein Studium mit anschließender Promotion überhaupt erst ermöglicht haben. Ihnen gebührt mein ganz besonderer Dank, ebenso wie meinem Bruder Dr. Thomas Weisemöller und seiner Frau Betty.

Unter den vielen Kollegen, die zum Gelingen der Arbeit beigetragen haben, möchte ich Thomas Kurpick hervorheben, der nicht nur der beste Kollege ist, den ich mir in einem gemeinsamen

[1] Andy und ich sind die einzigen, die sowohl den Probe- als auch den Abschlussvortrag gehört haben, aber ich kann allen Lesern versichern, dass das Feedback zum Probevortrag mir sehr geholfen hat.

Büro vorstellen kann, sondern auch ein langjähriger und guter Freund. Weiterhin geht ein besonderer Dank an Marita Breuer und Galina Volkova für den hervorragenden MontiCore-Support und für die Hilfe bei der Implementierung, sowie an Jan Oliver Ringert und Achim Lindt für ihre Beiträge zum Design der Sprachen in zahlreichen Diskussionsrunden.

Für das Korrekturlesen früher Fassungen der Dissertation bedanke ich mich sehr herzlich bei Dr. Martin Schindler, der den größten Teil der Arbeit gelesen hat, und bei André Fischer, Arne Haber, Thomas Kurpick, Antonio Navarro Pérez, Jan Oliver Ringert und Lise Voigt, die mich hier ebenfalls unterstützt haben.

Für die vielen hilfreichen Anmerkungen zu meiner Arbeit sowie fachliche Diskussionen, zahlreiche gute Ratschläge aus der „I3-Perspektive“, ihre enorme Hilfsbereitschaft und jederzeit ein offenes Ohr danke ich Prof. Dr.-Ing. Manfred Nagl und Dr. Anne-Thérèse Körtgen.

Ebenfalls bedanken möchte ich mich bei allen, die mir geholfen haben, meine Zeit am Lehrstuhl in so wunderbarer und positiver Erinnerung zu behalten. Hier dürfen sich alle Kollegen und ehemaligen Kollegen angesprochen fühlen. Besonders nennen möchte ich Dr. Markus Heller, der während meiner Zeit als studentische Hilfskraft mein ständiger Ansprechpartner war, und der meine Diplomarbeit betreut hat, sowie Silke Cormann, Angelika Fleck und Sylvia Gunder, die stets für einen angenehmen und reibungslosen Lehrstuhlbetrieb gesorgt haben. Bei den kickerbegeisterten Kollegen bedanke ich mich ebenfalls von Herzen und bitte darum, mir meine reduzierte Verfügbarkeit hierfür in den letzten Monaten der Promotion nachzusehen.

Abschließend danke ich allen sehr herzlich, die in Aachen rund um das Studium und die Promotion mit mir relaxt, gefeiert, ihre Freizeit verbracht oder einfach nur gut gelebt haben: André, Anna und Marcus, Diana, dem Eilendorder HC, Franzi und Andreas, Isabel und Bernd, Kai und Melanie mit Mika, dem KaWo 1, der Kiste, der Mensa V, Peter und Rebecca, Regina und Mo, Tanja, Thommy und Anke, Timo und Barbara, Torsten und Meike mit Flora, Ulla und Henning, sowie vielen anderen, die mich in den letzten Jahren begleitet haben.

Und nun wünsche ich eine gute Lektüre. Allen, deren Interesse diese Arbeit weckt, kann ich übrigens auch die anderen Bände von *Aachener Informatik-Berichte, Software Engineering* sehr empfehlen.

Inhaltsverzeichnis

1. Einleitung 1
1.1. Kontext und Einordnung der Arbeit . . . 1
1.2. Wissenschaftliche Fragestellung dieser Arbeit . . . 3
1.3. Wesentliche Ergebnisse . . . 4
1.4. Aufbau der Arbeit . . . 5

2. Anwendung und Entwicklung von Modelltransformationen 9
2.1. Modelle und domänenspezifische Sprachen im Software-Engineering . . . 9
2.1.1. Einsatz domänenspezifischer Sprachen . . . 9
2.1.2. Standardisierungsansätze zur Modellierung . . . 10
2.1.3. Grafische und textuelle Sprachen . . . 11
2.1.4. Ansätze zur Sprachdefinition . . . 12
2.2. Methodik der Nutzung von Modelltransformationen . . . 14
2.2.1. Nutzungsszenarien . . . 15
2.2.2. Nutzbarkeit von Transformationssprachen für Domänenexperten . . . 18
2.3. Kategorisierung von Transformationssprachen und -werkzeugen . . . 19
2.3.1. Unterscheidungskriterien . . . 19
2.3.2. Einordnung dieser Arbeit . . . 21
2.4. Wissenschaftliche Herausforderungen für Modelltransformationen . . . 22
2.5. Motivation der vorliegenden Arbeit . . . 24

3. Durchgängiges Beispiel: Vereinfachung hierarchischer Statecharts 25
3.1. Motivation und Anwendbarkeit . . . 25
3.2. Einführung in die Statechartsprache . . . 26
3.2.1. Zustände und Transitionen . . . 27
3.2.2. Stimuli, Aktionen und Guards . . . 28
3.2.3. Invarianten, Vor- und Nachbedingungen . . . 29
3.2.4. Entry- und Exit-Aktionen . . . 29
3.2.5. Do-Aktivitäten und interne Transitionen . . . 30
3.2.6. Die textuelle Syntax der Statechartsprache . . . 30
3.3. Semantikerhaltende Transformationsregeln . . . 33
3.3.1. Do-Aktivitäten eliminieren . . . 34
3.3.2. Transitionen an initiale Subzustände umleiten . . . 35
3.3.3. Transitionen aus finalen Subzuständen starten lassen . . . 36
3.3.4. Exit-Aktionen an ausgehende Transitionen verschieben . . . 36
3.3.5. Entry-Aktionen an eingehende Transitionen verschieben . . . 37

3.3.6. Interne Transitionen eliminieren 37
3.3.7. Zustandsinvarianten an Subzustände propagieren 38
3.3.8. Hierarchisch zergliederte Zustände entfernen 38
3.4. Verknüpfung der Regeln 39
3.5. Anforderungen an die Transformationssprache 41

4. Transformationsregeln in Objektdiagramm-Notation 43
4.1. Grundlagen 43
4.1.1. Konkrete Syntax, abstrakte Syntax und attributierte Graphen 43
4.1.2. Graphersetzungsregeln 45
4.2. Aufbau der Regeln 48
4.3. Effizientes Pattern-Matching 51
4.3.1. Reduktion des Suchraums 52
4.3.2. Kostenmodell und Optimierung des Suchplans 53
4.3.3. Implementierung des Pattern-Matching-Algorithmus 55
4.4. Negative Objekte und Listenobjekte 57
4.5. Ersetzung mit dem Entwurfsmuster Command 59
4.6. Verwandte Arbeiten 60
4.6.1. Verwendung von Constraint-Solvern 61
4.6.2. Abbildung auf das relationale Datenmodell 61
4.6.3. Suchplangesteuerte Ansätze 62

5. Eine Regelsprache mit domänenspezifischer Syntax 65
5.1. Syntax und Semantik der Regeln 65
5.1.1. Pattern Matching 66
5.1.2. Modifikationen des Modells 69
5.2. Ableitung der kontextfreien Syntax aus der Basis-DSL 72
5.2.1. Terme in Transformationsregeln 72
5.2.2. Umgang mit Attributen 73
5.3. Kontextbedingungen und Empfehlungen für Anwender 77
5.3.1. Typen der Schemavariablen 77
5.3.2. Disjunktheit der Matches von Listenknoten 78
5.3.3. Formatierungs- und Programmierrichtlinien 79
5.4. Übersetzung in Objektdiagramm-Notation 81
5.4.1. Generierung der linken und rechten Regelseite 81
5.4.2. Listen- und negative Knoten, Constraints und nichtisomorphe Matches 82
5.4.3. Implementierung des Transformators 85
5.5. Umgang mit Bezeichnern 86
5.5.1. Bezeichner bei der Mustersuche 87
5.5.2. Bezeichnerersetzung 87
5.5.3. Bezeichner bei der Generierung von Objektdiagrammregeln 88
5.6. Verwandte Arbeiten 89

6. Die Kontrollflusssprache für Modelltransformationen 93
6.1. Aufbau der Transformationsmodule 94
6.1.1. Module und Methoden 94
6.1.2. Sequenzielle Ausführung von Anweisungen 95
6.1.3. Bedingte Anweisungen und Schleifen 96
6.1.4. Eingebettete Java-Ausdrücke 97
6.1.5. Typsystem der Kontrollflusssprache 97
6.2. Einbettung der Transformationsregeln und Methodenaufrufe 98
6.3. Nichtdeterminismus 100
6.4. Codegenerierung 102
6.4.1. Struktur des generierten Codes 102
6.4.2. Ablauf der Codegenerierung 103
6.4.3. Abbildung des Nichtdeterminismus 106
6.5. Verwandte Arbeiten 106

7. Generierung von Transformationssprachen aus DSL-Grammatiken 111
7.1. Anwendungsszenario 111
7.2. Generierung der Regelgrammatik 112
7.2.1. Inhalt der Regelgrammatiken 113
7.2.2. Technische Umsetzung 117
7.3. Generierung des Transformators 117
7.3.1. Semantik der DSLs und Semantik der generierten Regelsprachen 118
7.3.2. Arbeitsweise und Aufbau des Transformators 119
7.3.3. Technische Umsetzung 119
7.4. Einbettung in die Kontrollflusssprache und generierte Werkzeuge 125

8. Nutzung von Transformationen und Transformationssprachen 129
8.1. Generierung der Transformationssprache durch den Sprachentwickler 129
8.2. Aufruf der generierten Werkzeuge und Workflows 130
8.2.1. DSLTools für Transformationen 131
8.2.2. Workflows 131
8.3. Integration von Transformationen in den Codegenerierungsprozess 132
8.3.1. Sprachen, Transformationen und Modelle 133
8.3.2. Transformationen in der Codegenerierung für Statecharts 133
8.4. Fallbeispiele aus der Vereinfachung hierarchischer Statecharts 135
8.4.1. Verschieben von Elementen 136
8.4.2. Kopieren von Elementen 136
8.4.3. Veränderung gebundener Bezeichner 137
8.4.4. Verwendung von Kontrollstrukturen 138
8.4.5. Erstellung eines ausführbaren Programms 139

9. Die Entwicklung der Modelltransformationsengine 141
9.1. Projektstruktur und agile Entwicklung mit MontiCore 141
9.1.1. Innere Struktur der Transformationsprojekte 141
9.1.2. Testgetriebene Entwicklung der MontiCore-Transformationsengine . . 143
9.2. Prototypenbasierte Entwicklung der Generatoren 144
9.2.1. Prototypen- und Generatorprojekte mit Modultests 145
9.2.2. Systemtest 146
9.3. Testgetriebene Weiterentwicklung und Wartung 147
9.3.1. Testen einfacher Generatoren 148
9.3.2. Testen des Generators für generierte Regelsprachen 149
9.4. Vor- und Nachteile der Ansätze 150

10. Epilog 153
10.1. Zusammenfassung 153
10.2. Ausblick 155
10.2.1. Anwendungen der Ergebnisse dieser Arbeit 155
10.2.2. Konzeptuell verwandte Problemstellungen 156
10.2.3. Technische Erweiterungen 157
10.3. Abschließende Bemerkungen 159

Literaturverzeichnis 161

A. Glossar und Abkürzungsverzeichnis 175

B. Notationen 181

C. Grammatiken 183
C.1. Grammatik der Statechartsprache 183
C.2. Grammatik der Transformationsregeln auf Statecharts 185

D. Transformation zur Vereinfachung hierarchischer Statecharts 211

E. Lebenslauf 219

Kapitel 1.

Einleitung

Domänenspezifische Sprachen erlauben es, bei der Erstellung von Softwaresystemen Experten der jeweiligen Anwendungsdomäne in die Entwicklung mit einzubeziehen und somit Expertenwissen besser zu nutzen. Diese Einbindung von Experten ist nicht auf einzelne Aktivitäten des Entwicklungsprozesses begrenzt, sondern kann immer dann in besonderem Maße stattfinden, wenn eine domänenspezifische Sprache (Englisch: *Domain Specific Language*, kurz *DSL*) eingesetzt wird. Tatsächlich existiert bereits ein breites Portfolio an DSLs aus verschiedenen Domänen zum Einsatz in nahezu allen Phasen des Softwareentwicklungsprozesses.

Die Verwendung einer DSL geht in der Regel mit der Notwendigkeit einher, Modelle beziehungsweise Programme in der Sprache zu transformieren. Durch Transformationen lassen sich beispielsweise evolutionäre Entwicklungsschritte in der Softwareentwicklung, Refactorings, Migrationen auf neue Sprachen und Werkzeuge oder Regeln zur Sicherung der Konsistenz heterogener Dokumente beschreiben.

Die Sprachen zur Beschreibung von Transformationsregeln sind jedoch bisher nicht domänenspezifisch und somit für Domänenxperten nur schwer verständlich und nutzbar. Das Ziel dieser Arbeit besteht darin, zu einer gegebenen domänenspezifischen Sprache S in einem automatisierten Verfahren eine Transformationssprache T abzuleiten, deren Notation sich eng an der von S orientiert, und die somit auch für Domänenxperten verständlich und anwendbar ist.

Dieses erste Kapitel ordnet zunächst die Arbeit in den umgebenden Kontext ein. Dazu wird in den folgenden zwei Abschnitten ein Überblick über die Forschungsthemen am Lehrstuhl für Software Engineering und über das MontiCore-Projekt gegeben. Im letzten Abschnitt folgt die Beschreibung der Gliederung dieses Arbeit und der Inhalte der folgenden Kapitel.

1.1. Kontext und Einordnung der Arbeit

Die Arbeit wurde am Lehrstuhl für Software Engineering der RWTH Aachen angefertigt. Die modellbasierte Softwareentwicklung sowie die Entwicklung unter Anwendung domänenspezifischer Sprachen gehören zu den Forschungsschwerpunkten des Lehrstuhls. Die wichtigsten Anwendungen dieser Ansätze liegen in den Bereichen automotive Software, Energiemanagement und Cloud Computing [NR+10].

Themen rund um Graph- und Modelltransformationen haben am Lehrstuhl eine langjährige Tradition. Hervorzuheben ist hier vor allem das Werkzeug PROGRES, das seit den späten 1980er Jahren entstanden ist und dort über rund 20 Jahre kontinuierlich weiterentwickelt wurde [Sch88, Sch91, Zün96a, RW08].

Mögliche Anwendungen für einen neuen Transformationsansatz liegen vor allem im Bereich der textuellen Modellierungssprachen. So stellen die Entwicklung und Anwendung des UML-Profils UML/P Kernthemen der Forschung am Lehrstuhl dar [Rum11, Rum04]. Mit der Dissertation [Sch12] wurden vor dieser Arbeit und teilweise parallel dazu die in der UML/P enthaltenen Sprachen mit einer textuellen Syntax implementiert. Im Gegenzug nutzt auch der hier vorgestellte Ansatz Elemente aus [Sch12], insbesondere zur Strukturmodellierung und zur Codegenerierung. Ebenfalls zu erwähnen sind die Architekturbeschreibungssprachen MontiArc und AJava, die in Ergänzung zur UML/P vor allem für die Modellierung eingebetteter, aber auch verteilter Systeme eingesetzt werden [HRR10]. Eine Reihe weiterer DSLs befindet sich am Lehrstuhl derzeit im Einsatz oder der Entwicklung. Die meisten dieser Sprachen sind den Domänen Automotive Software oder Energiemanagement zuzuordnen.

Die vorliegende Arbeit ist damit sowohl durch die Nutzung am Lehrstuhl für Software Engineering entwickelter Sprachen und Werkzeuge als auch durch die Bereitstellung von Lösungen für zukünftige Arbeiten in die Projektlandschaft am Lehrstuhl eingebunden.

Die vorgenannten Sprachen wurden mit Hilfe des MontiCore-Frameworks entwickelt. MontiCore [Kra10] ist ein Werkzeug und Rahmenwerk zur Entwicklung und Verarbeitung domänenspezifischer Sprachen. Auch die Ergebnisse der vorliegenden Arbeit wurden einerseits mit MontiCore entwickelt. Andererseits sind sie auch als Erweiterungen von MontiCore zu betrachten, die zusätzliche Funktionalität für mit MontiCore entwickelte Sprachen zur Verfügung stellen.

MontiCore entstand zunächst am Institut für Software Systems Engineering der TU Braunschweig. Seit 2009 wird es am Lehrstuhl für Software Engineering der RWTH Aachen weiterentwickelt.

Der wesentliche Einsatzzweck von MontiCore ist die Entwicklung textueller Modellierungssprachen, insbesondere auch der UML/P. Kern des MontiCore-Projektes ist ein daher ein compilererzeugendes Werkzeug, das aus einer angereicherten Grammatik einen Parser, Klassen der abstrakten Syntax sowie zusätzliche Infrastruktur zur Analyse und Verarbeitung von Modellen generiert [Kra10]. Die Infrastruktur unterstützt die Ausführung einzelner zustandsloser Verarbeitungsschritte für Modelle, genannt Workflows, durch das Rahmenwerk. Die Workflows können dabei sowohl Schritte zur Analyse der Modelle, etwa zum Parsen oder zum Überprüfen von Kontextbedingungen, als auch zur Synthese, beispielsweise zur Codegenerierung, sein. Die generierten Klassen der abstrakten Syntax bieten zusätzliche Methoden an, die ebenfalls der komfortablen Verarbeitung der Modelle dienen. So enthalten diese Klassen beispielsweise Methoden, die in Kombination mit dem Rahmenwerk eine Traversierung des abstrakten Syntaxbaums (Englisch: *abstract syntax tree*, kurz *AST*) nach dem Entwurfsmuster Visitor [GHJV95] erlauben. Workflows zur Codegenerierung werden in MontiCore nach einem templatebasierten Ansatz realisiert, der in [Sch12] entwickelt wurde. Neben den eigentlichen Templates sieht dieser Ansatz auch die Verwendung eines Template-Operators zur Ablaufsteuerung sowie sprachspezifischer Kalkulatoren vor, welche die Implementierung komplexerer Berechnungen in Java ermöglichen. Dies trägt zu effizienterer Entwicklung dieser Berechnungen und besserer Lesbarkeit der Templates bei.

Besonderheiten bei der Sprachentwicklung mit MontiCore sind die generierten Klassen der abstrakten Syntax, die viele Konzepte aus der Objektorientierung für die Sprachentwicklung adaptieren, ferner die integrierte Spezifikation von konkreter und abstrakter Syntax, sowie die

in [Völ11] entwickelten Konzepte zur kohärent kompositionalen Entwicklung von Sprachen. Diese ermöglichen im Vergleich mit anderen Werkzeugen zur Sprachentwicklung eine wesentlich weitergehende Wiederverwendung von Artefakten aus anderen Sprachen. So verwendet eine Vielzahl mit MontiCore entwickelter Sprachen eine gemeinsame Implementierung von Literalen primitiver Datentypen oder ein gemeinsames Basis-Typsystem. Die Wiederverwendung solcher Komponenten ist dabei nicht auf einzelne Aspekte der Sprache beschränkt, sondern ist kohärent für konkrete und abstrakte Syntax, Symboltabellenaufbau, Überprüfung von Kontextbedingungen und teilweise auch Erstellung von Editoren sowie Codegeneratoren möglich.

MontiCore bietet somit bereits umfangreiche Unterstützung für die Entwicklung domänenspezifischer Sprachen sowie für die Verarbeitung der Modelle in diesen Sprachen. Für die Verarbeitung der Modelle kommt bisher hauptsächlich die General-Purpose-Sprache (Englisch: general purpose language, GPL) Java zum Einsatz. Für die Codegenerierung wird zusätzliche die Templatesprache FreeMarker [Fre11] verwendet.

Für modellgetriebene Entwicklungsprozesse ist aber oftmals die Verwendung spezieller Transformationssprachen zweckmäßig, die bisher in MontiCore nicht vorhanden waren. Differenziertere Betrachtungen zu Transformationen im Entwicklungsprozess und Transformationssprachen folgen in den Abschnitten 2.1, 2.2, 2.3 und 2.4. Grundsätzlich sind In-Place-Transformationen, die auf einem einzelnen Modell operieren, zu unterscheiden von Modell-zu-Modell-Transformationen, die separate Quell- und Zielmodelle haben, welche häufig auch in verschiedenen Sprachen vorliegen können. In-Place-Transformationen können allerdings als wichtiges Hilfsmittel bei der Implementierung von Modell-zu-Modell-Transformationen genutzt werden, wie etwa in [Kön09] beschrieben wird.

Der wesentliche Beitrag dieser Arbeit liegt daher in der Erweiterung von MontiCore um In-Place-Transformationen. Die Besonderheit des gewählten Ansatzes liegt in der vereinfachten Nutzbarkeit der Transformationssprachen für Domänenexperten, wie sie im folgenden Abschnitt zusammengefasst wird und im Abschnitt 2.5 noch weitergehend motiviert wird.

1.2. Wissenschaftliche Fragestellung dieser Arbeit

In den vorangegangenen Abschnitten wurde bereits erläutert, dass durch das MontiCore-Rahmenwerk die effiziente Entwicklung domänenspezifischer Modellierungssprachen ermöglicht wird. Diese Sprachen erlauben es, Domänenexperten in den Entwicklungsprozess eines Softwareproduktes mit einzubeziehen. Eine wichtige Ergänzung zu Modellierungssprachen stellen Modelltransformationssprachen dar, die bisher aber für Domänenexperten schwer verständlich und somit nur eingeschränkt nutzbar sind und in agilen Softwareentwicklungsprozessen nicht angewendet werden.

Durch die Einführung domänenspezifischer Transformationssprachen ließe sich die Verständlichkeit und Nutzbarkeit von Modelltransformationen für Domänenexperten verbessern. Eine Verwendung domänenspezifischer Sprachen in agilen Prozessen würde dann eine leichtgewichtige Nutzung der DSLs, eine leichtgewichtige Nutzung der Transformationssprache, sowie eine flexible und schnelle Anpassbarkeit der DSL und der Transformationssprache erfordern. Von der Ähnlichkeit zwischen der DSL und der zugehörigen Transformationssprache könnten neben den

Domänenexperten auch Programmierer, Software-Architekten weitere an der Produktentwicklung beteiligte Personen profitieren.

Der Aufwand, parallel zu einer Modellierungssprache in einem losgelösten Entwicklungsprozess eine geeignete Transformationssprache zu erstellen und zu warten, ist jedoch beträchtlich. Ein Ansatz, mit dem sich eine solche Transformationssprache aus bestehenden Dokumenten des DSL-Entwicklungsprozesses generieren lässt, würde diesen Aufwand auf ein Minimum reduzieren. So könnte ein erheblicher Mehrwert für Nutzer domänenspezifischer Sprachen geschaffen werden, der nur mit sehr geringem zusätzlichen Aufwand bei der Sprachentwicklung verbunden wäre.

Die zentrale Fragestellung, die in dieser Arbeit beantwortet werden soll, lautet daher:

Wie lässt sich aus der Beschreibung einer textuellen domänenspezifischen Sprache S in einer Grammatik automatisiert eine Transformationssprache T ableiten, die auch für Domänenexperten verständlich ist, und die ihnen durch syntaktische Ähnlichkeit mit S möglichst vertraut ist?

Die Generierung domänenspezifischer Transformationssprachen ist dabei an Voraussetzungen geknüpft oder lässt sich durch Erweiterungen ergänzen, auf die in separaten Kapiteln eingegangen wird. So werden im Rahmen dieser Arbeit auch eine Transformationsengine zur Ausführung von Regeln in generischer Notation (Kapitel 4) und Kontrollflussanteile von Transformationssprachen (Kapitel 6) beschrieben, die jeweils nicht domänenspezifisch sind. Bei der Erstellung dieser Komponenten konnte auf viele Erfahrungen und Publikationen aus anderen Projekten aufgebaut werden, die sich ebenfalls mit Modelltransformationen befassen. Die zentrale Neuerung dieser Arbeit liegt also in der Generierung der Transformationssprache mit einer domänenspezifischen Notation.

1.3. Wesentliche Ergebnisse

Diese Arbeit liefert die folgenden wesentlichen wissenschaftlichen Ergebnisse, von denen die wichtigsten auch deutlich vereinfacht in Abbildung 1.1 dargestellt sind:

- Zu einer gegebenen textuellen Modellierungssprache lässt sich eine dazu passende Sprache für Transformationsregeln angeben. Diese Sprache hat eine der Modellierungssprache ähnliche konkrete Syntax. Sie ist somit für Experten der Modellierungssprache wesentlich verständlicher als eine generische, auf der abstrakten Syntax der Modelle basierende Sprache.

- Domänenspezifische Transformationssprachen lassen sich für viele Sprachen, die mit Hilfe des Werkzeugs MontiCore definiert wurden, automatisiert erstellen, also generieren. Automatisiert erstellt werden sowohl die Syntax der Transformationssprache als auch ein Codegenerator, der die Regeln in der Sprache ausführbar macht.

- Diese Codegeneratoren können als Zielsprache eine herkömmliche, auf der abstrakten Syntax basierende Transformationssprache verwenden. Diese kann wiederum mit Hilfe

eines zweiten Codegenerators nach Java übersetzt werden. Durch den mehrstufigen Generierungsprozess wird die Komplexität zwischen den beiden Codegeneratoren aufgeteilt und somit insgesamt reduziert.

- Die Transformationsregeln lassen sich in eine übergeordnete Kontrollflusssprache einbetten, sodass mehrere Transformationen zu komplexen Programmen verbunden werden können und ihr Ablauf einfacher gesteuert werden kann. Der Code beziehungsweise die Konfiguration dieser Einbettung lässt sich für generierte Regelsprachen ebenfalls automatisiert aus der zugrunde liegenden Modellierungssprache ableiten.

- Zur Erstellung aller in dieser Arbeit entstandenen Komponenten wurden zwei Methoden eingesetzt, die beide eine agile und insbesondere testgetriebene Entwicklung von Sprachen und Generatoren ermöglichen. Die erste Methode wurde im initialen Aufbau der hier entstanden Projekte eingesetzt, während die zweite, hieraus hervorgegangene Methode besser für die aktuelle Weiterentwicklung und Wartung geeignet ist.

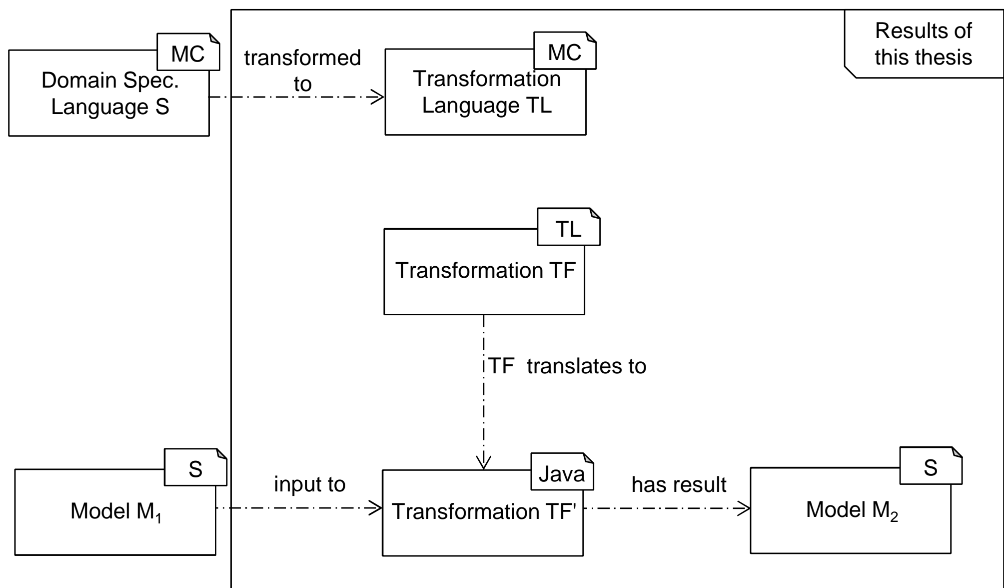


Abbildung 1.1.: Modelle, Transformationen, Sprachen und Generierung in dieser Arbeit

1.4. Aufbau der Arbeit

Diese Arbeit ist in zehn Kapitel gegliedert, die wie folgt aufgebaut sind:

In *Kapitel 1, Einleitung*, wurde bis zu diesem Punkt das Dissertationsvorhaben in die Forschungsziele des Lehrstuhls Software Engineering und den Kontext des MontiCore-Projektes eingeordnet sowie die wissenschaftliche Fragestellung, die in dieser Arbeit geklärt wird, vorgestellt. Außerdem wurde bereits ein Ausblick auf die wichtigsten Ergebnisse der Arbeit gegeben.

Das *Kapitel 2, Anwendung und Entwicklung von Modelltransformationen*, gibt einen Überblick über die methodische Verwendung domänenspezifischer Sprachen, modellgetriebener Entwicklung und Modelltransformationen, und es enthält eine Einführung in die aktuellen wissenschaftlichen Fragestellungen für Modelltransformationen, insbesondere in die in der vorliegenden Arbeit behandelten Themen.

Ein durchgängiges Beispiel für die folgenden Ausführungen wird in *Kapitel 3, Durchgängiges Beispiel: Vereinfachung hierarchischer Statecharts*, vorgestellt. Dieses Beispiel behandelt semantikerhaltende Transformationen auf hierarchischen Statecharts, die zur Vorbereitung eines Codegenerierungsprozesses eingesetzt werden können.

In *Kapitel 4, Transformationsregeln in Objektdiagramm-Notation*, wird eine generische, auf Objektdiagrammen basierende Sprache für Transformationsregeln vorgestellt. Dazu werden zunächst die kontextfreie Syntax der Sprache und Kontextbedingungen erläutert. Anschließend wird die Übersetzung der Transformationsregeln nach Java präsentiert. Hierbei liegt der Schwerpunkt auf effizientem Pattern Matching, das durch einen suchplangesteuerten Algorithmus umgesetzt wurde, sowie der Verwendung des Command-Entwurfsmusters, das für das in Kapitel 6 vorgestellte Backtracking erforderlich ist.

Das *Kapitel 5, Eine Regelsprache mit domänenspezifischer Syntax*, beschreibt eine domänenspezifischer Sprache für Modelltransformationsregeln. Aufbauend auf dem Beispiel hierarchischer Statecharts wird vorgestellt, wie die kontextfreie Syntax einer solchen Sprache erstellt und aus der Grammatik der Statechartsprache systematisch abgeleitet werden kann. Nach der Erläuterung von Kontextbedingungen für die Transformationsregeln folgt die Beschreibung der Codegenerierung. Diese basiert auf einem Transformator, der Regeln in konkreter Syntax in die in Kapitel 4 vorgestellte Objektdiagramm-Notation übersetzt.

In *Kapitel 6, Die Kontrollflusssprache für Modelltransformationen*, wird eine Kontrollflusssprache vorgestellt, die eingesetzt wird, um Transformationsregeln, die in den jeweiligen Regelsprachen vorliegen, zu komplexen Programmen zu komponieren. Erläutert werden zunächst die kontextfreie Syntax der Sprache, die Einbettung von Transformationsregeln in die Kontrollflusssprache und Kontextbedingungen. Im Folgenden werden die nichtdeterministischen Kontrollstrukturen sowie der Ansatz zum Backtracking vorgestellt.

Kapitel 7, Generierung von Transformationssprachen aus DSL-Grammatiken, behandelt die automatische Generierung einer domänenspezifischer Transformationssprache zu einer gegebenen DSL. Wiederum am Beispiel hierarchischer Statecharts wird erläutert, wie sich eine Regelsprache in konkreter Syntax mithilfe des MontiCore-Frameworks vollautomatisch aus der zugrunde liegenden Sprache generieren lässt. Dabei wird nicht nur die Syntax der generierten Sprache, sondern auch ein Transformator, der Regeln in die Objektdiagramm-Notation übersetzen und so letztlich ausführbar machen kann, aus der Grammatik der DSL generiert.

In *Kapitel 8, Nutzung von Transformationen und Transformationssprachen*, werden die Methodik zur Generierung von Transformationssprachen am Beispiel der Statechart-Transformationssprache und die Methodik zur Nutzung konkreter Transformationen am Beispiel ihrer Einbindung in einen Codegenerierungsprozess vorgestellt. Dieses Kapitel beschreibt also die Nutzung von Modelltransformationen durch den Anwender.

In *Kapitel 9, Die Entwicklung der Modelltransformationsengine*, wird die Methodik vorgestellt, die zur Umsetzung des in dieses Arbeit beschriebenen Transformationsansatzes entwickelt

und angewendet wurde. Besonderes Augenmerk wird dabei auf die testgetriebene Entwicklung und Qualitätssicherung von Generatoren gelegt.

Kapitel 10, Epilog, gibt schließlich eine abschließende Zusammenfassung der Arbeit und erläutert offene Fragestellungen, die aufbauend auf den Ergebnissen dieser Arbeit untersucht werden können.

Kapitel 2.

Anwendung und Entwicklung von Modelltransformationen

Die Entwicklung und Verbesserung von Ansätzen zur Modelltransformation ist in einer Vielzahl von Projekten Gegenstand der Forschung. Aufbauend auf einem Überblick über den Einsatz von Modellen und domänenspezifischen Sprachen in der Softwareentwicklung gibt dieses Kapitel einen Überblick über die bestehenden Ansätze. Im Anschluss werden die aktuellen wissenschaftlichen Fragestellungen zu Modelltransformationen herausgearbeitet, und die vorliegende Arbeit wird diesbezüglich eingeordnet, motiviert und mit verwandten Ansätzen verglichen.

2.1. Modelle und domänenspezifische Sprachen im Software-Engineering

Die modellgetriebene Entwicklung hat sich in den letzten Jahren als wichtiger Ansatz zur effizienten Erstellung qualitativ hochwertiger Softwaresysteme etabliert. Neben vergleichsweise selten anzutreffenden rein modellgetriebenen Entwicklungsprozessen ist vor allem eine Kombination modellgetriebener Ansätze mit objektorientierter Programmierung verbreitet [Fow10, Kap. 8, 27, 54–57].

2.1.1. Einsatz domänenspezifischer Sprachen

Bei der Beschreibung von Modellen lassen sich General-Purpose-Modellierungssprachen von domänenspezifischen Sprachen unterscheiden. *General-Purpose-Sprachen* eignen sich zur Beschreibung eines breiten Spektrums an Softwaresystemen, deren Einsatzbereich in beliebigen Anwendungsdomänen liegen kann. *Domänenspezifische Sprachen* dienen dagegen der Beschreibung von Softwaresystemen oder einzelner Aspekte solcher Systeme in einer bestimmten fachlichen oder technischen Domäne. Viele dieser Sprachen sind nicht ausführbar, sondern dienen der Spezifikation oder Strukturmodellierung. Aber auch die ausführbaren DSLs sind oft nicht Turing-vollständig und somit weniger ausdrucksmächtig als GPLs. Sie sind jedoch auf die Bedürfnisse der Anwender in der jeweiligen Domäne zugeschnitten und erlauben es, bestimmte Eigenschaften oder Vorgänge effizient und präzise zu beschreiben. Gelegentlich kommen domänenspezifische Sprachen können innerhalb der Domäne auch unabhängig von der Softwareentwicklung zum Einsatz. Sie sind auch für Domänenexperten nutzbar, die nur über grundlegende Programmierfähigkeiten und Kenntnisse in der Softwareentwicklung verfügen. Gerade

die Einbeziehung solcher Domänenexperten in den Softwareentwicklungsprozess stellt einen erheblichen Mehrwert dar, denn sie macht das Fachwissen aus der jeweiligen Domäne für die Softwareentwicklung besser nutzbar.

Modelle und Dokumente in domänenspezifischen Sprachen lassen sich in verschiedenen Phasen des Softwareentwicklungsprozesses einsetzen. So sind in der Anforderungsanalyse UML-Aktivitätsdiagramme [OMG10c] ein verbreitetes Beschreibungsmittel für Prozessabläufe. Im Systementwurf sind Architekturbeschreibungssprachen wie ACME [GMW97] verbreitet. Für die Implementierung von Steuergerätesoftware im Automobilbereich hat sich Matlab/Simulink [MLSL] zum de-facto-Standard entwickelt [SKSS07]. Des Weiteren existiert eine Reihe von Sprachen zur formalen Spezifikation [OMG10a], ebenso zur Validierung [GHR+03] und Verifikation [Jac02].

2.1.2. Standardisierungsansätze zur Modellierung

Mit der *Unified Modeling Language* (UML), deren Version 1.0 im Jahr 1997 durch die OMG veröffentlicht wurde, hat ein Konsortium aus Softwareherstellern und Forschungseinrichtungen den Versuch unternommen, einen Standard für Modellierung im Software Engineering zu etablieren. Die 2007 verabschiedete UML 2, die inzwischen in Version 2.3 vorliegt, ist inzwischen zu einer in Forschung, Lehre und Industrie weit verbreiten Modellierungssprache beziehungsweise Sammlung von Modellierungssprachen geworden.

Neben einer einheitlichen Modellierungssprache schlägt die OMG ebenfalls eine standardisierte Methodik für modellgetriebene Entwicklunsgprozesse vor [OMG03]. Die *Model Driven Architecture* (MDA) sieht eine strikte Trennung der technologieunabhängigen Spezifikation der Funktionalität eines Systems von seinen technischen Eigenschaften vor. Die funktionalen Eigenschaften werden technologieunabhängig in einem *Computation Independent Model* (CIM) beschrieben. Aus dem CIM entsteht in einem mehrstufigen Prozess durch Anreicherung mit technischen Informationen zunächst ein *Platform Independent Model* (PIM) und anschließend ein *Platform Specific Model* (PSM). Die Generierung des PIM und PSM aus den vorgelagerten Modellen und den technikspezifischen Anteilen wird mit Hilfe von *Modelltransformationen* durchgeführt. Dieser Transformationsschritt von PIM nach PSM kann in mehrere Schritte unterteilt werden, um sukzessive weitere plattformspezifische Informationen hinzuzufügen [AK05]. Das Ergebnis eines Transformationsschrittes dient dann im folgenden Schritt als Eingabemodell. Am Ende dieses Prozesses steht ein PSM, das alle für die Erstellung des Systems relevanten Informationen enthält. Es ist somit als Implementierung des Systems zu verstehen.

Mit der Anreicherung plattformunabhängiger Modelle um technische Informationen sind Modelltransformationen durch die Model Driven Architecture bereits grundlegend motiviert. Entsprechend existiert von Seiten der OMG auch ein Standard für Modelltransformationen, der in der *Meta Object Facility Query/View/Transformation* (MOF QVT oder abkürzend QVT) enthalten ist [OMG08b]. Die im QVT-Standard beschriebenen Transformationssprache ist unterteilt in den Kern-Anteil MOF QVT Core, den deklarativen Teil MOF QVT Relational, der sich auf den Kernteil semantikerhaltend abbilden lässt, und den imperativen Anteil MOF QVT Operational.

Der QVT-Standard deckt nur einen Teil der möglichen Einsatzgebiete von Modelltransformationen ab. Bei QVT handelt es sich um eine hybride Sprache zur Beschreibung exogener Modelltransformationen. Exogene Transformationen zeichnen sich dadurch aus, dass Quell-

und Zielmodell nicht notwendigerweise in der selben Sprache beschrieben sind. Eine hybride Transformationssprache stellt eine Kombination aus deklarativen Elementen, die ohne Kontrollstrukturen auskommen, und imperativen Elementen dar. QVT wurde im Hinblick auf die Implementierung von MDA-Prozessen entwickelt. Daneben existiert aber eine Fülle weitere Anwendungen für Modelltransformationen. Auf diese Anwendungen sowie verschiedene Arten von Transformationssprachen, die sich neben den oben genannten Kriterien auch nach vielen weiteren Gesichtspunkten kategorisieren lassen, wird in Abschnitt 2.3 ausführlicher eingegangen. Dem breiten Anwendungsspektrum entsprechend existiert bereits eine Vielzahl verschiedener Transformationsansätze. Neben QVT gibt es hier jedoch bisher keine weiteren Standards, die allgemeine Akzeptanz gefunden haben.

2.1.3. Grafische und textuelle Sprachen

Bei den Modellierungssprachen lassen sich zunächst grafische Sprachen von textuellen unterscheiden. Mischformen textueller und grafische Sprachen sind ebenfalls möglich. So existiert für die UML/P sowohl eine textuelle als auch eine grafische Notation [Rum11, Rum04], und Modelle in der grafischen UML, wie sie von der OMG beschrieben wurde, können um textuelle Bedingungen (Englisch: constraints) in der Object Constaint Language (OCL) [OMG10a] angereichert werden.

Auch domänenspezifische Sprachen existieren sowohl in textueller als auch in grafischer Ausprägung. Die Entscheidung zwischen einer textuellen und einer grafischen Notation bei der Entwicklung einer domänenspezifischen Sprache hängt von verschiedenen Faktoren ab [Pet95]. Unter anderem sind hier die in der Sprache zu beschreibenden Artefakte, die Präferenzen der zukünftigen Nutzer, die Einbindung in bestehende Prozesse und Werkzeuglandschaften sowie technische Faktoren zu berücksichtigen. Grafische Modellierungssprachen stehen beispielsweise in dem Ruf, dass ihre Modelle schneller zu erfassen sind. Allerdings gibt es auch hier Gegenbeispiele, wie etwa an der aussagenlogischen Formel in Abbildung 2.1 zu sehen ist.

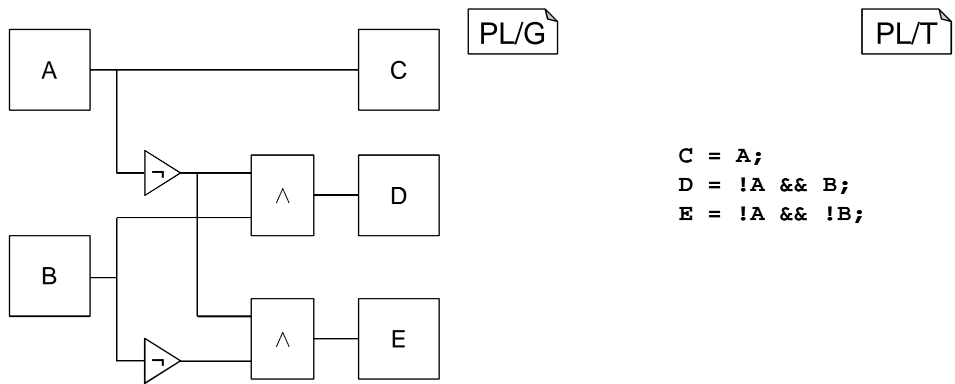


Abbildung 2.1.: Textuelle und grafische Notation aussagenlogischer Formeln

In den folgenden Kapiteln dieser Arbeit werden vor allem textuelle Modelle und Modellierungssprachen behandelt. Für die Fokussierung auf textuelle Sprachen gab es die folgenden Gründe:

- Textuelle Sprachen, vor allem Modellierungssprachen und die Programmiersprache Java, spielen im Umfeld der vorliegenden Arbeit eine herausragend wichtige Rolle. So können die hier vorgestellten Ansätze und Ergebnisse beispielsweise auf die textuelle Variante der UML/P [Sch12] oder die Architekturbeschreibungssprachen MontiArc und AJava [HRR10] angewendet werden.
- Die Implemtierungsergebnisse zu dieser Ausarbeitung sind in das Rahmenwerk MontiCore [Kra10, Völ11] eingebunden, das hauptsächlich für die Definition textueller Sprachen entwickelt wurde und eingesetzt wird.
- Die bestehenden Ansätze zur Definition grafischer und textueller konkreter Syntax unterscheiden sich erheblich. Ein Transformationsansatz, der sowohl grafische als auch textuelle konkrete Syntax reflektiert, erscheint daher wesentlich komplexer als ein Ansatz, der sich rein auf textuelle Sprachen konzentriert.

2.1.4. Ansätze zur Sprachdefinition

Seit der Entwicklung der ersten höheren Programmiersprachen in den 1950er Jahren wurde eine inzwischen nicht mehr zu überblickende Anzahl von Programmier- und Modellierungssprachen entwickelt.

In Analogie zu natürlichen Sprachen haben formale Sprachen, die durch Softwaresysteme verarbeitet werden können, Syntax und Semantik [HR04, Rum98]. Bei der Syntax wird zwischen konkreter und abstrakter Syntax unterschieden, sodass sich insgesamt drei Grundbestandteile einer formalen Sprache ergaben [Kle07]:

- Die konkrete Syntax definiert die gültigen Wörter einer Sprache und ihre Repräsentation gegenüber dem Benutzer.
- Die abstrakte Syntax beschreibt die rechnerinterne Darstellung der Wörter einer Sprache.
- Die Semantik einer Sprache beschreibt die Bedeutung der gültigen, also syntaktisch korrekten, Wörter.

Bedingt durch die Vielzahl und Vielfalt existierender Sprachen haben sich auch verschiedene Ansätze zur Sprachdefinition entwickelt. Im Bereich der Syntax sind vor allem metamodellbasierte und grammatikbasierte Ansätze verbreitet. Verbreitete Verfahren zur Semantikdefinition sind denotationale, operationale und translationale Semantiken. Die verschiedenen Ansätze zur Semantikdefinition sind jedoch für das Verständnis dieser Arbeit nur von untergeordneter Bedeutung. Für weiterführende Informationen hierzu wird daher auf die entsprechende Literatur verwiesen, etwa [SK95].

Abbildung 2.2 zeigt den Unterschied zwischen konkreter und abstrakter Syntax am Beispiel eines einfachen Automaten. Teil a) zeigt den Automaten in einer grafischen konkreten Syntax, Teil b) den selben Automaten in einer textuellen konkreten Syntax. Teil c) zeigt die abstrakte Syntax dieses Automaten als Objektdiagramm.

Grammatikbasierte Ansätze zur Syntaxdefinition verwenden Grammatiken zur Beschreibung der konkreten und abstrakten Syntax einer Sprache. Häufig ist die Syntaxdefinition in reguläre

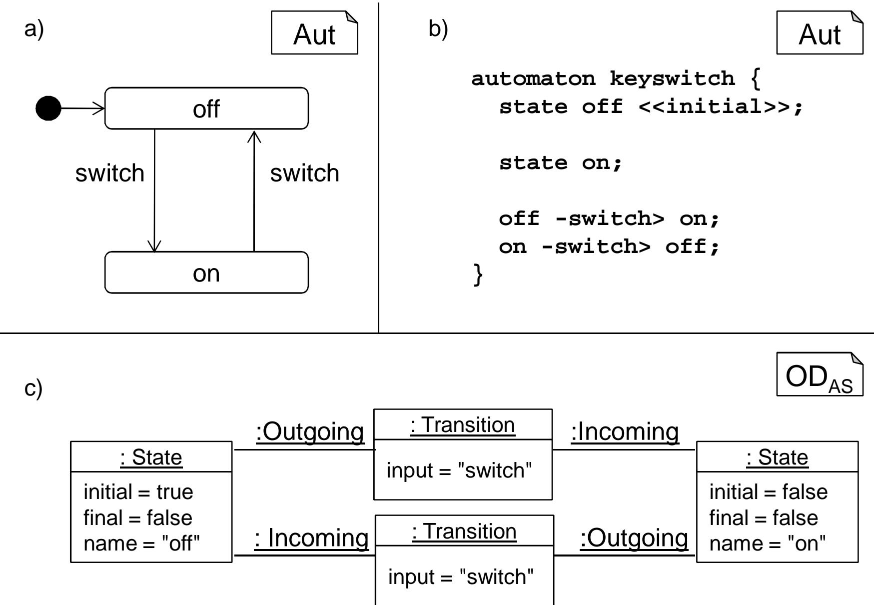


Abbildung 2.2.: Konkrete und abstrakte Syntax eines endlichen Automaten

Lexerproduktionen und kontextfreie Parserproduktionen unterteilt [ASU86]. Die im erstgenannten Teil definierten Lexeme dienen dann als Terminalsymbole in den Parserproduktionen. Aus den Lexer- und Parserproduktionen lassen sich mit compilererzeugenden Werkzeugen automatisiert die konkrete und abstrakte Syntax einer Sprache ableiten [ASU86]. Bekannte, diesem Ansatz folgende Werkzeuge sind lex/yacc [LMB92], Bison [Bis11] und ANTLR [ANT11]. Auch MontiCore ist ein Werkzeug zur grammatikbasierten Sprachdefinition, wobei hier als Besonderheit Lexer- und Parserproduktionen in der selben Grammatik spezifiziert werden können.

Metamodellbasierte Ansätze verwenden ein Modell zur Definition der abstrakten Syntax einer Sprache, das Metamodell. Dieses Modell definiert nur die abstrakte Syntax der Sprache. Zur Definition der konkreten Syntax kommen häufig Editorrahmenwerke zum Einsatz, die die Auswahl einer Repräsentation für jedes Element der abstrakten Syntax ermöglichen. Bekannte Werkzeuge, die einen metamodellbasierten Ansatz verfolgen, sind beispielsweise MetaEdit [Met11], EMF/GMF [BSM+03, GMF11] und MOFLON [AKRS06, WKS10].

Ergänzt werden sowohl die Grammatiken als auch die Metamodelle in der Regel durch Kontextbedingungen für die Sprache. Hier finden sich etwa Bedingungen zur Eindeutigkeit von Bezeichnern, Einschränkungen, die sich aus dem Typsystem der Sprache ergeben, oder Bedingungen, die sich in einer Grammatik oder einem Metamodell nicht komfortabel ausdrücken lassen, wie etwa die Widerspruchsfreiheit von Modifikatoren. Zur Formulierung der Kontextbedingungen wird oft eine Constraintsprache wie OCL oder eine Programmiersprache eingesetzt. Bei textuellen Sprachen werden häufig Symboltabellen oder Attributgrammatiken als zusätzliche Infrastruktur verwendet, wodurch die Überprüfung der Kontextbedingungen wesentlich vereinfacht wird. Für Sprachdefinitionen in MontiCore kommt Java zur Beschreibung der Kontext-

bedingungen zum Einsatz. Die Berechnung von Symboltabellen und Attributgrammatiken wird ebenfalls unterstützt.

Grammatikbasierte Ansätze haben sich zur Definition textueller Sprachen durchgesetzt. Ihr wesentliche Vorteil liegt in der automatischen Ableitung der konkreten Syntax aus der Grammatik, die für Metamodelle nicht oder nur ohne sprachspezifischen „syntactic sugar“ möglich ist. Für grafische Sprachen kommen dagegen meist metamodellbasierte Ansätze zum Tragen. Diese haben den Vorteil, dass Modelle in der abstrakten Syntax nicht als Bäume, sondern als attributierte Graphen repräsentiert werden, was die automatisierte Verarbeitung von Programmen und Modellen sowie die Anbindung von Editorframeworks in vielen Fällen vereinfacht. Nichtsdestotrotz ist die Definition grafischer Sprachen auf Basis einer Grammatik oder die Metamodellierung textueller Sprachen grundsätzlich möglich. Tabelle 2.3 listet die Gemeinsamkeiten und Unterschiede der beiden Ansätze auf.

Sprachdefinition	**metamodellbasiert**	**grammatikbasiert**
Primäre Beschreibungsmittel	Klassen, Kompositionen, Assoziationen	Nichtterminale, Terminale, Produktionen, in MontiCore auch Assoziationen
Weitere Bedingungen	abgeleitete Attribute und Constraints, z.B. OCL	Symboltabellen, Attributgrammatiken und Kontextbedingungen
Abstrakte Syntax	getypter Graph	getypter oder ungetypter Baum, in MontiCore getypter Graph mit spannendem Baum
Verbindung der Modellelemente	explizite Links	Bezeichnerbindung, in MontiCore optional auch explizite Links

Tabelle 2.3.: Vergleich metamodell- und grammatikbasierter Sprachdefinitionen

Bisher noch nicht erwähnt wurden die Unterschiede im Vorgehen bei der Syntaxdefinition: Während bei grammatikbasierten Ansätzen der Sprachentwickler typischerweise zunächst die konkrete Syntax der Sprache erstellt, beginnt man bei metamodellbasierten Ansätzen häufig mit der abstrakten Syntax der Sprache.

Eine weitere Besonderheit in MontiCore, die ebenfalls der Tabelle 2.3 zu entnehmen ist, ist die Verwendung von Assoziationen innerhalb der Grammatik. Diese erlaubt das Einfügen von Links zwischen den Knoten im abstrakten Syntaxbaum (AST), sodass ein abstrakter Syntaxgraph entsteht, in dem der AST einen Spannbaum darstellt.

2.2. Methodik der Nutzung von Modelltransformationen

Für die Verarbeitung von Modellen spielen – unabhängig davon, ob es sich um grafische oder textuelle Modelle handelt und wie die Modellierungssprache definiert wurde – Transformationen eine wichtige Rolle. Als Spezialfall von Modelltransformationen können Transformationen auf

Programmen in einer Programmiersprache angesehen werden; häufig ist die Abgrenzung zwischen Modell und Programm auch unscharf, beispielsweise bei Beschreibungen von Prozessen in Workflowsprachen.

Als *Modelltransformationen* werden im Folgenden in Anlehnung an [CH06] solche Programme bezeichnet, die eines oder mehrere Modelle erzeugen oder verändern. Diese Modelle werden als Zielmodelle bezeichnet. Die Eingabe einer Transformation wird dagegen als Quellmodell bezeichnet. Häufig, aber nicht notwendigerweise, hat eine Transformation genau ein Quell- und ein Zielmodell.

Modelltransformationen können in der modellbasierten oder modellgetriebenen Entwicklung auf vielfältige Weise eingesetzt werden. So können sie als abstrahierende Transformationen Informationen aus einem Modell entfernen, als verfeinernde Transformationen das Modell mit zusätzlichen Informationen anreichern, oder als semantikerhaltende Transformationen die Informationen erhalten, aber syntaktisch anders ausdrücken.

2.2.1. Nutzungsszenarien

Transformationen laufen meist innerhalb eines Werkzeugs ab, das von einem Modellierer oder Programmierer genutzt wird. Die Sichtbarkeit der Transformationen und ihre Beeinflussbarkeit durch den Nutzer kann jedoch stark variieren. Von Transformationsabläufen, die vor dem Benutzer verborgen sind, bis hin zur Ausführung vom Benutzer erstellter Transformationen sind vielfältige Einsatzszenarien denkbar, die im Folgenden erläutert werden.

Nutzung innerhalb von Werkzeugen

Ein erheblicher Anteil der Transformationen, die in der Praxis eingesetzt werden, läuft innerhalb eines Anwendungsprogramms ab, möglicherweise unbewusst für den Benutzer. Typische Beispiele hierfür sind Optimierungen, die Compiler auf Programmcode oder Datenbankanfragen vornehmen. Abbildung 2.4 zeigt die typische Nutzung in einem solchen Szenario.

Dabei erstellt der Nutzer zunächst ein Dokument d und übergibt dieses dann zur weiteren Verarbeitung an ein Werkzeug, beispielsweise einen Compiler [ASU86]. Dieser kann vor der Transformation andere Aktionen ausführen, zum Beispiel Analysen des Eingabemodells. Am Ende dieser Aktionen steht das Dokument d', das nicht notwendigerweise von d verschieden sein muss und als Eingabe der Transformation dient. Die Transformation, beispielsweise die Zwischencodeoptimierung im Compiler, erzeugt aus dem Eingabedokument d' ein Ausgabedokument d''. Dieses Dokument kann innerhalb des Werkzeugs weiter verarbeitet werden. Am Ende dieser Verarbeitung steht das Dokument d''', im Beispiel des Compilers der erzeugte Maschinencode. Dieses Dokument wird an den Benutzer übergeben, der es dann weiter verarbeitet, zum Beispiel den kompilierten Code ausführt oder an einen Endanwender ausliefert.

Verwendung parametrierbarer Transformationen

Ein weiteres Szenario der Nutzung von Transformationen, in dem ein Benutzer eine Transformation explizit aufruft und an diese Transformation nicht nur Eingabedokumente, sondern auch

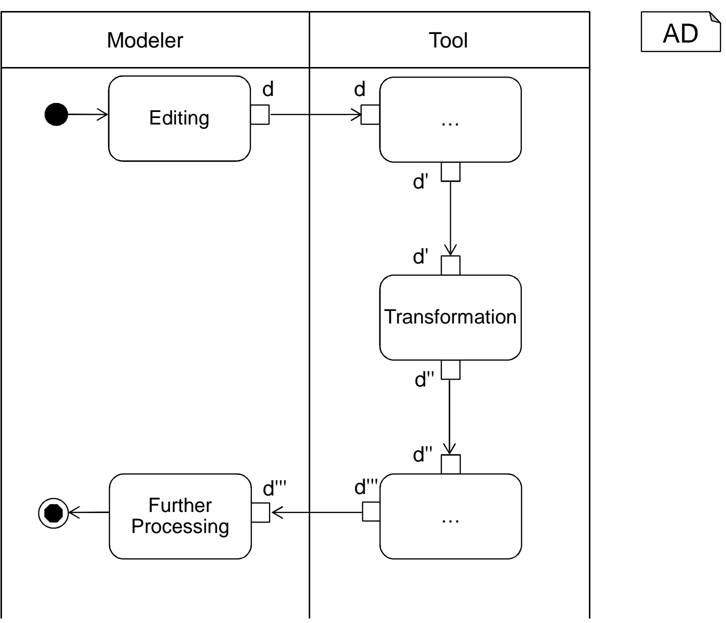


Abbildung 2.4.: Transformationen innerhalb eines Werkzeugs

weitere Parameter übergeben kann, ist in Abbildung 2.5 dargestellt. Ausprägungen dieses Szenarios finden sich etwa bei Refactoring-Operationen in integrierten Entwicklungumgebungen, zum Beispiel der Einführung einer Konstante für ein Zeichenketten-Literal.

Der Nutzer erstellt in diesem Szenario ebenfalls zunächst ein Dokument, im genannten Beispiel den Programmcode. Will er in diesem Programm eine Konstante für ein Zeichenketten-Literal einführen, so hat er zunächst verschiedene Parameter pm dieser Transformation anzugeben. Hierunter fällt das konkrete Literal im Eingabedokument d, ein Name für die Konstante, sowie deren Sichtbarkeit. Unter Berücksichtigung der übergebenen Argumente transformiert das Werkzeug das Eingabedokument in die Ausgabe d', die im Folgenden vom Benutzer weiter verarbeitet wird.

Aufgrund der erforderlichen Benutzereingaben sind solche Transformationen zur Verwendung in automatisierten Abläufen wie der Codegenerierung nicht geeignet. Sie können jedoch innerhalb von Editoren oder im Rahmen interaktiver Arbeitsschritte verwendet werden.

Benutzerdefinierte Transformationen

Wiederum anders verläuft die Nutzung von Transformation im Szenario, das in Abbildung 2.6 gezeigt wird. Hier ist der Nutzer des Werkzeugs gleichzeitig derjenige, der die Transformationen entwickelt.

Nach der Erstellung des Modells oder Programms im Dokument d erstellt der Benutzer eine Transformationsregel t in einer geeigneten Sprache. Alternativ kann er eine Transformationsregel auswählen, die er bereits zuvor erstellt hat. Diese Transformation kann zusätzlich wie für Abbildung 2.5 erläutert parametrisiert werden, in Abbildung 2.6 ist dieser Schritt jedoch nicht

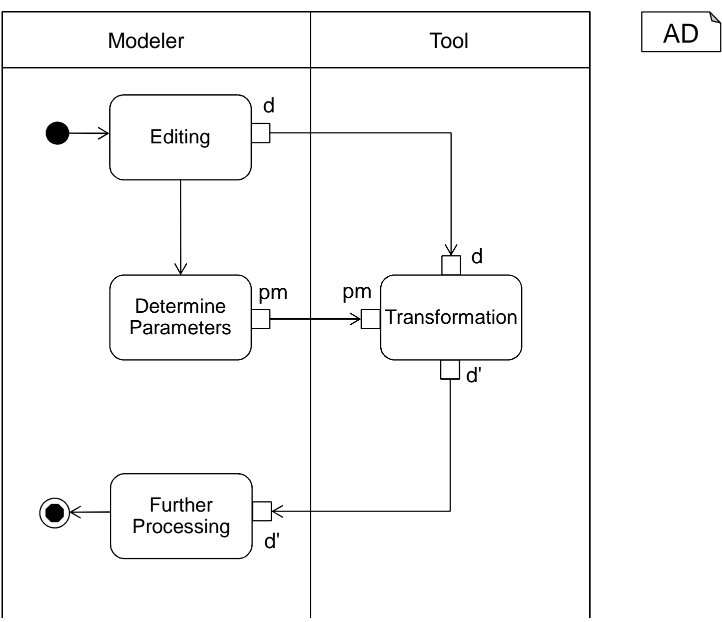


Abbildung 2.5.: Aufruf interaktiver Transformationen

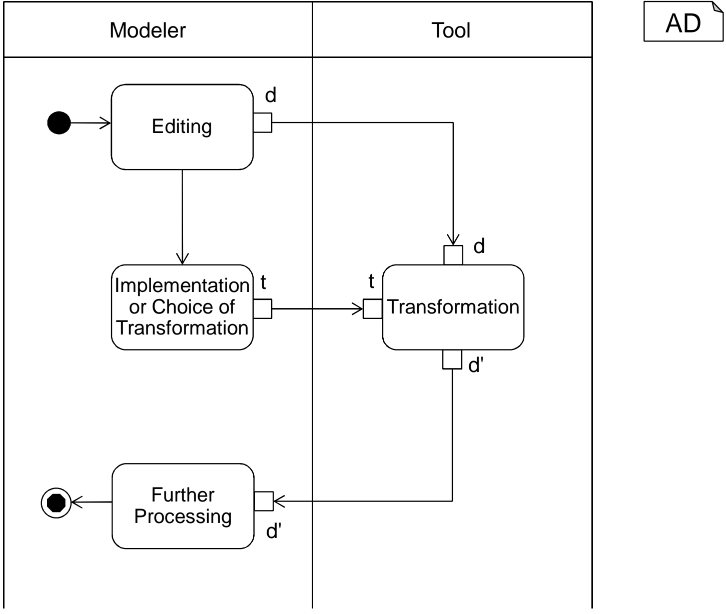


Abbildung 2.6.: Erstellung und Aufruf von Transformationen durch den Nutzer

aufgeführt. Durch die Ausführung der Transformation entsteht das Dokument d', das vom Benutzer weiter verarbeitet werden kann.

Diese Art der Transformationen ist für die vorliegende Arbeit besonders relevant, da sie voraussetzt, dass der Modellierer oder Programmierer, der die Transformationen selbst erstellt, die entsprechende Transformationssprache beherrscht. Ein Beispiel für die in Abbildung 2.6 gezeigte Nutzungsform ist die Verwendung von Makros innerhalb von Tabellenkalkulationen. Aber auch im Bereich der Softwaremodellierung finden sich entsprechende Beispiele. Stellvertretend sei hier die Makro-Engine des UML-Werkzeugs Magic Draw genannt [MD11]. Alternativ zur Programmierung können Makros häufig auch mit einem Rekorder aufgezeichnet werden, sind dann aber nicht parametrierbar und somit kaum wiederverwendbar.

Folglich bietet die Implementierung von Transformationen in einer geeigneten Sprache hier erhebliche Vorteile. Daher wird im Folgenden darauf eingegangen, inwieweit Transformationssprachen für Transformationsnutzer verständlich und nutzbar gestaltet werden können.

2.2.2. Nutzbarkeit von Transformationssprachen für Domänenexperten

Die Nutzbarkeit einer Transformationssprache hängt nicht nur von der Sprache selbst, sondern auch von den Fähigkeiten und Präferenzen ihrer Nutzer ab [SK03]. Durch das Zuschneiden einer Sprache auf die Bedürfnisse der Anwender lässt sich folglich die Benutzbarkeit dieser Sprache verbessern.

Im Bereich der Transformationssprachen sind somit zunächst Transformationssprachen, die nicht auf eine bestimmte Domäne oder auf einen bestimmten Nutzerkreis zugeschnitten wurden, zu unterscheiden von solchen Sprachen, die im Hinblick auf die Benutzung in einem bestimmten Anwendungsgebiet oder durch eine bestimmte Personengruppe entworfen wurden. In Analogie zur Unterscheidung von General-Purpose- und domänenspezifischen Programmier- beziehungsweise Modellierungssprachen lassen sich hier *General-Purpose-Transformationssprachen* und *domänenspezifische Transformationssprachen* voneinander unterscheiden.

General-Purpose-Transformationssprachen sind auf Modelle oder Programme in einer Vielzahl von Sprachen anwendbar. Sie verwenden zur Beschreibung der Transformationen jedoch eine einheitliche, von der Sprache der transformierten Modelle unabhängige Notation, wie etwa ATL [JK05] und VIATRA2 [VB07]. Diese Notation bezieht sich auf die abstrakte Syntax der transformierten Modelle, zum Beispiel durch Objektdiagramme der abstrakten Syntax. Sie unterscheidet sich dadurch jedoch meist erheblich von der Notation der zu transformierenden Artefakte, sodass sie für Domänenexperten nur schwer verständlich ist. Der Vorteil dieser Sprachen liegt in der universellen Einsetzbarkeit: Eine existierende General-Purpose-Transformationssprache kann ohne Anpassung auf Modelle in einer neuen Modellierungssprache angewendet werden.

Hiervon abgrenzen lassen sich spezifische Sprachen, die zur Verwendung in einem bestimmten Werkzeug oder einer Sammlung von Werkzeugen entwickelt wurden und dort von den Benutzern eingesetzt werden. Diese Sprachen können den transformierten Artefakten sehr ähnlich sein, wie beispielsweise die Java Tools Language (JTL) [CGM06]. Sie können sich aber auch von diesen deutlich unterscheiden, sind jedoch trotzdem exklusiv zur Nutzung innerhalb dieser Werkzeuge bestimmt, was etwa auf Matlab Scripts [GB03, Kap. 8] zutrifft. Der Vorteil dieser Sprachen liegt darin, dass sie für die entsprechende Nutzergruppe optimal zugeschnitten werden

können. Als Nachteil steht dem jedoch der erhebliche Aufwand zur Entwicklung einer solchen Sprache gegenüber.

Eine Sonderrolle innerhalb der domänenspezifischen Transformationssprachen können somit Sprachen einnehmen, die automatisch aus den Sprachen der zu transformierenden Artefakte generiert werden. Diese Sprachen sind ebenfalls auf die Verwendung in einer bestimmten Domäne zugeschnitten. Sie können sich jedoch, da sie aus der jeweiligen Basissprache abgeleitet sind, nicht beliebig von dieser Sprache unterscheiden. Für die zuvor genannten Matlab-Scripts etwa ist dieser Ansatz folglich nicht geeignet. Die Vorteile einer solchen Sprache sind jedoch ihre gute Anpassung an die Domäne, sofern syntaktische Ähnlichkeit zwischen den Modellen und den Transformationen gewünscht ist, und der vernachlässigbar geringe Aufwand zur Erstellung der Sprache.

Die Generierung und Verwendung domänenspezifischer Transformationssprachen bilden daher den Schwerpunkt dieser Arbeit, wobei auf die Generierierung insbesondere in Kapitel 7 eingegangen wird.

2.3. Kategorisierung von Transformationssprachen und -werkzeugen

Aufgrund der vielfältigen Einsatzzwecke von Modelltransformationen hat sich auch eine Vielzahl von Transformationssprachen mit unterschiedlichen Eigenschaften herausgebildet. Eine umfangreiche Betrachtung und Taxonomie der wichtigsten Sprachen findet sich in [CH06]. In diesem Abschnitt werden lediglich die Kriterien vorgestellt, die für Modelltransformationen in MontiCore oder für das Verständnis dieser Arbeit und zur Abgrenzung gegen verwandte Ansätze wesentlich sind.

Zur Beschreibung von Transformationen sind zunächst einmal alle GPLs, wie etwa Java, C oder Haskell geeignet. Es existiert aber auch eine Vielzahl von *Transformationssprachen*, die speziell zur Implementierung von Modelltransformationen entworfen wurden.

Viele dieser Transformationssprachen beschreiben kleine Ersetzungsschritte in *Transformationsregeln*. Größere Programme können dann mit geeigneten Kompositionsmechanismen aus diesen Regeln und eventuell vorhandenen weiteren Sprachelementen zusammengesetzt werden.

2.3.1. Unterscheidungskriterien

Zunächst lassen sich unidirektionale Transformationen von bidirektionalen unterscheiden. Unidirektionale Transformationen können lediglich Informationen vom Quellmodell in das Zielmodell abbilden. Bidirektionale Transformationen können auch in umgekehrter Richtung ausgeführt werden. Die Übersetzung von UML-Klassendiagrammen und ER-Modellen in die jeweils andere Sprache ist ein häufig anzutreffendes Beispiel einer bidirektionalen Modell-zu-Modelltransformation.

Ein wesentlicher Vorteil unidirektionaler Transformationen ist, dass sich mit Ihnen leicht Informationen zu einem Modell hinzufügen oder aus dem Modell entfernen lassen, sodass diese Transformationen verfeinernd oder abstrahierend sein können.

Des weiteren lassen sich In-Place-Transformationen von Modell-zu-Modell-Transformationen unterscheiden. Bei In-Place-Transformationen sind Quell- und Zielmodell identisch oder haben zumindest gemeinsame Elemente[1], während bei Modell-zu-Modell-Transformationen separate Quell- und Zielmodelle verwendet werden. Modell-zu-Modell-Transformationen lassen sich wiederum unterteilen in endogene Transformationen, bei denen Quell- und Zielmodell in der selben Sprache vorliegen, und exogene Transformationen, bei denen dies nicht gefordert ist. In-Place-Transformationen lassen sich durch Modell-zu-Modell-Transformationen nachbilden, indem zunächst der Inhalt des Quellmodells in ein leeres Zielmodell kopiert wird. Dies geht allerdings mit erhöhtem Speicher- und Zeitbedarf bei der Ausführung der Transformation einher.

Eingesetzt werden In-Place-Transformationen vor allem zum Refactoring, zum Beispiel der semantikerhaltenden Vereinfachung von Statecharts, oder zur Beschreibung des Laufzeitverhaltens von Modellen, wie etwa der Zustandsübergänge in Automaten oder Petri-Netzen. Modell-zu-Modell-Transformationen dagegen werden vorwiegend zur Propagation von Information in andere Dokumente verwendet, etwa bei der Inferenz von Klassendiagrammen aus Objektdiagrammen. Weiterhin lassen sich Transformationsansätze unterscheiden bezüglich der internen Repräsentation von Modellen sowie der in den Transformationen beschrieben Modellfragmengte, die als *Muster* (Englisch: *Pattern*) bezeichnet werden. Stringbasierte Ansätze operieren dabei auf Zeichenketten, während bei termbasierten Systemen die Bäume der abstrakten Syntax verwendet werden. Noch einen Schritt weiter gehen Graphersetzungssysteme, bei denen attributierte und getypte Graphen verwendet werden. Bei termbasierten Ansätzen wird in der Regel ein einzelner Baum als Muster verwendet, während bei Graphersetzungssystemen auch Varianten existieren, bei denen das Muster nicht zusammenhängend sein muss.

In imperativen Ansätzen wird die Ausführung einzelner Programmteile durch Kontrollstrukturen gesteuert. Deklarative Transformationssprachen kommen dagegen völlig ohne Kontrollstrukturen aus. Auch wenn in komplexen Transformationen deklarativer Sprachen dann oft mehrere Regeln anwendbar sind, können die Resultate dieser Transformationen dennoch determiniert sein. Dies ist dann der Fall, wenn entweder die Reihenfolge, in der Regeln angewendet werden, eindeutig bestimmt ist, oder wenn die Transformationsregeln in einem komplexen Programm konfluent sind. Ebenfalls möglich ist eine Mischform imperativer und deklarativer Ansätze, wie beispielsweise in PROGRES [Zün92] und Fujaba [FNTZ00]. Hier wird die auszuführende Regel imperativ bestimmt, die Mustersuche innerhalb der Regeln ist aber deklarativ beschrieben.

In Tabelle 2.7 sind Beispiele für die verschiedenen Kategorien von Transformationssprachen aufgeführt. Diese Tabelle ist jedoch nicht vollständig, sondern dient nur dem Verweis auf exemplarische Arbeiten, die stellvertretend aus einer größeren Anzahl von Transformationsansätzen ausgewählt wurden.

---

[1] Hier wird der Begriff „In-Place“ also anders verwendet als bei In-Place-Algorithmen, die neben den Eingabedaten nur einen konstanten Platzbedarf haben.

[2] Ab der Version 3 werden zusätzlich In-Place-Transformationen unterstützt.

Unterscheidungskriterium	Ausprägung	Beispielsprachen
Erstellung des Zielmodells	in-Place	PROGRES [Sch91], Fujaba [FNTZ00]
	Modell-zu-Modell	ATL [JK05][2], Henshin [BEJ10]
Quell- und Zielsprache	endogen	PROGRES, Fujaba
	exogen	TGG [Sch94], QVT relational [OMG08a]
Ausführungsrichtung	unidirektional	AGG [Tae04], Fujaba, PROGRES
	bidirektional	TGG, QVT relational
Modellrepräsentation	String	FreeMarker [Fre11]
	Term (Baum)	Stratego/XT [BKVV08]
	Graph	PROGRES, Fujaba, AGG, TGG, VIATRA [VB07]
Ablaufsteuerung	imperativ	FreeMarker, Java
	deklarativ	TGG, QVT relational
	hybrid	Fujaba, PROGRES

Tabelle 2.7.: Kategorisierung ausgewählter Modelltransformationsansätze

2.3.2. Einordnung dieser Arbeit

Der in dieser Arbeit entwickelte Transformationsansatz ist folgendermaßen zu kategorisieren: Die generierten Sprachen erlauben die Beschreibung von In-Place-Transformationen, die somit auch endogen sind. Diese Transformationen werden nur unidirektional ausgeführt.

Die unidirektionale Ausführung ist für In-Place-Transformationen weitaus verbreiteter als die bidirektionale, und sie ist für die angestrebten Einsatzzwecke der Transformationsengine zu bevorzugen. Die Hauptgründe für die Entwicklung einer In-Place-Transformationsengine liegen in der Konzentration auf generatives Forward-Engineering im Umfeld dieser Arbeit, in der Möglichkeit, Abstraktionen oder Verfeinerungen zu beschreiben. Weitere Gründe sind die einfachere Implementierung von Modellmodifikationen in solchen Sprachen, sowie die effizientere Ausführung der Regeln, die beide darin begründet sind, dass nicht weite Teile des Quellmodells in das Zielmodell kopiert werden müssen.

Eine Engine für Modell-zu-Modell-Transformationen lässt sich auf Basis der in dieser Arbeit entstandenen In-Place-Transformationsengine entwickeln. Hierauf wird in Abschnitt 10.2 ein Ausblick gegeben.

Weil die Modelle als Graph vorliegen, der einen AST als spannenden Baum enthält, ist der Ansatz nicht term- sondern graphorientiert. Der Grund dafür ist zum einen, dass nicht wie bei Termersetzungssystemen üblich nur ein einzelner Teilbaum als Muster angegeben werden kann, sondern auch mehrere Teil-ASTs ein Muster bilden können. Der Zusammenhang zwischen diesen Teilgraphen wird über Zusatzbedingungen wie zum Beispiel Bezeichnergleichheit hergestellt. Ein weiterer Grund für die Wahl eines graphbasierten Ansatzes ist die Unterstützung für Assoziationen in MontiCore-Grammatiken, durch welche die ASTs zu beliebigen zyklischen Syntaxgraphen erweitert werden.

Die Ablaufsteuerung erfolgt hybrid: Die einzelnen Graphersetzungsregeln sind deklarativ, können aber durch Elemente einer imperativen Kontrollflusssprache miteinander kombiniert werden. So lässt sich das vergleichsweise hohe Abstraktionsniveau deklarativer Ersetzungsregeln mit einer für den Benutzer leicht verständlichen Ablaufsteuerung kombinieren.

Neben den genannten Gründen für die Wahl eines graphbasierten Ansatzes mit hybrider Ablaufsteuerung ergab sich die Entscheidung für eine derartige Transformationssprache vor allem aus dem geplanten Anwendungszweck, nämlich der Implementierung von Refactorings und semantikverfeinernden Transformationen auf Modellen. Diese werden in der Regel nur unidirektional ausgeführt. Sie finden weiterhin stets innnerhalb einer Sprache statt, und sie werden aus Effizienzgründen idealerweise in-Place ausgeführt.

2.4. Wissenschafltiche Herausforderungen für Modelltransformationen

Bereits um 1970 begannen Wissenschaftler, sich mit Graphersetzungssystemen zu beschäftigen, da man erkannt hatte, dass die bekannten Chomsky-Grammatiken, deren Ersetzungsregeln lineare Zeichenketten manipulieren, für eine Reihe interessanter Anwendungen nicht ausreichend ausdrucksstark sind [PR69, Sch70, Pra71]. Aufbauend auf diesen frühen Arbeiten entstanden im Laufe der letzten vier Jahrzehnte viele Ansätze zur Modelltransformation, die auf Graphersetzungen basieren. Unter anderem fallen hierunter auch die Transformationssysteme, die in Tabelle 2.7 in der Liste der Ansätze mit graphbasierter Modellrepräsentation aufgeführt sind.

Mit der zunehmenden Verbreitung dieser Transformationssprachen und ihrer Nutzung im größeren Maßstab rückten in den vergangenen Jahren auch die Steigerung der Effizienz bei der Entwicklung und Ausführung von Transformationen, Maßnahmen zur Qualitätssicherung sowie die Nutzbarkeit entsprechender Sprachen und Werkzeuge in den Vordergrund. So finden sich in den Konferenzbänden der Tagungen, die sich mit den Themen modellgetriebene Entwicklung und Modelltransformationen beschäftigen, zahlreiche Beiträge zu Fragestellungen, die sich aus dem zunehmenden Einsatz von „Modellierung-im-Großen“, und insbesondere auch „Modelltransformationen-im-Großen“ ergeben. Als Konferenzen sind zu letzterem Thema unter anderem die AGTIVE [SNZ08] und die ICMT [TG10] zu nennen. In etwas geringerem Maße trifft dies auch auf die ICGT [ERRS10], bei welcher der formale Ansatz der Graphtransformationen im Vordergrund steht, und die MODELS [PRH10a, PRH10b], die das gesamte Themengebiet der Modellierung umfassender behandelt, zu. Auch auf Konferenzen, die sich mit Software Engineering ohne einen spezielleren Fokus beschäftigen, finden sich entsprechende Beiträge [CB07].

Im Bereich der Qualitätssicherung von Modelltransformationen werden derzeit erste Verfahren zum systematischen Testen von Transformationen entwickelt. Wichtige Herausforderungen hierbei werden in [BGF+10] zusammengefasst. Bestehende Ansätze beschäftigen sich beispielsweise mit Überdeckungskriterien auf den möglichen Eingabemodellen [FBMLT07, EKT09] oder vergleichen das Verhalten ausführbarer generierter Modelle oder Programme mit einer Interpretation des Eingabemodells [Stü06]. Offene Fragen sind jedoch unter anderem, wie sich Überdeckung auf deklarativen Transformationen geeignet sicherstellen lässt, und wie sich die Beschreibung der Testfälle vom Abstraktionsniveau objektorientierter Programmiersprachen auf das eines modellgetriebenen Ansatzes anheben lässt.

Neben dem Testen von Transformationen gibt es auch Ansätze zur formalen Verifikation von Transformationen, etwa [HL07, BBKR08, Wag09]. Diese Ansätze sind jedoch häufig sehr aufwändig zu implementieren, oder sie lassen sich nur auf eingeschränkte Klassen von Transformationen, Modellen oder Spezifikationen anwenden.

Eine Verbesserung der Effizienz bei der Regelausführung lässt sich vor allem durch Optimierungen beim Pattern Matching erreichen, da hier das NP-vollständige Problem der Teilgraphisomorphie zu lösen ist[3]. Trotz der NP-Vollständigkeit lässt sich dieses Problem für viele interessante Anwendungsfälle in akzeptabler Zeit lösen. Mögliche Ansätze hierzu sind die Reduktion auf ein Erfüllbarkeitsproblem [Rud00] oder suchplangesteuerte Verfahren [Zün96b]. Eine weitere Steigerung der Effizienz lässt sich durch modellsensitive Ansätze erreichen, die Eigenschaften des Hostgraphen in den Suchplan mit einbeziehen [BKG08].

Des Weiteren gibt es auch außerhalb des Pattern-Matchings Möglichkeiten zur Effizienzverbesserung, auf die hier jedoch nicht näher eingegangen werden soll, wie etwa bei der Auswahl der anzuwendenden Regel, sofern diese nicht eindeutig bestimmt ist, oder beim Backtracking an nichtdeterministischen Kontrollstrukturen. Ein möglicher Einstiegspunkt in vertiefende Literatur zur Performance-Optimierung von Transformationssystemen ist [VSV05].

Ansätze zur Effizienzsteigerung gibt es nicht nur für die Ausführungszeit, sondern auch für die vom Entwickler zu investierende Zeit beziehungsweise den anfallenden Aufwand bei der Erstellung von Modelltransformationen. So wird zum einen versucht, durch die Entwicklung statischer Analysen und syntaxsensitiver oder gar -gesteuerter Entwicklungsumgebungen dem Entwickler beim Editieren frühzeitig Rückmeldung über Fehler in Transformationen zu geben [NS91, Opt06, Küs06]. Zum anderen gibt es auch Bemühungen, die Wiederverwendbarkeit von Transformationen zu verbessern, zum Beispiel durch Spezialisierungskonzepte [OMG08b, Abschnitt 9.14], Generizität [LKAS09] oder das Einweben von Informationen, die sich aus den Metamodellen der Quell- und Zielsprache ableiten lassen [DDFV09].

Zu den Verbesserungen bei der Usability sind zum einen die bereits erwähnten statischen Analysen und Komfortfunktionen der Entwicklungsumgebungen zu zählen. Darüber hinaus beschäftigt sich aber auch eine Reihe von Ansätzen mit der Frage, wie die Lesbarkeit von Modelltransformationen für die Nutzer von Modellierungssprachen verbessert werden kann, und wie solche Nutzer auch ohne tiefgehende Kenntnisse in der abstrakten Syntax der Sprache Transformationen implementieren können. Hier sind grundsätzlich drei Kategorien von Ansätzen voneinander zu unterscheiden: Erstens kann zu einer Sprache eine zugehörige Transformationssprache entwickelt werden, wie dies etwa bei Transformationen mathematischer Ausdrücke in Kalkülen der Fall ist [BEH$^+$87], aber sich prinzipiell auch auf Modellierungssprachen wie die UML anwenden lässt [BW06, Sch06]. Zweitens kann aus konkreten Beispiel für Quell- und Zielmodell die zugehörige Transformation inferiert werden [Var06, KW07]. Drittens lässt sich eine Transformationssprache aus der konkreten und abstrakten Syntax der Modellierungssprache generieren [GMP09, Grø09].

Der in dieser Arbeit vorgestellte Ansatz fällt in die dritte Kategorie. Im Vergleich zu vorher bestehenden Ansätzen ist als Alleinstellungsmerkmal die Generierung der Transformationssprache

[3] Die Reduktion der Teilgraphisomorphie auf das Cliquenproblem in Polynomialzeit ist trivial: Eine Clique mit k Knoten existiert in einem Graphen genau dann, wenn es einen Teilgraphen gibt, der isomorph zum vollständigen Graphen mit k Knoten ist. Eine direkte Reduktion auf 3-SAT beschreibt Cook in [Coo71].

auf textuelle DSLs angewandt, voll automatisiert und für existierende Modellierungssprachen durchgeführt worden. Eine genauere Abgrenzung gegen verwandte Arbeiten findet diesbezüglich im Abschnitt 5.6 statt.

2.5. Motivation der vorliegenden Arbeit

Wie im Abschnitt 2.1.4 skizziert wurde, bieten moderne Werkzeuge ausgeklügelte Konzepte an, die es erlauben, domänenspezifische Sprachen auch zur Verwendung in einzelnen oder wenigen Projekten mit akzeptablem Aufwand zu erstellen. Dieser Aufwand amortisiert sich in vielen Fällen durch bessere Einbindung von Domänenexperten in den Softwareentwicklungsprozess und erhöhte Effizienz bei der Produktentwicklung unter Verwendung einer DSL [KMB+96, GK03, Met07].

Domänenexperten können die so erstellten Sprachen nutzen und Modelle oder Programme in den Sprachen erstellen. Automatisierte Änderungen der Modelle lassen sich jedoch nur in Programmen oder Transformationen beschreiben, die gegen die abstrakte Syntax der DSL operieren. Da die entsprechenden Programmier- und Transformationssprachen nur die abstrakte, aber nicht die konkrete Syntax der DSLs widerspiegeln, sind sie für Domänenexperten meist unverständlich.

Die werkzeuggestützte Modellmanipulation ist aber für die effektive Verwendung von DSLs von großer Wichtigkeit. Dabei reichen die mit dem Werkzeug ausgelieferten Editieroperationen für komplexe Anwendungen nicht aus. Vielmehr müssen Domänenexperten in der Lage sein, auch umfangreiche Operationen auf den Modellen selbst zu implementieren. Der Bedarf, Domänenexperten an der Erstellung komplexer Transformationen mitwirken zu lassen, wird unter anderem aus dem hohen Verbreitungsgrad von Skripten und Skriptsprachen, beispielsweise zum Einsatz in Tabellenkalkulationen, Computeralgebrasystemen oder Datenbankanwendungen, ersichtlich. Er ist auch in einer Reihe wissenschaftlicher Aufsätze in den letzten Jahren unter den Schlagwörtern „End-User Software Enginering“ [BCP+03] oder „End-User Development“ [LPKW06] aufgegriffen worden.

Darüber hinaus stellen Transformationssprachen, die die konkrete Syntax der zugrunde liegenden DSL widerspiegeln, auch für die Sprach- und Werkzeugentwickler einen Mehrwert dar. So lassen sich Transformationen oft kompakter notieren, und die Lesbarkeit der Transformationsregeln wird erhöht, was wiederum die Wartbarkeit des Werkzeugs verbessert.

Mit der Einführung domänenspezifischer Notationen wird also vor allem in erheblichem Umfang die Nutzbarkeit von Modelltransformationen verbessert. Damit adressiert diese Arbeit eine Fragestellung, die gemeinsam mit der verbesserten Effizienz bei der Ausführung von Transformationen sowie der Qualitätssicherung zu den wichtigen offenen Forschungsfragen im Bereich der Modelltransformationen zu zählen ist.

Kapitel 3.

Durchgängiges Beispiel: Vereinfachung hierarchischer Statecharts

In diesem Kapitel wird ein Transformationsprozess auf hierarchischen Statecharts vorgestellt, der in den folgenden Kapiteln als durchgängiges Beispiel dient. Die einzelnen Schritte dieses Prozesses sind [Rum11] entnommen, werden hier jedoch anders als in der Originalliteratur miteinander kombiniert und um zusätzliche Bedingungen für ihre Anwendbarkeit erweitert.

Das Kapitel beginnt mit einer Motivation dieses Prozesses und erläutert seine Anwendung als vorbereitenden Schritt für die Codegenerierung aus Statecharts in Abschnitt 3.1. Nach einer informellen Einführung in die Syntax und die Semantik der Statechartsprache in Abschnitt 3.2 folgt in Abschnitt 3.3 eine Auflistung von Transformationsregeln, die jeweils kleine Vereinfachungsschritte auf Statecharts beschreiben. In Abschnitt 3.4 wird dann die Verknüpfung der einzelnen Regeln zu einem Gesamtprozess vorgestellt. Abschließend werden in Abschnitt 3.5 die Anforderungen diskutiert, die sich aus den Transformationsregeln und ihrer Verknüpfung an eine Transformationssprache beziehungsweise an die Transformationsengine, die zur Ausführung der Regeln verwendet wird, ergeben.

3.1. Motivation und Anwendbarkeit

Durch die in diesem Kapitel beschriebenen Regeln werden hierarchische Statecharts in semantisch äquivalente Statecharts ohne Hierarchie transformiert. Neben der Hierarchie wird auch die Verwendung weiterer Sprachkonzepte eliminiert, ohne die Semantik des Statecharts dabei zu verändern.

Eine semantikerhaltende Transformation ist dadurch charakterisiert, dass sie das beobachtbare Verhalten eines Modells nicht verändert. Genauer gesagt werden in der Abbildung, durch welche die Semantik einer Sprache definiert ist [HR04], die Eingabe und das Resultat einer solchen Transformation auf den selben Wert abgebildet.

Auf eine Formalisierung der Semantik von Statecharts wird an dieser Stelle verzichtet. Die Syntax und das semantische Mapping der UML/P-Statecharts aus [Rum11] werden in [Sch12] und [CGR08] beschrieben.

In dieser Arbeit wird zur einfacheren Illustration des Ansatzes eine leichtgewichtige Variante der Statecharts verwendet. Eine Einführung in die Syntax erfolgt in Abschnitt 3.2, hier wird auch eine vereinfachte EBNF-Grammatik für die Sprache angegeben. Die vollständige MontiCore-Grammatik der Statechartsprache befindet sich im Anhang C.1. Die Semantik orientiert sich an der in [CGR08] vorgestellten und wird bei Bedarf zusätzlich in natürlicher Sprache erläutert.

Die semantikerhaltenden Transformationen auf Statecharts können zur Vorbereitung der Codegenerierung dienen. Da durch die Transformationen alle Verwendungen bestimmter Sprachkonzepte, wie etwa der Hierarchie, eliminiert werden, muss ein Codegenerator, der auf transformierten Statecharts operiert, diese Konzepte nicht mehr berücksichtigen. Durch die Vorverarbeitung in einem Transformationsschritt wird also die Implementierung von Codegeneratoren vereinfacht, indem die Komplexität teilweise in die Transformation ausgelagert wird. Insbesondere ist diese teilweise Auslagerung der Komplexität auch unabhängig von der Zielplattform. Somit kann ein Transformator zur Vorbereitung der Eingabe verschiedener Codegeneratoren eingesetzt werden, wie in Abbildung 3.1 illustriert wird. Die von einem Anwendungsmodellierer erstellte Beschreibung eines Statecharts wird hier transformiert und anschließend in Codegenerierungsschritten für drei verschiedene Zielsprachen als Eingabe verwendet.

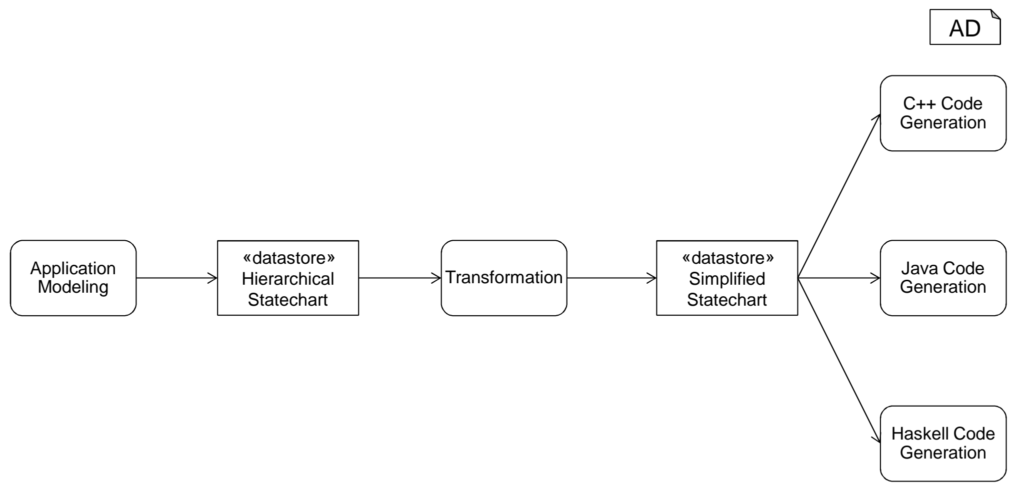


Abbildung 3.1.: Verwendung eines Transformators für mehrere Codegeneratoren

Dieser komplexitätsreduzierende Schritt ist also nur einmal zu implementieren, kann aber in verschiedenen Codegenerierungsprozessen mit unterschiedlichen Zielplattformen eingesetzt werden. Somit wird mit der Durchführung der in diesem Kapitel beschriebenen Transformation nicht nur ein einzelner Codegenerator vereinfacht, sondern auch ein potenziell über mehrere Softwareentwicklungsprozesse wiederverwendbares Artefakt geschaffen.

3.2. Einführung in die Statechartsprache

Statecharts sind zustandsbasierte Modelle, durch die sich das Verhalten von Objekten einer Klasse beschreiben lässt. Außerdem können sie auch zur Beschreibung des inneren Verhaltens von Methoden verwendet werden. Diese Methoden-Statecharts spielen für die Beispiele in der vorliegenden Arbeit jedoch keine Rolle und werden daher auch in diesem Kapitel außer Acht gelassen.

Formal beruhen Statecharts auf den Mealy-Automaten [Mea55]. Sie erweitern diese jedoch um zusätzliche Konzepte, die im Folgenden noch erläutert werden.

Die Syntax der in diesem Abschnitt vorgestellten Statechartsprache orientiert sich sowohl an der grafischen als auch an der textuellen Syntax der UML/P Statecharts. Die grafische Syntax wurde in [Rum11] vorgestellt, die textuelle Syntax aufbauend hierauf in [Sch12]. Zur Vereinfachung der Beispiele wurden in dieser Arbeit jedoch einige Veränderungen vorgenommen. Im Folgenden wird die Notation der wichtigsten Konzepte erläutert. Für eine vollständige Beschreibung der UML/P-Statecharts wird auf [Rum11, Sch12] verwiesen.

In diesem Abschnitt wird zunächst die Statchartsprache in einer grafischen Notation vorgestellt sowie die Bedeutung der Elemente in natürlicher Sprache informell erklärt. Anschließend folgt die Beschreibung der in den folgenden Kapiteln überwiegend verwendete textuelle Fassung. Alle Verweise auf grafische Notationen beziehen sich auf die grafische Notation für UML/P-Statecharts. Diese weisen einige Unterschiede zu den von der OMG definierten UML-State-Machines [OMG10b, OMG10c] auf, die jedoch für die Beispiele in dieser Arbeit nicht wesentlich sind.

Zur Erläuterung der Konzepte der Statechartsprache wird ein vereinfachtes Modell der Steuerung einer Fußgängerampel verwendet, das in Abbildung 3.2 dargestellt ist.

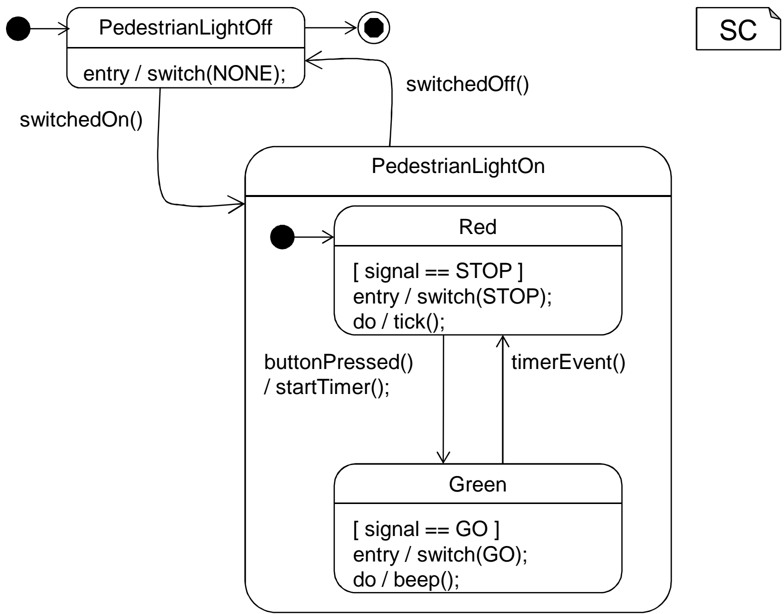


Abbildung 3.2.: Vereinfachtes Modell der Steuerung einer Fußgängerampel

3.2.1. Zustände und Transitionen

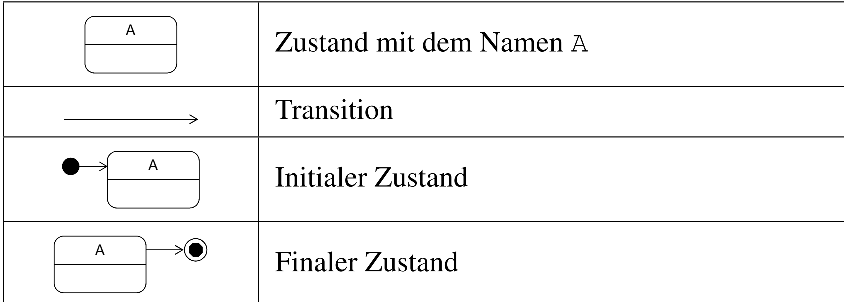
Die Kernelemente der Statechartsprache bilden *Zustände* und *Transitionen*. In der grafischen Notation werden Zustände als Rechtecke mit gerundeten Kanten dargestellt und mit dem Namen des Zustands beschriftet. Transitionen werden als solide Pfeile notiert. Zustände können hierarchisch als Ober- und Unterzustände geschachtelt sein, alternativ sind auch die Begriffe Super-

und Subzustand gebräuchlich. Wenn in einem Statechart zur Ausführungszeit ein Unterzustand erreicht wird, so ist stets gleichzeitig auch dessen Oberzustand aktiv.

Das in Abbildung 3.2 dargestellte Statechart hat die Zustände `PedestrianLightOff` und `PedestrianLightOn` sowie die Zustände `Red` und `Green` als Unterzustände von `PedestrianLightOn`. Transitionen verlaufen jeweils in beide Richtungen zwischen `PedestrianLightOff` und `PedestrianLightOn` sowie zwischen `Red` und `Green`.

Zustände können als *Anfangszustände* (auch *initiale* Zustände) und *Endzustände* (auch *finale* Zustände) markiert werden. Alle Objekte, deren Verhalten durch ein Statechart beschrieben werden, befinden sich zu Beginn ihres Lebenszyklus in dessen initialem Zustand. Beim Erreichen eines finalen Zustands kann das Objekt aus dem Speicher entfernt werden[1]. Davon abweichend ist die Bedeutung dieser Markierungen für Unterzustände: Bei hierarchisch geschachtelten Zuständen wird beim Erreichen einer Oberzustandes automatisch ein initialer Unterzustand aktiviert, und beim Erreichen eines finalen Subzustandes wird dessen Superzustand verlassen.

Das Statechart der Fußgängerampel hat den Anfangs- und Endzustand `PedestrianLightOff`. Beim Betreten des Zustands `PedestrianLightOn` wird zunächst der Unterzustand `Red` aktiviert. Da `PedestrianLightOn` über keinen ausgewiesenen Endzustand verfügt, kann dieser Zustand von jedem beliebigen Subzustand aus verlassen werden. Tabelle 3.3 fasst die Notationen von Zuständen und Transitionen zusammen.



Notation	Bedeutung
A	Zustand mit dem Namen `A`
→	Transition
●→ A	Initialer Zustand
A →◉	Finaler Zustand

Tabelle 3.3.: Notation von Zuständen und Transitionen

3.2.2. Stimuli, Aktionen und Guards

Transitionen können mit *Stimuli* beschriftet werden. Stimuli sind Ereignisse, in der Regel Methodenaufrufe, die das Schalten der entsprechenden Transition auslösen. Beim Auftreten des Stimulus-Ereignisses wechselt das Objekt also vom Quellzustand der Transition in den Zielzustand. Fehlt die Angabe eines Stimulus, so kann die Transition spontan schalten, also ohne Auftreten eines bestimmten Ereignisses ausgelöst werden.

Die Stimuli in Abbildung 3.2 sind `switchedOn()` für den Wechsel von `PedestrianLightOff` nach `PedestrianLightOn`, `switchedOff()` für die entgegengesetzte Tran-

[1]Wird aus einem Statechart Code in einer Sprache mit automatischer Speicherbereinigung generiert, wie dies beispielsweise in Java der Fall ist, so hat die Markierung finaler Zustände meistens keinen Einfluss, da das Objekt vom System entfernt wird, sobald es nicht mehr referenziert wird.

sition, `buttonPressed()` für den Wechsel von `Red` nach `Green` und `timerEvent()` für die Transition von `Green` nach `Red`.

Des Weiteren können Transitionen *Aktionen* zugeordnet werden, die beim Schalten der Transition ausgeführt werden. Eine Aktionsbeschriftung besteht in der grafischen Variante der Statecharts aus einem einleitenden Schrägstrich und einer Folge von Java-Statements.

Im Beispiel aus Abbildung 3.2 ist die einzige Aktion an einer Transition der Methodenaufruf `startTimer();` an der Transition von `Red` nach `Green`. Eine Übersicht zu den Beschriftungen an Transitionen gibt Abbildung 3.4.

myCall() →	Transition mit dem Methodenaufruf `myCall` als Stimulus
/ someAction(...); →	Transition mit dem Methodenaufruf `someAction(...);` als Aktion

Tabelle 3.4.: Beschriftungen an Transitionen

3.2.3. Invarianten, Vor- und Nachbedingungen

Zustandsinvarianten sind Bedingungen, die zur Ausführungszeit gelten müssen, während sich das Statechart im entsprechenden Zustand befindet. Diese Bedingungen werden innerhalb des Zustands in eckigen Klammern notiert und können als boolesche Ausdrücke in Java angegeben werden.

Vorbedingungen und *Nachbedingungen* sind ebenfalls boolesche Ausdrücke, die in eckigen Klammern notiert werden. Sie beziehen sich jedoch nicht auf einen Zustand, sondern auf eine Transition. Eine Vorbedingung gibt eine zusätzliche Einschränkung für die Schaltbereitschaft der Transition an, eine Nachbedingung beschreibt eine Zusicherung, die nach dem Schalten der Transition gilt.

Das Beispiel-Statechart in Abbildung 3.2 enthält die Invarianten `signal == STOP` und `signal == GO` in den Zuständen `Red` beziehungsweise `Green`; Vor- und Nachbedingungen sind im Beispiel nicht enthalten, ihre Notation kann aber Tabelle 3.5 entnommen werden.

A [n >= 1]	Zustand mit dem booleschen Ausdruck `n>=1` als Invariante
[true] myCall() / someAction(...); [o != null] →	Stimulus, Aktion sowie Vorbedingung `true` und Nachbedingung `o != null` an einer Transition

Tabelle 3.5.: Notation von Zustandsinvarianten und Bedingungen an Transitionen

3.2.4. Entry- und Exit-Aktionen

Entry-Aktionen werden beim Betreten eines Zustands ausgeführt, *Exit-Aktionen* beim Verlassen des Zustands. Notiert werden solche Aktionen durch die Angabe des Schlüsselwortes `entry`

beziehungsweise `exit`, gefolgt von einem Schrägstrich und einer Folge von Java-Statments, den eigentlichen Aktionen.

Im Statechart der Fußgängerampel sind `switch(NONE);`, `switch(STOP);` und `switch(GO);` als Entry-Aktionen für die Zustände `PedestrianLightOff`, `Red` und `Green` enthalten. Exit-Aktionen werden nicht verwendet. In Tabelle 3.6 sind die Notationen von Entry- und Exit-Aktionen zusammengefasst.

A entry / doSomething(...);	Zustand mit dem Methodenaufruf `doSomething(...);` als Entry-Aktion
A exit / somethingElse(...);	Zustand mit dem Methodenaufruf `somethingElse(...);` als Exit-Aktion

Tabelle 3.6.: Notation von Entry- und Exit-Aktionen

3.2.5. Do-Aktivitäten und interne Transitionen

Do-Aktivitäten sind einem Zustand zugeordnete Aktionen, die regelmäßig ausgeführt werden, solange sich das Objekt in diesem Zustand befindet. Sie werden durch das Schlüsselwort `do`, einen Schrägstrich, sowie die eigentliche Aktion als eine Folge von Java-Statements notiert.

Interne Transitionen sind Transitionen, nach deren Schalten das Objekt wieder in dem Zustand ist, der die interne Transition enthält. Im Gegensatz zu anderen Transitionen löst das Schalten einer internen Transition aber keine Entry- oder Exit-Aktionen aus.

In Abbildung 3.2 enthalten die Zustände `Red` und `Green` je eine Do-Aktivität mit den Anweisungen `tick();` beziehungsweise `beep();`. Die Notation von Do-Aktivitäten und internen Transitionen ist in Tabelle 3.7 zusammengefasst.

A do / executeMe(...);	Zustand mit dem Methodenaufruf `executeMe(...);` als Do-Aktivität
A tEvent() / tCounter++;	Interne Transition mit dem Methodenaufruf `tEvent()` als Stimulus und der Anweisung `tCounter++;` als Aktion

Tabelle 3.7.: Notation von Do-Aktivitäten und internen Transitionen

3.2.6. Die textuelle Syntax der Statechartsprache

Für die in diesem Abschnitt bisher vorgestellten Elemente der Statecharts wird in den folgenden Kapiteln meistens eine textuelle Notation verwendet, die in diesem Unterabschnitt vorgestellt wird. Einen ersten Überblick hierzu gibt Abbildung 3.8. Die wichtigsten Produktionen der Grammatik dieser Sprache sind in Abbildung 3.9 in der Erweiterten Backus-Naur-Form (EBNF, [II96]) dargestellt.

Statechart

```
1  statechart PedestrianLight {
2
3    state PedestrianLightOff <<initial>><<final>> {
4      entry : { switch(NONE); }
5    }
6
7    state PedestrianLightOn {
8
9      state Red <<initial>> {
10       [ signal == STOP ]
11       entry : { switch(STOP); }
12       do : { tick(); }
13     }
14
15     state Green {
16       [ signal == GO ]
17       entry : { switch(GO); }
18       do : { beep(); }
19     }
20
21     Red -> Green : buttonPressed()
22       / { startTimer(); }
23     ;
24
25     Green -> Red : timerEvent() ;
26   }
27
28   PedestrianLightOff -> PedestrianLightOn : switchedOn();
29   PedestrianLightOn -> PedestrianLightOff : switchedOff();
30
31 }
```

Abbildung 3.8.: Textuelle Fassung des Statecharts aus Abbildung 3.2

```
EBNF
1  SCStructure ::= Statechart | State ;
2
3  Statement ::= BlockStatement | ExpressionStatement ;
4
5  Statechart ::= 'statechart' Name '{' (State | Transition )* '}' ;
6
7  EntryAction ::= 'entry' ':' BlockStatement ;
8
9  ExitAction ::= 'exit' ':' BlockStatement ;
10
11 DoAction ::= 'do' ':' BlockStatement ;
12
13 InternTransition ::=
14   '-intern>' (':' (Name ('(' (Argument (',' Argument )* ) ')' )? )?
15     ('[' Expression ']' )?
16     ('/' BlockStatement ('[' Expression ']' )? )? ';' | ';'
17   ) ;
18
19 State ::=
20   'state' Name ('<<' ('initial'| 'final') '>>' )*
21   (
22     ('{' ('[' Expression ('&&' Expression )* ']' )?
23       (EntryAction )? (DoAction )? (ExitAction )?
24       (State | Transition | InternTransition )* '}'
25     ) | ';'
26   ) ;
27
28 Transition ::=
29   Name '->' Name
30   (':' (Name ('(' ((Argument (',' Argument )* )? ) ')' )? )?
31     ('[' Expression ']' )?
32     ('/' BlockStatement ('[' Expression ']' )? )? ';'
33     | ';'
34   ) ;
35
36 Argument ::= Name Name ;
37
38 BlockStatement ::= '{' (Statement )* '}' ;
39
40 ExpressionStatement ::= Expression ';' ;
```

Abbildung 3.9.: Wichtige Produktionen der Statechart-Grammatik

Die Definition eines Statecharts beginnt mit dem Schlüsselwort `statechart`, gefolgt vom Namen und einer Liste von Zuständen und Transitionen in beliebiger Reihenfolge, die innerhalb eines Blocks aus geschweiften Klammern definiert werden. Die Definition des Statecharts `PedestrianLight` erstreckt sich über alle Zeilen der Abbildung 3.8.

Zustandsdefinitionen werden durch das Schlüsselwort `state` eingeleitet. Hierauf folgt der Name des Zustands, sowie weitere Details innerhalb eines Blocks in geschweiften Klammern. Diese weiteren Details können im Zustand enthaltene Transitionen und Subzustände sein. Ferner sind dies Zustandsinvarianten, Entry- und Exit-Aktionen, Do-Aktivitäten und interne Transitionen. In Abbildung 3.8 werden die Zustände `PedestrianLightOn` und `PedestrianLightOff` sowie `Red` und `Green` als Unterzustände von `PedestrianLightOn` definiert.

Anfangs- und Endzustände werden durch die Stereotpyen `<<initial>>` beziehungsweise `<<final>>` gekennzeichnet, wie im Beispiel in den Zeilen 3 und 9 zu sehen ist.

Transitionen werden durch den Namen des Quellzustandes, einen Pfeil in ASCII-Art `->` und den Namen des Zielzustandes beschrieben. Vor einem abschließenden Semikolon können optional noch der Stimulus, eine Precondition, eine Aktion und eine Postcondition angegeben werden, die von den zwingend erforderlichen Angaben durch einen Doppelpunkt abgetrennt werden.

Ein Stimulus ist je nach Verwendungszweck des Statecharts entweder ein einfacher Name oder der Name einer Methode gefolgt von einer Parameterliste, zum Beispiel `switchedOn()` in Zeile 28 des Beispielstatecharts.

Aktionen bestehen aus einem Schrägstrich und einem Block-Statement in einer an Java angelehnten Notation. Um die Beispielgrammatik einfach zu halten, wird hier jedoch nicht der volle Sprachumfang von Java unterstützt. Zeile 22 der Abbildung enthält mit `/ { startTimer(); }` ein Beispiel einer Aktion.

Invarianten sowie Pre- und Postcondition sind boolesche Ausdrücke. Auch hier ist die Notation an Java angelehnt, es wird aber wiederum nicht der volle Sprachumfang unterstützt. Das Statechart der Fußgängerampel enthält Invarianten in den Zeilen 10 und 16.

Do-Aktivitäten, Entry- und Exit-Aktionen werden durch die Schlüsselwörter `do`, `entry` beziehungsweise `exit` und einen Doppelpunkt eingeleitet. Es folgt ein Block-Statement, das syntaktisch der Syntax für Block-Statements in Aktionen entspricht. Abbildung 3.8 enthält Entry-Aktionen in den Zeilen 4, 11 und 17.

Interne Transitionen werden durch einen beschriften Pfeil in ASCII-Art `-intern>` eingeleitet. Es folgt ein Doppelpunkt sowie optional Stimulus, Precondition, Aktion und Postcondition. Ab dem Doppelpunkt entsprechen interne Transitionen syntaktisch den oben beschriebenen Transitionen.

3.3. Semantikerhaltende Transformationsregeln

Die im letzten Abschnitt beschriebenen Konzepte der Statechartsprache bieten dem Nutzer an vielen Stellen die Möglichkeit, bei der Beschreibung eines Sachverhalts zwischen verschiedenen Konzepten zu wählen. Tatsächlich kann auf die Nutzung vieler Konzepte verzichtet werden, ohne die Ausdrucksmächtigkeit der Sprache zu reduzieren. Nichtsdestotrotz ist die Beibehaltung der Konzepte in der Sprache sinnvoll, da ihre Nutzung beispielsweise die Lesbarkeit der Modelle verbessern kann, oder die interne Strukturierung der Statecharts überhaupt erst ermög-

licht. Codegeneratoren lassen sich jedoch in vielen Fällen einfacher entwickeln, wenn solche Konzepte in den Modellen nicht verwendet werden.

Jede der in diesem Abschnitt vorgestellten Transformationsregeln dient der Eliminierung eines einzelnen Konzeptes aus Statecharts. Dabei werden alle Vorkommen dieses Konzeptes durch eine semantisch äquivalente Verwendung anderer Konzepte ersetzt. Hierbei können die folgenden Beobachtungen gemacht werden:

- Die Regeln beziehen sich nur auf einen Ausschnitt des Statecharts - typischerweise einen Zustand oder eine Transition sowie die hierin enthaltenen oder unmittelbar hiermit verbundenen Elemente. Zur vollständigen Eliminierung eines Konzeptes aus einem Modell müssen die Regeln also mehrfach angewendet werden.
- Einige der hier vorgestellten Regeln verwenden im Ersetzungsteil Konzepte, die durch andere Regeln eliminiert werden. Für die vollständige Entfernung dieser Elemente ist also die Reihenfolge, in der die Regeln ausgeführt werden, von Bedeutung.
- Die Anwendbarkeit bestimmter Regeln unterliegt Voraussetzungen, die erst durch die Ausführung anderer Regeln geschaffen werden. Diese können zwar in den Regeln angegeben werden. Damit die Regel zur Laufzeit tatsächlich erfolgreich angewendet werden kann, muss aber dafür Sorge getragen werden, dass die Voraussetzungen im Vorfeld geschaffen werden. Auch hier spielt also auf die Reihenfolge bei der Ausführung der Regeln eine Rolle.
- Ebenfalls in den folgenden Beispielen enthalten sind Regeln, die keine Konzepte eliminieren. Diese dienen dann der Vorbereitung anderer Regeln, indem sie die Voraussetzungen für deren Anwendbarkeit schaffen.

Der Fokus dieses Abschnittes liegt auf der Beschreibung einzelner Regeln anhand konkreter Beispiele, den Voraussetzungen für ihre Anwendbarkeit sowie den Ersetzungen, die durch die Regeln vorgenommen werden. Die korrekte Reihenfolge und mehrfache Anwendung von Regeln wird in Abschnitt 3.4 beschrieben. Einige der hier vorgestellten Ersetzungen sind so implementiert, dass die Ausführung vorgelagerter Regeln Voraussetzung ist, damit die folgenden Ersetzungen semantikerhaltend angewendet werden können.

3.3.1. Do-Aktivitäten eliminieren

Do-Aktivitäten, also Aktionen, die während des Verweilens in einem Zustand regelmäßig ausgeführt werden, lassen sich durch interne Transitionen ersetzen, die durch den Ablauf eines Timers stimuliert werden. Dieser Timer muss beim Betreten des Zustandes und nach dem Schalten der internen Transition gestartet und beim Verlassen des Zustands gestoppt werden. Abbildung 3.10 zeigt die Eliminierung der Do-Aktivität aus dem Zustand `Red`.

Zu beachten ist zum einen, dass für die Do-Aktivität auch dann eine neue interne Transition angelegt wird, wenn bereits eine andere interne Transition existiert. Des Weiteren soll die Anweisung `timer.set(...);` als letzte Anweisung beim Betreten des Statecharts und an der internen Transition ausgeführt werden. Die Anweisung `timer.stop(...);` soll dagegen beim Verlassen des Zustands vor allen anderen Anweisungen ausgeführt werden.

```
Statechart
1 state Red <<initial>> {
2   entry : {
3     switch(STOP);
4   }
5   do : {
6     tick();
7   }
8 }
```

```
Statechart
1 state Red <<initial>> {
2   entry : {
3     switch(STOP);
4     timer.set(this, delay);
5   }
6   exit: {
7     timer.stop(this);
8   }
9
10  -intern> : timeout / {
11    tick();
12    timer.set(this, delay);
13  };
14 }
```

Abbildung 3.10.: Äquivalente Zustände mit Do-Aktiviät (links) und interner Transition (rechts)

3.3.2. Transitionen an initiale Subzustände umleiten

Diese Ersetzung fällt in die Kategorie der Regeln, die keine syntaktischen Konzepte der Statechartsprache aus dem Modellen eliminieren, sondern nur der Vorbereitung weiterer Schritte, insbesondere des Entfernens der Hierarchie, dienen.

Eine Transition, die einen Superzustand als Ziel hat, kann durch Transitionen zu jedem initialen Subzustand dieses Zustands ersetzt werden. Dabei müssen Quellzustand und Beschriftung der neuen Transitionen mit denen der alten Transition übereinstimmen. Sind alle neuen Transitionen angelegt, können die alte Transition und die Markierung `<<initial>>` der Subzustände entfernt werden. Hat ein Superzustand keine explizit markierten Initialzustände, so können alle seine Subzustände als Anfangszustände interpretiert werden. Abbildung 3.11 zeigt das Umleiten einer Transition vom Zielzustand `On` zu dessen initialem Subzustand `Red`.

```
Statechart
1 state Off { ... }
2
3 state On {
4   state Red <<initial>> { ... }
5 }
6
7 Off -> On : switchedOn();
```

```
Statechart
1 state Off { ... }
2
3 state On {
4   state Red { ... }
5 }
6
7 Off -> Red : switchedOn();
```

Abbildung 3.11.: Äquivalente Statechart-Ausschnitte: Transition endet in Superzustand (links) oder in initialem Subzustand (rechts)

Zu beachten ist, dass diese Transformation zunächst für alle initialen Subzustände und alle eingehenden Transitionen eines Zustands die neuen Transitionen erzeugen muss, bevor die ursprünglichen Transitionen entfernt und das `initial`-Attribut gelöscht werden können.

3.3.3. Transitionen aus finalen Subzuständen starten lassen

Wie auch durch die Weiterleitung von Transitionen an initiale Subzustände wird durch diese Regel kein syntaktisches Konzept aus den Statecharts eliminiert.

Eine Transition, die in einem Superzustand startet, kann durch Transitionen ersetzt werden, die in dessen finalen Subzuständen beginnen. Hier müssen Zielzustand und Beschriftung der neuen Transitionen mit denen der alten Transition übereinstimmen. Nach dem Anlegen der neuen Transitionen können die `<<final>>`-Markierung und die alte Transition entfernt werden.

Auch hier ist zu beachten, dass ein Zustand mehrere finale Subzustände und mehrere ausgehende Transitionen haben kann. In diesem Fall müssen zunächst alle neuen Transitionen angelegt werden, bevor das `final`-Attribut gelöscht und die alten Transitionen entfernt werden können.

Abbildung 3.12 zeigt die Anwendung dieses Ersetzungsschrittes auf den hierarchischen Zustand `On`. Da dieser Zustand auf der linken Seite keinen explizit markierten finalen Subzustand hat, werden alle Subzustände implizit als Endzustände angesehen. Deshalb beginnt in den beiden Subzuständen `Red` und `Green` auf der rechten Seite je eine Transition. Zu beachten ist also, dass die Regel in dieser Form nur dann auf einen Modellausschnitt anwendbar ist, wenn es sich bei keinem der betreffenden Subzustände um einen Endzustand handelt.

Statechart
```
1 state Off { ... }
2
3 state On {
4   state Red { ... }
5   state Green { ... }
6 }
7
8 On -> Off : switchedOff();
```

Statechart
```
1 state Off { ... }
2
3 state On {
4   state Red { ... }
5   state Green { ... }
6 }
7
8 Red -> Off : switchedOff();
9 Green -> Off : switchedOff();
```

Abbildung 3.12.: Äquivalente Statechart-Ausschnitte: Transition beginnt in Superzustand ohne finalen Subzustand (links) oder in dessen Subzuständen (rechts)

3.3.4. Exit-Aktionen an ausgehende Transitionen verschieben

Exit-Aktionen werden beim Verlassen eines Zustands ausgeführt. Da im Folgenden auch die Aktionen der entsprechenden Transition angestoßen werden, ist es möglich, an jeder ausgehenden Transition eine Kopie der Exit-Aktion zu erstellen. Diese muss die erste Aktion sein, die an der jeweiligen Transition ausgeführt wird. Anschließend kann die Exit-Aktion aus dem Zustand entfernt werden.

In Abbildung 3.13 ist das Entfernen der Exit-Aktion `timer.stop(this);` aus dem Zustand `Green` dargestellt. Diese Aktion ist im äquivalenten Modellausschnitt auf der rechten Seite an der Transition zu finden, die vom Zustand `Green` nach `Red` führt.

Treten Exit-Aktionen auch in hierarchisch gegliederten Zuständen auf, so sind sie auch beim Verlassen aus einem Subzustand heraus zu berücksichtigen. In Kombination mit Bedingungen

Statechart
```
1 state Green {
2   ...
3   exit: {
4     timer.stop(this);
5   }
6 }
7 state Red { ... }
8
9 Green->Red : buttonPressed();
```

Statechart
```
1 state Green { ... }
2 state Red { ... }
3
4 Green->Red : buttonPressed() / {
5   timer.stop(this);
6 };
```

Abbildung 3.13.: Äquivalente Statechart-Ausschnitte mit Exit-Aktion (links) und Aktion an ausgehender Transition (rechts)

für die Aktionen lässt sich die Semantik der UML/P-Statecharts hinsichtlich der Ausführungs- beziehungsweise Auswertungsreihenfolge an dieser Stelle durch Stereotypen beeinflussen; eine detaillierte Betrachtung würde das Beispiel an dieser Stelle aber zu umfangreich werden lassen, weshalb Exit-Aktionen für Superzustände hier nicht betrachtet werden und für Details auf [Rum11] verwiesen wird.

3.3.5. Entry-Aktionen an eingehende Transitionen verschieben

Die Eliminierung von Entry-Aktionen verläuft weitgehend analog zum Entfernen der Exit-Aktionen. Allerdings müssen die Entry-Aktionen als letzte Aktion der eingehenden Transitionen ausgeführt werden.

Abbildung 3.14 zeigt das Entfernen der Entry-Aktion aus dem Zustand `Red`. Eine Kopie der Aktion wurde an der einzigen eingehenden Transition von `Green` nach `Red` angelegt.

Statechart
```
1 state Green { ... }
2 state Red {
3   ...
4   entry: {
5     timer.start(this, delay);
6   }
7 }
8
9 Green->Red : buttonPressed();
```

Statechart
```
1 state Green { ... }
2 state Red { ... }
3
4 Green->Red : buttonPressed() / {
5   timer.start(this, delay);
6 };
```

Abbildung 3.14.: Äquivalente Statechart-Ausschnitte mit Entry-Aktion (links) und Aktion an eingehender Transition (rechts)

3.3.6. Interne Transitionen eliminieren

Interne Transitionen sind Transitionen innerhalb eines Zustandes. Da beim Schalten einer internen Transition keine Entry- oder Exit-Aktionen ausgeführt werden dürfen, wird in [Rum11]

die Ersetzung interner Transitionen durch Einführung eines Subzustandes und einer zusätzlichen Transition beschrieben. Durch vorherige Eliminierung der Entry- und Exit-Aktionen wie beschrieben kann jedoch auf die Einführung des Unterzustandes verzichtet werden.

Die interne Transition des Zustands `Green` aus Abbildung 3.15 lässt sich durch eine echte Transition ersetzen, die von `Green` nach `Green` führt und mit dem selben Stimulus sowie der selben Aktion beschriftet ist, die auch für die interne Transition gelten.

Statechart

```
1 state Green {
2   -intern> : timeout / { beep(); };
3 }
```

Statechart

```
1 state Green { }
2 Green->Green : timeout / {
3   beep();
4 };
```

Abbildung 3.15.: Äquivalente Statechart-Ausschnitte mit interner Transition (links) und Transition außerhalb des Zustandes (rechts)

3.3.7. Zustandsinvarianten an Subzustände propagieren

Zustandsinvarianten eines Superzustandes müssen auch in allen Subzuständen gelten. Durch Anwendung dieser Transformation lässt sich das Konzept der Invarianten zwar nicht elimieren. Die Invarianten werden jedoch in allen Zuständen explizit gemacht. Somit dient dieser Schritt der Vorbereitung eines späteren Schrittes, nämlich des in Abschnitt 3.3.8 beschriebenen Entfernens der Hierarchie.

Falls in einem Unterzustand noch keine Invariante existiert, so kann die Invariante des Oberzustands in den Unterzustand kopiert werden. Anderenfalls ist die bestehende Invariante mit der des Superzustands zu konjugieren. Nachdem eine Invariante in alle Subzustände propagiert wurde, kann sie im Superzustand entfernt werden.

Abbildung 3.16 zeigt auf der linken Seite den Superzustand `PedestrianLightOn` mit der Invariante `status == ON`. Auf der rechten Seite wurde diese Invariante mit den bestehenden Invarianten der Zustände `Red` und `Green` konjugiert sowie aus dem Oberzustand `PedestrianLightOn` entfernt.

3.3.8. Hierarchisch zergliederte Zustände entfernen

Das Entfernen der Superzustände aus einem Statechart ist nur dann semantikerhaltend, wenn verschiedene Voraussetzungen erfüllt sind, die beispielsweise durch die bisher beschriebenen Transformationen geschaffen werden können. So dürfen zum Beispiel die zu entfernenden Zustände keine eingehenden oder ausgehenden Transitionen mehr haben, und die Zustandsinvarianten müssen an Unterzustände propagiert worden sein. Liegen alle Voraussetzungen vor, kann ein hierarchisch zergliederter Zustand entfernt werden. Alle in diesem Zustand enthaltenen Elemente sind danach in dem Oberzustand des entfernten Zustands enthalten, oder auf oberster Ebene im Statechart, falls ein solcher Oberzustand nicht existiert.

Statechart

```
1 state PedestrianLightOn {
2   [ status == ON ]
3
4   state Red {
5     [ signal == STOP ]
6   }
7
8   state Green {
9     [ signal == GO ]
10  }
11 }
```

Statechart

```
1 state PedestrianLightOn {
2   state Red {
3     [ status == ON
4       && signal == STOP ]
5   }
6
7   state Green {
8     [ status == ON
9       && signal == GO ]
10  }
11 }
```

Abbildung 3.16.: Äquivalente Statechart-Ausschnitte mit Invariante am Superzustand (links) und an allen Subzuständen (rechts)

Abbildung 3.17 zeigt, wie der Zustand `PedestrianLightOn` aus dem Statechart `PedestrianLight` entfernt wird. Die in diesem Zustand enthaltenen Subzustände `Red` und `Green` befinden sich anschließend auf oberster Ebene im Statechart.

3.4. Verknüpfung der Regeln

Die im Kapitel 3.3 beschriebenen Regeln lassen sich grundsätzlich in der genannten Reihenfolge ausführen. Dabei kann jede Ersetzung so oft ausgeführt werden, bis es im Modell keine passenden Abschnitte mehr gibt, auf die sich die Ersetzung anwenden ließe. Bei einigen Ersetzungen, wie etwa dem Weiterleiten von Transitionen an initiale Subzustände, ist es jedoch erforderlich, vor der Ausführung mehrere oder alle möglichen Anwendungsstellen zu bestimmen.

Es sind aber auch von der angegebenen Reihenfolge abweichende Varianten in der Ausführung möglich; jedoch unter Einhaltung einiger Randbedingungen:

- Beim Eliminieren der Do-Aktivitäten (Abschnitt 3.3.1) entstehen Entry- und Exit-Aktionen sowie interne Transitionen. Diese sind nach dem Eliminieren der Do-Aktivitäten gemäß den Abschnitten 3.3.4 und 3.3.5 zu behandeln.
- Die angegebene Ersetzung der internen Transitionen (Abschnitt 3.3.6) durch echte Transitionen ist ohne Einführung eines zusätzlichen Subzustandes nur dann wie angegeben möglich, wenn der Zustand neben der internen Transition weder Entry- noch Exit-Aktionen beinhaltet.
- Das Verschieben von Entry- und Exit-Aktionen an eingehende und ausgehende Transitionen (Abschnitte 3.3.4 und 3.3.5) ist nur dann semantikerhaltend, wenn zuvor alle Markierungen initialer und finaler Subzustände entsprechend der Abschnitte 3.3.2 und 3.3.3 beseitigt wurden.
- Die Entfernung der Hierarchie (Abschnitt 3.3.8) setzt die Ausführung aller vorherigen Schritte voraus, sofern sich diese auf das betreffende Modell anwenden lassen.

Statechart

```
1  statechart PedestrianLight{
2    state PedestrianLightOff
3      <<initial>><<final>>{
4    }
5    state PedestrianLightOn{
6      state Red{
7        [signal==STOP]
8      }
9      state Green{
10       [signal==GO]
11     }
12   }
13
14   Red->Green:
15     buttonPressed()/{ ... } ;
16   Green->Red:
17     timerEvent()/{ ... } ;
18   PedestrianLightOff->Red:
19     switchedOn()/{ ... } ;
20   Red->PedestrianLightOff:
21     switchedOff()/{ ... } ;
22   Green->PedestrianLightOff:
23     switchedOff()/{ ... } ;
24   Red->Red:
25     timeout()/{ ... } ;
26   Green->Green:
27     timeout()/{ ... };
28 }
```

Statechart

```
1  statechart PedestrianLight{
2    state PedestrianLightOff
3      <<initial>><<final>>{
4    }
5    state Red{
6      [signal==STOP]
7    }
8    state Green{
9      [signal==GO]
10   }
11
12   Red->Green:
13     buttonPressed()/{ ... } ;
14   Green->Red:
15     timerEvent()/{ ... } ;
16   PedestrianLightOff->Red:
17     switchedOn()/{ ... } ;
18   Red->PedestrianLightOff:
19     switchedOff()/{ ... } ;
20   Green->PedestrianLightOff:
21     switchedOff()/{ ... } ;
22   Red->Red:
23     timeout()/{ ... } ;
24   Green->Green:
25     timeout()/{ ... };
26 }
```

Abbildung 3.17.: Äquivalente Statecharts mit Superzustand (links) und nach Entfernung der Hierarchie (rechts)

- Die Regel zum Propagieren von Invarianten an Subzustände (Abschnitt 3.3.7) kann jederzeit semantikerhaltend ausgeführt werden. Insbesondere lässt sie sich auch mit der Regel zum Entfernen der Hierarchie aus Abschnitt 3.3.8 kombinieren. Da dieser Schritt als letzter vor dem Entfernen eines Superzustands ausgeführt wird, kann anstatt der Invariante im Superzustand dann auch der gesamte Superzustand entfernt werden. Diese Kombination bietet den Vorteil, dass nicht verfolgt oder ermittelt werden muss, welche Invarianten bereits propagiert wurden.

- Es genügt, das Entfernen von Superzuständen – gegebenenfalls in Kombination mit dem Propagieren von Invarianten – auf oberster Hierarchiestufe im Statechart auszuführen. Alle anderen möglichen Anwendungen der Regel werden hierdurch in der Hierarchie um eine Ebene nach oben verschoben, so dass bei wiederholter Ausführung auch hierdurch ein flaches Statechart entsteht.

- Alternativ können auch alle Regeln nur auf oberster Ebene des Statecharts ausgeführt werden. Dann ist jedoch die Ausführung aller Regeln unter Beachtung der Anwendungsbedingungen so oft zu wiederholen, bis keine der Transformationen mehr anwendbar ist.

In der im Rahmen dieser Arbeit entwickelten Beispielimplementierung wurden die Ersetzungen in der im Abschnitt 3.3 beschriebenen Reihenfolge ausgeführt. Einzige Ausnahme hierzu stellen die Regeln zum Propagieren der Invarianten und zum Entfernen der Superzustände dar, die miteinander kombiniert wurden, sodass das separate Entfernen der Invarianten entfällt. Der vollständige Quellcode der Transformation zur Vereinfachung hierarchischer Statecharts befindet sich im Anhang D.

In dieser Implementierung ist ein wesentlicher Teil der Vereinfachungsregeln für Statecharts umgesetzt, weitere Möglichkeiten finden sich in [Rum11]. Diese lassen auch mit Hilfe der im Folgenden beschriebenen Transformationssprachen umsetzen, gegebenenfalls mit passenden Werkzeugen zur Ergänzung. So ist etwa für die Behandlung von Invarianten ein rein auf der Syntax beruhender Ansatz problematisch, und es sollten geeignete Hilfsmittel eingesetzt werden, um Probleme wie Widerspruchsfreiheit oder wechselseitigen Ausschluss auf den booleschen Ausdrücken zu entscheiden.

3.5. Anforderungen an die Transformationssprache

Der Prozess des Vereinfachens hierarchischer Statecharts dient als grundlegendes Beispiel für die Entwicklung einer Transformationssprache, weil er Transformationsregeln unterschiedlicher Komplexität zur Ausführung auf Modellen in einer praxisrelevanten Sprache beinhaltet. Aus der Umsetzung der im Abschnitt 3.3 beschriebenen Schritte sowie der im Abschnitt 3.4 beschriebenen Verknüpfung der Regeln ergibt sich bereits eine Reihe von Anforderungen an Transformationssprachen. Diese fanden auch Anwendung und wurden ergänzt in zusätzlichen Beispielen, die während der Entwicklung der Transformationsengine für MontiCore erstellt wurden.

- Einzelne Ersetzungsschritte sollen in Transformationsregeln beschrieben werden können. Diese Regeln sollen entsprechend den verbreiteten Ansätzen zur Graphersetzung aus einem Muster- und einem Ersetzungsteil bestehen. Eine integrierte Notation beider Teile kann helfen, doppelten Code in den beiden Teilen der Regel zu vermeiden; jedoch sollen Mustersuche und Ersetzung voneinander unterscheidbar bleiben.

- Die konkrete Syntax der Transformationsregeln soll sich an der konkreten Syntax der zugrunde liegenden Modelle, also der Statecharts orientieren. Die damit einhergehende Einschränkung der Ausdrucksmächtigkeit (beispielsweise können keine Schnittstellen der abstrakten Syntax als Typen verwendet werden) soll dadurch kompensiert werden, dass zusätzliche Constraints für die Anwendbarkeit der Regeln und Änderungsoperationen auch gegen die abstrakte Syntax der Modellierungssprache programmiert werden können.

- Transformationen sollen strukturiert programmiert werden können. Die Transformationssprache soll also Kontrollstrukturen anbieten, welche die bedingte oder wiederholte Ausführung von Anweisungen erlauben. Die Ausführung einer Transformationsregel ist eine

solche Anweisung. Ferner soll es möglich sein, Transformationen in parametrierbare Methoden zu zerlegen, die mit geeigneten Argumenten aufgerufen werden können. Durch diese strukturierte und prozedurale Programmierung wird etwa die Bestimmung aller möglichen Anwendungsstellen von Regeln vor der ersten Ausführung vereinfacht, was beispielsweise die Implementierung des Weiterleitens von Transitionen an Initialzustände (vgl. Abschnitt 3.3.2) deutlich vereinfacht.

Aus diesen drei zentralen Anforderungen ergibt sich eine Unterteilung der Transformationssprache in drei Komponenten, an der sich auch die Struktur der folgenden Kapitel orientiert.

Die erste Komponente ist eine auf Objektdiagrammen basierende Sprache zur Beschreibung von Transformationsregeln. Kern dieser Komponente ist ein Codegenerator, der – ähnlich zu bestehenden Ansätzen der Graphersetzung – einzelne Regeln in Code einer General-Purpose-Programmiersprache übersetzt. Die auf Objektdiagrammen basierende Transformationssprache sowie die zugehörigen Werkzeuge werden in Kapitel 4 beschrieben.

Die zweite Komponente enthält den domänenspezifischen Anteil der Transformationssprache für Statecharts, die Transformationsregeln. Dieser Teil der Transformationssprache basiert syntaktisch auf der Statechartsprache. Solche Regeln können in die vorgenannte, objektdiagrammbasierte Sprache übersetzt und so zur Ausführung gebracht werden. Auf die domänenspezifische Regelsprache und die Übersetzung in Objektdiagramm-Notation wird in Kapitel 5 detailliert eingegangen.

Beide Arten von Transformationsregeln können in eine Kontrollflusssprache eingebettet werden, welche Kernbestandteil der dritten Komponente ist. Diese Sprache erlaubt die Verknüpfung mehrerer Regeln miteinander. In den generierten Code des Kontrollflussanteils einer Transformation wird der aus den Regeln generierte Code eingebunden, und zur Laufzeit der Regeln wird er an den entsprechenden Stellen aufgerufen. Die Kontrollflusssprache für Transformationen in MontiCore wird in Kapitel 6 beschrieben.

Kapitel 4.

Transformationsregeln in Objektdiagramm-Notation

Die in diesem Kapitel vorgestellte Transformationssprache erlaubt es, Modelltransformationsregeln durch zwei Objektdiagramme der UML/P [Rum11, Sch12] zu beschreiben, die dann als linke und rechte Seite einer Graphersetzungsregel interpretiert werden. Als Wirtsgraph bei der Ausführung der Regeln dient der abstrakte Syntaxbaum oder -graph des zu transformierenden Modells. Das Vorgehen bei der Ausführung der Regeln orientiert sich im Wesentlichen an bestehenden Ansätzen, da die effiziente Ausführung von Graphersetzungsregeln schon seit längerem Gegenstand der Forschung ist (vgl. auch Kapitel 2).

Im ersten Abschnitt des Kapitels erfolgt eine Einführung in die Grundlagen der Graphersetzungsregeln, die den folgenden Abschnitten zugrunde liegen. Es folgt eine informelle Beschreibung der Sprache für Transformationsregeln in Objektdiagrammmnotation. Genauer wird anschließend auf die effiziente Mustersuche eingegangen, welche die wesentliche Schwierigkeit bei der Ausführung der linken Regelseite darstellt. Im Abschnitt 4.4 werden die weiterführenden Konzepte der negativen Knoten und Listenknoten vorgestellt, im darauf folgenden Abschnitt der in dieser Arbeit gewählte Ansatz zur Ersetzung, der auf dem Entwurfsmuster Command beruht. Abschließend erfolgt ein Vergleich mit anderen Modelltransformationsengines.

4.1. Grundlagen

Um die Methoden der Graphersetzungssysteme auf Modelltransformationen anwenden zu können, ist es erforderlich, Modelle als Graphen und Transformationsregeln als Graphersetzungsregeln auffassen zu können. Die Abbildung von Modellen einer in MontiCore definierten Sprache auf Graphen wird hier nicht vollständig formalisiert, sondern nur exemplarisch für ein Beispielmodell angegeben, was für das Verständnis der folgenden Ausführungen aber ausreichend ist. Zur abstrakten Syntax mit MontiCore definierter Sprachen wird auch auf [Kra10] verwiesen.

4.1.1. Konkrete Syntax, abstrakte Syntax und attributierte Graphen

Die konkrete und abstrakte Syntax eines ausgewählten Statecharts wurden bereits in Abbildung 2.2 eingeführt. Zur weiteren Betrachtung verwenden wir die Statechartsprache aus Kapitel 3 als Beispiel. Abbildung 4.1 zeigt Ausschnitte eines Statecharts in konkreter Syntax und ein Objektdiagramm der abstrakter Syntax dieses Statecharts.

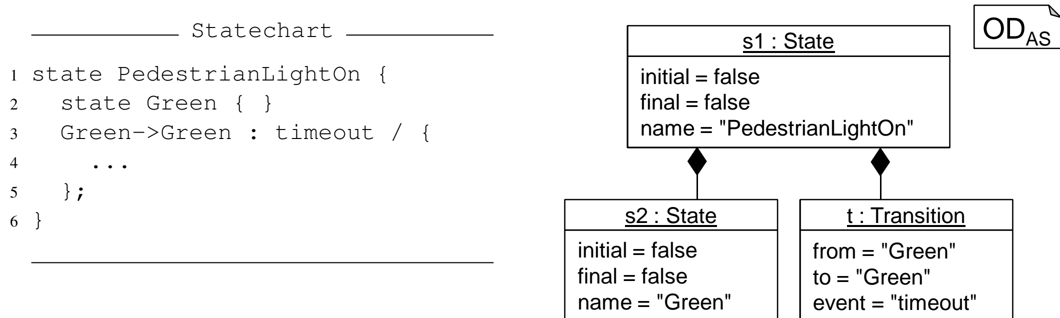


Abbildung 4.1.: Konkrete und abstrakte Syntax eines Statechart-Ausschnitts

Nach dem Parsen liegt ein MontiCore-Modell als abstrakter Syntaxbaum vor, der als Spezialfall eines gerichteten, attributierten, knoten- und kantenmarkierten Graphen aufgefasst werden kann. Werden zusätzlich Assoziationen verwendet (vgl. [Kra10, S. 50 ff.]), so kann das Modell als Graph mit dem AST als Spannbaum aufgefasst werden.

Der Begriff des gerichteten, attributierten, knoten- und kantenmarkierten Graphen wird in dieser Arbeit entsprechend der folgenden Definition verwendet:

Definition 1 *Ein* gerichteter, attributierter, knoten- und kantenmarkierter Graph*, kurz GAKK-Graph, ist ein 7-Tupel* $G := (NL, EL, A, N, E, l, av)$*, und es ist*

- NL *eine endliche Menge von Knotenmarkierungen / -typen / -labeln*
- EL *eine endliche Menge von Kantenmarkierungen / -typen / -labeln*
- A *eine endliche Menge von Attribut(nam)en*
- N *eine endliche Menge von Knoten(bezeichnern)*
- $E \subseteq N \times EL \times N$ *die Kantenrelation*
- $l : N \to NL$ *die Knotenbeschriftungsfunktion*
- $av : N \times A \rightsquigarrow \{true, false\} \cup \mathbb{N} \cup \mathrm{CHAR}^* \cup \ldots$ *die (partielle) Attributwertfunktion.*

Die abstrakte Syntax eines Modells lässt sich nicht nur als Objektdiagramm wie in Abbildung 4.1 darstellen, sondern auch als GAKK-Graph interpretieren. Ein Objektdiagramm kann dann als Visualisierung eines Graphen angesehen werden. In einem GAKK-Graphen zu einem Modell sind:

- NL die Klassen der abstrakten Syntax
- EL die Assoziationen (und auch Kompositionen) der abstrakten Syntax
- A die Menge aller Attribute aller Klassen
- N alle Objekte des abstrakten Syntaxbaums oder -graphen
- $E \ni (s, e, t) \Leftrightarrow$ es gibt im AST einen e-markierten Link von s nach t
- l bildet jedes Objekt auf seine Klasse ab
- av bildet jedes Objekt und alle für dessen Typ gültigen Attribute auf ihren Wert ab.

Mit dieser Interpretationsvorschrift ergibt sich beispielsweise für den Statechartausschnitt aus Abbildung 4.1 ein Graph $G_{SC} := (NL_{SC}, EL_{SC}, A_{SC}, N_{SC}, E_{SC}, l_{SC}, av_{SC})$ mit

$$\begin{aligned}
NL_{SC} &= \{State, Transition\}\\
EL_{SC} &= \{StateContainsState, StateContainsTransition\}\\
A_{SC} &= \{initial, final, name, from, to, event\}\\
N_{SC} &= \{s_1, s_2, t\}\\
E_{SC} &= \{(s1, StateContainsState, s2), (s1, StateContainsTransition, t)\}\\
l_{SC} &: s_1 \mapsto State, s_2 \mapsto State, t \mapsto Transition\\
av_{SC} &: (s_1, initial) \mapsto false, (s_1, final) \mapsto false, (s_1, name) \mapsto \text{„}PedestrianLightOn\text{“},\\
&\quad (s_2, initial) \mapsto false, (s_2, final) \mapsto false, (s_2, name) \mapsto \text{„}Green\text{“},\\
&\quad (t, from) \mapsto \text{„}Green\text{“}, (t, to) \mapsto \text{„}Green\text{“}, (t, event) \mapsto \text{„}timeout\text{“}
\end{aligned}$$

Zu Abbildungen von Modellen auf Objektdiagramme der abstrakten Syntax und auf gerichtete und attributierte Graphen ist anzumerken, dass eine solche Abbildung sowohl auf vollständige Modelle als auch auf Modellfragmente angewendet werden kann. Dies ist eine Grundvoraussetzung für die Anwendung von Graphersetzungsregeln zur Modelltransformation. Wie im Folgenden erläutert wird, lässt sich eine Graphersetzung durch zwei Modellfragmente, die linke und rechte Regelseite (Englisch: left hand side, abgekürzt LHS, bzw. right hand side, abgekürzt RHS), beschreiben, die auf ein vollständiges Modell, den Wirtsgraphen, angewendet wird.

4.1.2. Graphersetzungsregeln

Graphersetzungsregeln sind eine deklarative Beschreibungsform für Änderungsoperationen auf Graphen; im Zuge dieser Arbeit werden dabei nur GAKK-Graphen betrachtet. Der Graph, der durch eine Ersetzungsregel verändert wird, heißt *Wirtsgraph* oder *Hostgraph*.

Graphersetzungsregeln bestehen aus einer linken und einer rechten Regelseite. Informell lässt sich die linke Regelseite als ein Muster im Wirtsgraphen beschreiben, das durch die rechte Regelseite ersetzt wird.

Zur Formalisierung von Graphersetzungsregeln existieren verschiedene Ansätze; diese bauen etwa auf der Mengentheorie [Nag79], der Kategorientheorie [EM85] oder der Prädikatenlogik [Sch91] auf. Mengentheoretische Ansätze verwenden zur Beschreibung der Effekte von Ersetzungsregeln Operationen wie Durchschnitt, Vereinigung oder Differenz. Der kategorientheoretische Ansatz formalisiert Transformationsregeln mit Hilfe von Graphmorphismen. Der logische Ansatz verwendet Formeln zur Beschreibung von Graphen und betrachtet Ersetzungsregeln als Manipulationen der Formelmenge, die den Hostgraphen beschreibt.

Auch innerhalb dieser Ansätze gibt es verschiedene Varianten, die jedoch im Folgenden nicht weiter von Bedeutung sind. In diesem Kapitel wird im Wesentlichen eine vereinfachte Form des kategorientheoretischen Ansatzes nach [EEPT06] zur Erläuterung verwendet[1], die ein anschauliches Fundament für die Implementierung einer Modelltransformations-Engine bietet.

[1] Verzichtet wird hier im Wesentlichen auf die Angabe des vollständigen Kontextes der zu löschenden und der zu verschiebenden Knoten, was die Regeln wesentlich kompakter und intuitiver macht, dafür aber zum Verlust einiger Vorteile dieses Ansatzes führt, wie etwa der Invertierbarkeit von Regeln.

Zur Erläuterung benötigen wir die Begriffe des Morphismus und des Pushouts, die an dieser Stelle nur informell und nur für GAKK-Graphen eingeführt werden. Für eine formale Definition wird zum Beispiel auf [Gol84] verwiesen.

Definition 2 *(informell) Ein* Morphismus *ist im Kontext dieser Arbeit eine Abbildung zwischen zwei GAKK-Graphen, welche die Relationen auf den Graphen erhält, also mit der Kantenrelation, der Knotenbeschriftungsfunktion und der Attributwertfunktion verträglich ist.*

Definition 3 *Seien G, A und B Graphen sowie $a : G \to A$ und $b : G \to B$ Morphismen. Ein Graph P heißt* Pushout *von a und b, falls zwei Morphismen $p_a : A \to D$ und $p_b : B \to D$ existieren, so dass $D = p_a(a(G)) = p_b(b(G))$.*

Die oberen Diagramme in Abbildung 4.2 zeigen eine vereinfachte Variante der Regel zum Weiterleiten von Transitionen an initiale Subzustände[2] (vgl. Abschnitt 3.3.2). Die eigentliche Transformationsregel besteht aus der linken Regelseite `L` und der rechten Regelseite `R`. Hiervon abgeleitet ist der Kern `K` der Regel, der Durchschnitt der linken und rechten Regelseite.

Die unteren Diagramme dieser Abbildung zeigen einen Ausschnitt `G` aus einem Statechart, auf das die Regel angewendet wird, also aus dem Wirtsgraphen, außerdem das Ergebnis `G'` der Regelausführung, sowie den Durchschnitt `D` von `G` und `G'`.

Bei der Ausführung der Regel wird zunächst ein der linken Regelseite entsprechendes Muster im Wirtsgraphen gesucht; diese Musterentsprechung heißt *Match* der Regel.

Definition 4 *Sei $L := (NL_L, EL_L, A_L, N_L, E_L, l_L, av_L)$ ein GAKK-Graph, die linke Regelseite, und $G := (NL_G, EL_G, A_G, N_G, E_G, l_G, av_G)$ ein GAKK-Graph, der Wirtsgraph. Das Bild einer Abbildung $m := (m_{NL}, m_{EL}, m_A, m_N)$ heißt Match für L in G, falls*

$$\begin{aligned} m_{NL} &: NL_L \to NL_G && \textit{die Identität} \\ m_{EL} &: EL_L \to EL_G && \textit{die Identität} \\ m_A &: A_L \to A_G && \textit{die Identität} \\ m_N &: N_L \to N_G && \textit{injektiv} \end{aligned}$$

und m_N ein Morphismus bezüglich der Kantenrelationen E_L und E_G, der Knotenbeschriftungsfunktionen l_L und l_G sowie der Attributwertfunktionen av_L und av_G ist.

Eine Transformationsregel (L, R) ist auf einen Wirtsgraphen G nur dann anwendbar, wenn ein Match für ihre linke Regelseite existiert. Das Resultat der Regelausführung G' wird dann entsprechend Abbildung 4.3 so bestimmt, dass gilt:

$$\begin{aligned} K &= L \cap R \\ D &= (G \setminus m(L) \cup m(K)), \end{aligned}$$

[2] Da nur die Weiterleitung einer einzelnen Transition an einen einzelnen Subzustand erfolgt, ist die Regel in dieser Form alleine nicht semantikerhaltend.

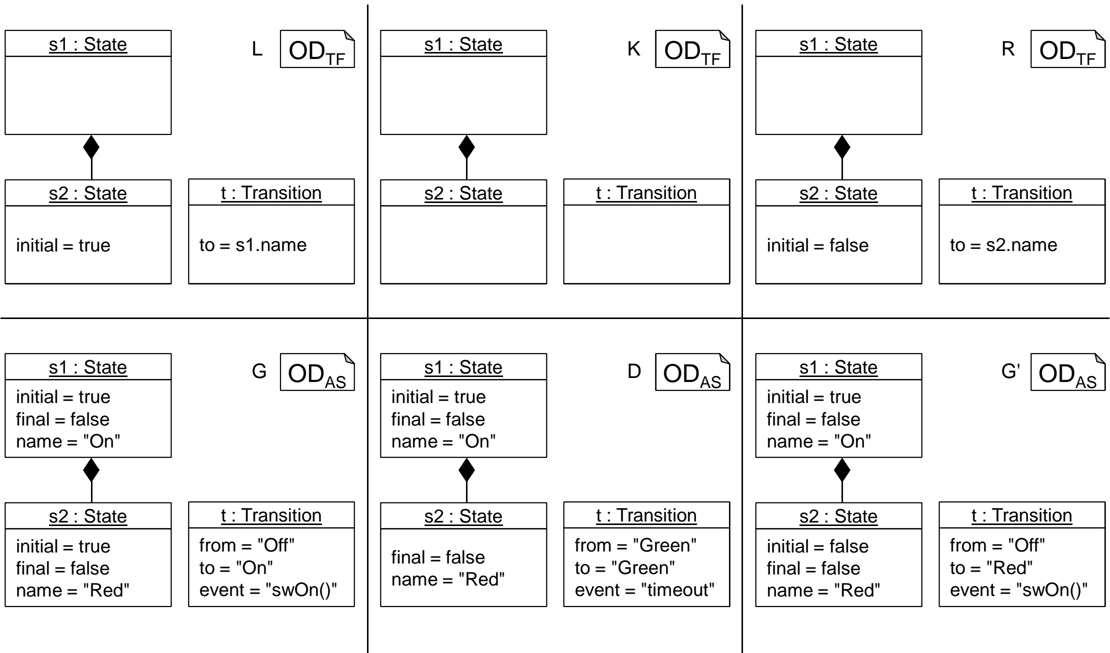


Abbildung 4.2.: Transformationsregel in grafischer Objektdiagramm-Notation und Anwendung auf einen Wirtsgraphen

wobei alle gezeigten Abbildung Morphismen bezüglich der in Definition 4 geforderten Relationen und Funktionen sind, und PO_L und PO_R Pushouts sind. Der Einfachheit halber wurde auf die Darstellung von Knoten- und Kantenmarkierungen sowie Attribute in diesem Beispiel verzichtet.

Aufgrund der geforderten Pushout-Eigenschaften für PO_L und PO_R wird dieses Verfahren in der Literatur als der *Double-Pushout-Ansatz* bezeichnet.

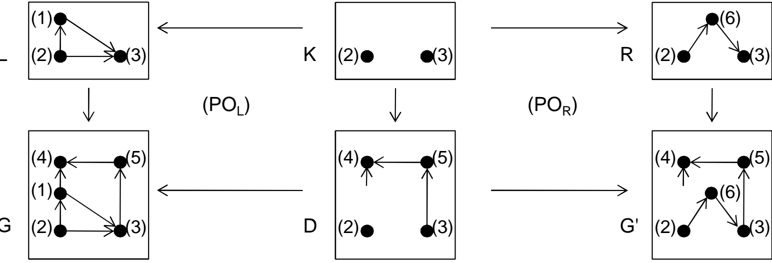


Abbildung 4.3.: Anwendung einer Graphersetzungsregel, die eine hängende Kante zur Folge hätte (Beispiel angelehnt an [EEPT06])

Die Ausführung einer Ersetzungsregel nach dem Double-Pushout-Ansatz kann allerdings – abhängig vom Hostgraphen – in hängenden Kanten resultieren, also Kanten, deren Quell- oder

Zielknoten nicht existiert. Die in Abbildung 4.3 dargestellte Ausführung einer Regel würde beispielsweise den Knoten (1) in G entfernen, die Kante $((1),(4))$ bliebe jedoch erhalten.

Hängende Kanten können aus verschiedenen Gründen problematisch sein. Zunächst sind sie formal unzulässig, weil gemäß Definition 1 die Kantenrelation $E \subseteq N \times EL \times N$ sein muss; folglich ist die Regel auf dem Hostgraphen aus dem Beispiel nicht nach dem Double-Pushout-Ansatz ausführbar. Des Weiteren sind Graphen beziehungsweise Modelle mit hängenden Kanten in vielen Fällen nicht das gewünschte Ziel einer Ersetzungsregel. Außerdem sind bei der Implementierung von Graphen beziehungsweise Modellen in einer objektorientierten Sprache die Kanten oft nicht als eigenständige Objekte, sondern lediglich als Zeiger realisiert. Dies ist auch in MontiCore der Fall. Hier wird durch `null`-Zeiger die Abwesenheit einer Kante dargestellt, und für hängende Kanten gibt es keine Repräsentation.

Eine Weiterentwicklung des Double-Pushout-Ansatzes, bei der das Problem hängender Kanten nicht auftritt, ist der in Abbildung 4.4 dargestellte Single-Pushout-Ansatz [LE91, EHK+97]. Hier wird das Ergebnis der Ersetzungsregel durch den Pushhout bestimmt, der direkt aus L und den Morphismen $L \to R$ und $L \to G$ konstruiert wird. Im Gegensatz zum Double-Pushout-Ansatz sind beim Single-Pushout-Ansatz nicht alle Morphismen total. Wie das Beispiel zeigt, können die Abbildungen $L \to R$ und $G \to G'$ auch partielle Morphismen sein. Durch die Pushout-Konstruktionwerden hängende Kanten automatisch entfernt, wodurch die in Abbildung 4.4 gezeigte Regel nach dem Single-Pushout-Ansatz ausführbar ist, obwohl ihre beiden Regelseiten L und R mit denen aus Abbildung 4.3 übereinstimmen.

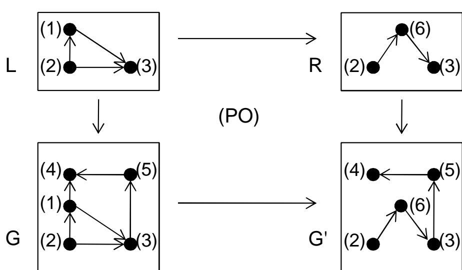


Abbildung 4.4.: Single-Pushout-Diagramm zur Regel aus Abbildung 4.3 (Beispiel angelehnt an [EEPT06])

Auf eine vollständige Formalisierung des Single-Pushout-Ansatzes wird an dieser Stelle verzichtet. Im Folgenden Abschnitt wird die Anwendung der hier beschriebenen Grundlagen auf Modelltransformationen in MontiCore erläutert.

4.2. Aufbau der Regeln

Wie im letzten Abschnitt erläutert wurde, können einzelne Transformationregeln durch Graphersetzungsregeln beschrieben werden, die aus einer linken und einer rechten Regelseite bestehen. Mit den Objektdiagrammen der UML/P stand im Kontext dieser Arbeit bereits eine geeignete Sprache zur Verfügung, auf die zur Beschreibung von Transformationsregeln aufgebaut werden konnte.

Die kontextfreie Syntax der UML/P Objektdiagramme wurde daher mittels Sprachvererbung (vgl. [Völ11]) wiederverwendet, sodass die Grammatik für Transformationsregeln sehr einfach gehalten werden konnte. Hinzugefügt wurden lediglich die Kombination zweier Diagramme zu einer Regel, zusätzlichen Anwendungsbedingungen und besondere Anweisungen für das Pattern Matching sowie einige technische Ergänzungen.

In den Kapiteln 5 und 7 wird erläutert, wie sich Sprachen für Modelltransformationen, die eine domänenspezifische Syntax verwenden, aus gegebenen Modellierungssprachen generieren lassen. Diese Sprache sind zur Beschreibung von Transformationsregeln durch den Benutzer besser geeignet als die in diesem Kapitel vorgestellte Sprache auf Basis von Objektdiagrammen. Die hier vorgestellte Sprache wird jedoch von den domänenspezifischen Sprachen als Zielplattform für die Codegenerierung verwendet, sodass die Definition dieser Sprache für die Transformationsengine in MontiCore zwar erforderlich ist, auf die Einführung besonderen Komforts oder „syntaktischen Zuckers“ jedoch an dieser Stelle verzichtet werden kann.

Wie Abbildung 4.5 zeigt, besteht eine Transformationsregel aus vier Elementen, die im Folgenden vorgestellt werden: Der linken und rechten Regelseite, den Folding-Anweisungen und zusätzlichen Constraints für die Anwendbarkeit der Regel.

MontiCore-Grammatik

```
1 ODRule =
2   "pattern" Lhs:ODDefinition
3   ("replacement" Rhs:ODDefinition)?
4   ("folding" "{" FoldingSet* "}")?
5   ("where" "{" Constraint:BooleanExpression "}")?;
```

Abbildung 4.5.: Auszug aus der MontiCore-Grammatik für Transformationsregeln in Objektdiagramm-Notation

Jede Regel beginnt mit dem Schlüsselwort `pattern` (Abbildung 4.5, Zeile 2). Es folgt das Objektdiagramm der linken Regelseite, auf das in der abstrakten Syntax über das Attribut `Lhs` zugegriffen werden kann. Die Regel `ODDefinition`, auf die hier verwiesen wird, ist in der Obergrammatik, also der Grammatik der UML/P-Objektdiagramme, definiert, von der die Grammatik für Transformationsregeln alle Produktionen erbt.

Als Typen der Objekte im Objektdiagramm müssen die von MontiCore generierten Java-Klassen der abstrakten Syntax angegeben werden. Diese können ebenfalls als Typen von Attributen verwendet werden. Als Attributwerte können beliebige Java-Ausdrücke angegeben werden, insbesondere auch Literale.

Für Links müssen der Name der Assoziation und ein Rollenname angegeben werden, da sonst zu dem Link die Bedingung für das Matching nicht immer eindeutig bestimmt werden kann. Im Fall von Kompositionen entfällt der Assoziationsname; hier ist der Rollenname eindeutig und somit ausreichend.

Der optionale Ersetzungsteil der Regel wird durch das Schlüssewort `replacement` eingeleitet (Abbildung 4.5, Zeile 3). Das Attribut `Rhs` verweist ebenfalls auf ein Objektdiagramm der UML/P. Fehlt die Angabe der rechten Regelseite, so beschreibt die Regel lediglich eine Mustersuche; dieser Fall wird also so behandelt, als ob linke und rechte Regelseite identisch wären. Eine leere rechte Regelseite bedeutet hingegen, dass alle Objekte und Links des Matches aus

dem Hostgraphen entfernt werden. Für die Angabe von Typen sowie die Definition von Links und Attributen gelten die gleichen Bedingungen wie auf der linken Regelseite.

Auf das Schlüsselwort `folding` folgt eine Liste beliebig vieler Folding-Mengen, durch die nicht-isomorphe Matches zugelassen werden können (Abbildung 4.5, Zeile 4). Eine Folding-Menge besteht aus mindestens zwei Objektnamen. Für Objekte, die in einer Folding-Menge stehen, wird beim Matching nicht überprüft, ob die entsprechenden Objekte im Hostgraphen voneinander verschieden sind. So lässt sich beispielsweise bei der Mustersuche in Statecharts nach einer Transition sowie ihrem Quell- und einem Zielzustand steuern, ob Quell- und Zielzustand identisch sein dürfen. Die Angabe von Folding-Mengen ist optional.

Letzter und ebenfalls optionaler Bestandteil ist die Angabe eines Constraints für die Anwendbarkeit der Regel, der durch das Schlüsselwort `where` eingeleitet wird (Abbildung 4.5, Zeile 5). Der Rumpf des Constraints kann ein beliebiger boolescher Java-Ausdruck sein. Ein solcher Ausdruck gibt eine zusätzliche Bedingung an, die für alle Matches zu der Regel gelten muss. In der Implementierung ist die Einbindung des Constraints über Spracheinbettung (vgl. [Völ11]) gelöst.

In Abbildung 4.6 ist die vereinfachte Transformationsregel zum Umleiten von Transitionen an initiale Subzustände (vgl. Abschnitt 4.1.2 und Abbildung 4.2) in der textuellen Fassung dargestellt. Im Vergleich zu den bisher gezeigten Versionen der Regel ist in dieser Fassung auch der Quellzustand der Transition angegeben. In den Zeilen 3 bis 14 sind insgesamt vier Objekte definiert, die drei Zustände `state_1` bis `state_3` und eine Transition `transition_1`.

Durch das Beispiel in der Abbildung sind auf der linken Regelseite (Zeilen 1–18) drei Attributwerte festgelegt: Für `state_3` ist gefordert, dass das boolesche Attribute `initial` den Wert `true` hat (Zeile 8). Hier ist der Attributwert also durch ein Boolean-Literal fest vorgegeben. Für `transition_1` sind die Werte der String-Attribute `from` und `to` gegeben (Zeilen 12 f.). Bei diesen Werten handelt es sich jedoch nicht um feste Literale, sondern um Verweise auf Attribute anderer Objekte. Diese Namen erlauben es, Verbindungen zu anderen Objekten im AST zu beschreiben. Durch die Verwendung solcher Referenzen wird zwar sichergestellt, dass die Transition in den angegebenen Zuständen beginnt beziehungsweise endet, es wird jedoch kein fixer Name für diese Zustände festgelegt.

Durch die Komposition in Zeile 16 und die Angabe des Rollennamens `(states)` ist festgelegt, dass `state_3` ein Unterzustand von `state_2` sein muss.

Die rechte Regelseite (Zeilen 18–34) unterscheidet sich von der linken in zwei Punkten; dies sind genau die Änderungen, die bei Ausführung der Regel vorgenommen werden, falls ein Match für die linke Seite existiert. Erstens hat das Attribut `initial` des Zustandes `state_3` den Wert `false` (Zeile 25). Zweitens entspricht der Wert des `to`-Attributs der Transition `transition_1` jetzt dem Wert des `name`-Attributs von `state_3` (Zeile 29), also des vormals initialen Subzustandes.

Die Angabe unveränderter Attribute, wie `transition_1.from`, kann auf der rechten Regelseite entfallen. Die Angabe der unveränderten Objekte und Links ist dagegen erforderlich, weil Objekte und Links, die nur auf der linken Regelseite angegeben sind, durch die Regel gelöscht werden.

```
Transformationsregel in Objektdiagramm-Notation
1  pattern objectdiagram lhs{
2
3    state_1:ASTState;
4
5    state_2:ASTState;
6
7    state_3:ASTState{
8      boolean initial = true;
9    }
10
11   transition_1:ASTTransition{
12     String from = state_1.name;
13     String to = state_2.name;
14   }
15
16   composition state_2 -- (states) state_3 ;
17
18 }replacement objectdiagram rhs{
19
20   state_1:ASTState;
21
22   state_2:ASTState;
23
24   state_3:ASTState{
25     boolean initial = false;
26   }
27
28   transition_1:ASTTransition{
29     String to = state_3.name;
30   }
31
32   composition state_2 -- (states) state_3 ;
33
34 }
```

Abbildung 4.6.: Weiterleiten von Transitionen an initiale Subzustände

4.3. Effizientes Pattern-Matching

Die Mustersuche in Graphen wird bei naiver Implementierung schnell ineffizient. So enthält der AST des Statecharts in Abbildung 3.2 beispielsweise 37 Objekte. Für die Zuordnung der vier Objekte der linken Regelseite in Abbildung 4.6 zu diesen Objekten ergeben sich bereits $37^4 = 1.874.161$ Möglichkeiten.

Die Anzahl möglicher Matches lässt sich bereits durch einfache Maßnahmen deutlich reduzieren. Die Gruppierung der Objekte nach ihrem Typ führt beispielsweise zu vier Zuständen und vier Transitionen, auf die sich für die Abbildung der drei Zustände und der einen Transition der oben genannten Regel nur noch $4^3 \cdot 4^1 = 256$ Möglichkeiten ergeben. Dieses Rechenbei-

spiel zeigt aber die Bedeutung eines effizienten Algorithmus' zur Mustersuche für die effiziente Regelausführung.

Tatsächlich beruht die Mustersuche in GAKK-Graphen auf dem Problem der Teilgraphisomorphie, für das sich die NP-Vollständigkeit durch Reduktion auf CLIQUE und 3-SAT nachweisen lässt [Coo71].

Im Folgenden wird ein Algorithmus vorgestellt, der aus [BKG08] übernommen und im Rahmen dieser Arbeit sowie in [Höl10] für MontiCore angepasst und implementiert wurde. Der ausschlaggebende Grund für die Auswahl dieses Algorithmus war die bessere Laufzeit, die im Vergleich zu anderen Ansätzen in verschiedenen Fallstudien ermittelt wurde [TBB+08, GBG+06]. Diese Laufzeitverbesserung wird durch die Erstellung eines *modellsensitiven Suchplans* erreicht, der neben der Transformationsregel auch Informationen aus dem Hostgraphen mit einbezieht, um eine optimale Strategie für die Mustersuche zu wählen.

4.3.1. Reduktion des Suchraums

Bevor mit der eigentlichen Suche nach einem Match begonnen wird, erfolgt die Bestimmung eines Suchplans. Dieser Suchplan legt die Reihenfolge fest, in der Entsprechungen für die einzelnen Elemente der linken Regelseite im Hostgraphen gesucht werden. Ein möglicher Suchplan für die Transformationsregel aus Abbildung 4.6 wäre zum Beispiel

$$[state_1, state_2, transition_1, state_3].$$

Der Suchplan ist als Kellerspeicher implementiert, die Kellerspitze ist rechts dargestellt. Bei diesem Suchplan wird also als erstes ein mögliches Match für `state_3` gesucht.

Abbildung 4.7 zeigt einen gerichteten Baum, der einen Ausschnitt aus dem Suchraum für die Anwendung der Transformationsregel auf das Statechart `PedestrianLight` aus Abbildung 3.8 darstellt. Die Knoten repräsentieren die möglichen Matches für die einzelnen Objekte, jede Zeile repräsentiert also die Suche nach einem Match für ein Objekt. Die Kanten zeigen jeweils zu den nachfolgend auszuprobierenden Kombinationen. Die Suche nach einem Match für das gesamte Muster kann als Tiefensuche in diesem Baum verstanden werden.

Bei einer vollständigen Tiefensuche in dem Baum, die Kindknoten von links nach rechts traversiert, wird also zunächst versucht, `state_3` auf `PedestrianLightOff` zu matchen. Hierfür werden dann alle möglichen Matches für `transition_1`, `state_2` und `state_1` ausprobiert. Da hierfür kein Match existiert, wird versucht, `state_3` auf `PedestrianLightOn` zu matchen. Auch hierfür werden alle möglichen Matches der übrigen Objekte der linken Regelseite durchprobiert, erneut ohne ein gültiges Match zu finden.

Als Nächstes ergibt sich die Zuordnung von `state_3` zum Zustand `Red` und von `transition_1` zu der Transition, die von `PedestrianLightOff` nach `PedestrianLightOn` führt. Für diese Zuordnung wird zuerst versucht, `state_2` auf `PedestrianLightOff` abzubilden, hierfür scheitern aber alle möglichen Zuordnungen von `state_1`. Der nächste Kandidat für `state_2` ist `PedestrianLightOn`. Hierzu ergibt sich für den ersten Kandidaten für `state_1`, nämlich `PedestrianLightOff` ein gültiges Match, das in der Abbildung durch den fettgedruckten Pfad hervorgehoben ist.

Alle Pfade, die rechts vom Match liegen, werden bei der Suche nach einem einzelnen Match nicht mehr traversiert. In Abbildung 4.7 sind diese Pfade gestrichelt dargestellt.

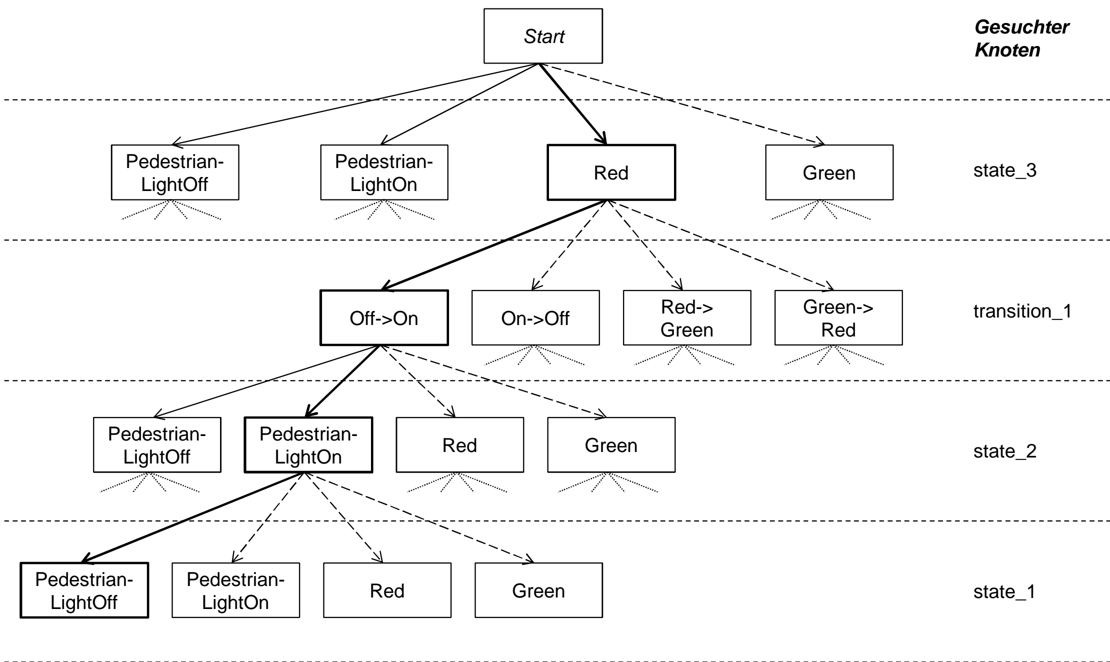


Abbildung 4.7.: Ausschnitt des Suchraums zur Transformationsregel aus Abbildung 4.6

Der Aufwand für die Suche nach diesem Verfahren ist jedoch immer noch hoch. Eine erste Verbesserung lässt sich dadurch erreichen, dass Bedingungen für die einzelnen Objekte möglichst frühzeitig geprüft werden. So muss `state_3` ein Anfangszustand sein. Dies kann bereits unmittelbar nach dem Matching von `state_3` geprüft werden; so lässt sich die Traversierung aller Teilbäume unter `PedestrianLightOn` in der Zeile zu `state_3` vermeiden. Andere Bedingungen können dagegen erst überprüft werden, nachdem für beide an der Bedingung beteiligten Objekte ein Kandidat gesucht wurde, so etwa die Bedingung, dass `state_2` der Zielzustand von `transition_1` ist.

4.3.2. Kostenmodell und Optimierung des Suchplans

Weitere Verbesserungen der Laufzeit lassen sich erreichen, wenn zur Bestimmung des Matches für ein Objekt Informationen aus den Kandidaten für andere Objekte verwendet werden. In der Regel, die in Abbildung 4.6 dargestellt ist, wird gefordert, dass `state_3` ein Unterzustand von `state_2` ist. Wird wie in Abbildung 4.7 zunächst `state_3` gematcht, so kommen als Belegung für `state_2` nicht mehr alle Zustände in Frage, sondern nur noch der Oberzustand des aktuellen Kandidaten für `state_3`. (Falls ein solcher nicht existiert, kann der Kandidat für `state_3` nicht zu einem gültigen Match führen.)

Verallgemeinert man diese Beobachtung, so kommt man zu dem Ergebnis, dass die Kosten, also die erforderliche Rechenzeit zum Matchen eines Objektes, von den zuvor bestimmten Objekten abhängt. Typischerweise greift man sich ein einzelnes bereits gematchtes Objekt heraus und versucht, über mit diesem Objekt verbundene Kanten zu einem weiteren Objekt zu gelangen.

Die Kosten hierfür hängen zum einen von der Navigierbarkeit und Multiplizität der Kantentypen im Graphschema ab. Zum anderen können auch die Anzahl der Knoten im Hostgraphen und ihre Eigenschaften zur Verbesserung der Kostenabschätzung herangezogen werden. So lassen sich beispielsweise die beiden Anfangszustände im Beispiel schneller traversieren als alle vier Transitionen, weswegen es sinnvoll ist, `state_3` vor `transition_1` zu matchen. Abbildung 4.8 zeigt die erwarteten Kosten für das Matching der Objekte aus dem Beispiel.

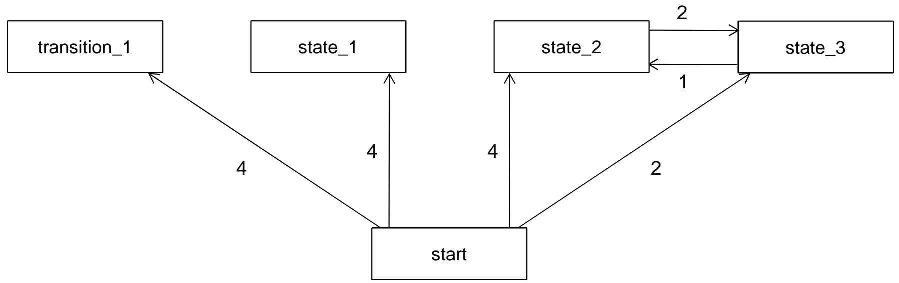


Abbildung 4.8.: Kostenmodell zur Regel `ForwardToInitial` und dem Statechart `PedestrianLight`

Die Kanten sind mit den Kosten zur Bestimmung des Zielobjektes beschriftet, falls für das Quellobjekt bereits ein Kandidat existiert. Die Kantengewichte für das Beispiel ergeben sich wie folgt:

- Ausgehend vom Startknoten, der auch verwendet werden kann, wenn noch kein Objekt gematcht wurde, kann `state_3` mit den Kosten 2 (der Anzahl der Initialzustände) erreicht werden. Alle anderen Objekte können mit den Kosten 4 (Anzahl der Transitionen beziehungsweise Zustände im Graphen) erreicht werden.
- Von `state_3` aus kann `state_2` mit den Kosten 1 erreicht werden, da ein Zustand höchstens einen Oberzustand haben kann.
- Umgekehrt betragen die Kosten zum Erreichen des Zustands `state_2` von `state_3` aus 2, denn der einzige Zustand im Hostgraphen, der überhaupt Subzustände hat, verfügt über zwei Unterzustände.

Für einen Suchplan S wie in Abbildung 4.7 seien nun n die Anzahl der Zeilen und c_i der ausgehende Kantengrad der Knoten in der i-ten Zeile. Die Gesamtkosten des Suchplans $c(S)$ für den Fall, dass alle Matches bestimmt werden sollen, sind dann

$$c(S) = \sum_{k=1}^{n} \prod_{i=1}^{k} c_i. \tag{4.1}$$

Falls für alle Objekte Kandidaten zum Matching existieren, sind in diesem Term alle $c_i > 0$. Den größten Einfluss auf die Gesamtkosten hat dann die Suche nach dem letzten Objekt, also

der letzte Summand. Für die Kosten dieser Suche $c(S_n)$ gilt

$$c(S_n) = \prod_{i=1}^{n} c_i. \tag{4.2}$$

Diese Kosten gilt es also zu minimieren. Da alle $c_i > 0$ sind, ist der Term genau dann minimal, wenn auch

$$log(c(S_n)) = \sum_{i=1}^{n} log(c_i) \tag{4.3}$$

minimal ist. Beschriftet man den Graphen in Abbildung 4.7 mit den logarithmischen Kosten, so lässt sich die Suche nach dem optimalen Suchplan auf die Bestimmung eines minimalen Spannbaums über diesem Graphen reduzieren. Da die Graphen in MontiCore-Modellen Bäume sind, sofern nicht Assoziationen in der Grammatik verwendet werden, wurden noch einige Anpassungen dieses Verfahrens vorgenommen, die jedoch für die folgenden Ausführungen nicht relevant sind.

Jede Kante des minimalen Spannbaums entspricht beim Pattern-Matching der Suche nach einem Objekt. Da die Kosten für das Matching der Objekte, die früh gesucht werden, einen besonders großen Einfluss auf die Gesamtkosten haben (vgl. Gleichung 4.1), ist es sinnvoll, Objekte mit geringen Kosten möglichst früh zu matchen.

Für die Ausführung der Regel `ForwardToInitial` über dem Statechart `PedestrianLight` ergibt sich aus diesen Optimierungen, dass für `state_3` nur Initialzustände durchsucht werden und für `state_2` nur der Oberzustand von `state_3`, sowie ein vorzeitiger Abbruch der Tiefensuche auf Pfaden, an denen die Namen für Start und Ziel von `transition_1` nicht mit den Namen der jeweiligen Zustände übereinstimmen.

Abbildung 4.9 zeigt den deutlich verkleinerten Suchraum nach der Optimierung des Pattern-Matchings. Knoten, an denen die Suche frühzeitig abgebrochen werden kann, sind durchgestrichen dargestellt. Insgesamt hat der Baum, auf dem die Tiefensuche stattfindet, nur noch 20 Blätter, von denen der Pfad zum fünften Blatt die Knoten des ersten und einzigen Matches enthält. Dies stellt gegenüber dem vollständigen Durchprobieren aller Objekte beinahe eine Verbesserung um den Faktor 10^5 und gegenüber der einfachen Implementierung, bei der nur nach Typen gruppiert wird, immerhin noch um Faktor zwölf dar. Allerdings entsteht durch die Optimierung ein gewisser Overhead, der sich in der bisherigen Anwendung aber als akzeptabel erwiesen hat. Auf die Details wird im folgenden Implementierungsabschnitt eingegangen.

4.3.3. Implementierung des Pattern-Matching-Algorithmus

Die Optimierungen lassen sich grundsätzlich in zwei Kategorien unterteilen: Solche, die nur auf der Transformationsregel beruhen und solche, die auch Informationen aus dem Hostgraphen mit einbeziehen. In die erste Kategorie fällt beispielsweise das frühzeitige Abbrechen der Suche, falls Bedingungen für Attributwerte verletzt sind. Zur zweiten Kategorie gehört dagegen das frühzeitige Matchen von Objekten mit geringen Kosten, da die erwarteten Kosten für das Matching der Objekte vom Hostgraphen abhängig sind.

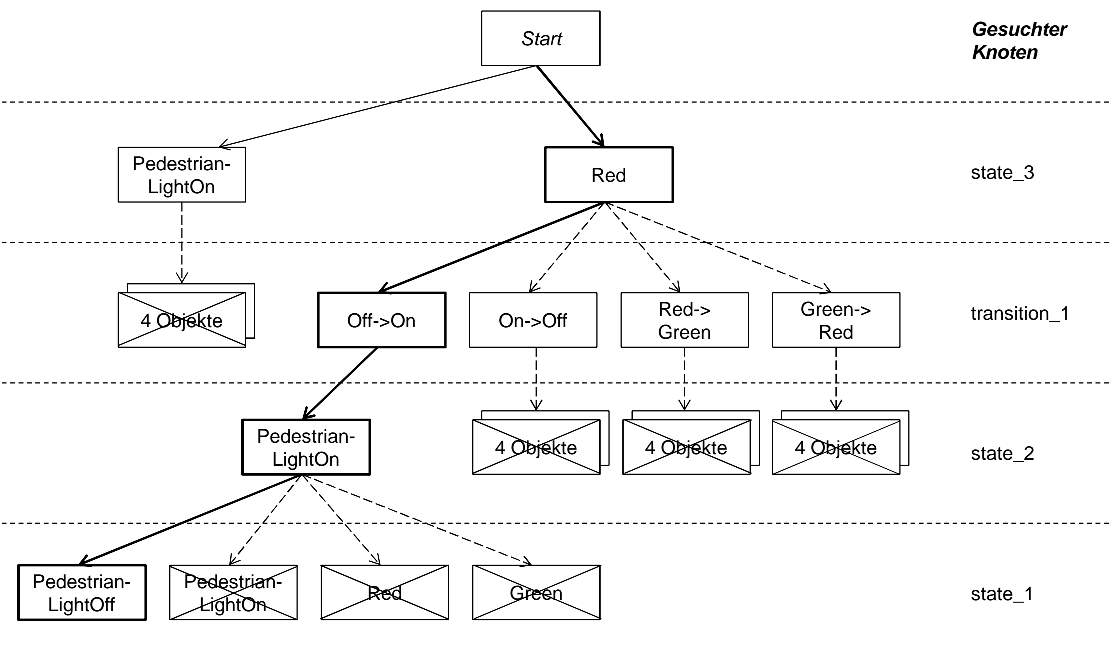


Abbildung 4.9.: Optimierter Suchraum zur Transformationsregel aus Abbildung 4.6

Die Ausführung der Regeln kann entweder durch Codegenerierung und die Ausführung des generierten Codes, durch einen Interpreter oder durch eine Mischform dieser beiden Ansätze erfolgen. Die Ausführung generierten Codes hat gegenüber dem Interpreter-Ansatz zwei wesentliche Vorteile: Zum einen ist die Ausführung eines Kompilats meist schneller als die Interpretation der Quellen [HCJ+97, JBSM08], und zum anderen ist das Resultat der Codegenerierung einsehbar und verarbeitbar, zum Beispiel durch statische Analysen, Tests oder Transformationen.

Ein rein generativer Ansatz würde für modellsensitive Suchpläne aber eine Just-In-Time-Übersetzung der Regeln und das dynamische Nachladen auf der Zielplattform über Reflektionsmechanismen vor der Ausführung einer Regel erfordern.

Die Implementierung des Pattern-Matching-Algorithmus ist daher zwar aus der jeweiligen Transformationsregel generiert. Vor Beginn des Matchings erfolgt aber für die modellsensitive Optimierung zusätzlich eine Analyse des Hostgraphen, mit deren Ergebnis der aus der Regel generierte Code konfiguriert wird.

Das Klassendiagramm in Abbildung 4.10 zeigt wichtige Elemente des generierten Codes einer Transformationsregel. Zu jeder Regel wird eine eigene Klasse generiert[3]. Diese Klasse verfügt mit dem Attribut `allMatches` über eine Liste aller gefundenen Matches zu dieser Regel. Diese sind jeweils in einem Objekt gespeichert, dessen Typ die innere Klasse `Match` ist. Ein `Match`-Objekt enthält zu jedem Objekt der Regel einen Verweis auf das entsprechende Objekt im Hostgraphen.

[3] Jede Instanz dieser Klasse repräsentiert eine Ausführung der Regel und hält auch Informationen über den Status der Ausführung. Dies ist insbesondere erforderlich, wenn eine Regel rückgängig gemacht werden soll oder mehrfach ausgeführt werden muss. Hierauf wird im Kapitel 6 näher eingegangen.

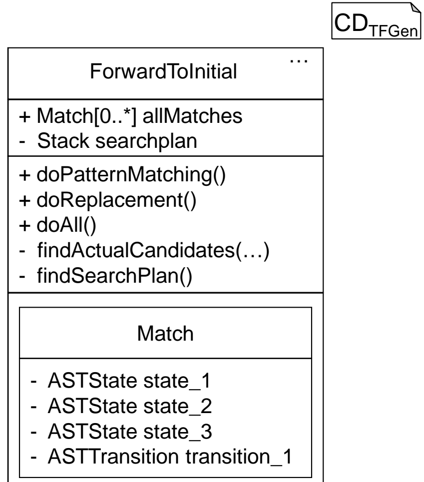


Abbildung 4.10.: Klassendiagramm des aus einer Transformationsregel generierten Codes

Die private Methode `findSearchPlan` wird zu Beginn des Pattern-Matchings aufgerufen und berechnet den Suchplan, der in der Instanzvariable `searchplan` gespeichert wird. Spätere Neuberechnungen der Kandidatenmengen zu einzelnen Regelobjekten werden von der Methode `findActualCandidates` vorgenommen.

Daneben verfügt die Klasse über die öffentlichen Methoden `doPatternMatching` zum Durchführen der Mustersuche, `doReplacement` zum Durchführen der Ersetzung und `doAll`, welche die beiden vorgenannten Methoden kapselt. Die Durchführung des Ersetzungsteils wird im folgenden Abschnitt beschrieben.

4.4. Negative Objekte und Listenobjekte

Mit den bisher vorgestellten Elementen der Regelsprache lassen sich bereits viele Transformationsregeln aus dem Prozess der Vereinfachung hierarchischer Statecharts umsetzen. Oft sind aber weiterführende Konzepte hilfreich, von denen an dieser Stelle zwei vorgestellt werden: die negativen Objekte, mit denen sich beschreiben lässt, was im Kontext eines Matches nicht existieren darf, und die Listenobjekte, die auf eine beliebige Anzahl von Knoten des Hostgraphen gematcht werden können.

Abbildung 4.11 zeigt die Verwendung eines negativen Knotens (Zeile 5) und zweier Listenknoten (Zeilen 9 und 21) in einer Transformationsregel[4]. Diese Regel transformiert alle Subzustände eines Zustands in Endzustände, falls für diesen Zustand noch kein expliziter Endzustand exitiert. Hierbei handelt es sich um einen Teil der in Abschnitt 3.3.3 beschriebenen Transformation.

[4]Die Regel ist in der hier angegebenen Form nicht parsebar, weil sie das Schlüsselwort `final` als Attributnamen enthält. Wird sie jedoch – wie in Kapitel 5 beschrieben wird – generiert, liegt bereits der Syntaxbaum zu dieser Regel vor, und die Verwendung des Attributnamens ist unproblematisch.

Transformationsregel in Objektdiagramm-Notation

```
1 pattern objectdiagram lhs{
2
3   state_1:mc.testcases.statechart._ast.ASTState;
4
5   <<not>> state_2:mc.testcases.statechart._ast.ASTState{
6     boolean final = true;
7   }
8
9   <<list>> $L:mc.testcases.statechart._ast.ASTState;
10
11  composition state_1 -- (states) state_2 ;
12
13  composition state_1 -- (states) $L ;
14
15 }replacement objectdiagram rhs{
16
17  state_1:mc.testcases.statechart._ast.ASTState;
18
19  state_2:mc.testcases.statechart._ast.ASTState;
20
21  <<list>> $L:mc.testcases.statechart._ast.ASTState{
22    boolean final = true;
23  }
24
25  composition state_1 -- (states) state_2 ;
26
27  composition state_1 -- (states) $L ;
28
29 }where {
30   !match.$L.isEmpty()
31 }
```

Abbildung 4.11.: Zwei Listenknoten und ein negativer Knoten in einer Transformationsregel

Das negative Objekt, markiert durch den Stereotype <<not>>, gibt an, dass die Regel nur dann matcht, wenn alle Objekte ohne Stereotyp ein gültiges Match haben, aber das negierte Objekt nicht.

Weil negative Objekte kein Match haben, können sie auch durch die rechte Regelseite nicht erzeugt oder modifiziert werden. Negative Objekte dürfen daher nur auf der linken Regelseite vorkommen.

Das Listenobjekt in Zeile 9, markiert durch den Stereotyp <<list>>, wird auf eine Liste von Objekten im Hostgraphen gematcht, wobei versucht wird, eine Liste mit einer möglichst großen Anzahl von Objekten zu matchen.

Das Listenobjekt in Zeile 21 trägt den gleichen Bezeichner wie das Listenobjekt in Zeile 9. Es beschreibt also Änderungen von Attributwerten, in diesem Fall der Attributwerte aller Objekte in der Liste. Für jedes dieser Objekte wird das Attribut `final` auf den Wert `true` gesetzt.

Abbildung 4.12 veranschaulicht das Matching von Listenobjekten. Das Pattern auf der linken Seite hat ein Match, sofern für ein beliebiges $n \in \mathbb{N}_0$ ein Match zu dem entsprechenden Muster auf der rechten Seite gefunden werden kann. Ist dies für mehrere n möglich, so werden möglichst viele Objekte dem Match hinzugefügt.

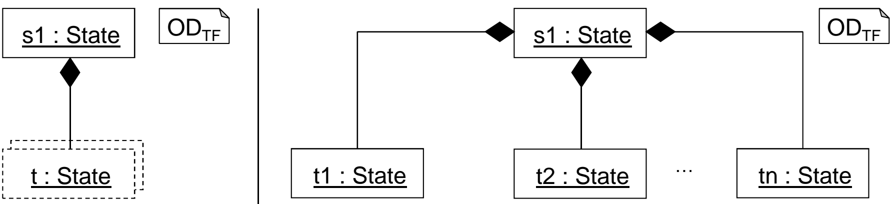


Abbildung 4.12.: Pattern mit Listenobjekt (links) und Übersetzung in Listen von Objekten

Listenobjekte dürfen in der rechten Regelseite nur verwendet werden, wenn auf der linken Seite ein Listenobjekt gleichen Namens existiert.

4.5. Ersetzung mit dem Entwurfsmuster Command

Obwohl der Aufruf einer Regel innerhalb eines komplexeren Programms als Ausdruck aufgefasst werden kann, wird durch den Codegenerator für jede Transformationsregel eine eigene Java-Klasse erstellt. Um den Anweisungscharakter der Regeln zu erhalten, ist es sinnvoll, die Instanziierung einer Regel und ihre Ausführung als Kommando im Sinne des Entwurfsmusters Command [GHJV95] zu kapseln. Ein weiterer Grund, dieses Entwurfsmuster zu verwenden, ist die Notwendigkeit, die von einer Regel vorgenommenen Änderungen rückgängig zu machen, worauf in Kapitel 6 noch näher eingegangen wird.

Für die Transformationsengine in MontiCore wird eine leicht angepasste Version dieses Entwurfsmusters verwendet, von der eine konkrete Ausprägung für das Beispiel der Vereinfachung von Statecharts im Klassendiagramm in Abbildung 4.13 dargestellt ist.

Die abstrakte Klasse `ODRule` ist die gemeinsame Oberklasse aller aus Transformationsregeln generierten Java-Klassen. Sie ersetzt die Klasse `Commmand` aus [GHJV95]. Jede aus einer Transformationsregel generierte Klasse ist somit eine konkrete Kommandoklasse.

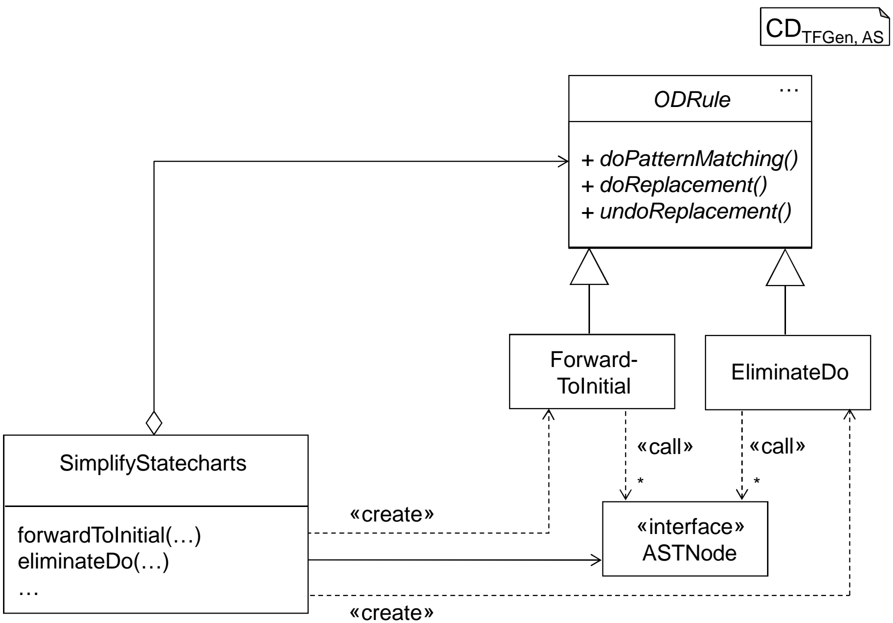


Abbildung 4.13.: Adaption des Entwurfsmusters Command für Transformationsregeln

Ein komplexes Transformationsprogramm, im Beispiel `SimplifyStatecharts`, operiert auf einem Wirtsgraphen, repräsentiert durch seinen Wurzelknoten vom Typ `ASTNode`. Es erzeugt bei Bedarf Instanzen der Transformationsregeln, im Beispiel `ForwardToInitial` und `EliminateDo`. Diese Erzeugung und die Ausführung einer Regel erfolgt immer innerhalb einer gleichnamigen Methode.

Die Transformationsregeln rufen dann ihrerseits beim Pattern-Matching und bei der Ersetzung Methoden auf den AST-Knoten, zum Beispiel Zugriffsmethoden auf Attribute, auf. Zusätzlich werden Factories [GHJV95] zum Erzeugen von AST-Knoten verwendet, die jedoch der Übersicht wegen in der Abbildung nicht aufgeführt sind.

Im Vergleich zu dem in [GHJV95] beschrieben Command-Muster entfällt außerdem die Unterscheidung zwischen Client und Invoker, da sowohl die Erzeugung als auch der Aufruf der Transformationsregeln aus dem Hauptprogramm erfolgen.

4.6. Verwandte Arbeiten

Neben dem in diesem Kapitel bisher betrachteten Ansatz, in dem der Suchbaum auf geeignete Weise umstrukturiert und durch geeignete Abbildungen effektiv verkleinert wird, existieren einige alternative Ansätze, um das NP-vollständige Probleme der Graphmustersuche möglichst effizient zu lösen. Die wichtigsten Ansätze werden in diesem Abschnitt vorgestellt.

4.6.1. Verwendung von Constraint-Solvern

Die Suche nach einem isomorphen Teilgraphen zu einem gegebenen Muster lässt sich als Constraint-Satisfaction-Problem formulieren. Zur Lösung dieses Problems können dann Constraint-Solver eingesetzt werden, die über umfangreiche Optimierungsverfahren zur Lösungssuche verfügen. Ansätze dieser Art werden zum Beispiel in [Rud00] und [LV02] beschrieben.

Bei einem *Constraint-Satisfaction-Problem*, kurz CSP, geht es darum, für eine Menge von Variablen $V = v_1, \ldots, v_n$ Werte festzulegen. Diese Werte müssen aus einer bestimmten Menge, der *Domäne* D_{v_i} der Variablen, stammen, wobei die Domänen verschiedener Variablen identisch sein können. Zum anderen müssen die Werte der Variablen eine gegebene Menge von Constraints erfüllen. Ein Constraint c ist eine Relation über mehreren Domänen, also $c \subseteq D_{v_{r_1}} \times \ldots \times D_{v_{r_k}}$, die zum Beispiel durch eine aussagenlogische Formel gegeben sein kann.

Für das Graphmuster aus Abbildung 4.2, eine Abbildung Γ der Variablen auf ihre Werte und den Hostgraphen $G = (NL, EL, A, N, E, l, av)$ erhält man zum Beispiel das CSP mit

$$
\begin{aligned}
V &:= \{s_1, s_2, t, e\} \\
D_v &:= N \text{ für } v \in \{s_1, s_2, t\} \\
D_e &:= E \\
C &:= c_{NL}, c_{EL}, c_{av} \\
c_{NL} &\text{ gegeben durch } NL(\Gamma(s_1)) = NL(\Gamma(s_2)) = State \wedge NL(\Gamma(t)) = Transition \\
c_{EL} &\text{ gegeben durch } EL(\Gamma(e)) = StateContainsState \\
c_{av} &\text{ gegeben durch } av(\Gamma(s_2), initial) = true \wedge av(\Gamma(t), to) = av(\Gamma(s_1), name)
\end{aligned}
$$

Werkzeuge, die einen CSP-basierten Ansatz zum Pattern-Matching verfolgen, sind zum Beispiel AGG [Tae04] und VIATRA [VB07]. Ein wesentlicher Nachteil dieses Ansatzes ist, dass hier keine Optimierungen vorgenommen werden, die sich auf die abstrakte Syntax der Modellierungssprache beziehen, etwa die bevorzugte Verwendung navigierbarer Assoziationen oder Multiplizitäten. Dies könnte ein Grund dafür sein, dass die Performance CSP-basierter Werkzeuge in den betrachteten Benchmarks weniger gut war als die von Engines mit suchplangesteuertem Matching [TBB+08].

4.6.2. Abbildung auf das relationale Datenmodell

Eine weitere Möglichkeit, sich existierende Ansätze zur Optimierung zunutze zu machen, ist die Persistierung der Modelle in relationalen Datenbanken und die Abbildung der Mustersuche in Modellen in eine Anfragesprache für Datenbanken, zum Beispiel auf SQL. Sowohl die Datenstruktur als auch die Anfragen werden von den gängigen Datenbankmanagementsystemen optimiert [Vos00, Kapitel 8 und 18], wobei die angewandten Optimierungsschritte durch den Nutzer beeinflussbar sein können.

In der Abbildung werden Klassen auf Relationen, Attribute auf Felder dieser Relationen, und Objekte mit ihren Attributwerten auf Tupel abgebildet. Assoziationen werden auf zweiwertige Relationen abgebildet, und die Links dieser Assoziationen auf Tupel in den entsprechenden

Tabellen. Zu diesen Relationen und Tupeln werden dann im Datenbanksystem Tabellen und Zeilen angelegt.

Abbildung 4.14 zeigt die Tabellen, die sich so für die Klasse `State` und die Komposition `StateContainsState` ergeben. Die Einträge in diesen Tabellen entsprechen den hierzu gehörenden Instanzen im Modell, also den Objekten und Links aus dem Statechart der Fußgängerampel.

State			
id	**name**	**initial**	**final**
1	Off	true	true
2	On	false	false
3	Red	true	false
4	Green	false	false

StateContainsState	
composite	**component**
2	3
2	4

Abbildung 4.14.: Auszug der Tabellen und -einträge zum Statechart `PedestrianLight`

Die Abwicklung von Anfragen über Graphen durch relationale Datenbankmangementsysteme wird zum Beispiel in [CYD$^+$08] und [ZCO09] beschrieben. Diese Ansätze verwenden außerdem optimierte JOIN-Verfahren, um auch Muster, die längere Pfade enthalten, effizient finden zu können, und sie haben gute Ergebnisse für Anfragen auf sehr großen Graphen geliefert. Für Modelle kleiner und mittlerer Größe, deren ASTs auch im Hauptspeicher gehalten werden können, erscheint der zusätzliche Aufwand, das Modell nach dem Parsen in eine Datenbank zu kopieren, jedoch zu groß, sodass diese Ansätze für MontiCore nicht vielversprechend sind.

4.6.3. Suchplangesteuerte Ansätze

Modelltransformationswerkzeuge, die das Pattern-Matching ohne Zuhilfenahme externer Werkzeuge auf der vorhandenen Repräsentation des Modells im Hauptspeicher ausführen, verwenden häufig Heuristiken zur Erstellung eines Suchplans. In den meisten Fällen wird der hierdurch entstehende zusätziche Aufwand bei der Codegenerierung oder Initialisierung durch die bessere Laufzeit bei der Mustersuche überkompensiert.

Zu den suchplangesteuerten Ansätzen zählen unter anderem die in diesem Kapitel vorgestellte Engine in MontiCore, sowie die in [FNTZ00] und in [Dör95] vorgestellten Verfahren. Besonders viele Möglichkeiten der modellunabhängigen Optimierung sind in PROGRES umgesetzt worden [Zün96b, SWZ99]. Hier werden zunächst für verschiedene Operationen die Kosten auf Basis gängiger Modellgrößen und -eigenschaften geschätzt. Des Weiteren wird – ebenfalls unter Berücksichtigung der angenommenen Modelleigenschaften – die erwartete Anzahl der Operationsaufrufe für die möglichen Suchpläne bestimmt. Diese Werte werden dann jeweils miteinander multipliziert und über alle Operationen summiert, um den Suchplan mit minimalen Kosten zu bestimmen.

Einige Optimierungsschritte aus den existierenden suchplangesteuerten Ansätzen – jedoch nicht alle – wurden in MontiCore übernommen. Eine Sonderrolle kommt dabei den modellsensitiven Optimierungen zu, die im Wesentlichen [BKG08] entnommen sind. Durch weitere

modellunabhängige Optimierungen kann in zukünftigen Arbeiten bei Bedarf aber vermutlich noch eine weitere Effizienzsteigerung erreicht werden.

Die Verwendung einer bestehenden Implementierung bot sich für die Transformationsengine in MontiCore nicht an. Erstens sind die von MontiCore generierten Klassen der abstrakten Syntax zu den bestehenden Werkzeugen nicht kompatibel; man hätte hierfür eine Adapterschicht generieren müssen, um die Engine auch für bestehende Sprachen verwenden zu können. Zweitens wären bei der Verwendung einer anderen Engine auch die Lizenzen dieser Werkzeuge zu beachten. Eine Ausnahme hierzu stellt PROGRES dar, das zwar lizenzrechtlich unproblematisch wäre, aber auf den Plattformen, auf denen MontiCore verwendet wird, aus technischen Gründen nicht mehr mit vertretbarem Aufwand eingesetzt werden kann.

Kapitel 5.

Eine Regelsprache mit domänenspezifischer Syntax

Aktuelle Modelltransformationssprachen, so auch die meisten der in Tabelle 2.7 genannten Sprachen und die in Kapitel 4 vorgestellte Sprache zur Beschreibung von Transformationsregeln, verwenden meist eine an Objektdiagramme oder attributierte Graphen angelehnte Notation, sofern sie auf Graphersetzungssystemen basieren. Diese Notationen orientieren sich an der abstrakten Syntax der zu transformierenden Modelle.

Die im Rahmen dieser Arbeit entwickelten Transformationssprachen, von denen in diesem Kapitel die Sprache zur Beschreibung von Transformationsregeln auf Statecharts exemplarisch betrachtet wird, grenzen sich von den oben genannten Sprachen insofern ab, als dass ihre Syntax sich eng an der konkreten Syntax der in Kapitel 3 vorgestellten Statechartsprache orientiert, also domänenspezifisch ist. Die Transformationssprachen sind somit nicht mehr auf beliebige Modelle, sondern nur noch auf Modelle einer bestimmten DSL anwendbar. Jedoch besteht ein systematischer Zusammenhang zwischen der Syntax dieser Basissprache und der Transformationssprache, sodass sich Transformationssprachen aus Modellierungssprachen systematisch erstellen und – wie in Kapitel 7 genauer erläutert wird – sogar generieren lassen.

Dieses Kapitel beginnt mit einer Einführung in die kontextfreie Syntax der Transformationsregeln. Anschließend wird der Zusammenhang zwischen dieser Syntax und der zugrunde liegende Modellierungssprache analysiert. Im Folgenden wird eine Auswahl wichtiger kontextsensitiver Bedingungen für Transformationsregeln vorgestellt.

Die Abschnitte 5.4 bis 5.5 behandeln die Codegenerierung für Transformationsregeln in domänenspezifischer Syntax. Hier wird zunächst die Übersetzung der Regeln in die in Kapitel 4 beschriebene Sprache auf Basis der UML/P-Objektdiagramme vorgestellt. Anschließend wird auf einen besonderen Aspekt bei der Übersetzung, nämlich den Umgang mit Bezeichnern, näher eingegangen. Zum Abschluss des Kapitels erfolgt ein Vergleich der hier vorgestellten Sprache mit bestehenden Ansätzen zu Modelltransformationen und Termersetzungen in konkreter Syntax.

5.1. Syntax und Semantik der Regeln

Die domänenspezifische Regelsprache verfügt in weiten Teilen über die Funktionalität, die auch von der Sprache in Objektdiagramm-Notation bereitgestellt wird. In diesem Abschnitt werden die wichtigsten Konzepte vorgestellt. Ein wichtiger Unterschied ist dabei, dass die linke und

die rechte Regelseite nicht mehr voneinander separiert sind, sondern dass eine integrierte Notation verwendet wird, in der Unterschiede zwischen der linken und rechten Regelseite besonders gekennzeichnet sind.

5.1.1. Pattern Matching

Eine Regel in domänenspezifischer Syntax besteht im Wesentlichen aus einer Liste von Termen. Terme ohne besondere Kennzeichnung einer Ersetzung gelten dabei als Elemente, die auf der linken und rechten Regelseite identisch vorhanden sind.

Abbildung 5.1 zeigt eine Regel, die keine Modifikationen am Modell vornimmt. Diese Regel prüft lediglich, ob eine Transition existiert, die an einem Zustand mit initialem Unterzustand endet.

Statechart-Transformationsregel

```
1 state $source;
2
3 state $outer {
4   state $inner <<initial>>;
5 }
6
7 $source -> $outer;
```

Abbildung 5.1.: Regel in konkreter Syntax ohne Modifikationen

Die angegebene Regel kommt ohne Konzepte der abstrakten Syntax der Statechartsprache aus. Im Gegensatz zu den Regeln in Objektdiagramm-Notation werden beispielsweise keine Typen der abstrakten Syntax und keine Attributnamen verwendet. Tatsächlich wäre diese Regel ein gültiger Rumpf eines Statecharts; dies gilt aber nicht für alle Regeln, wie an weiteren Beispielen in diesem Kapitel zu sehen ist.

Die Regel in Abbildung 5.1 enthält drei Zustände und eine Transition, die sowohl Elemente der linken als auch auf der rechten Regelseite sind. Anstatt konkreter Bezeichner sind als Namen der Zustände Schemavariablen angegeben, die durch ein `$`-Zeichen eingeleitet werden.

Eine *Schemavariable* ist ein Platzhalter, der für variable Elemente im durch die Regel transformierten Modell steht, zu denen nicht alle Details angegeben sind. Bei der Ausführung der Regel werden alle Schemavariablen mit konkreten Elementen aus dem Hostgraphen belegt.

Obwohl die Bezeichner in der Transformationsregel in Abbildung 5.1 nicht fest sind, so wird doch gefordert, dass der Quell- und Zielname der Transition in Zeile 7 mit den Namen der Zustände in den Zeilen 1 beziehungsweise 3 übereinstimmen. Dies wird aus der Namensgleichheit der jeweiligen Schemavariablen abgeleitet.

Es wird weiter gefordert, dass der in Zeile 4 angegebene Zustand ein Initialzustand ist. Auch dies lässt sich in der konkreten Syntax der Statechartsprache ausdrücken, nämlich durch die Angabe von `<<initial>>`.

Außerdem muss das Match des Zustands mit dem Namen `$outer` (Zeilen 3-5) ein Oberzustand des Initialzustands in Zeile 4 sein. In der konkreten Syntax der Statechartsprache wie auch

in der Transformationsregel wird dies dadurch ausgedrückt, dass der Subzustand innerhalb des Rumpfes seines Superzustands definiert wird.

Über weitere Eigenschaften des Matches wird durch die Regel keine Aussage getroffen. So können die gematchten Zustände beispielsweise auch initial oder final sein, ohne dass dies in der Regel angegeben ist. Sie können Subzustände, Invarianten oder Aktionen enthalten, und Transitionen können Beschriftungen tragen, die nicht in der Transformationsregel angegeben sind. Auf die explizite Forderung der Nichtexistenz solcher Elemente wird weiter unten in diesem Abschnitt noch eingegangen.

Für viele Anwendungen reicht die konkrete Syntax der Modellierungssprache zur Beschreibung der Transformationsregeln nicht aus. So ist es oft sinnvoll, nicht nur für Namen im Modell, sondern auch für ganze Terme in der Regel Schemavariablen anzulegen. Dies erlaubt zum einen, in zusätzlichen Anwendungsbedingungen auf diese Terme zuzugreifen. Zum anderen können so auch Terme definiert werden, die eine abstrakte Klasse oder eine Schnittstelle als Typ haben, und denen aus der Modellierungssprache keine konkrete Syntax eindeutig zugeordnet werden kann, oder Listen von Termen definiert und benannt werden.

Abbildung 5.2 zeigt eine Transformationsregel, die eine Mischform aus konkreter und abstrakter Syntax verwendet. Neben der in Abbildung 5.1 gewählten Form in konkreter Syntax gibt es für die Definition von Termen zwei weitere Möglichkeiten: erstens die Angabe eines *Schematyps* und eines Bezeichners, der *Schemavariablen*, gefolgt von einem Semikolon. Dies ist in Abbildung 5.2 in Zeile 3 für die `Expression $E` der Fall. Zweitens kann eine Mischung aus dieser Form und konkreter Syntax verwendet werden. Diese Mischform besteht aus dem Typ und der Schemavariablen des Terms, gefolgt vom Rumpf der Definition. Dieser Rumpf ist in doppelten eckigen Klammern eingeschlossen, womit auch im Folgenden transformationsspezifische Sprachkonstrukte hervorgehoben werden. Innerhalb dieses Blocks werden weitere Details zum Term, im Beispiel in den Zeilen 1-5 zum Zustand `$S`, spezifiziert. Hierfür können wiederum Terme in allen drei Darstellungsformen verwendet werden, auch beliebige Kombinationen dieser Formen sind erlaubt.

Statechart-Transformationsregel

```
1 State $S [[
2   state $s_name {
3     [  Expression $E; ]
4   }
5 ]]
6 where { match.$S.getStates().size() % 2 == 0 }
```

Abbildung 5.2.: Transformationsregel, in der sowohl abstrakte als auch konkrete Syntax verwendet wird

In diesem Beispiel sind drei Formen der Abweichung von der konkreten Syntax der Statechartsprache zu sehen: Erstens gibt es transformationsspezifische Sprachanteile, die nicht aus der Statechartsprache stammen, wie beispielsweise explizite Bezeichner für Terme oder die doppelten eckigen Klammern. Zweitens ist für die Terme `$S` und `$E` jeweils der Schematyp `State` beziehungsweise `Expresssion` angegeben, der sich auf die abstrakte Syntax der Statecharts bezieht. Diese Art der Definition erlaubt es zum einen, auch Terme zu definieren, deren Typ eine

abstrakte Klasse oder Schnittstelle ist. Zum anderen wird hierdurch explizit ein Bezeichner für diesen Term vergeben. Dieser erlaubt es, wie in Zeile 6 der Abbildung 5.2, zusätzliche Constraints für das Pattern Matching anzugeben. Diese Constraints sind Java-Ausdrücke, welche die Ausdruchsmächtigkeit der Regelsprache weiter erhöhen; sie stellen die dritte Abweichung von der konkreten Syntax der Statecharts dar. Der Ausdruck im Beispiel stellt sicher, dass der gematchte Zustand eine gerade Anzahl von Unterzuständen hat. Dies ließe sich weder in konkreter Syntax noch durch Objektdiagramme der abstrakten Syntax ausdrücken.

Als Alternative zu den booleschen Java-Ausdrücken hätten auch OCL-Ausdrücke gewählt werden können, deren Kombination mit Objektdiagrammen in [Rum11] und [Sch12] beschrieben wird. Jedoch bieten die verfügbaren Java-Bibliotheken, insbesondere die MontiCore-Laufzeitumgebung, einen beträchtlichen Mehrkomfort bei der Spezifikation der Constraints, weshalb im Rahmen dieser Arbeit Java als Sprache gewählt wurde.

Negative Objekte und Listenobjekte

Auch in der domänenspezifischen Syntax ist es möglich, negative Objekte und Listenobjekte anzugeben (vgl. Abschnitt 4.4). Diese werden durch die Schlüsselwörter `not` und `list` eingeleitet. Optional können wie oben beschrieben der Schematyp des Terms und ein Bezeichner angegeben werden. Der folgende Rumpf steht dann innerhalb doppelter eckiger Klammern. So lässt sich eindeutig bestimmen, auf welches Objekt sich das `not` beziehungsweise `list` bezieht.

Die in Abbildung 5.3 dargestellte Regel zeigt die Verwendung eines negativen und eines Listenobjekts. Sie matcht einen Zustand, der keinen initialen Unterzustand hat. Zu diesem Zustand wird außerdem die Liste aller seiner Unterzustände gematcht. Durch den Ausdruck in Zeile 5 wird zusätzlich gefordert, dass diese Liste nicht leer sein darf. Die anonyme Schemavariable `$_` in Zeile 3 matcht hier beliebige Bezeichner; Details zu anonymen Schemavariablen finden sich in Abschnitt 5.5.

```
Statechart-Transformationsregel
1 state $s_name {
2   not [[ state $i <<initial>>; ]]
3   list<State> $SUB [[ state $_ ; ]]
4 }
5 where { !match.$SUB.isEmpty() }
```

Abbildung 5.3.: Regel mit negativem Objekt und Listenobjekt

Zusätzliche Constraints und nichtisomorphes Matching

In den Abbildungen 5.2 und 5.3 war bereits zu sehen, dass auch die Regeln in domänenspezifischer Notation `where`-Blöcke enthalten können, in denen zusätzliche Constraints für das Pattern-Matching angegeben werden. Dabei können als Objekte nur die in der Regel definierten Schemavariablen für Objekte verwendet werden, wobei auch Listen und negative Objekte

zulässig sind. Ansonsten entsprechen die Java-Ausdrücke in `where`-Blöcken syntaktisch und semantisch denen der Transformationsregeln in Objektdiagramm-Notation (vgl. Abschnitt 4.2).

Ebenso können in Regeln in domänenspezifischer Notation auch nichtisomorphe Matches erlaubt werden. Diese werden wie bei Regeln in Objektdiagramm-Notation durch das Schlüsselwort `folding` eingeleitet und entsprechen ihnen auch sonst in der kontextfreien Syntax (vgl. Abschnitt 4.2). Analog zu den Einschränkungen für `where`-Blöcke können auch hier innerhalb des Blocks nur Schemavariablen für Terme aus der Transformationsregel als Objektbezeichner verwendet werden.

5.1.2. Modifikationen des Modells

Die im Unterabschnitt 5.1.1 vorgestellten Regeln führen zur Laufzeit Mustersuchen im Hostmodell durch. Für viele Transformationen, unter anderem die in Kapitel 3 beschrieben Vereinfachung hierarchischer Statecharts, ist es aber erforderlich, mit den Transformationsregeln auch Veränderungen des Modells zu beschreiben.

Die Änderungsoperationen auf Modellen sind dabei aus den atomaren Operationen Erzeugen von Objekten, Löschen von Objekten, Setzen von Attributwerten, Erzeugen von Links und Löschen von Links zusammengesetzt[1]. Alle Änderungen am Modell werden also als eine Kombination dieser Operationen dargestellt. Im Folgenden wird die Notation dieser Operationen in Transformationsregeln in domänenspezifischer Syntax vorgestellt. Dabei werden die Regeln für die Vereinfachung hierarchischer Statecharts als Beispiel verwendet, und es wird die Zusammensetzung dieser Regeln aus atomaren Änderungsoperationen erläutert.

Generell beginnen Änderungsoperationen mit zwei öffnenden eckigen Klammern `[[`. Es folgt ein im Modell zu suchendes Teil-Muster, das wie oben beschrieben aufgebaut ist. Durch den Operator `:-` wird die Ersetzung für dieses Muster eingeleitet. Ersetzungen entsprechen in der kontextfreien Syntax den Mustern; kontextsensitiv dürfen abweichend aber beispielsweise nur Schemavariablen verwendet werden, die an anderer Stelle definiert wurden. Das Ende der Änderungsoperation wird mit zwei schließenden eckigen Klammern `]]` gekennzeichnet.

Abbildung 5.4 zeigt eine einfache Ersetzungsregel, die einen Zustand mit dem Namen `Red` durch einen Zustand mit dem Namen `Green` ersetzt. Zu beachten ist hier, dass der ganze Zustand ersetzt wird. Sind im ursprüngliche Zustand also die Werte weiterer Attribute wie `initial` oder `final` gesetzt oder sind in ihm weitere Zustände oder Transitionen enthalten, so sind diese Informationen und Objekte nach Ausführung der Regel nicht mehr vorhanden. Ebenfalls zu beachten ist, dass lediglich die Zustände selbst umbenannt werden, etwaige Referenzen auf diese Zustände, zum Beispiel als Quell- oder Zielzustand einer Transition, jedoch nicht angepasst werden.

Erzeugen und Löschen von Termen

Die in Abbildung 5.4 gezeigte Regel beinhaltet bereits das Löschen und Erzeugen je eines Terms, da sich das Ersetzen eines Zustands aus diesen beiden Operationen zusammensetzt. Ein zu lö-

[1] In der Implementierung erfolgt in MontiCore-Modellen das Erzeugen und Löschen von Links in der Regel auch über das Setzen von Attributwerten. Da es vor allem bei den Kompositionsbeziehungen eine besondere Rolle spielt, wird es hier aber einzeln betrachtet.

Statechart-Transformationsregel

```
1 [[
2   state Red;
3 :-
4   state Green;
5 ]]
```

Abbildung 5.4.: Einfache Transformationsregel mit Ersetzung

schender Term ist ein Term, der innerhalb einer Transformationsregel nur im linken Teil einer Ersetzung vorkommt. Analog kommt ein zu erzeugender Term nur im rechten Teil einer Regel vor.

In Abbildung 5.5 ist das Anlegen einer leeren Entry-Aktion gezeigt. Diese Transformation wird als vorbereitender Schritt der Vereinfachung hierarchischer Statecharts für alle Zustände ausgeführt, für die noch keine Entry-Aktion spezifiziert ist.

Statechart-Transformationsregel

```
1 state $s_name {
2   [[ :- entry : { } ]]
3 }
```

Abbildung 5.5.: Erzeugen einer leeren Entry-Aktion

Abbildung 5.6 zeigt das Löschen einer Exit-Aktion aus einem Zustand, das ausgeführt werden kann, nachdem eine Kopie der Aktion an allen ausgehenden Transitionen des Zustands angelegt wurde (vgl. Abschnitt 3.3.4). Hierbei werden auch alle in der Exit-Aktion enthaltenen Anweisungen entfernt, da sie als Unterknoten im AST gemeinsam mit der Aktion gelöscht werden.

Statechart-Transformationsregel

```
1 state $s_name {
2   [[ exit : { }  :- ]]
3 }
```

Abbildung 5.6.: Löschen einer Exit-Aktion

Operationen auf atomaren Termen

Einige der Terme, die während der Vereinfachung hierarchischer Statcharts modifiziert werden, beziehen sich auf nicht weiter zu zerlegende Terme, die in der abstrakten Syntax auf Attributwerte von Objekten abgebildet werden. Aus Sicht des Statechart-Parsers sind diese Terme bestimmte Arten von Tokens, wie etwa die Namen von Zuständen oder die Schlüsselwörter `initial` und `final`.

Während Transformationssprachen, die sich auf die abstrakte Syntax beziehen, solche Terme anders behandeln als Objekte oder Gruppen von Objekten, entfällt aus der Sicht des Benutzers

diese Unterscheidung bei Transformationen in konkreter Syntax. In der Implementierung müssen sie jedoch gesondert behandelt werden, wie in Abschnitt 5.4 erläutert wird.

In Analogie zur Syntax für das Löschen und Erzeugen von Objekten wird auch beim Setzen von Attributen die Ersetzung in doppelten eckigen Klammern eingefasst. Die linke und rechte Regelseite sind auch hier durch den Ersetzungsoperator :- getrennt.

Zeile 2 in Abbildung 5.7 zeigt das Löschen des initial-Flags eines Subzustands. Das Setzen erfolgt analog durch den Term `[[ :- initial ]]`.

In Zeile 4 der selben Abbildung ist die Veränderung eines Bezeichners zu sehen. In der Ersetzung `[[ $outer :- $inner ]]` werden zwei Schemavariablen verwendet. Die Angabe von `$outer` auf der linken Seite erzwingt, dass der Quellzustand der gematchten Transition vor der Regelausführung mit dem in Zeile 1 definierten Namen des äußeren Zustandes übereinstimmt, da dieser in der Regel ebenfalls als `$outer` bezeichnet ist. Die Angabe von `$inner` auf der rechten Seite sorgt dafür, dass der Zielzustand durch die Regel auf den Namen des in Zeile 2 definierten inneren Zustands `$S$` gesetzt wird, der hier ebenfalls als `$inner` bezeichnet ist.

Statechart-Transformationsregel

```
1 state $outer {
2   State $S [[ state $inner << [[ initial :- ]] >>; ]]
3 }
4 Transition $T [[ $source -> [[ $outer :- $inner ]]; ]]
```

Abbildung 5.7.: Setzen von Attributen in der Regel `redirectTransitionToSubstate`

Erzeugen und Löschen von Links

Durch das Erzeugen und Löschen von Links können Zeiger auf andere Objekte im AST modifiziert werden. Da das Assoziatonskonzept von MontiCore keine Auswirkungen auf die konkrete Syntax der Sprachen hat, sind bei den Regeln in konkreter Syntax hiervon nur die Kompositionen im AST betroffen. Durch das Erzeugen und Löschen von Links werden also im Wesentlichen Terme innerhalb des AST verschoben.

Abbildung 5.8 zeigt das Verschieben einer Transition. Zeile 2 ist syntaktisch zunächst identisch zum Erzeugen eines Terms, und Zeile 5 ist identisch zum Löschen eines Terms, also zum Entfernen der Knoten aus dem AST. Weil der Bezeichner `$T` der beiden Terme aber übereinstimmt, beschreibt diese Regel das Verschieben des Terms von einer beliebigen Position in den Zustand `$S`, wobei eventuelle Kindknoten der Transition im AST und Attributwerte erhalten bleiben. Die Position der Transition `$T` kann beliebig sein, da für `$T` in Zeile 5 auf der linken Seite kein übergeordneter Knoten angegeben ist.

Steht ein zu verschiebender Term innerhalb eines zu löschenden Terms, so hat das Verschieben Priorität über das Löschen. Der entsprechende Term wird also zunächst an die angegebene Stelle im Modell verschoben, anschließend wird der übergeordnete Term gelöscht.

Generell werden beim Löschen eines Terms aus dem AST auch dessen adjazente Kanten entfernt (vgl. Abschnitt 4.1.2). Dies ist zum einen sinnvoll unter Berücksichtigung der Tatsache, dass die Regeln auf der konkreten Syntax der Statechartsprache beruhen, und in der konkreten

Statechart-Transformationsregel

```
1 State $S [[ state $s_name {
2   [[ :- Transition $T; ]]
3 } ]]
4
5 [[ Transition $T; :- ]]
```

Abbildung 5.8.: Verschieben einer Transition in einen Zustand durch die Regel `moveTransitionToState`

Syntax hängende Kanten beispielsweise für Kompositionsbeziehungen nicht ausgedrückt werden können. Darüber hinaus sind in der abstrakten Syntax MontiCore-basierter Sprachen Links technisch durch Zeiger auf andere Objekte umgesetzt, sodass auch hier die Darstellung hängender Kanten schwerfallen würde.

5.2. Ableitung der kontextfreien Syntax aus der Basis-DSL

Die möglichst enge Anlehnung an die Notation der Statechartsprache war bei der Entwicklung der Sprache für Transformationsregeln ein wichtiges Ziel. Da die beiden Sprachen syntaktisch einander stark ähneln, liegt es nahe, auf die Definition der Syntax der Statechartsprache bei der Erstellung der Regelsprache zurückzugreifen. Die Syntax von Sprachen wird in MontiCore durch kontextfreie Grammatiken definiert. Tatsächlich lässt sich, wie im Folgenden erläutert wird, die Grammatik der Regelsprache systematisch aus der Grammatik der Statechartsprache ableiten.

Dabei wird an verschiedenen Stellen auf Elemente der Grammatiken der Statechartsprache und der Sprache für Transformationsregeln auf Statecharts verwiesen. Diese Elemente sind hier nur so umfangreich und detailliert aufgeführt, wie zum Verständnis des Abschnitts erforderlich ist. Die vollständige Grammatik der Statecharts ist in Anhang C.1 abgebildet. Die hieraus generierte Grammatik der Sprache für Transformationsregeln ist in Anhang C.2 dargestellt.

5.2.1. Terme in Transformationsregeln

Die zentralen Elemente einer Transformationsregel sind die in ihr beschriebenen Terme. Innerhalb von Transformationsregeln für Statecharts kann es sich dabei unter anderem um Zustände, Transitionen, Statecharts, Expressions oder Anweisungen handeln. Ein Term kann wiederum in unmodifizierter Form zur Mustersuche verwendet werden, oder es kann durch die im vorangegangenen Abschnitt vorgestellten transformationsspezifischen Sprachkonstrukte als Teil einer Ersetzung ausgezeichnet sein, eine negative Anwendungsbedingung beschreiben oder für ein Listenobjekt stehen.

Es ist also festzustellen, dass für jedes Nichtterminal der Statechartsprache mehrere entsprechende Sprachkonstrukte in der Transformationssprache existieren. In der Grammatik der Regelsprache sind diese durch separate Nichtterminale umgesetzt, die eine gemeinsame Schnittstelle implementieren. So existiert in der Regelsprache das Nichtterminal `State`, das die Nichtterminale `State_Pattern`, `State_Negation`, `State_List` und `State_Replacement`

als Alternativen hat. Darüber hinaus implementieren die aus diesen Nichtterminalen generierten Klassen der abstrakten Syntax noch die Java-Schnittstellen `IPattern`, `INegation`, `IList` und `IReplacement`, die unabhängig von einem bestimmten Nichtterminal der DSL Muster, negative Anwendungsbedingungen, Listenknoten oder Ersetzungen klassifizieren, und die durch die Laufzeitumgebung der Transformationsengine bereitgestellt werden.

Die Produktionen, die sich aus dem Nichtterminal `State` der Basissprache ergeben, sind in Abbildung 5.9 vereinfacht dargestellt. Zeile 1 der Abbildung zeigt die Schnittstellenproduktion `State`. Es folgt die Produktion `State_Pattern`, auf deren rechter Seite die drei in Abschnitt 5.1.1 vorgestellten Möglichkeiten zur Notation eines Musters zu finden sind.

In den Zeilen 4-13 ist die Notation in konkreter Syntax spezifiziert. Diese entspricht weitgehend der rechten Seite der entsprechenden Produktion in der Statechartgrammatik. Zu beachten ist dabei, dass hier auch Nichtterminale wie `State` und `Transition` in Zeile 9 verwendet werden, die sich dann auf die entsprechenden Schnittstellenproduktionen beziehen. Des weiteren haben die Attribute `Name`, `Initial` und `Final` in Zeile 4 gegenüber der Statechartgrammatik abweichende Typen. Dies ist durch die Notwendigkeit bedingt, in den Transformationsregeln Attributwerte setzen zu können, und wird im Folgenden noch erläutert. Der Zustandsrumpf wurde in der Abbildung gekürzt (s. Zeile 8), es fehlen die Aktionen und Invarianten. Darüber hinaus sind auch einige technische Anweisungen für den Parser nicht dargestellt.

In den Zeilen 14 und 15 sind die beiden anderen möglichen Notationen für ein `State_-Pattern` aufgeführt. Zeile 14 ist dabei gekürzt, sie beschreibt die Mischform aus konkreter und abstrakter Syntax und enthält folglich auch die in den Zeilen 4 bis 13 aufgeführten Elemente. Zeile 15 beschreibt die rein auf der abstrakten Syntax basierende Notation.

Die hier beschriebenen Produktionen der Regelsprache ergeben sich systematisch aus der zugrunde liegenden Basissprache. Die Ableitungsvorschrift hierfür wird in Kapitel 7 erläutert.

Abbildung 5.10 fasst die Klassen der abstrakten Syntax der Statecharttransformationssprache zusammen, wobei nur die Klassen aufgeführt sind, die sich aus dem Nichtterminal `State` ergeben oder mit diesen Klassen in Verbindung stehen. Das Präfix `AST` wird von MontiCore bei der Codegenerierung automatisch erstellt und kennzeichnet alle aus der Grammatik abgeleiteten Klassen der abstrakten Syntax.

5.2.2. Umgang mit Attributen

Die bisher vorgestellten Elemente der Regelsprache erlauben es, Zustände zu matchen, zu erzeugen und zu löschen. Durch das Löschen oder Hinzufügen eingebetteter Zustände und Transitionen (vgl. Abbildung 5.9, Zeilen 9 und 24-26) lassen sich auch Kompositionslinks erzeugen oder entfernen. Darüber hinaus muss es aber auch möglich sein, durch die Regeln Attributwerte zu setzen. Im Fall der in Abbildung 5.9 dargestellten Zustände betrifft dies das Bezeichner-Attribut `Name` und die booleschen Attribute `Initial` und `Final`.

Für das Matching und die Ersetzung der Attributwerte werden in der Grammatik der Regelsprache ebenfalls entsprechende Produktionen erstellt, die für das Attribute `Initial` in Abbildung 5.11 zu sehen sind. Im Vergleich zu den Produktionen, die zu einem Nichtterminal der DSL angelegt wurden (vgl. Abbildung 5.9), werden andere externe Schnittstellen implementiert, die für Attribute spezifisch sind, und es gibt keine Produktion für Listen von `Initial`-Attributen.

MontiCore-Grammatik

```
1  interface State;
2
3  State_Pattern implements State astimplements /mc.tfcs.ast.IPattern =
4    ("state" Name:TfIdentifier ("<<" (Initial | Final) ">>")*
5      (
6        (
7          "{"
8             ...
9             (States:State | Transitions:Transition | ...)*
10         "}"
11       ) | ";"
12     )
13   )
14   | "State" SchemaVarName:IDENTVAR? "[[" ("state" ...) "]]"
15   | "State" SchemaVarName:IDENTVAR ";"
16 ;
17
18 State_List implements State astimplements /mc.tfcs.ast.IList =
19   "list" ("<" "State" ">")? SchemaVarName:IDENTVAR? "[[" State "]]";
20
21 State_Negation implements State astimplements /mc.tfcs.ast.INegation =
22   "not" ("<" "State" ">")? SchemaVarName:IDENTVAR? "[[" State "]]";
23
24 State_Replacement implements State
25     astimplements /mc.tfcs.ast.IReplacement =
26   ("[[" lhs:State? ":-" rhs:State? "]]");
```

Abbildung 5.9.: Produktionen zum Nichtterminal `State` in Transformationsregeln

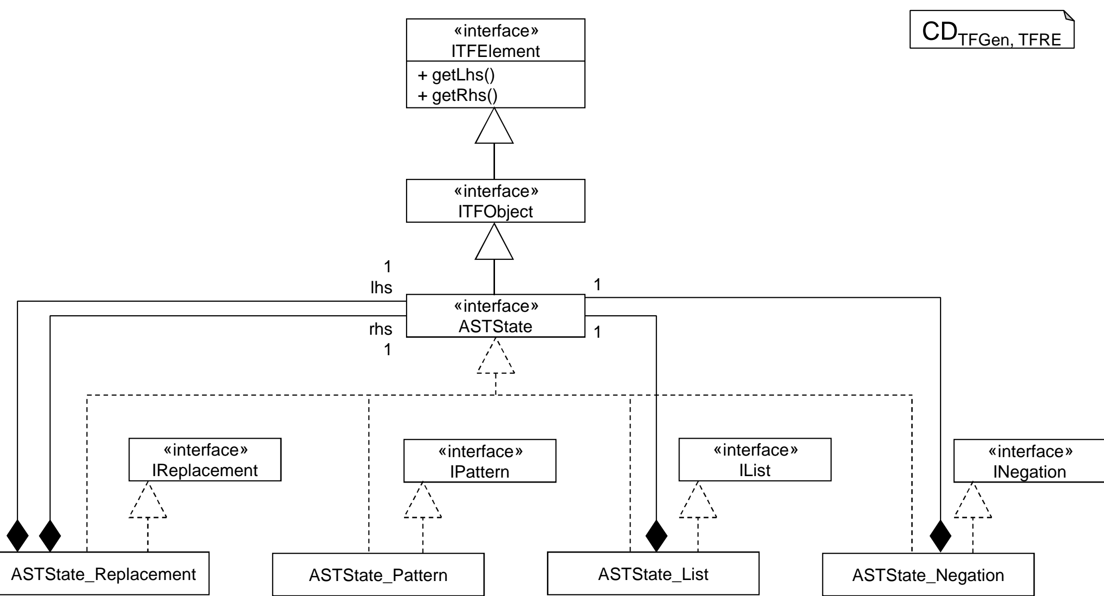


Abbildung 5.10.: Abstrakte Syntax der Zustände in Transformationsregeln

MontiCore-Grammatik

```
1 interface Initial;
2
3 Initial_Pattern implements Initial
4     astimplements /mc.tfcs.ast.IAttributePattern =
5   Initial:["initial"];
6
7 Initial_Negation implements Initial
8     astimplements /mc.tfcs.ast.IAttributeNegation =
9   "not" "[[" Initial_Pattern "]]";
10
11 Initial_Replacement implements Initial
12     astimplements /mc.tfcs.ast.IAttributeReplacement =
13   ("[[" lhs:Initial_Pattern ":-" "]]")
14   | ("[[" ":-" rhs:Initial_Pattern "]]")
15 ;
```

Abbildung 5.11.: Produktionen zum Attribut `Initial` der Produktion `State`

Abbildung 5.12 gibt einen Überblick über die Klassen, die zum Attribut `Initial` generiert wurden oder mit diesen Klassen in Verbindung stehen. Die Produktionen und Klassen für das `Final`-Attribut entsprechen den gezeigten Produktionen beziehungsweise Klassen bis auf den Namen.

Die Zustände der Statechartsprache verfügen neben den geschachtelten Elementen sowie den booleschen Attributen auch noch über einen Bezeichner, der im Attribut `Name` gehalten wird. Abbildung 5.13 zeigt die Produktionen, die für das Matching und die Modifikation von Bezeichnern benötigt werden. Die Produktion `TfIdentifier` in den Zeilen 1-3 ermöglicht es, in Transformationsregeln anstelle eines Bezeichners auch die Ersetzung eines Bezeichners anzugeben (vgl. das Setzen des Zielzustands einer Transition in Abbildung 5.7). Die in den Zeilen 5-9 beschriebene lexikalische Produktion `IDENTVAR` erlaubt es, anstatt fixer Bezeichner auch Schemavariablen zu verwenden. Hierauf wird in Abschnitt 5.5 näher eingegangen.

Nun sind noch die Sprachelemente, die sich auf Objekte beziehen, mit den Sprachelementen zur Modifikation von Attributen zu verknüpfen. Das Klassendiagramm in Abbildung 5.14 fasst die AST-Klassen der Statecharttransformationssprache zusammen, die sich nach [Kra10] aus den Produktionen in den Abbildungen 5.9, 5.11 und 5.13 ergeben. Die Abbildung veranschaulicht auch, wie die Kompositionen und Attribute zu den Klassen angelegt wurden. Als Typ der Attribute und Assoziationsenden wurde jeweils die Schnittstelle verwendet, die für den entsprechenden Typ des Attributs oder der Assoziation in der DSL erstellt wurde. Dabei genügt es, die Attribute und Assoziationen in den implementierenden Klassen der Schnittstelle `IPattern` anzulegen, weil die übrigen Klassen jeweils solche Objekte kapseln und somit alle Zugriffe an die gekapselten Objekte delegieren können.

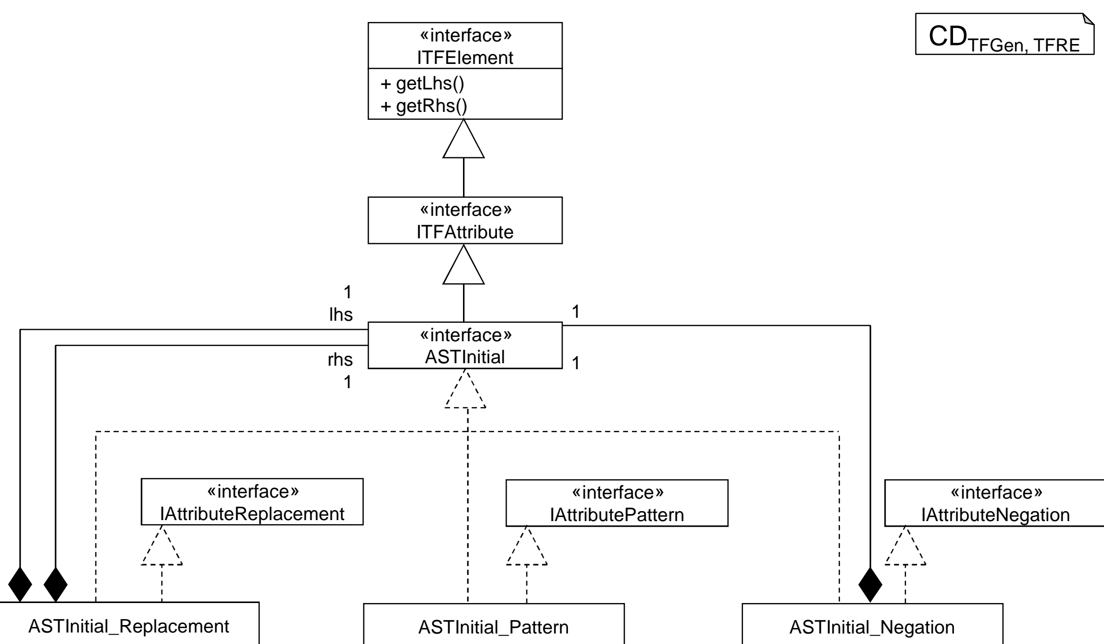


Abbildung 5.12.: Abstrakte Syntax der Transformationselemente zum Attribut `Initial` der Statechartsprache

MontiCore-Grammatik

```
1 TfIdentifier =
2   identifier:IDENTVAR
3   | ("[[" identifier:IDENTVAR ":-" newIdentifier:IDENTVAR "]]");
4
5 ident IDENTVAR ... =
6   '$' ('a'..'z'  | 'A'..'Z'  | '_' | '0'..'9'  | '$')*
7   | ('a'..'z'  | 'A'..'Z'  | '_')
8       ('a'..'z'  | 'A'..'Z'  | '_' | '0'..'9'  | '$')*
9   | '\\' '$' ('a'..'z'  | 'A'..'Z'  | '_' | '0'..'9'  | '$')*;
```

Abbildung 5.13.: Produktionen zu Bezeichner-Attributen der Statechart-Grammatik

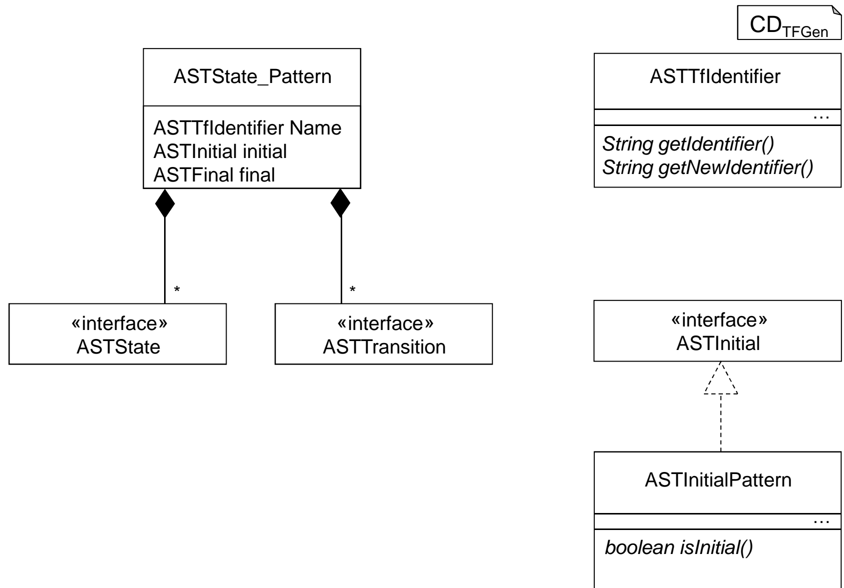


Abbildung 5.14.: Attribute und Kompositionen in der Regelsprache

5.3. Kontextbedingungen und Empfehlungen für Anwender

Neben den bisher erwähnten Kontextbedingungen für Transformationsregeln existieren noch weitere, von denen die wichtigsten in diesem Abschnitt vorgestellt werden. Die hier vorgestellte Auswahl sowie die bisher in MontiCore implementierten Regeln sind jedoch noch nicht vollständig und können im Rahmen zukünftiger Arbeiten weiter ausgebaut werden.

Die statische Analyse von Modelltransformationsregeln ist in mehreren Gruppen aktuell Gegenstand der Forschung. Exemplarisch soll hier auf die Arbeiten zu Typkonzepten in Transformationen [ZHV09] und einen Ansatz zur Typinferenz in Graphersetzungsregeln [vE08] verwiesen werden. Darüber hinaus gibt es auch Bestrebungen, die Qualität von Transformationen durch formale Methoden nachzuweisen [ABK07, CLSAT09], die jedoch im Folgenden nicht näher betrachtet werden.

Des Weiteren werden in diesem Abschnitt Hinweise für Nutzer von Transformationssprachen in domänenspezifischer Syntax gegeben, die zwar in Bezug auf die Korrektheit der Transformationen nicht erforderlich sind, deren Einhaltung aber die Lesbarkeit und Wartbarkeit der Programme verbessern soll.

5.3.1. Typen der Schemavariablen

Zu Schemavariablen in Transformationsregeln können Schematypen angegeben werden, die sich entweder aus den Produktionen in der Grammatik der Basissprache ergeben, wie beispielsweise die Typen `State` und `Transition`, oder die aus der Infrastruktur für MontiCore-basierte Sprachen abgeleitet sind, wie der Schematyp `Name` für Bezeichner.

Abbildung 5.15 zeigt eine Regel, in der eine Kombination aus Schemavariablen für Bezeichner und für Objekte verwendet wird. Ein Match für diese Regel besteht aus einer Transition und deren Quellzustand, wobei zusätzlich gefordert ist, dass beide Objekte im selben Vaterknoten enthalten sein müssen, der entweder ein Zustand oder das Statechart sein kann[2].

Statechart-Transformationsregel

```
1 State $S [[ state $s_name; ]]
2
3 Transition $T [[ $s_name -> $target; ]]
4
5 where {
6   match.$S.getParentElement() == match.$T.getParentElement()
7 }
```

Abbildung 5.15.: Schemavariablen für Objekte und Bezeichner

Durch die Verwendung der Schemavariablen `$s_name` in den Zeilen 1 und 3 ist sichergestellt, dass der Bezeichner, der für den Zustand als Name angegeben ist, mit dem Bezeichner übereinstimmt, der die Quelle der Transition beschreibt. Die Schemavariablen `$S` und `$T` beziehen sich dagegen auf das Zustands- beziehungsweise Transitionsobjekt, also auf die AST-Knoten. Im Contraint in Zeile 6 können daher auch Methoden der Klassen `ASTState` und `ASTTransition` verwendet werden. Die Übereinstimmung der Rückgabewerte von `getParentElement()` stellt sicher, dass beide Objekte einen gemeinsamen Vaterknoten haben.

Zu den Schemavariablen für Objekte ergeben sich neben der genannten Abgrenzung von Bezeichnern auch Einschränkungen aus den Typen der zu matchenden Objekte. So kann in den Constraints zu diesen Objekten nur auf Attribute und Typen zugegriffen werden, die für den entsprechenden Typ gültig und in der Transformationsregel sichtbar sind. Für Ersetzungen ist bereits durch die kontextfreie Analyse sichergestellt, dass ein Objekt nur durch ein Objekt gleichen Typs ersetzt werden kann. Beispielsweise ist in Zeile 26 der Abbildung 5.9 spezifiziert, dass Zustände innerhalb eines Statecharts nur durch Zustände ersetzt werden können. Dies lässt sich kontextfrei jedoch nicht mehr sicherstellen, wenn ein Objekt wie in Abbildung 5.8 in zwei verschiedenen Ersetzungen verwendet und somit verschoben wird. Jedoch können auch hier auf der rechten Regelseite nur Objekte verwendet werden, die auf der linken Regelseite den durch die Ersetzung vorgegebenen Typ oder einen Subtyp hiervon haben.

5.3.2. Disjunktheit der Matches von Listenknoten

Werden in einer Regel mehrere Listenknoten angegeben, so müssen die Matches zu diesen Knoten disjunkt sein, sofern keine Folding-Regel angegeben wurde. Erfüllt ein Element im Hostgraphen die Bedingungen mehrerer Listenknoten, so ist durch die Regel nicht eindeutig bestimmt, in welcher Liste es nach dem Pattern-Matching enthalten ist. In der aktuellen Implementierung wäre das Objekt in der Liste enthalten, die gemäß dem Suchplan früher gematcht wird (vgl.

[2] Listen von AST-Knoten, die von MontiCore für alle iterierten Attribute generiert werden, werden in diesem Sinne nicht als der Vaterknoten eines AST-Knotens betrachtet.

Abschnitt 4.3). Dadurch würde die Regel jedoch unverständlich, sodass empfohlen wird, mehrere Listenknoten nur dann zu verwenden, wenn sichergestellt ist, dass die möglichen Matches zur Ausführungszeit disjunkt sind. Dies ist jedoch zur Compilezeit der Regel nur in Spezialfällen entscheidbar, sodass keine Kontextbedingung angegeben werden kann, welche diese Eigenschaft korrekt und vollständig umsetzt.

Die in Abbildung 5.16 gezeigte Regel matcht einen Zustand, dessen eingehende Transitionen und alle initialen Unterzustände. Für diese Regel und die Statechartsprache ist die Disjunktheit der Matches zu den Listen `$SUB` und `$T` dadurch garantiert, dass die Typen `State` und `Transition` keinen gemeinsamen Untertypen haben. Dies ist zwar für Klassenproduktionen in MontiCore zur Compilezeit der Regel entscheidbar. Generell lässt sich diese Aussage jedoch nicht immer zuverlässig treffen; beispielsweise kann für zwei Schnittstellentypen ein gemeinsamer Untertyp zur Compilezeit der Regel nicht bekannt sein, da durch Mehrfachvererbung eine Klasse beide Schnittstellen implementieren kann.

Statechart-Transformationsregel

```
1 state $outer {
2   list<State> $SUB [[ state $_ << initial >>; ]]
3 }
4
5 list<Transition> $T [[ $_ -> $outer; ]]
```

Abbildung 5.16.: Disjunkte Listen in einer Transformationsregel

5.3.3. Formatierungs- und Programmierrichtlinien

Die Einhaltung von Formatierungs- und Programmierrichtlinien kann wesentlich zur besseren Lesbarkeit von Programmcode und somit zur Wartbarkeit von Software beitragen. Gute Wartbarkeit ist eminent wichtig, da die Wartung von Software den Großteil des Software-Lebenszyklus bestimmt und somit ein ausschlaggebender Kostenfaktor ist. Für die Programmierung in GPLs sind solche Richtlinien gängige Praxis. Beispiele und weitere Erläuterungen finden sich unter anderem in [Bec97, Kapitel 7] oder [Sun99].

Die grundsätzliche Notwendigkeit eines guten Stils lässt sich auch auf die Modellierung übertragen, wobei gerade für die textuelle Modellierung viele bewährte Vorgehensweise aus der Programmiereung anwendbar sind. So lassen sich für ein Projekt Konventionen zur Benennung von Variablen, Groß- und Kleinschreibung, Einrückung, zu Zeilenumbrüchen oder Kommentaren treffen.

Für die in diesem Kapitel behandelte Sprache zur Beschreibung von Transformationsregeln auf hierarchischen Statecharts ist die Definition von Formatierungsrichtlinien etwas schwieriger als für die zugrunde liegende Modellierungssprache. Der Grund hierfür ist die Kombination der transformationsspezifischen Elemente mit der konkreten Syntax der Statechartsprache. Einerseits soll die Struktur der Transformationsregeln leicht erkennbar sein. Andererseits soll der Vorteil, dass die Muster in konkreter Syntax leicht zu erfassen sind, nicht durch beliebige Formatierung zunichte gemacht werden – eine Herausforderung, die übrigens auch beim Schreiben von Templates für Codegeneratoren auftritt.

In der Transformationssprache für Statecharts sind die transformationsspezifischen Elemente die doppelten eckigen Klammern, der Ersetzungsoperator `:-`, die `where`- und `folding`-Blöcke sowie die Markierungen von Listen- und negativen Knoten. Auch die Schemavariablen für Objekte fallen in diese Kategorie, da sie nicht der konkreten Syntax der Statechartsprache entnommen sind, nicht jedoch die Schemavariablen für Bezeichner.

Zur besseren Unterscheidung der Schemavariablen für Objekte von denen für Bezeichner wird empfohlen, Schemavariablen für Objekte in Großbuchstaben zu schreiben. Abbildung 5.17 wird bei der Vereinfachung hierarchischer Statecharts zur Suche nach intialen Subzuständen verwendet. Hier ist die Großschreibung auf die Objekte `$SUPER`, `$SUB` und `$T` anzuwenden, während sich `$outer` auf einen Bezeichner bezieht und somit klein geschrieben wird.

Statechart-Transformationsregel

```
1 State $SUPER [[ state $outer {
2   list<State> $SUB [[ state $_ << initial >>; ]]
3 } ]]
4
5 Transition $T [[ $_ -> $outer; ]]
6
7 where { !match.$SUB.isEmpty() }
```

Abbildung 5.17.: Hervorhebung von Schemavariablen für Objekte durch Großbuchstaben

Für die Zeilenumbrüche ist es empfehlenswert, die Konventionen aus der Statechartsprache wenn möglich zu übernehmen. So ergeben sich die drei Zeilen, in denen der Zustand `$SUPER` definiert wird aus der Formatierung seiner Details in konkreter Syntax (Zeilen 1–3), während die Transition `$T` in einer einzigen Zeile beschrieben werden kann (Zeile 5).

Insbesondere beim Ersetzen von Objekten kann es vorkommen, dass im Vergleich zu Modellen in der Statechartsprache zusätzliche Zeilenumbrüche sinnvoll sind. Die Regel in Abbildung 5.18 veranschaulicht dies für die Eliminierung der Do-Aktivitäten aus Statecharts. Das Ersetzen der Entry- und Exit-Aktionen (Zeilen 2–6 und 10–14) ließe sich in je einer Zeile nicht übersichtlich aufschreiben. Hier empfiehlt es sich, die transformationsspezifischen Elemente und die Modellobjekte in einzelne Zeilen zu schreiben, wobei die Modellobjekte um eine Ebene tiefer eingerückt werden als die transformationsspezifischen Elemente. In diesem Sinne werden auch die Schemavariablen für Objekte, zum Beispiel `$B_ENTRY` als Modellobjekte betrachtet, sodass sie genau den Platz entsprechender Elemente in konkreter Syntax einnehmen.

Werden Objekte durch die Regel erzeugt oder gelöscht, so ist es oft platzsparender und vergleichbar übersichtlich, die transformationsspezifischen Elemente in die erste und letzte Zeile der DSL-Elemente zu schreiben. Für die einzeilige Definition einer Do-Aktivität ist dies in Zeile 8 zu sehen. Für mehrzeilige Objekte wie die interne Aktion in den Zeilen 16–19 werden die Formatierungskonventionen der Statechartsprache übernommen. Dies betrifft neben den Zeilenumbrüchen auch die Einrückung, sodass die Anweisungen in den Zeilen 17 und 18 um eine Ebene tiefer gestellt sind als die Definition der internen Transition.

```
Statechart-Transformationsregel
1  state $s_name {
2    [[
3      entry : BlockStatement $B_ENTRY;
4    :-
5      entry : { BlockStatement $B_ENTRY; timer.set(this, delay); }
6    ]]
7
8    [[ do : BlockStatement $B_DO; :- ]]
9
10   [[
11     exit : BlockStatement $B_EXIT;
12   :-
13     exit : { timer.stop(this); BlockStatement $B_EXIT; }
14   ]]
15
16   [[ :- -intern>: timeout / {
17            timer.set(this, delay);
18            BlockStatement $B_DO;
19         }; ]]
20 }
```

Abbildung 5.18.: Formatierte Transformationsregel

5.4. Übersetzung in Objektdiagramm-Notation

Um die Codegenerierung für Transformationsregeln in konkreter Syntax beherrschbar zu gestalten, erfolgt anstatt einer direkten Codegenerierung nach Java eine Übersetzung in die Objektdiagramm-basierte Notation von Transformationsregeln, die in Kapitel 4 vorgestellt wurde. Die Codegenerierung aus Regeln in abstrakter Syntax ist bereits ein gut untersuchtes Forschungsfeld, in dem auf bestehende Ergebnisse und Publikationen aufgebaut werden konnte (vgl. Kapitel 4).

Die Übersetzung in Regeln in Objektdiagramm-Notation konnte auch deshalb gut beherrschbar gehalten werden, weil die abstrakte Syntax der Regelsprache für Statecharts systematisch aus der Syntax der Statechartsprache abgeleitet wurde und ihr sogar strukturell ähnlich ist. Als Konsequenz daraus ist die in diesem Kapitel vorgestellte Generierung von Regeln in abstrakter Syntax weniger komplex als eine direkte Codegenerierung nach Java.

5.4.1. Generierung der linken und rechten Regelseite

Die Übersetzung von Regeln in konkreter Syntax in die Objektdiagramm-Notation erfordert vor allem die Erstellung einer linken und rechten Regelseite, die sich aus der integrierten Darstellung der Regel in konkreter Syntax ergibt. Dabei gilt die in Abschnitt 5.1 beschriebene Konvention, dass nicht besonders gekennzeichnete Bereiche der Regel zur linken und zur rechten Regelseite gehören. Abweichungen hiervon werden durch einen Ersetzungsblock in doppelten eckigen Klammern und den Ersetzungsoperator :− gekennzeichnet.

Abbildung 5.19 zeigt eine Regel zum Weiterleiten einer Transition an einen initialen Subzustand, die gleichzeitig die `initial`-Markierung des Subzustands entfernt. In dieser Abbildung

ist die Regel in konkreter Syntax dargestellt. Abbildung 5.20 zeigt die hieraus generierte Regel in Objektdiagramm-Notation.

```
Statechart-Transformationsregel
1 state $outer {
2   State $S [[ state $inner << [[ initial :- ]] >>; ]]
3 }
4 Transition $T [[ $source -> [[ $outer :- $inner ]]; ]]
```

Abbildung 5.19.: Regel zum Umleiten von Transitionen in konkreter Syntax

Die generierte Transformationsregel enthält drei Objekte: die Zustände `state_1` und `$S` sowie die Transition `$T`. Der Name `state_1` wurde dabei vom Transformator generiert, weil für den Zustand mit dem Namen `$outer` (Abbildung 5.19, Zeilen 1–3) in der Regel in konkreter Syntax kein Objektname angegeben ist. Diese Objekte sind in der generierten Regel auf beiden Seiten vorhanden, da sie nicht innerhalb eines Ersetzungsblocks spezifiziert sind.

Ebenfalls auf beiden Regelseiten enthalten ist die Kompositionsbeziehung zwischen den Zuständen `state_1` und `$S`. Die Kompositionsbeziehung ergibt sich daraus, dass in der Regel in konkreter Syntax `$S` ein Unterzustand des Zustands zu `state_1` ist.

Unterschiede zwischen der linken und der rechten Regelseite ergeben sich dagegen bei den Attributen: durch die Ersetzung `[[ initial :- ]]` in Zeile 2 der Abbildung 5.19 muss das boolesche Attribut `initial` des entsprechenden Zustandes von `true` auf den Wert `false` gesetzt werden. Dies ist in Abbildung 5.20 in den Zeilen 6 und 20 berücksichtigt.

Der Bezeichner `to` der Transition `$T` wird durch die Ersetzungsoperation `[[ $outer :- $inner ]]` vom Namen des Objekts `state_1`, auf den `$outer` gematcht wird, auf das Match von `$inner` gesetzt, was gleichzeitig der Wert des Attributs `name` von `$S` ist. Dies ist in Abbildung 5.20 in den Zeilen 10 und 24 angegeben.

Das Löschen und Erzeugen von Regeln wird in der Objektdiagramm-Notation dadurch angegeben, dass ein Objekt nur auf der linken oder nur auf der rechten Regelseite existiert. Die Abbildungen 5.21 und 5.22 sind aus den Regeln in konkreter Syntax generiert, die in den Abbildungen 5.5 und 5.6 vorgestellt wurden. Zu beachten ist hier, dass nicht nur Objekte erzeugt oder gelöscht werden, sondern dass auch entsprechend der Vorgaben aus den Regeln in konkreter Syntax Kompositionslinks angelegt beziehungsweise entfernt werden. Dies ist durch die Links in der Zeilen 13–15 der Abbildung 5.21 und den Zeilen 9–11 der Abbildung 5.22 sichergestellt, die jeweils auf der anderen Regelseite fehlen.

Das Verschieben von Objekten lässt sich durch geeignete Kombinationen von Erzeuge- und Löschoperationen spezifizieren, wie in Kapitel 5.1.2 bereits erläutert wurde.

5.4.2. Listen- und negative Knoten, Constraints und nichtisomorphe Matches

Die Syntax der Transformationsregeln sieht vor, dass Listenknoten und negative Knoten durch die Schlüsselwörter `list` und `not` eingeleitet werden und außerdem ein echtes Objekt innerhalb des Blocks in doppelten eckigen Klammern enthalten. Bei der Übersetzung in Objekt-

Transformationsregel in Objektdiagramm-Notation

```
1 pattern objectdiagram lhs{
2
3   state_1:mc.testcases.statechart._ast.ASTState;
4
5   $S:mc.testcases.statechart._ast.ASTState{
6     boolean initial = true;
7   }
8
9   $T:mc.testcases.statechart._ast.ASTTransition{
10    String to = state_1.name;
11  }
12
13  composition state_1 -- (states) $S ;
14
15 }replacement objectdiagram rhs{
16
17  state_1:mc.testcases.statechart._ast.ASTState;
18
19  $S:mc.testcases.statechart._ast.ASTState{
20    boolean initial = false;
21  }
22
23  $T:mc.testcases.statechart._ast.ASTTransition{
24    String to = $S.name;
25  }
26
27  composition state_1 -- (states) $S ;
28 }
```

Abbildung 5.20.: Generierte Regel zum Umleiten von Transitionen in Objektdiagramm-Notation

Transformationsregel in Objektdiagramm-Notation

```
1 pattern objectdiagram lhs{
2
3   state_1:mc.testcases.statechart._ast.ASTState;
4
5 }replacement objectdiagram rhs{
6
7   state_1:mc.testcases.statechart._ast.ASTState;
8
9   entryAction_1:mc.testcases.statechart._ast.ASTEntryAction;
10
11  blockStatement_1:mc.testcases.statechart._ast.ASTBlockStatement;
12
13  composition state_1 -- (entryAction) entryAction_1 ;
14
15  composition entryAction_1 -- (block) blockStatement_1 ;
16
17 }
```

Abbildung 5.21.: Generierte Regel zum Erzeugen einer Entry-Aktion

Transformationsregel in Objektdiagramm-Notation

```
1 pattern objectdiagram lhs{
2
3   state_1:mc.testcases.statechart._ast.ASTState;
4
5   exitAction_1:mc.testcases.statechart._ast.ASTExitAction;
6
7   blockStatement_1:mc.testcases.statechart._ast.ASTBlockStatement;
8
9   composition state_1 -- (exitAction) exitAction_1 ;
10
11  composition exitAction_1 -- (block) blockStatement_1 ;
12
13 }replacement objectdiagram rhs{
14
15  state_1:mc.testcases.statechart._ast.ASTState;
16
17 }
18
```

Abbildung 5.22.: Generierte Regel zum Löschen einer Exit-Aktion

diagramm-Notation genügt es daher, das enthaltene Objekt zu übersetzen und dann mit dem entsprechenden Stereotypen zu versehen.

Abbildung 5.23 zeigt eine Regel in konkreter Syntax, die einen negativen und einen Listenknoten enthält. Die hieraus generierte Regel in Objektdiagramm-Notation wurde bereits in Abbildung 4.11 gezeigt.

```
Statechart-Transformationsregel
1 state $outer {
2   not [[ state $inner <<final>>; ]]
3   list $L [[ state $_ << [[ :- final ]] >> ; ]]
4 } where {
5   !match.$L.isEmpty()
6 }
```

Abbildung 5.23.: Konkrete Syntax mit Listen- und negativen Knoten zu Abbildung 4.11

Aus diesen beiden Abbildungen ist auch ersichtlich, dass die Constraints, die vom Benutzer als Java-Ausdrücke gegen die abstrakte Syntax der Statechartsprache implementiert werden, unmodifiziert übernommen werden. Ebenfalls kopiert werden können die `folding`-Anweisungen für nichtisomorphe Matches, da die Objektbezeichner aus der handgeschriebenen Regel in die generierte Regel übernommen werden; das Matching der Beispielregel soll jedoch isomorph sein, sodass hier keine `folding`-Anweisung verwendet wird.

5.4.3. Implementierung des Transformators

Die Übersetzung von Transformationsregeln in konkreter Syntax in objektdiagrammbasierte Regeln kann als Modell-zu-Modell-Transformation aufgefasst werden. Die in den vorigen Abschnitten vorgestellte Abbildung zeigt, dass sich die Elemente in der generierten Regel jeweils aus kleinen Teil-ASTs der Transformationsregel in konkreter Syntax ergeben. Die Übersetzungstransformation operiert also lokal auf überschaubaren Unterbäumen. Zur Implementierung der Übersetzung wurde daher das Entwurfsmuster Visitor [GHJV95] verwendet, von dem für MontiCore eine angepassten Version existiert. Für dieses Muster wird von MontiCore Infrastruktur in die Klassen der abstrakten Syntax generiert sowie durch die Laufzeitumgebung zur Nutzung in Werkzeugen bereitgestellt [Kra10].

Abbildung 5.24 zeigt die Verwendung des Entwurfsmusters Visitor zur Generierung von Transformationsregeln in Objektdiagramm-Notation, wobei davon abstrahiert wird, dass zur Erzeugung von Objekten eigentlich das Entwurfsmuster der Fabrikmethode [GHJV95] eingesetzt wird.

Bei Traversierung einer Regel in konkreter Syntax erzeugt der generierte `Statechart-Rule2ODVisitor` zunächst eine Regel in abstrakter Syntax mit zwei leeren Objektdiagrammen. Dies ist in der abstrakten Oberklasse `Rule2ODVisitor` implementiert, welche die Codeanteile enthält, die von der Statechartsprache unabhängig sind.

Anschließend durchläuft der Visitor die gesamte Transformationsregel und legt zu jedem `IPattern`-AST-Knoten ein `ASTODObject` im Objektdiagramm an. Dies findet für jeden Typ von Knoten in einer eigenen `visit`-Methode statt, die in der Abbildung aus Platzgründen le-

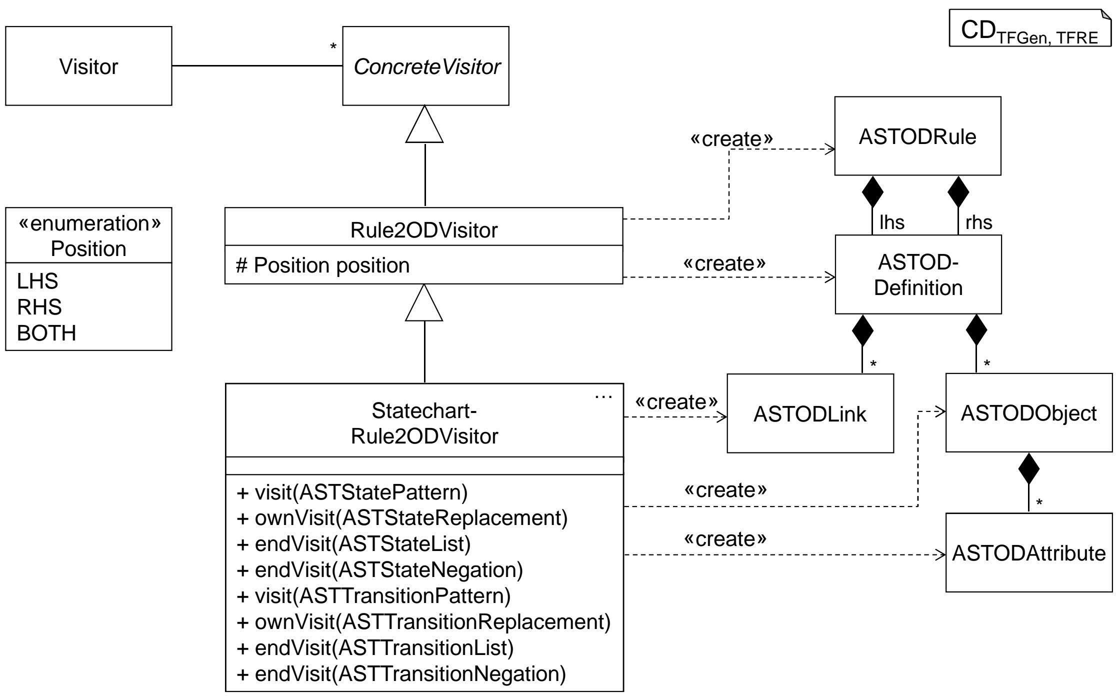


Abbildung 5.24.: Übersetzung der Regeln mit dem Entwurfsmuster Visitor

diglich für Zustände und Transitionen dargestellt ist. Das weiteren werden in diesen Methoden `ASTODAttributes` erzeugt, welche die Bedingungen für Attributwerte aus der Regel in konkreter Syntax widerspiegeln.

Durchläuft der Visitor einen Ersetzungsknoten, also ein Objekt, dessen Klasse `IReplacement` implementiert, so wird die Traversierung unterbrochen. Anschließend wird zuerst das Attribut `Position` auf den Wert `LHS` gesetzt. Dies sorgt dafür, dass während der folgenden Traversierung des Musterteils der Ersetzung nur im Objektdiagramm zur linken Regelseite Objekte erzeugt werden. Anaolg erfolgt die Traversierung des Ersetzungsteils, nachdem `Position` auf `RHS` gesetzt wurde. Für die Traversierung weiterer Knoten wird das Attribut abschließend wieder auf den Standardwert `BOTH` gesetzt.

Die Markierung der negativen Knoten und Listenknoten erfolgt über das Hinzufügen von Sterotypen zu den entsprechenden Objekten. Dies kann erst nach dem Erzeugen der Objekte erfolgen, sodass beim Erreichen eines negativen oder Listenknotens zuerst die entsprechenden Kindknoten traversiert werden. Nachdem diese besucht wurden, wird in der `endVisit`-Methode der Stereotyp dem erzeugten Objekt hinzugefügt.

5.5. Umgang mit Bezeichnern

Wie aus den bisherigen Ausführungen bereits ersichtlich ist, spielen Bezeichner in textuellen Sprachen eine herausragend wichtige Rolle. Sie erlauben es, Objekten eindeutige Namen zu ge-

ben und andere Objekte über ihren eindeutigen Namen zu referenzieren. Daher sind Bezeichner auch in den Transformationssprachen von zentraler Bedeutung. Der Umgang mit ihnen wird deshalb in diesem Abschnitt gesondert erläutert.

5.5.1. Bezeichner bei der Mustersuche

Innerhalb der Regeln können Bezeichner zum einen an den Stellen auftreten, die in der konkreten Syntax der Statechartsprache einen Bezeichner erfordern. Zum anderen können sie auch als explizite Bezeichner für Objekte vergeben werden (vgl. Abschnitt 5.1.1). Orthogonal hierzu wird zwischen drei verschiedenen Arten von Bezeichnern unterschieden:

- *Schemavariablen* beginnen mit einem `$`-Zeichen. Sie stehen für beliebige, aber feste Objekte oder Bezeichner im Match. Wird eine Schemavariable innerhalb der Regel mehrfach verwendet, wie `$source` und `$outer` in Abbildung 5.1, so handelt es sich bei den Matches im Modell um gleiche Bezeichner oder um das selbe Objekt. Ob es sich bei den Bezeichnern im Modell um ein definierendes oder referenzierendes Auftreten handelt, ist dabei unbedeutend.

- Die *anonyme Schemavariable* `$_` stellt eine Ausnahme zur oben genannten Konvention da. Sie darf beliebig oft verwendet werden, ohne dass im Match Gleichheit der Bezeichner oder Identität der Objekte gefordert wird.

- Alle Bezeichner, die nicht mit einem `$`-Zeichen beginnen, sind *feste Bezeichner*, die genau wie angegeben im Match vorhanden sein müssen. Diese Benennung ist nur für Bezeichner zulässig, aber nicht für Objekte. Damit auch feste Bezeichner angegeben werden können, die mit dem `$`-Zeichen beginnen, kann ein Backslash als Escape-Zeichen vorangestellt werden. So wird durch den Bezeichner `\$a` in der Transformationsregel angegeben, dass der entsprechende Bezeichner im Match `$a` sein muss.

5.5.2. Bezeichnerersetzung

Ein Beispiel zum Setzen von Bezeichnern wurde bereits in Abschnitt 5.1.2 gegeben. Ergänzend sind hier noch einige Kontextbedingungen zur Verwendung von Schemavariablen auf der rechten Regelseite anzugeben.

Da anonymen Schemavariablen im Match kein fester Wert zugewiesen wird, dürfen sie im Ersetzungsteil einer Regel, also auf der rechten Regelseite, nicht verwendet werden. Andere Schemavariablen sind auf der rechten Regelseite zulässig, wenn auf der linken Regelseite diese Schemavariable auch verwendet wird und das entsprechende Objekt oder der Bezeichner einen kompatiblen Typ hat. Alternativ kann der Wert einer Schemavariablen auch als Parameter an die Regel übergeben werden, dies wird in Kapitel 6 erläutert. Auch in diesem Fall kann die Schemavariable auf der rechten Regelseite verwendet werden. Feste Werte können für Bezeichner jederzeit auf der linken und auf der rechten Regelseite verwendet werden.

5.5.3. Bezeichner bei der Generierung von Objektdiagrammregeln

Bei der Generierung von Regeln in Objektdiagramm-Notation werden Bezeichner für Objekte aus der Regel in konkreter Syntax übernommen. Dadurch können sie auch in `folding`-Mengen und in Constraints unmodifiziert übernommen werden, sodass für diese Teile der Regel keine Übersetzung stattfinden muss. In den Abbildungen 5.19 und 5.20 ist zu sehen, dass die Objektbezeichner `$S` und `$T` bei der Generierung übernommen wurden.

Neben den Objekten mit expliziten Bezeichnern kommen in den Regeln in domänenspezifischer Notation auch noch Elemente in konkreter Syntax vor, für welche kein expliziter Objektname vergeben wurde. Auch für die hieraus erzeugten Objekte in der generierten Regel ist jedoch ein Name erforderlich, um etwa Links angeben zu können, an denen diese Objekte beteiligt sind. So wurde für den Zustand in der Zeilen 1–3 der Abbildung 5.19 das Objekt `state_1` generiert (Zeilen 3 und 17 in Abbildung 5.20), das an den Kompositionen in den Zeilen 13 und 27 beteiligt ist.

Der Name für ein generiertes Objekt, das in konkreter Syntax nicht explizit benannt ist, ergibt sich aus dem Typ des Objekts und einer fortlaufenden Nummer. Dieses Schema wird auch zur Benennung von Objekten verwendet, die aus anonymen Schemavariablen generiert wurden.

Von den Namen der Objekte abzugrenzen sind wiederum die Bezeichner im Modell beziehungsweise die Schemavariablen für solche Bezeichner. In Abbildung 5.19 wird für den Namen des Zustands in Zeile 1 die Schemavariable `$outer` verwendet. Diese wird ebenfalls als Name des Zielzustands der in Zeile 4 beschriebenen Transition angegeben. Trifft bei der Übersetzung in Objektdiagramm-Notation der Visitor auf eine Schemavariable für einen Bezeichner, so wird das zugehörige Attribut, im Beispiel `state_1.name`, in einer Map gespeichert. Bei weiteren Vorkommen dieses Bezeichners wird dann festgestellt, dass es sich um eine bereits bekannte Schemavariable handelt. Hieraus ergibt sich dann für die Transition `$T` in den Zeilen 23–25 der Abbildung 5.20 die Bedingung für das Attribut `String to = $S.name;`.

Ob diese Bedingung an der Transition angegeben wird, oder ob eine äquivalente Bedingung für den Zustand generiert wird, hängt von der Reihenfolge ab, in welcher der Visitor die Regel in konkreter Syntax definiert. Dies hat jedoch nur syntaktisch einen Einfluss auf die generierte Regel, die Semantik ist dagegen von der Reihenfolge der Traversierung unabhängig.

Tabelle 5.25 fasst noch einmal die Behandlung der Bezeichner aus den Abbildungen 5.19 und 5.20 zusammen. Dabei ist zu beachten, dass bei Schemavariablen für Bezeichner eines der Zielelemente aus den soeben genannten Gründen nicht in der generierten Regel auftritt.

Quellelement	Typ	Generierte Elemente
unbenannt	`State`	`state_1`
`$S`	`State`	`$S`
`$T`	`Transition`	`$T`
`$outer`	`Name`	`$T.to` (nur LHS), `state_1.name`
`$inner`	`Name`	`$T.to` (nur RHS), `$S.name`
`$source`	`Name`	`$T.from`

Tabelle 5.25.: Bezeichner bei der Übersetzung der Regel aus Abbildung 5.19

5.6. Verwandte Arbeiten

Die Idee, Benutzer ohne tiefgehende Programmierkenntnisse in die Lage zu versetzen, Programme zu erstellen oder ihren Bedürfnissen anzupassen, zieht sich durch eine Vielzahl von softwaretechnischen Publikationen der letzten Jahrzehnte [Shn82, CHK+93, Lie01] und geht sogar soweit, dass auch Kindern die Erstellung von Programmen ermöglicht werden soll [Sta11, LEG11].

Das Bestreben, Transformationen in konkreter Syntax der zu transformierenden Modelle zu notieren und so für Domänenexperten verständlich zu machen, kann als Spezialfall dieser Bemühungen betrachtet werden. So gibt es auch im Bereich der Modell- und Programmtransformationen bestehende Ansätze, die Gemeinsamkeiten mit der vorliegenden Arbeit aufweisen. Diese Gemeinsamkeiten und auch die Unterschiede werden im Folgenden herausgearbeitet. Wegen des engen Zusammenhangs mit der Generierung von Transformationssprachen in konkreter Syntax wird dieser Aspekt im Bezug auf verwandte Arbeiten auch hier besprochen. Die Generierung von Transformationssprachen wie der hier beschriebenen Statecharttransformationssprache mit MontiCore, insbesondere die technische Umsetzung, wird in dieser Arbeit in Kapitel 7 beschrieben.

Eine naheliegende Möglichkeit, Transformationen in einer domänenspezifischen Syntax auszudrücken, ist es, zu einer DSL eine spezielle Transformationssprache zu erstellen.

Steel und Drogemuller beschreiben eine solche Sprache für Gebäudemodelle in [SD11]. Diese unterscheiden sich technisch zwar substanziell von den in dieser Arbeit betrachteten Modellen. Jedoch untermauert der Ansatz den Bedarf, Transformationssprachen auch für Domänenexperten verständlich und beschreibbar auszugestalten.

Eine eigens für eine bestimmte Modellierungssprache nutzbare Transformationssprache wird auch von Schmidt in [Sch06] für die UML beschrieben. Allerdings ist zu diesem Ansatz keine Implementierung bekannt.

Einen Schritt weiter gehen Baar und Whittle in [BW06]. Hier wird das Metamodell einer Transformationssprache aus dem Metamodell einer Modellierungssprache generiert. Für diese generierte Transformationssprache ließe sich beispielsweise eine auf Objektdiagrammen basierende Notation automatisiert erstellen. Während andere auf Objektdiagrammen basierende Transformationssprachen jedoch nur eine solche generische Notation nutzen, liegt ein wesentlicher Vorteil des generierten Metamodells darin, dass sich mit Editorframeworks hierzu leicht eine domänenspezifische konkrete Syntax definieren lässt. Allerdings ist [BW06] auf die Syntax der Transformationen beschränkt und enthält keine Angaben über die Ausführung der Regeln oder über eine Implementierung.

Eine Alternative zur Definition einer speziellen Transformationssprache ist die Ableitung von Transformationsregeln aus exemplarischen Quell- und Zielmodellen. Für Modell-zu-Modell-Transformationen wird dies zum Beispiel von Kindler und Wagner in [KW07] beschrieben. Bei Ansätzen dieser Art dienen Quell- und Zielmodell als Ausgangsversion für die linke und rechte Regelseite. Diese können dann beispielsweise durch Entfernen von Attributen oder Ersetzen konkreter Klassen durch Supertypen generalisiert werden. Hierbei editiert der Nutzer allerdings rein auf der abstrakten Syntax der Sprachen basierende Regeln, und er verwendet eine generische Notation für Transformationsregeln, anstatt spezifische Sprachen zu generieren. Auch zu

diesem Ansatz ist keine Implementierung bekannt. Von der vorliegenden Arbeit grenzt er sich außerdem dadurch ab, dass er exogene Modell-zu-Modelltransformationen behandelt, während hier In-Place-Transformationen beschrieben werden.

Die Ableitung von Transformationsregeln aus Beispielen wird auch als Inferenz von Transformationsregeln bezeichnet, und es gibt viele weitere Ansätze, die dieser Kategorie zuzuordnen sind und die auch bereits implementiert wurden. Der zentrale Nachteil solcher Ansätze ist, dass die inferierten Regeln entweder nur auf einer sehr eingeschränkten Klasse von Modellen matchen, oder eine Generalisierung erforderlich ist, die nach der Generierung der Beispielregel händisch durchgeführt wird, und bei welcher der Nutzer sich mit der abstrakten Syntax der zu transformierenden Modelle auseinandersetzen muss. Als Beispiel seien hier die Arbeiten von D. Varró [Var06], Kappel, Langer, Wimmer et al. [BLS+09, LWK10], Sun, White und Gray [SWG09] oder Nanda et al. [NMSS09] genannt, die Liste ließe sich jedoch noch fortsetzen.

Die Generierung von Modelltransformationssprachen in konkreter Syntax wurde bisher vor allem für grafische Sprachen untersucht. R. Grønmo stellt in [GMP09, Grø09] einen Ansatz zur Generierung grafischer Transformationssprachen aus grafischen DSLs vor, der weitgehend, aber nicht voll automatisiert ist [Grø09, Kap. 6.2].

Ebenfalls auf grafische Sprachen zielen Kühne et al. in [KMS+10] ab. Im diesem Ansatz werden spezifische Transformationssprachen zu gegebenen grafischen Modellierungssprachen generiert. Ferner ist eine Implementierung auf Basis der Transformationsengine von AToM3/MoTif entstanden. Die Autoren argumentieren, dass eine Anpassung der generierten Sprache sinnvoll ist, um die Ausdrucksmächtigkeit der Transformationssprache zu erhöhen. Unklar bleibt allerdings, ob diese Anpassung technisch zwingend erforderlich ist, oder ob auch eine einfache Transformationssprache vollautomatisch generiert werden kann. Neben der Verwendung einer grafischen Notation unterscheidet diesen Ansatz von der vorliegenden Arbeit auch, dass eindeutige Bezeichner für jedes Modellelement vergeben werden müssen. Außerdem handelt es sich auch bei dieser Arbeit um exogene Transformationen. Folglich gibt es hier auch keine integrierte Notation der linken und rechten Regelseite.

Im weiteren Sinne zu den Transformationsregeln in konkreter Syntax für grafische Sprachen ist auch die Generierung syntaxgesteuerter grafischer Editoren zu zählen, die Bardohl et al. in [BEW04] beschreiben. Allerdings liegt der Fokus hier auf kleinen Änderungsoperationen, die händisch von einem Benutzer in einem Editor ausgeführt werden, und weniger auf der Automatisierung komplexer Transformationen.

Modelle und Programme in einer textuellen Notation werden häufig nicht mit Graphersetzungssystemen, sondern mit Termersetzungssystemen transformiert. Hierfür beschreibt Visser in [Vis02], wie sich Termersetzungen durch Regeln beschreiben lassen, in denen die konkrete Syntax der zu transformierenden Terme verwendet wird. Termersetzungssysteme zielen allerdings typischerweise auf die Manipulation kleiner und zusammenhängender Teil-ASTs ab, während in graphbasierten Ansätzen die Regeln auf verschiedenen Teilgraphen operieren können, die beliebig über den abstrakten Syntaxbaum oder -graphen verteilt sind oder sogar aus verschiedenen Hostgraphen stammen können.

Ebenfalls den Transformationen in konkreter Syntax ähnlich ist die Angabe von Deltas zu einem Modell in einer speziellen Delta-Modellierungssprache. In [HRRS11] werden Deltas für Modelle in der textuellen Architekturbeschreibungssprache MontiArc behandelt, wobei sich die

Sprache zur Beschreibung von Deltas syntaktisch eng an der Architekturbeschreibungssprache orientiert. Sie ergänzt allerdings diese Sprache unter anderem um Operatoren für das Hinzufügen oder Entfernen von Komponenten. Zusätzlich werden auch Operatoren eingeführt, die semantisch sinnvolle komplexe Transformationen beschreiben, etwa das automatische Anlegen von Verbindungen zwischen Komponenten. Allerdings ist diese Sprache manuell erstellt und nicht aus der Architekturbeschreibungssprache automatisch abgeleitet.

Eine Transformationssprache für Java, die Muster in konkreter Syntax unterstützt, beschreiben Appeltauer und Kniesel in [AK08]. Zwar sind Transformationen in dieser Sprache nicht auf andere Sprachen als Java anwendbar. Jedoch unterstreicht die Transformation von Java-Programmen die Nützlichkeit und Anwendbarkeit von Regeln in konkreter Syntax für umfangreiche, praxisrelevante Sprachen. Eine Sprache mit vergleichbaren Anwendungsmöglichkeiten ist JTL, das Cohen et al. in [CGM06] beschreiben. In beiden Ansätzen liegt der Fokus jedoch auf der Mustersuche in konkreter Syntax; Ersetzungen werden nur rudimentär unterstützt.

In Tabelle 5.26 sind noch einmal die wichtigsten Gemeinsamkeiten und Unterschiede des in dieser Arbeit vorgestellten Ansatzes im Vergleich mit ausgewählten verwandten Arbeiten in einer Übersicht zusammengefasst.

Quelle/ Werkzeug	**Ansatz/Transformationssprache**	**Modelle**	**Transformationen**	**Implementierung**
[Sch06]	manuelle Sprachdefinition	UML 2	in-Place, graphbasiert	nein
[Var06], [KW07], [BLS+09, LWK10]	Regelinferenz	grafische	exogen, graphbasiert	nur [BLS+09, LWK10]
[GMP09, Grø09]	größtenteils generierte Sprache	grafische	in-Place, graphbasiert	ja
[KMS+10]	generierte Sprache, manuelle Anpassungen	grafische	exogen, graphbasiert	ja
[BEW04]	syntaxgesteuerter Editor	grafische	in-Place, graphbasiert	ja
[HRRS11]	manuelle Sprachdefinition	MontiArc	Deltas in konkreter Syntax, komplexe Operatoren	ja
[AK08], [CGM06]	manuelle Sprachdefinition	Java	Muster in konkreter Syntax	ja
[Vis02]	generierte Sprache	textuelle	in-Place, termbasiert	ja
MontiCore	generierte Sprache	textuelle	in-Place, graph- und bezeichnerbasiert	ja

Tabelle 5.26.: Auswahl verwandter Arbeiten zu Transformationen in konkreter Syntax

Kapitel 6.

Die Kontrollflusssprache für Modelltransformationen

Die Transformationsregeln, die sich mit den Sprachen beschreiben lassen, die in den Kapiteln 4 und 5 vorgestellt wurden, erlauben vor allem die Beschreibung einfacher Ersetzungsregeln. Für praxisrelevante Anwendungen, etwa die in Kapitel 3 beschriebenen semantikerhaltenden Vereinfachung, Refactorings oder semantikverfeinernde Transformationen ist es jedoch erforderlich, mehrere Regeln geeignet zu kombinieren. Dabei kann die Entscheidung, ob und welche Regeln im Folgenden anzuwenden sind, von früheren Transformationsergebnissen oder von Modelleigenschaften abhängen.

Die dynamische Verkettung von Anweisungen, insbesondere der Ausführung von Regeln, kann entweder implizit oder explizit erfolgen [CH06] und wird für Transformationsregeln als Regel-Scheduling bezeichnet. Bei implizitem Scheduling gibt der Entwickler der Transformationen keine Reihenfolge der Regeln vor, sondern diese wird zur Laufzeit von der Transformationsengine bestimmt. Beim externen Scheduling bietet die Transformationssprache dagegen Konzepte zur Steuerung der Regelauswahl an. Das Scheduling der Regeln wird also hier bereits zur Designzeit der Transformation durch den Kontrollfluss definiert, die tatsächlich Abfolge hängt allerdings im Allgemeinen vom transformierten Modell ab und wird somit erst zur Laufzeit festgelegt.

Die Beschreibung des Kontrollflusses zur Designzeit kann zum Beispiel durch Aktivitätsdiagramme [FNTZ00] oder vergleichbare Notationen geschehen, in denen nach einer Regel ein Sprung oder bedingter Sprung zur nächsten auszuführenden Regel angegeben wird. Sprunganweisungen sind insbesondere bei grafischen Sprachen verbreitet.

Für textuelle Sprachen gelten Sprunganweisungen nach der von Dijkstra angestoßenen Diskussion [Dij68] dagegen als schwer verständlich, sie verschlechtern somit die Wartbarkeit von Softwaresystemen. Stattdessen haben sich Kontrollstrukturen durchgesetzt, die auch der in diesem Kapitel beschriebenen Kontrollflusssprache zugrunde liegen.

Das Design der Sprache ist in vielen Punkten an die Programmiersprache Java [Fla05] angelehnt. Diese und weitere an Java angelehnte Sprachen wie die UML/P spielen im Umfeld der vorliegenden Arbeit eine wichtige Rolle, sodass sich die entworfene Sprache in eine Sammlung nach ähnlichen Kriterien entworfener Sprachen eingliedert.

Die Entwicklung der Sprache und die Implementierung erfolgte größtenteils im Rahmen der Masterarbeit [Sch11]. Dort werden auch Aspekte der Implementierung detaillierter erläutert, als es im Rahmen dieses Kapitels möglich ist.

6.1. Aufbau der Transformationsmodule

Die angestrebte Ähnlichkeit zu Java und die bessere Verständlichkeit gaben den Ausschlag zur Entscheidung für eine explizite Definition des Kontrollflusses. Darüber hinaus wurden weitere Entscheidungen zum Sprachdesign getroffen, die in diesem Abschnitt gemeinsam mit den realisierenden Konstrukten vorgestellt werden.

6.1.1. Module und Methoden

Ein Programm in der Transformationsssprache lässt sich in mehrere Module unterteilen. Jedes Modul ist dabei in einer separaten Datei abgelegt. Die Module entsprechen in der Granularität etwa den Java-Klassen. Moduldefinitionen sind die CompilationUnits (vgl. [Kra10]) der Kontrollflusssprache. Sie können daher eine Paketdeklaration und `import`-Anweisungen enthalten.

Die eigentliche Moduldefinition wird durch das Schlüsselwort `module` eingeleitet. Es folgt ein Name sowie ein Block der im Modul definierten Methoden. Im Gegensatz zu Java-Klassen enthalten Module aber außer Methoden keine weiteren Member-Elemente. Abbildung 6.1 zeigt ein Modul mit drei Methodendeklarationen.

Statechart-Transformation

```
1  package mc.tf.sc;
2
3  module SimplifyStatecharts {
4
5    main() {
6      ...
7    }
8
9    forwardToInitial() {
10     ...
11   }
12
13   transformation moveTransitionToState(Transition $T, State $S) {
14     ...
15   }
16 }
```

Abbildung 6.1.: Ein Modul mit drei Methoden

Die Kontrollflusssprache kennt zwei Varianten der Methodendeklaration: Anweisungsmethoden und Transformationsregelmethoden. Anweisungsmethoden, in der Abbildung in den Zeilen 5 ff. und 9 ff. zu sehen, bestehen aus einem Methodennamen, einer Parameterliste und einem `BlockStatement`, also im Wesentlichen einer Liste von Anweisungen. Transformationsregelmethoden werden, wie in Zeile 13 zu sehen ist, zur Unterscheidung durch das Schlüsselwort `transformation` eingeleitet. Sie haben ebenfalls einen Namen und eine Parameterliste. Anstelle des `BlockStatements` besteht ihre Implementierung jedoch aus einer Transformationsregel, die für Statecharts in einer der beiden in den Kapiteln 4 und 5 vorgestellten Regelsprachen

implementiert sein kann. Für die Transformation von Modellen in anderen Sprachen können Regeln in Objektdiagramm-Notation oder in einer spezifischen Regelsprache verwendet werden.

Die Methodendeklarationen in der Kontrollflusssprache enthalten keinen Rückgabetyp. Per Konvention sind alle Methoden vom Typ `boolean`, und der Rückgabewert gibt an, ob die Ausführung der Methode erfolgreich war. Die Ausführung von Regelmethoden ist erfolgreich, wenn die enthaltene Ersetzung angewendet werden konnte. Anweisungsmethoden sind erfolgreich, wenn alle Anweisungen ausgeführt werden konnten. Diese Konvention erlaubt es, alle Methodenaufrufe auch als Bedingungen in Kontrollstrukturen zu verwenden. Des Weiteren wird der Rückgabewert im generierten Code genutzt, um im Falle fehlgeschlagener Anweisungen Backtracking auszulösen. Hierauf wird in Abschnitt 6.3 noch näher eingegangen.

Die Rückgabe von Objekten und Werten muss folglich immer über Parameter erfolgen. Damit so beispielsweise auch zuvor nicht initialisierte Variablen gesetzt werden können oder primitive Datentypen zurückgegeben werden können, verwendet die Kontrollflusssprache *Referenzparameter* (Englisch: *call by reference*) anstatt der in Java üblichen *Werteparameter* (Englisch: *call by value*).

Im generierten Code ist dieser Unterschied technisch durch ein Wrapper-Objekt der Klasse `ObjectWrapper` gelöst. Dieses Objekt kapselt ein Zeigerattribut auf das tatsächliche Objekt. Hat dieser Zeiger beim Aufruf einer Transformationsregelmethode den Wert `null`, so kann ein von der Methode neu erzeugtes Objekt über den entsprechenden Referenzparameter zurückgegeben werden. Wird dagegen ein tatsächliches Objekt innerhalb einer Transformationsregelmethode gelöscht, so hat das Zeigerattribut nach der Ausführung den Wert `null`. Um die Löschung des Objektes aus dem Hauptspeicher kümmert sich dann der Garbage-Collector von Java.

Der Rumpf von Anweisungsmethoden ist an Blockstatements aus Java angelehnt und wird im Folgenden beschrieben. Die Einbettung von Transformationsregeln wird hier noch nicht betrachtet; sie ist der zentrale Aspekt für die Kombination der Kontrollflusssprache mit den Regelsprachen und wird daher in Abschnitt 6.2 gesondert erläutert.

6.1.2. Sequenzielle Ausführung von Anweisungen

Der Rumpf einer Anweisungsmethode besteht im einfachsten Fall aus einer Folge von Anweisungen. Neben den Kontrollstrukturen `if` und `loop`, die im Folgenden noch erläutert werden, kann eine Anweisung auch die Auswertung eines Java-Ausdrucks sein. Dies macht es möglich, in den Anweisungsmethoden auch Java-Methodenaufrufe, Initialisierungen von Variablen oder andere Ausdrücke aus Java einzubetten.

So zeigt Abbildung 6.2 eine Sequenz von Anweisungen aus der Vereinfachung hierarchischer Statecharts. Zunächst wird wie in Java eine Variable vom Typ `Logger` deklariert und mit dem Ergebnis eines Java-Methodenaufrufs initialisiert (Zeile 2). Anschließend erfolgt ein Methodenaufruf auf diesem Objekt (Zeile 4) sowie der Aufruf einer weiteren Anweisungsmethode, die im selben Modul definiert ist.

Der hier gezeigten Methode `main` kommt in den Transformationsmodulen eine besondere Rolle zu: Sie ist der Einstiegspunkt in die Ausführung des aus der Regel generierten Java-Programms. Dabei akzeptiert dieses Programm den Namen einer Eingabedatei als Parameter. Diese enthält das Modell, auf dem die Transformation aufgerufen wird. Daneben gibt es aber

Statechart-Transformation

```
1 main() {
2   Logger log = mc.MCG.getLogger();
3   ...
4   log.finer("propagateInvariantsAndRemoveHierarchy");
5   propagateInvariantsAndRemoveHierarchy();
6 }
```

Abbildung 6.2.: Ausschnitt der `main`-Methode aus der Transformation zur Vereinfachung hierarchischer Statecharts

noch die Möglichkeit, das zu transformierende Modell als Java-Parameter beim Aufruf des generierten Codes zu übergeben, unter anderem um Transformationen im DSLTool-Framework von MontiCore ausführen zu können, oder um den generierten Code aus anderen Java-Prgrammen aufzurufen.

6.1.3. Bedingte Anweisungen und Schleifen

In vielen Fällen hängt die Auswahl einer anzuwendenden Regel im Transformationsprozess von Eigenschaften des Modells oder von einem zu einem früheren Zeitpunkt berechneten Ergebnis ab. In imperativen Programmiersprachen werden vergleichbare Sachverhalte durch die Einführung von Kontrollstrukturen gelöst, in Java beispielsweise durch `if`-Statements sowie `while`- und `for`-Schleifen.

Im Kontext von Modelltransformationen ist es häufig zusätzlich erforderlich, eine Anweisung oder ein Folge von Anweisungen zu iterieren, das heißt so lange anzuwenden, bis sie nicht mehr anwendbar ist. Dies ließe sich zwar durch eine `while`-Schleife, die das boolesche Literal `true` als Bedingung hat, lösen, dies ist jedoch wenig elegant.

Für Transformationen in MontiCore wurden daher die Kontrollstrukturen `loop` und `if` eingeführt. Durch Kombination dieser beiden Konstrukte lassen sich Anweisungen erstellen, die zu den aus imperativen Sprachen bekannten `while`-Schleifen äquivalent sind. Zwar werden für die Angabe solcher Schleifen dann zwei Kontrollstrukturen benötigt, sodass die Programme etwas größer werden. Im Gegenzug wird aber die Kontrollflusssprache einfacher gehalten, da keine zusätzliche Kontrollstruktur eingeführt wird.

Darüber hinaus können auch `for`-Schleifen verwendet werden, jedoch nur zur Iteration über eine Menge oder Liste, aber nicht in der aus Java ebenfalls bekannten Form mit je einem Initialisierungs-, Bedingungs- und Update-Ausdruck.

Wie Kontrollstrukturen innerhalb von Anweisungsmethoden eingesetzt werden könne, zeigt Abbildung 6.3, die der Methode zum Entfernen der Hierarchie aus Statecharts, also dem letzten Schritt der Vereinfachung, entnommen ist. In Zeile 1 der Abbildung wird `loop` in Kombination mit einer `if`-Anweisung verwendet, womit das Verhalten einer `while`-Schleife nachgebildet wird. Hier wird der boolesche Rückgabewert eines Methodenaufrufs ausgewertet, um zu entscheiden, ob und wie oft die folgenden Anweisungen ausgeführt werden sollen. Beim Betreten des Blocks zu dieser `if`-Anweisung wird dann die `for`-Schleife in Zeile 2 benutzt, um über alle

Elemente der Liste `$substates` zu iterieren und diese jeweils in Zeile 4 als Parameter eines Methodenaufrufs zu verwenden.

Statechart-Transformation

```
1 loop if (findTopLevelStateWithSubstates($s, $substates)) {
2   for (State $next : $substates) {
3     ...
4     pullUpState($next);
5   }
6   $s.delete();
7 }
```

Abbildung 6.3.: Verwendung von Kontrollstrukturen

6.1.4. Eingebettete Java-Ausdrücke

Die Ausdrücke in der Kontrollflusssprache werden durch Spracheinbettung [Völ11] aus Java übernommen. Die Übernahme ist dabei nicht auf die kontextfreie Grammatik beschränkt, sondern betrifft gleichzeitig auch das Typsystem und weitere Kontextbedingungen.

Die eingebetteten Java-Ausdrücke machen in der praktischen Anwendung einen erheblichen Teil des Programmcodes aus. Hierunter fallen alle Methodenaufrufe, Zuweisungen, Feldzugriffe, Konstruktoraufrufe, Präfix-, Postfix- und Infix-Ausdrücke. Durch letztere stehen unter anderem auch die gängigen booleschen Operatoren zur Verfügung.

Betrachtet man die loop-Schleife aus der Methode `startFromFinal` in Abbildung 6.4, so können hier die folgenden Java-Ausdrücke identifiziert werden:

- Die Bedingung in Zeile 1 ist ein Methodenaufruf mit einfachem Namen. Obwohl es sich bei dieser Methode um eine Transformationsregel handelt, ist der Aufruf syntaktisch von anderen Methodenaufrufen nicht zu unterscheiden, also ebenfalls ein Java-Ausdruck.
- Auch bei dem Variablennamen `$substates` in Zeile 2 handelt es sich um einen Java-Ausdruck. Die Variablendeklaration `State $substate` ist ebenfalls aus Java entnommen, jedoch kein Ausdruck.
- In Zeile 3 ist eine Variablenzuweisung zu sehen, deren linke Seite aus einer Variablendeklaration und deren rechte Seiten aus einem Methodenaufruf mit qualifizierten Namen besteht.
- In den Zeilen 4 und 5 ist jeweils ein Methodenaufruf mit einfachem Namen zu finden.
- Zeile 7 enthält einen Methodenaufruf mit qualifiziertem Namen.

6.1.5. Typsystem der Kontrollflusssprache

MontiCore unterstützt die in [Völ11] beschrieben durchgängig kompositionale Entwicklung von Sprachen. Neben der kontextfreien Syntax der Java-Ausdrücke konnte daher auch das Typsystem in weiten Teilen aus Java übernommen werden.

Statechart-Transformation

```
1 loop if (searchTransitionForFinalSubstates($t, $superstate, $substates)) {
2   for (State $substate : $substates) {
3     Transition $t_new = $t.deepClone();
4     moveTransitionToState($t_new, $superstate);
5     startTransitionFromSubstate($t_new, $substate);
6   }
7   $t.delete();
8 }
```

Abbildung 6.4.: Eingebettete Java-Ausdrücke

Für Typsicherheit bei Variablenzuweisungen, Methodenaufrufen mit Parametern etc. gelten daher die Kontextbedingungen aus Java. Jedoch ergeben sich aus den Grammatiken für die zu transformierenden Modelle zusätzliche Typen, nämlich ein eigener Typ für jede Produktion in der Symboltabelle der Grammatik. Als Name des Typs wird dabei der Name aus dem Symboltabelleneintrag übernommen.

Über die aus Java bekannten und von MontiCore für jede CompilationUnit (vgl. [Kra10]) zur Verfügung gestellten `import`-Anweisungen können weitere Java-Typen in ein Modul importiert werden. Hierdurch ist auch die Verwendung bereits kompilierter Java-Typen möglich, etwa aller Klassen aus den Bibliotheken der Java-Laufzeitumgebung. Da diese Klassen auf jedem System mit einer Java-Installation vorhanden sind, steht so für Modelltransformationen in MontiCore eine umfangreiche Klassenbibliothek zur Verfügung. Ebenfalls verfügbar sind die Klassen der MontiCore-Laufzeitumgebung, die auch auf jedem System, auf dem Transformationen entwickelt oder ausgeführt werden, installiert sein muss.

Die Verwendung von Klassen aus externen Bibliotheken ist in Abbildung 6.5 dargestellt. In Zeile 3 wird zunächst die Klasse `Logger` aus der Java-Laufzeitbibliothek importiert. Eine Variable dieses Typs wird in Zeile 8 deklariert und initialisiert, wobei als Ausdruck zur Initialisierung der Aufruf der statischen Methode `getLogger()` der Klasse `MCG` verwendet wird. Diese wird durch die MontiCore-Laufzeitumgebung zur Verfügung gestellt und im Beispiel über ihren voll qualifizierten Namen angesprochen.

6.2. Einbettung der Transformationsregeln und Methodenaufrufe

Wie in Abschnitt 6.1.1 bereits beschrieben wurde, erlaubt die Kontrollflusssprache neben der Definition der an Java-Methoden angelehnten Anweisungsmethoden auch die Beschreibung von Regelmethoden. Diese werden durch das Schlüsslwort `transformation` eingeleitet.

Anstelle eines Anweisungsblocks besteht der Rumpf solcher Methoden aus einer einzigen Transformationsregel. Diese kann entweder in der generischen, auf Objektdiagrammen basierenden Notation oder in einer domänenspezifischen Form, wie der Regelsprache für Statecharttransformationen vorliegen. Die konkrete Sprache wird in einer Language-Datei [Kra10] konfiguriert, sodass keine Anpassung der Grammatik erforderlich ist.

Statechart-Transformation

```
1 package mc.tf.sc;
2
3 import java.util.logging.Logger;
4
5 module SimplifyStatecharts {
6
7   main() {
8     Logger log = mc.MCG.getLogger();
9     log.finer("createEmptyActionBlock");
10    ...
11  }
12  ...
13 }
```

Abbildung 6.5.: Verwendung von Klassen aus der Java- und MontiCore-Laufzeitumgebung

Die Methodendeklaration in Abbildung 6.6 enthält zwei Parameter vom Typ `Transition` und `State`. Der Methodenrumpf ist in der domänenspezifischen Regelsprache implementiert, die in Kapitel 5 beschrieben wurde. Je nachdem, ob diese Parameter beim Aufruf der Methode mit Objekten belegt werden, können diese bereits fest als Bestandteil des Matches vorgegeben sein. Nach einer erfolgreichen Ausführung der Regel weisen alle als Argumente übergebenen Zeiger auf das Match des dem jeweiligen Formalparameter entsprechenden Objekts der Regel.

Statechart-Transformation

```
1 transformation moveTransitionToState(Transition $T, State $S) {
2   State $S [[ state $s_name {
3     [[ :- Transition $T; ]]
4   } ]]
5
6   [[ Transition $T; :- ]]
7 }
```

Abbildung 6.6.: Methode zum Verschieben einer Transition in einen Zustand

Wie bereits erwähnt wurde, ist der Rückgabewert eines Methodenaufrufs immer vom Typ `boolean`. Er gibt an, ob die Ausführung der Regel erfolgreich war, während die Rückgabe anderer Werte oder Objekte über die Parameter erfolgen muss.

In den Methodendeklarationen wird nicht zwischen Ein- und Ausgabeparametern entschieden. Dies hat zwar den Nachteil, dass Seiteneffekte einer Methode nicht aus ihrer Signatur ersichtlich sind. Jedoch gilt dies für viele imperative Programmiersprachen, die Referenzparameter erlauben, und es ermöglicht neben einer flexibleren Verwendung der Methoden auch eine größere syntaktische Ähnlichkeit der Methodendeklarationen zu Java. Des Weiteren wäre für die Verfeinerung von Methoden eine Unterscheidung kovarianter und kontravarianter Parameter voneinander beinahe zwingend erforderlich, allerdings ist eine Verfeinerungskonzept für die Kontrollflusssprache weder umgesetzt noch geplant.

Der Aufruf einer Methode ist ein boolescher Ausdruck, der in den meisten Fällen in eine Anweisung eingebettet ist, aber auch als Bedingung verwendet werden kann. Abbildung 6.7 zeigt in den Zeile 4 und 5 den Aufruf einer Methode mit drei Parametern. Wird diese Methode erfolgreich ausgeführt, sind alle drei Variablen belegt und der Rumpf der Schleife wird betreten. Bei den Methodenaufrufen in den Zeilen 8 und 9 werden diese Objekte beziehungsweise die Elemente der Liste `$substates` erneut als Eingabeparameter für Transformationsmethoden verwendet, unter anderem wird auch die in Abbildung 6.6 gezeigte Regel aufgerufen. Diese Methodenaufrufe sind nur dann erfolgreich, wenn das Match den Vorgaben durch die Argumente entspricht.

Statechart-Transformation

```
1  State $superstate;
2  List<State> $substates;
3  Transition $t;
4  loop if (searchTransitionForInitialSubstates($t,
5      $superstate, $substates)) {
6    for (State $substate : $substates) {
7      Transition $t_new = $t.deepClone();
8      moveTransitionToState($t_new, $superstate);
9      redirectTransitionToSubstate($t_new, $substate);
10   }
11   $t.delete();
12 }
```

Abbildung 6.7.: Aufruf von Methoden als Bedingung und innerhalb eines Anweisungsblocks

6.3. Nichtdeterminismus

Bei der Ausführung von Transformationen kann die Engine in bestimmten Schritten zwischen mehreren Alternativen auswählen. So können insbesondere bei der Ausführung einer Regel mehrere mögliche Matches existieren. Das resultierende Modell einer Transformation kann offensichtlich von der Auswahl der Matches abhängen. Wird durch ein solches Match das transformierte Modell derart beeinflusst, dass Matches späterer Regelausführungen entfernt werden, so kann die Wahl eines „schlechten“ Matches sogar dafür ausschlaggebend sein, dass die gesamte Transformation scheitert.

Dies lässt sich durch die Simulation nichtdeterministischen Verhaltens vermeiden. Dabei ergibt sich bei der Programmausführung anstatt einer Sequenz atomarer Operationen ein Baum, in dem zu jeder nichtdeterministischen Entscheidung ein Nachfolgeknoten für jede mögliche Auswahl existiert. Zur Simulation nichtdeterministischen Verhaltens muss in diesem Baum ein Pfad gesucht werden, der eine erfolgreiche Ausführung modelliert. Die bekanntesten Verfahren hierzu sind die Breitensuche und die Tiefensuche [RN03]. Da die Breitensuche für Bäume mit vielen Verzweigungen und einem hohen Verzweigungsgrad einen in der Praxis nicht zu bewältigenden Speicherbedarf hat, wird in MontiCore der Nichtdeterminismus mit einer Tiefensuche im Lösungsraum simuliert. Allerdings hat diese Tiefensuche im Allgemeinen den Nachteil, dass

sie nicht terminiert, wenn die Traversierung vor einer möglichen Lösung auf einen unendlichen Pfad stößt.

Die Tiefensuche führt zunächst die Anweisungen des Programms in gegebener Reihenfolge aus, wobei an jeder Verzweigung die erste mögliche Auswahl getroffen wird. Gerät die Tiefensuche an eine Stelle, an der eine erfolgreiche Fortsetzung des Programms nicht mehr möglich ist, da eine Anweisung fehlschlägt, so wird zur letzten Verzweigung zurückgesprungen, an der eine weitere Auswahl möglich ist. Dieser Rücksprung wird als *Backtracking* bezeichnet. An dieser Stelle wird eine andere Auswahl getroffen, und für diese Auswahl wird die Tiefensuche fortgesetzt.

Die Vereinfachung und weitere Transformationen auf Statecharts lassen sich zwar auch deterministisch mit vertretbarem Aufwand implementieren. Um jedoch für die Erläuterung des Backtrackings keine zusätzliche Beispielsprache einführen zu müssen, wird die in Abbildung 6.8 gezeigte Erweiterung der Regel `pullUpState` verwendet.

Statechart-Transformation

```
1 pullUpStateIfPossible(State $s) {
2   pullUpState($s);
3   !namingConflict();
4 }
5
6 transformation pullUpState(State $S) {
7   [[ State $S; :- ]]
8   statechart $sc_name {
9     [[ :- State $S; ]]
10  }
11 }
12
13 transformation namingConflict() {
14   statechart $_ {
15     state $conflicting_name;
16     state $conflicting_name;
17   }
18 }
```

Abbildung 6.8.: Backtracking beim Verschieben von Zuständen

Die Methode `pullUpStateIfPossible` ruft zunächst die Transformationsregel `pullUpState` auf. Diese verschiebt den als Parameter gegeben Zustand auf die oberste Ebene des Statecharts. Anschließend wird die seiteneffektfreie Regel `namingConflict` aufgerufen. Diese sucht auf oberster Ebene des Statecharts nach zwei Zuständen mit identischem Namen.

Da das Ergebnis des Aufrufs in Zeile 3 negiert wird, schägt die Anweisung genau dann fehl, wenn es einen Namenskonflikt gibt. In diesem Fall wird das Backtracking gestartet. Dieses besteht aus Kontrollfluss- und Datenflussbacktracking.

Im Kontrollfluss muss das Programm an die Stelle der letzten nichtdeterministischen Auswahl zurückspringen. Dies ist die Auswahl eines Matches durch die Transformationsregel `pullUpState`. Das Datenflussbacktracking muss alle Änderungen, die seit dieser Stelle am Modell vorgenommen wurden, rückgängig machen. Für Transformationsregeln kann dies durch

die `undoReplacement`-Methode, die durch das Verwendung des Entwurfsmusters Command verfügbar ist (vgl. Abschnitt 4.5), erfolgen. Für andere Anweisungen kann dies teilweise automatisch geschehen, etwa indem bei Variablenzuweisungen der alte Wert gespeichert wird. Für andere Anweisungen, insbesondere Aufrufe kompilierter Java-Methoden, muss eine explizite `undo`-Annotation angegeben werden, da deren Verhalten nicht automatisch analysiert werden kann. Da die Einführung von `undo`-Annotationen für die in dieser Arbeit als Beispiel verwendete Vereinfachung hierarchischer Statecharts nicht erforderlich ist, wird für Details hierzu auf [Sch11] verwiesen.

Die Angabe von `undo`-Annotationen ist syntaktisch optional. Jedoch schlägt die Regelausführung zur Laufzeit fehl, wenn das Backtracking an eine Stelle gerät, die nicht rückgängig gemacht werden kann.

In der Beispielregel wird vom Datenflussbacktracking der verschobene Zustand wieder an seine ursprüngliche Position verschoben. Anschließend wird versucht, ein weiteres Match für die Regel `pullUpState` zu finden. Da der Zustand jedoch in einem Parameter übergeben wurde und somit beim Matching der Regel schon vorgegeben ist, und weil nur ein einziges Statechart in einem AST existiert, kann es in diesem Fall kein weiteres Match geben. Somit kann auch die Methode `pullUpStateIfPossible` nicht erfolgreich ausgeführt werden.

Der Unterschied zu einer deterministischen Ausführung ist in diesem Fall also lediglich, dass die Änderungen am Modell automatisch rückgängig gemacht wurden. Im Allgemeinen kann der Nichtdeterminismus jedoch auch zu einer erfolgreichen Anwendung einer Transformation führen, die bei deterministischer Ausführung nicht gefunden worden wäre.

6.4. Codegenerierung

Die Transformationen in der Kontrollflusssprache und die eingebetteten Regeln werden in einem Codegenerierungsschritt nach Java übersetzt und somit ausführbar gemacht. Der generierte Java-Code lässt sich auch in bestehende Werkzeuglandschaften einbinden; wie dies für die Modellverarbeitung mit dem DSLTool-Framework von MontiCore erfolgt, wird in Kapitel 8 beschrieben.

Aufgrund der Spracheinbettung müssen bei der Codegenerierung mehrere Generatoren zum Einsatz kommen. Im einfachsten Fall liegen die eingebetteten Regeln in der Objektdiagramm-Notation vor. In dem Fall besteht die Codegenerierung aus der Übersetzung des Kontrollflussanteils nach Java und der Codegenerierung für die Regeln in Objektdiagramm-Notation. Werden dagegen Regeln in konkreter Syntax verwendet, so müssen diese zusätzlich in die Objektdiagramm-Notation übersetzt werden.

6.4.1. Struktur des generierten Codes

Aus einem Modul der Kontrollflusssprache wird bei der Codegenerierung genau eine Java-Datei erstellt. Diese enthält eine öffentliche Klasse für das Modul. Die aus dem Modul generierte Klasse enthält wiederum die aus den Regeln generierten Klassen als geschachtelte Klassen. Diese geschachtelten Klassen entsprechen in jedem Fall der in den Abschnitten 4.3 und 4.5

beschriebenen Struktur, da auch der Java-Code für Regeln in konkreter Syntax letztlich vom Codegenerator für Transformationsregeln in Objektdiagramm-Notation erzeugt werden.

Ein Ausschnitt aus der Struktur der generierten Java-Klasse zum Modul `SimplifyStatecharts` ist in Abbildung 6.9 dargestellt. Wie in Kapitel 4 bereits beschrieben wurde, enthalten die aus den Regeln generierten Klassen wiederum die geschachtelte Klasse Match, so dass sich insgesamt die gezeigte Struktur ergibt.

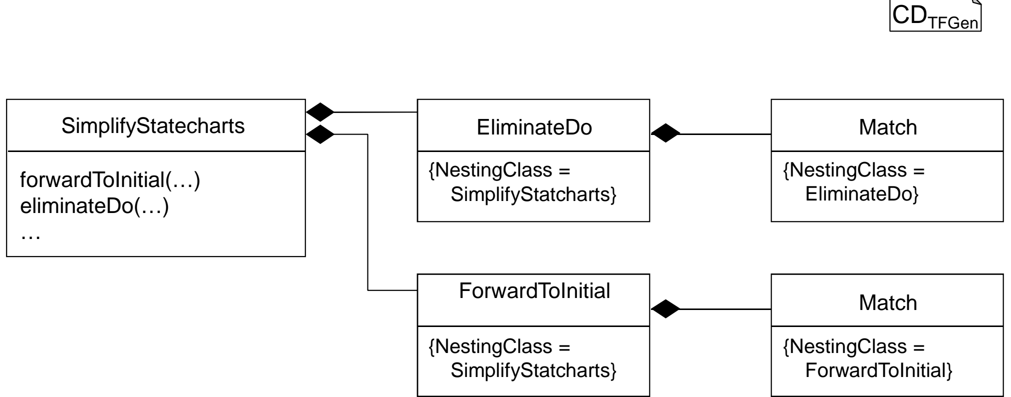


Abbildung 6.9.: Struktur der generierten Klassen zu den Modulen der Kontrollflusssprache

6.4.2. Ablauf der Codegenerierung

Zur Generierung der Klassen kommt die Templatesprache FreeMarker [Fre11] zum Einsatz. Eine Methodik und ein Framework zur Verwendung der Template-Engine werden in MontiCore durch die MontiCore Generation and Language Infrastructure (MontiCoreGLI) [Sch12] vorgegeben.

Durch den templatebasierten Ansatz werden die Java-Klassen in einer serialisierten Form erstellt, sodass es nötig ist, für die Einbettung der geschachtelten Klassen zwischen den Codegeneratoren für den Kontrollflussanteil und für die Regeln in Objektdiagramm-Notation zu wechseln.

Für die Regeln in konkreter Syntax gibt es dagegen zwei Möglichkeiten der Codegenerierung: Erstens können vor der Codegenerierung alle Regeln in konkreter Syntax in die Objektdiagramm-Notation transformiert werden. Zweitens kann hierauf verzichtet werden und beim Erreichen einer Regeldefinition in konkreter Syntax eine zweistufiger Codegenerator aufgerufen werden, der die Regel erst in Objektdiagramm-Notation und dann in Java übersetzt. Wegen des geringeren technischen Aufwandes wurde in dieser Arbeit der zweite Ansatz gewählt.

Der Unterschied zwischen den beiden Codegenerierungsprozessen ist in Abbildung 6.10 für die Verarbeitung eines Transformationsmoduls `m` dargestellt. Der obere Teil der Abbildung zeigt einen Workflow, in dem vor der Codegenerierung die eingebettete Regel `r` in eine äquivalenten Regel in Objektdiagramm-Notation `r'` transformiert wird. Im Workflow, der im unteren Teil dargestellt ist, erfolgt diese Transformation erst während der eigentlichen Codegenerierung. Der Nachteil des Ersetzens vor der Generierung besteht darin, dass nicht nur das Objektdiagramm in

den Quell-AST eingehängt werden muss, sondern auch die abgeleiteten Informationen, etwa aus Attributgrammatiken oder Symboltabellen, transformiert oder neu berechnet werden müssen, was vergleichsweise hohen Implementierungsaufwand erfordert.

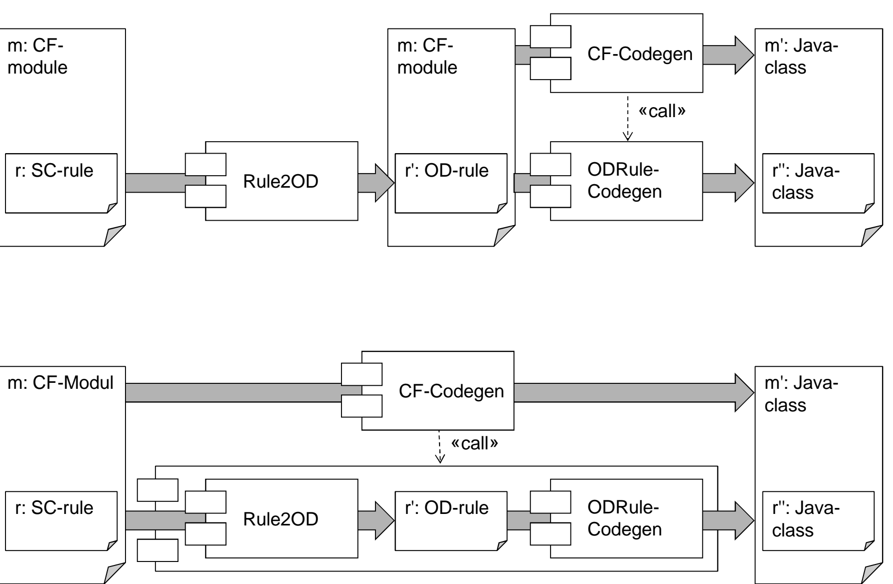


Abbildung 6.10.: Codegenerierung mit Transformation der Regeln vor Generierung aus dem Kontrollfluss (oben) und mit zweistufiger Generierung aus den Regeln (unten)

Technisch ist die Kombination der Codegeneratoren über Sprachvererbung gelöst [Völ11], über die sich auch die Anpassungen des Parsers und der statischen Analyse lösen lassen. Die Basissprache `CFLanguage` besteht dabei aus der Kontrollflusssprache mit eingebetteten Regeln in Objektdiagramm-Notation. Die Untersprache `StatechartTransformationLanguage` erweitert diese Sprache um eigene Workflows, insbesondere zur Codegenerierung, aus denen unter anderem der in Abschnitt 5.4.3 beschriebene Visitor zur Übersetzung in Objektdiagramm-Notation und die Klasse `FreeMarkerTemplateEngine` zur Generierung des Java-Codes aufgerufen werden.

Die Abbildungen 6.11 und 6.12 zeigen Struktur und Laufzeitverhalten des Codegenerators für Kontrollflussmodule mit eingebetteten Regeln in Statechart-Syntax. Im Klassendiagramm ist zu sehen, dass die Klasse `StatechartTransformationGenerationWorkflow` als Erweiterung der Standardklasse `CodegenWorkflow` eingesetzt wird. Ein solcher Workflow verwendet eine speziellen Visitor. Wie im Sequenzdiagramm dargestellt ist[1], nimmt dieser Visitor beim Erreichen einer Regelmethode zunächst die Übersetzung in ein Objektdiagramm vor und generiert aus diesem dann Java-Code. Der generierte Java-Code wird im aus Platzgründen nicht abgebildeten `ControlflowGenerationStorage`-Objekt zwischengespeichert und kann von dort aus durch die Template-Engine in die generierten Dateien eingefügt werden.

[1] Konstruktoraufrufe, Fabriken und weitere `visit`-Methoden wurden hier aus Platzgründen nicht abgebildet.

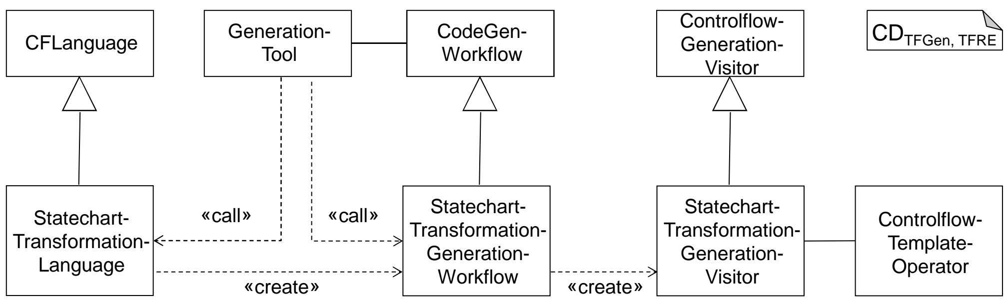


Abbildung 6.11.: Struktur des Codegenerators für Module mit eingebetteten Statechart-Regeln

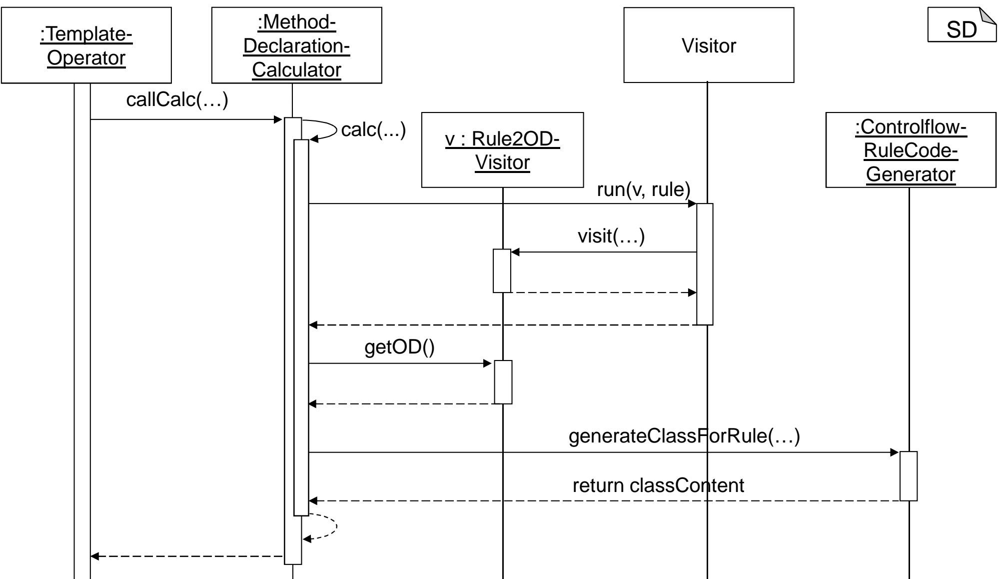


Abbildung 6.12.: Ablauf der Codegenerierung für Module mit eingebetteten Statechart-Regeln

6.4.3. Abbildung des Nichtdeterminismus

Die Umsetzung des Nichtdeterminismus bei der Codegenerierung erfolgt über die Abbildung von Wahlpunkten, an denen nichtdeterministische Entscheidungen simuliert werden, in den generierten Code. Für jeden dieser Wahlpunkte wird festgehalten, für welche Auswahl an dieser Stelle der weitere Programmablauf bereits ausgeführt wurde.

Abbildung 6.13 zeigt Teile des Codes, der aus dem Rumpf der Methode `pullUpStateIfPossible` generiert wurde. Für diese Liste von Anweisungen wird im Generat eine `while`-Schleife erzeugt, die so lange wiederholt wird, bis eine erfolgreiche Ausführung gefunden wurde oder alle möglichen Wahlpunkte erschöpft sind. Die nächste auszuführende Anweisung wird durch den Wert der Variablen `statement35_35` bestimmt, wobei der Wert 0 für ein aus technischen Gründen eingeführtes Dummy-Statement steht und alle weiteren $n \in \mathbb{N}$ jeweils für die n-te Anweisung. Der Index `35_35` ergibt sich dabei aus der Position des Statements in der Quellcode-Datei und stellt die Eindeutigkeit der Variablennamen sicher, wenn solche Variablen für mehrere Listen von Anweisungen eingeführt werden müssen.

Im einfachsten Fall, nämlich der erfolgreichen Ausführung jeder Anweisung im ersten Versuch, wird die `while`-Schleife nur einmal durchlaufen. Beispielsweise wird der Methodenaufruf von `pullUpState` in Zeile 14 ausgeführt. Ist dieser erfolgreich, so wird in Zeile 25 der Zähler `statement35_35` erhöht, so dass die folgende `case`-Anweisung (im Beispiel bereits `default`) betreten wird. Am Ende der letzten Anweisung wird im Fall einer erfolgreichen Ausführung mit dem `break while35_35;` in Zeile 31 die `while-Schleife` verlassen.

Schlägt innerhalb der Schleife eine Anweisung fehl, so wird die Variable `btInfo` auf den Wert `NEW_CHOICE` gesetzt. Dies signalisiert, dass Backtracking erforderlich ist. Durch das Dekrementieren der Variablen `statement35_35` in Zeile 43 wird zur vorherigen Anweisung zurückgesprungen. Dies wird so lange wiederholt, bis eine Anweisung eine neue Alternative erfolgreich ausführt. Danach wird `btInfo` wieder auf `EXEC` gesetzt, und die Anweisungen laufen wieder vorwärts ab.

Sollte keine erfolgreiche Ausführung der Anweisungen möglich sein, so wird die Variable `statement35_35` nach dem vollständigen Durchlauf des Suchbaums den Wert 0 haben. Dadurch wird die `while`-Schleife abgebrochen, und das Backtracking springt zu den Wahlpunkten zurück, die vor dem Aufruf der Methode `pullUpStateIfPossible` liegen.

6.5. Verwandte Arbeiten

In den verschiedenen Werkzeugen zur Modelltransformation werden sehr unterschiedliche Mechanismen zur Auswahl der Transformationsregeln und ihrer Anwendungsstellen eingesetzt.

Im Bereich der Modell-zu-Modell-Transformationen gibt es rein deklarative Sprachen, wie etwa die Tripel-Graph-Grammatiken [Sch94, KW07, Kön09], BOTL [MB03] oder QVT relational [OMG08b], die gänzlich ohne Kontrollstrukturen auskommen. Dabei kann ein objektgetriebener Scheduling-Ansatz verfolgt werden, bei dem die als Nächstes anzuwendende Regel implizit durch Angabe eines zu transformierenden Objekts bestimmt wird.

Der Unterschied zum in dieser Arbeit verwendeten regelbasierten Scheduling ist in Abbildung 6.14 dargestellt. Für In-Place-Transformationen bietet sich die Verwendung eines regelba-

Generierter Java-Code

```
int statement35_35 = (btInfo == BacktrackInfo.EXEC) ? 0 : -1;

while35_35 : while (true) {
  switch (statement35_35) {
    case 0:
      if (btInfo == BacktrackInfo.EXEC) {
        backtrackStack.push(new DummyRule());
      }
      else {
        break;
      }
      ++statement35_35;
    case 1:
      if (pullUpState(btInfo, $s)) {
        btInfo = BacktrackInfo.EXEC;
      }
      else {
        btInfo = BacktrackInfo.NEW_CHOICE;
      }
      ;

      if (btInfo == BacktrackInfo.NEW_CHOICE) {
        break;
      }
      ++statement35_35;

    default:
      ...
      ++statement35_35;

      break while35_35;
  }
  if (btInfo==BacktrackInfo.NEW_CHOICE) {
    if (statement35_35 == 0) {
      break while35_35;
    }
    else {
      if (wasCommitedBefore()) {
        throw new BacktrackCommitException();
      }
      ODRule backtrackStatement = backtrackStack.pop();
      backtrackStatement.undoReplacement();
      --statement35_35;
    }
  }
}

return (btInfo==BacktrackInfo.EXEC);
```

Abbildung 6.13.: Generierter Code zum Methodenrumpf von `pullUpStateIfPossible`

sierten Ansatzes an, da in vielen Situationen mehrere Regeln anwendbar sind. Weil die Ausführung der Regeln im Allgemeinen nicht konfluent ist, und weil eine beliebige Auswahl zu einem unerwünschten Ergebnis führen kann, wird für die In-Place-Transformationen in MontiCore ein regelbasierter Ansatz verwendet.

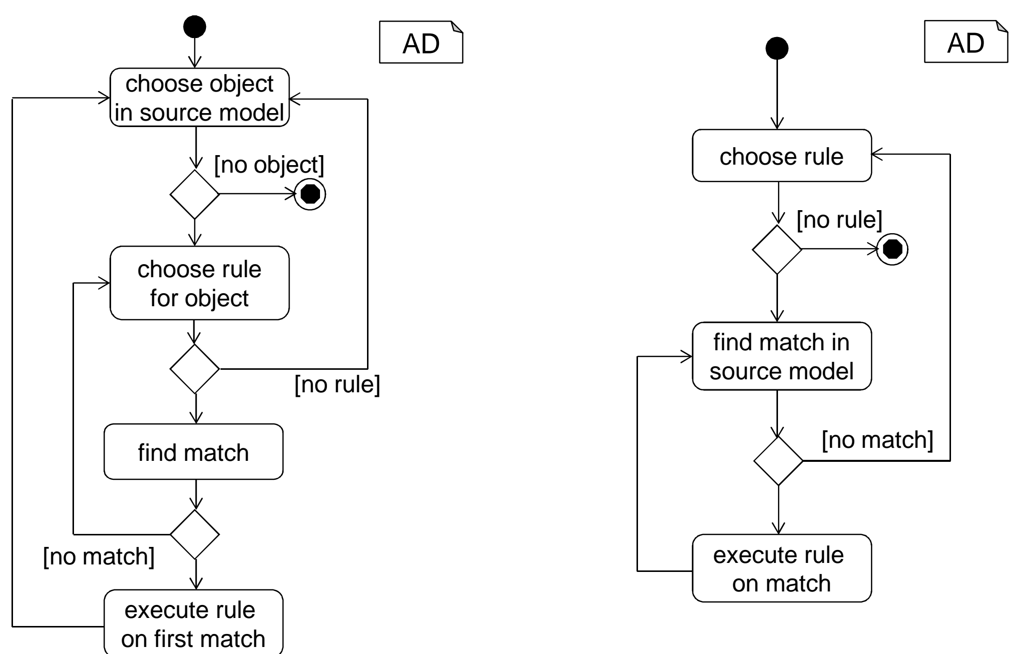


Abbildung 6.14.: Objekt- (links) und regelbasiertes (rechts) Scheduling

Die Motivation, die Auswahl von Regeln durch Bedingungen weiter zu verfeinern, ergibt sich aus den Anwendungen. So kann es für Refactorings in einem Werkzeug sinnvoll sein, eine Regel nicht wie in Abbildung 6.14 gezeigt auf alle möglichen Matches anzuwenden, sondern nur auf ausgewählte Stellen des Modells.

Die Umsetzung solcher Bedingungen durch Sprunganweisungen ist zwar für grafische Kontrollflusssprachen verbreitet, auch hier gibt es allerdings Ansätze zur Einführung von Kontrollstrukturen [KCS05]. In textuellen Sprachen führen Sprunganweisungen zu schlecht wartbarem Code, sodass die Einführung von Kontrollstrukturen praktisch unumgänglich ist. So finden sich Kontrollstrukturen, die mit den in diesem Kapitel vorgestellten Konzepten vergleichbar sind, beispielsweise auch in PROGRES [Zün92] und in QVT operational [OMG08b].

Die Simulation nichtdeterministischen Verhaltens ist vergleichsweise selten anzutreffen. Die in diesem Kapitel vorgestellte Lösung orientiert sich in weiten Teilen an der Implementierung in PROGRES [SZ92]. Jedoch werden nicht alle hier vorgestellten Konzepte unterstützt, beispielsweise fehlen boolesche Operatoren mit nichtdeterministischem Verhalten. In der Implementierung konnten im Vergleich zu PROGRES die Möglichkeiten der Objektorientierung genutzt werden, sodass Regeln bei mehrfacher Ausführung einfach instanziiert werden können. Auch

das Entwurfsmuster Command ist dem erst später erschienenen Gang-of-Four-Buch [GHJV95] entnommen.

Die Kontrollflusssprache für Transformationen in MontiCore übernimmt somit die wichtigsten Konzepte gängiger Modelltransformationssprachen. Ein besonderes, wenn auch nicht einzigartiges Feature stellt die Simulation nichtdeterministischen Verhaltens dar, wobei hier im Vergleich zu Vorarbeiten von den Konzepten moderner Programiersprachen und von Entwurfsmustern profitiert werden konnte.

Kapitel 7.

Generierung von Transformationssprachen aus DSL-Grammatiken

Die Kombination der in Kapitel 5 vorgestellten Regelsprache und der in Kapitel 6 beschriebenen Kontrollflusssprache erlaubt es, auch komplexe Transformationen auf Statecharts zu beschreiben. Die Verständlichkeit der Transformationsprogramme für Domänenexperten ist dabei gegenüber einer generischen Transformationssprache, wie etwa der in Kapitel 4 vorgestellten Regelsprache, deutlich verbessert.

Die Konzepte, die der Entwicklung der Regelsprache für Statecharts und der in Abschnitt 6.2 beschriebenen Einbettung in den Kontrollfluss zugrunde liegen, sind auch auf andere DSLs als Statecharts anwendbar. Die Entwicklung einer speziellen Transformationssprache zu jeder Modellierungssprache bedeutet jedoch für den Sprachentwickler einen beträchtlichen zusätzlichen Aufwand.

In diesem Kapitel wird daher – wiederum am Beispiel der Statechartsprache – vorgestellt, wie sich domänenspezifische Transformationssprachen vollautomatisiert aus der Beschreibung einer Sprache in MontiCore ableiten lassen. Nach einer kurzen Vorstellung des Szenarios, in dem diese Sprachgenerierung angewendet werden kann, wird zunächst die Generierung der Regelsprache aus der Grammatik und weiteren Artefakten der Modellierungssprache erläutert. Dies betrifft sowohl die Syntax der Regelsprache als auch den Codegenerator für diese Sprache, der die Regeln in die bekannte Objektdiagramm-Notation transformiert. Per Spracheinbettung wird die Regelsprache in die Kontrollflusssprache aus Kapitel 6 eingebettet, wobei auch die Konfiguration der Einbettung und der erforderliche Glue Code generiert werden.

7.1. Anwendungsszenario

Das Szenario der Erstellung und des Einsatzes einer generierten Transformationssprache ist in Abbildung 7.1 dargestellt. Der Stereotyp «`gen`» steht hier sowohl für generische als auch für generierte Komponenten, da diese gleichermaßen ohne zusätzlichen Aufwand für den Sprachentwickler im gezeigten Szenario eingesetzt werden können.

Neben den aus [Kra10] bekannten Artefakten wie Parser, Workflows und den Klassen der abstrakten Syntax lassen sich aus der Grammatik einer DSL, im Beispiel der Statechartgrammatik, auch die Grammatik und der Codegenerator der Transformationssprache generieren. Da die-

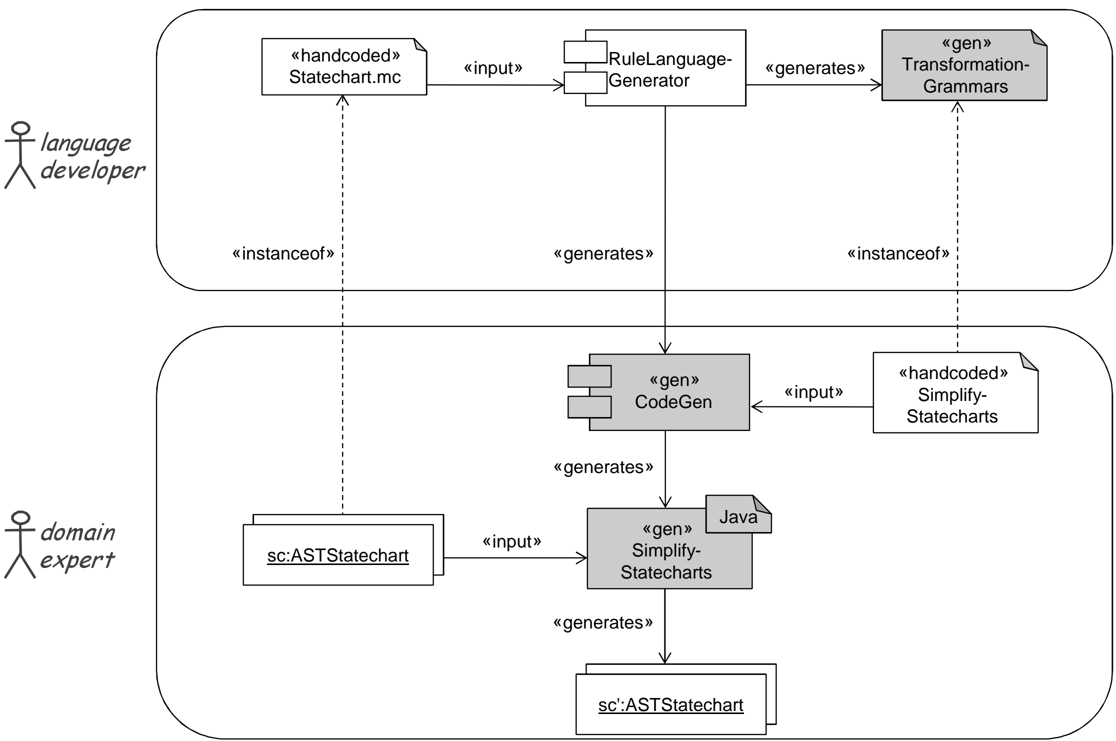


Abbildung 7.1.: Erstellung und Einsatz einer generierten Transformationssprache

se Artefakte automatisch von der Komponente `RuleLanguageGenerator` erstellt werden, entsteht für den Sprachentwickler hierdurch kein zusätzlicher Aufwand.

Ein Domänenexperte oder Modellierer kann jetzt Transformationen in dieser Sprache schreiben, im Beispiel die Transformation `SimplifyStatecharts`. Der generierte Codegenerator kann diese Transformationen in Java-Code übersetzen. Dieser kann auf Statechartmodellen im Hauptspeicher, also beispielsweise geparsten Modellen, ausgeführt werden. Die resultierenden transformierten Modelle, im Beispiel also die vereinfachten Statecharts, können dann beliebig weiterverarbeitet werden, etwa durch Codegenerierung oder Serialisierung.

7.2. Generierung der Regelgrammatik

Aus der Grammatik der Transformationsregeln werden unter anderem der Lexer, der Parser und die Klassen der abstrakten Syntax generiert. Jede Transformationsregel in konkreter Syntax muss zu ihrer entsprechenden Grammatik konform, also (kontextfrei) syntaktisch korrekt sein, um in weiteren Schritten wie der Codegenerierung verarbeitet werden zu können.

7.2.1. Inhalt der Regelgrammatiken

Da die Transformationsregeln in konkreter Syntax den zu transformierenden Modellen syntaktisch ähneln, weisen auch die Grammatiken viele Ähnlichkeiten auf. Aus den transformationsspezifischen Konstrukten wie Ersetzungen oder negativen und Listenknoten sowie der Möglichkeit, auch Objekte in abstrakter Syntax oder der in Abschnitt 5.1.1 eingeführten Mischform anzugeben, ergibt sich jedoch die Notwendigkeit einiger Änderungen. Auch technische Anpassungen wie ein erhöhter Lookahead des Parsers werden in die Grammatik der Transformationsregeln generiert.

Die Produktionsregeln einer EntryAction in einem Statechart und eines EntryAction-Objekts in einer Transformationsregel sind in Abbildung 7.2 gegenübergestellt. Solche konkreten Produktionen in MontiCore-Grammatiken werden als *Klassenproduktionen* bezeichnet. Die Produktion `EntryAction_Pattern` implementiert zunächst zwei Schnittstellen. Zum einen ist dies die Schnittstellenproduktion `EntryAction`, wodurch sichergestellt ist, dass ein `EntryAction_Pattern` in Transformationsregeln überall dort verwendet werden kann, wo auch in der Syntax der Statechartsprache eine `EntryAction` erlaubt ist. Zum anderen ist dies die externe Schnittstelle `IPattern`, die das Objekt als Muster kennzeichnet und so auch für generische Komponenten unterscheidbar von Listen- und negativen Knoten sowie Ersetzungen macht.

```
MontiCore-Grammatik
1 EntryAction= "entry" ":" Block:BlockStatement;
```

```
MontiCore-Grammatik
1 EntryAction_Pattern implements EntryAction
2     astimplements /mc.tfcs.ast.IPattern =
3   ("entry" ":" Block:BlockStatement)
4   | "EntryAction" SchemaVarName:IDENTVAR?
5     "[[" ("entry" ":" Block:BlockStatement) "]]"
6   | "EntryAction" SchemaVarName:IDENTVAR ";"
7 ;
```

Abbildung 7.2.: Vergleich einer Produktion aus der Statechartgrammatik (oben) mit einer generierten Produktion aus der Grammatik der Transformationsregeln (unten)

Betrachtet man die Produktion im unteren Teil genauer, so lässt sich zum einen feststellen, dass die drei Alternativen auf der rechten Regelseite genau die drei möglichen Notationen für Entry-Aktionen beschreiben (vgl. Abschnitt 5.1). Zum anderen sind alle Bestandteile dieser Transformationsregel entweder für Muster in Transformationsregeln spezifisch oder aus der Produktion in der Statechartgrammatik ableitbar. Auch wenn in dieser Regel nicht alle Möglichkeiten von MontiCore-Grammatiken ausgeschöpft wurden, so veranschaulicht dieses Beispiel doch, dass sich für Produktionsregeln in der Grammatik einer Modellierungssprache passende Regeln für Muster in den dazugehörigen Transformationsregeln generieren lassen.

Die linke Seite der Produktion in der Regelsprache besteht aus dem Namen der Produktion und den implementierten Schnittstellen. Der Name ergibt sich aus dem Namen der Produktion

in der DSL und dem Zusatz _Pattern, der Name der ersten Schnittstelle ist genau der Name der Produktion in der DSL, und die zweite Schnittstelle sowie alle weiteren Bestandteile sind für alle aus Klassenproduktionen abgeleiteten Produktionen von Mustern gleich.

Die Alternative in Zeile 3 beschreibt ein Muster in konkreter Syntax. Sie ist eine exakte Kopie der rechten Regelseite aus der DSL-Grammatik. Allerdings verweist das Nichtterminal BlockStatement in der Regelgrammatik nicht mehr auf eine Klassenproduktion aus der DSL-Grammatik, sondern auf die entsprechende Schnittstellenproduktion der Regelgrammatik.

Die Alternative, die in den Zeilen 4 und 5 abgebildet ist, beschreibt die Mischform aus konkreter und abstrakter Syntax. Das Schlüsselwort EntryAction hierfür ergibt sich aus dem Regelnamen in der DSL-Grammatik. Ihm folgt optional ein Objektbezeichner, der für alle Muster-Produktionen gleich ist. In den doppelten eckigen Klammern folgt wiederum eine Kopie der rechten Seite der DSL-Grammatik.

Die dritte Alternative ist in Zeile 6 spezifiziert. Sie entspricht zunächst der zweiten Alternative, nur wird anstatt des Objekts in den doppelten eckigen Klammern ein Semikolon eingefügt.

Abbildung 7.3 zeigt die Produktion der Ersezungsregeln für EntryActions. Für die linke Seite der Produktion kann die Zusammensetzung der linken Seite analog zu der unteren Regel in Abbildung 7.2 erfolgen. Die rechte Seite der Produktion enthält als konstante Anteile die doppelten eckigen Klammern und die Ersetzungsoperatoren. Variabel sind hier nur die Typen der Nichtterminale der linken und rechten Regelseite. Diese können offensichtlich aber leicht aus der Grammatik der DSL abgeleitet werden.

MontiCore-Grammatik

```
1 EntryAction_Replacement implements EntryAction
2     astimplements /mc.tfcs.ast.IReplacement =
3   ("[[" lhs:EntryAction? ":-" rhs:EntryAction? "]]") ;
```

Abbildung 7.3.: Die Produktion zu Ersetzungen von EntryActions

Auch die Produktionen für negative Knoten und Listenknoten können, wie aus der Regel in Abbildung 7.4 zu sehen ist, leicht generiert werden. Der Name der Regel, die implementierte Schnittstellenproduktion, das Typargument der Liste und das Nichtterminal in Zeile 4 ergeben sich aus dem Namen entsprechenden Regel in der DSL. Die weiteren Elemente sind für alle Produktionen zu Listen in Transformationssprachen identisch.

MontiCore-Grammatik

```
1 EntryAction_List implements EntryAction
2     astimplements /mc.tfcs.ast.IList =
3   "list" ("<" "EntryAction" ">")? SchemaVarName:IDENTVAR?
4   "[[" EntryAction "]]";
```

Abbildung 7.4.: Die Produktion zu Listen von EntryActions

Abbildung 7.5 fasst die Produktionen für die Schnittstelle, das Pattern, negative und Listenknoten sowie Ersetzungen zusammen, die für eine Produktion A aus der Basis-DSL in der Regelsprache generiert werden. Der Kommentar /* Copy of original syntax */ kenn-

zeichnet die Stellen, an denen die rechte Seite der Produktion A aus der Basis-DSL eingefügt wird.

MontiCore-Grammatik

```
1  interface A astextends /mc.tfcs.ast.ITFElement;
2
3  A_Pattern implements A astimplements /mc.tfcs.ast.IPattern =
4      (/* Copy of original syntax */)
5    | "A" SchemaVarName:IDENTVAR? "[[" (/* Copy of original syntax */) "]]"
6    | "A" SchemaVarName:IDENTVAR ";";
7
8  A_Replacement implements A astimplements /mc.tfcs.ast.IReplacement =
9    ("[[" lhs:A? ":-" rhs:A? "]]");
10
11 A_Negation implements A astimplements /mc.tfcs.ast.INegation =
12   "not" ("<" "A" ">")? SchemaVarName:IDENTVAR? "[[" A "]]";
13
14 A_List implements A astimplements /mc.tfcs.ast.IList =
15   "list" ("<" "A" ">")? SchemaVarName:IDENTVAR? "[[" A "]]";
```

Abbildung 7.5.: Zu einem Nichtterminal A der Basissprache generierte Produktionen

Eine zusätzliche Produktion in der Grammatik für Transformationsregeln ergibt sich aus der Möglichkeit, mehrere Objekte auf oberster Ebene in einer Regel spezifizieren zu können. Während in einem Modell genau ein Knoten auf oberster Ebene existiert, in der betrachteten Sprache etwa ein Nichtterminal vom Typ Statechart, gilt dies nicht für Transformationsregeln. So enthält beispielsweise die Regel in Abbildung 5.1 auf Seite 66 zwei Zustände und eine Transition, für die kein Vaterknoten im Modell angegeben ist.

Um solche Regeln parsen zu können, wird das in Abbildung 7.6 gezeigte Nichtterminal TfObjects eingeführt. Die entsprechende Regel wird beim Parsen von Transformationsregeln als Startsymbol verwendet. Sie produziert zu einer Liste von Objekten, wobei der Typ jedes Objekts beliebig aus den Typen der DSL gewählt werden kann. Zusätzlich enthält die rechte Seite dieser Regel auch den optionalen booleschen Ausdruck, der eine zusätzlichen Bedingung für die Anwendbarkeit der Regel darstellt. Da sich die Liste der Objekte leicht aus den Nichtterminalen der DSL-Grammatik ableiten lässt und alle weitere Elemente der Produktion unveränderlich sind, kann offensichtlich auch diese Regel generiert werden.

Der Generator der Grammatiken für Transformationssprachen erzeugt auch Produktionen, die es erlauben Muster, Ersetzungen und Negationen von Attributwerten anzugeben. Für diese in Abschnitt 5.2.2 vorgestellten Regeln zur Angabe von Attributwerten in Transformationsregeln lässt sich ebenfalls eine Unterscheidung in feste und variable Anteile finden, und die entsprechenden Produktionen in der Regelgrammatik lassen sich aus den Informationen in der DSL-Grammatik generieren.

Abbildung 7.7 zeigt die resultierenden Produktionen für ein boolesches Attribut B. Auch hier kennzeichnet der Kommentar /* Copy of original syntax */ in Zeile 11 die Stelle, an der die Original-Syntax des Attributs B aus der Basis-DSL eingefügt wird.

MontiCore-Grammatik

```
1  TfObjects = "{"
2    ((Statechart_Pattern | Statechart_List | Statechart_Replacement)
3      => Statechart
4    | (EntryAction_Pattern | EntryAction_List | EntryAction_Replacement)
5      => EntryAction
6    | (ExitAction_Pattern | ExitAction_List | ExitAction_Replacement)
7      => ExitAction
8    | (DoAction_Pattern | DoAction_List | DoAction_Replacement)
9      => DoAction
10   | ... )*
11   ("where" "{" Constraint:BooleanExpression "}")? "}"
12 ;
```

Abbildung 7.6.: Die Produktion zu Listen von `EntryActions`

MontiCore-Grammatik

```
1  interface B;
2
3  B_Negation implements B astimplements /mc.tfcs.ast.IAttributeNegation =
4    "not" "[[" B_Pattern "]]";
5
6  B_Replacement implements B
7      astimplements /mc.tfcs.ast.IAttributeReplacement =
8    ("[[" lhs:B_Pattern ":-" "]]") | ("[[" ":-" rhs:B_Pattern "]]");
9
10 B_Pattern implements B astimplements /mc.tfcs.ast.IAttributePattern =
11   B:[/* Copy of original syntax */];
```

Abbildung 7.7.: Zu einem booleschen Attribut `B` der Basissprache generierte Produktionen

7.2.2. Technische Umsetzung

Für die technische Umsetzung der Generierung von Grammatiken für Transformationssprachen sind wie für andere Codegenerierungen auch die beiden Ansätze der templatebasierten und der API-basierten Codegenerierung möglich.

Während bei der Generierung der Regelsprachen und der zugehörigen Werkzeuge in weiten Teilen ein templatebasierter Ansatz zum Einsatz kommt, bietet sich für die Generierung der Grammatiken ein API-basierter Ansatz an. Der Grund hierfür liegt darin, dass die Klassenproduktionen für die Muster, also zu den Unterklassen von `IPattern`, zum Großteil aus Kopien der Produktionen aus der DSL-Grammatik bestehen, die mit einem API-basierten Ansatz leicht erzeugt werden können.

Die API-basierte Generierung erfolgt im Workflow `DSL2TransformationLanguageTransformationWorkflow`, der seinerseits den Visitor `DSL2TransformationLanguageVisitor` verwendet, um die Grammatik der DSL zu traversieren und zu jedem Element die erforderlichen Artefakte in der Grammatik der Regelsprache anzulegen.

Der Ablauf des Workflows ist in Abbildung 7.8 auszugsweise dargestellt. Im Wesentlichen traversiert der konkrete Visitor `v` alle AST-Knoten in der geparsten Grammatik der DSL. Zu jedem Knoten erzeugt er mit Hilfe einer `ProductionFactory` geeignete Produktionen für die Grammatik der Regelsprache, wobei die Erzeugung in der Abbildung vereinfacht dargestellt ist und typischerweise entweder über Fabrikmethoden, über das Kopieren von Knoten der Quellgrammatik oder über das Parsen kleiner Grammatik-Artefakte erfolgt. Anzahl und Typ der erzeugten Knoten können je nach Art des besuchten Knotens variieren, jedoch werden immer nur AST-Knoten für die Grammatik der Regelsprache zurückgegeben. Diese Knoten werden dann vom Visitor an geeigneter Stelle in den AST der generierten Grammatik eingefügt.

Nach der Traversierung des ASTs der DSL kann der Workflow über die Zugriffsmethode `getTfLang` auf den AST der Regelgrammatik zugreifen. Die Grammatik schreibt er mit Hilfe eines nicht in der Abbildung dargestellten Pretty-Printers in eine Ausgabedatei. Dort kann sie weiterverarbeitet werden, etwa durch die MontiCore-Workflows zur Generierung eines Lexers und Parsers und der Klassen der abstrakten Syntax. Diese Artefakte sind auch für die weitere Verarbeitung von Modelltranformationsregeln erforderlich, vor allem für die Übersetzung in Objektdiagramm-Notation durch den Transformator, dessen Generierung im folgenden Abschnitt vorgestellt wird.

7.3. Generierung des Transformators

Um die Regelsprachen wie in Abschnitt 7.1 beschrieben effektiv nutzen zu können, genügt es nicht, ihre Grammatiken aus den zugrunde liegenden DSLs zu generieren. Zusätzlich müssen sie auch ausführbar sein, wozu sich die Abbildung auf eine bereits definierte, ausführbare Sprache eignet.

Diese Abbildung auf eine bereits definierte Sprache ist das semantische Mapping [HR04] der Regelsprachen, und als semantische Domäne, also als die bereits definierte Sprache und Zielmenge dieser Abbildung, wird die in Kapitel 4 beschriebene Sprache für Transformationsregeln in Objektdiagramm-Notation verwendet.

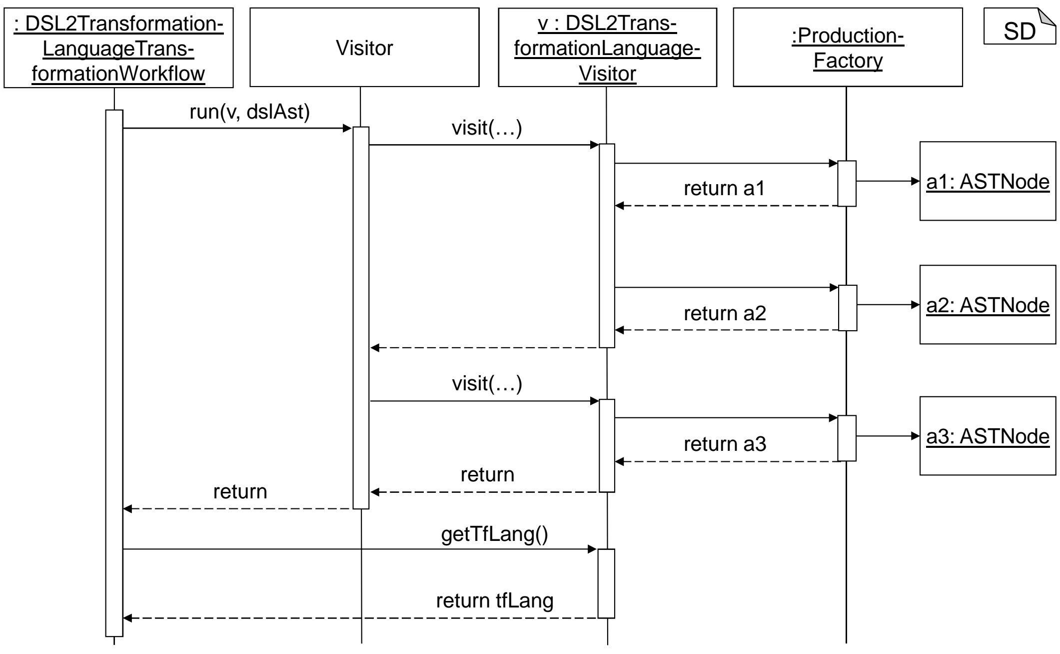


Abbildung 7.8.: Transformation der DSL-Grammatik in die Grammatik der Regelsprache

7.3.1. Semantik der DSLs und Semantik der generierten Regelsprachen

Damit die Definition des semantischen Mappings ohne zusätzlichen Aufwand für den Entwickler der DSL erfolgt, muss diese Abbildung für jede generierte Regelsprache automatisch erstellt werden. Als Quellartefakte dieser Generierung kommen also nur die vorhandenen Artefakte, im Wesentlichen die syntaktische und semantische Definition der Quellsprache in Betracht.

Da die Definition der Semantik in vielen Fällen jedoch nur informell oder in einer nicht automatisiert auswertbaren Form vorliegt, wird die Abbildung der Transformationsregeln in konkreter Syntax auf Regeln in Objektdiagramm-Notation lediglich aus der Syntax der zugrunde liegenden DSL abgeleitet, die in MontiCore-basierten Sprachen in einer Grammatik definiert ist.

Mit dieser Einschränkung auf die Grammatik als Quellartefakt der Generierung des Transformators gehen zwei Eigenschaften der generierten Regelsprachen einher: Erstens ist die Semantik der Regelsprache völlig unabhängig von der Semantik der zugrunde liegenden DSL; dies ist weniger schwerwiegend, denn aus dem Beispiel der Statechartsprache ist bereits intuitiv klar, dass diese beiden Sprachen sehr unterschiedliche Dinge beschreiben. Zweitens ist aber auch die Syntax der Regelsprache unabhängig von der Semantik der DSL. Daraus folgt beispielsweise, dass es nicht möglich ist, die Syntax der Regelsprache auf semantikerhaltende oder -verfeinernde Transformationsregeln einzuschränken. Dies wäre zwar etwa für die Vereinfachung von Statecharts sinnvoll, ist aber mit dem gewählten Ansatz beziehungsweise aufgrund der häufig informellen Beschreibung der Semantik der DSLs konzeptuell nicht möglich.

7.3.2. Arbeitsweise und Aufbau des Transformators

Ein Transformator, der Regeln in konkreter Syntax in die Objektdiagramm-Notation übersetzt, muss im Wesentlichen zu jeder Transformationsregel die Objektdiagramme der linken und rechten Regelseite erzeugen. Zusätzlich muss er auch die `folding`-Blöcke und die zusätzlichen Constraints transformieren. Diese können jedoch aus den Regeln in konkreter Syntax in die Regeln in Objektdiagramm-Notation kopiert werden, sodass der Fokus im Folgenden auf der Generierung der linken und rechten Regelseite liegt.

Die linke und die rechte Regelseite bestehen aus Objektdiagrammen, die isomorph zu einem Teil des ASTs nach dem Einlesen des Hostmodells beziehungsweise nach der Ausführung der Transformationsregel sind. Da die Transformationsregeln den Hostmodellen syntaktisch ähneln, ähneln auch die abstrakten Syntaxbäume der Regel und des Modells einander. Daher kann die Erzeugung eines Objektdiagramms, das einen solchen Ausschnitt aus dem Hostgraphen beschreibt, systematisch erfolgen.

In Abbildung 7.9 ist oben eine einfache Transformationsregel gezeigt, die lediglich eine Mustersuche auf dem Hostgraphen durchführt. Im linken Teil der Abbildung ist ein Ausschnitt des ASTs dieser Transformationsregel nach dem Parsen gezeigt. Der rechte Teil zeigt eine zur oberen Regel äquivalente Regel in Objektdiagramm-Notation.

Der Transformator für Transformationsregeln muss also aus dem AST auf der linken Seite die Regel auf der rechten Seite erzeugen können. Zu beachten ist allerdings, dass es sich beim Objektdiagramm auf der linken Seite um ein Objektdiagramm der abstrakten Syntax der Regelsprache für Statecharttransformationen handelt. Das Objektdiagramm auf der rechten Seite zeigt dagegen die konkrete Syntax der zu generierenden Transformationsregel – wenn auch nicht in der üblichen textuellen, sondern in einer grafischen Notation. Der Transformator operiert dagegen in beiden Sprachen auf der abstrakten Syntax, jedoch zeigt die Grafik die strukturellen Ähnlichkeiten.

Der AST auf der linken Seite enthält zunächst für jedes zu matchende Objekt ein Objekt, entspricht insofern also dem zu erzeugenden Objektdiagramm. Unterschiede gibt es allerdings bei den Attributen: Hier ist für jeden Attributwert im AST auf der linken Seite ein eigenes Objekt angelegt worden. Dieses Vorgehen erlaubt es, den alten und den neuen Attributwert im Falle einer Ersetzung zu speichern. Zusätzliche Objekte werden auch für Ersetzungen, negative und Listenknoten angelegt, die jedoch in der gezeigten Regel nicht vorkommen.

Der zu generierende Transformator ist als Visitor implementiert, der die geparste Regel traversiert. Die genaue Funktionsweise ist am Beispiel der Statechart-Regeln in Abschnitt 5.4 dokumentiert. Im Folgenden wird beschrieben, wie der Quellcode eines solchen Transformators aussieht, und wie er sich aus der Grammatik der Statechartsprache generieren lässt.

7.3.3. Technische Umsetzung

Die Visitoren zur Übersetzung in Objektdiagramm-Notation werden mit der MontiCore Generation and Language Infrastructure [Sch12] und der Templatesprache FreeMarker [Fre11] generiert. Als Zielsprache bei der Codegenerierung wird Java verwendet. Einen Überblick über den Aufbau der generierten Visitor-Klasse gibt Abbildung 7.10.

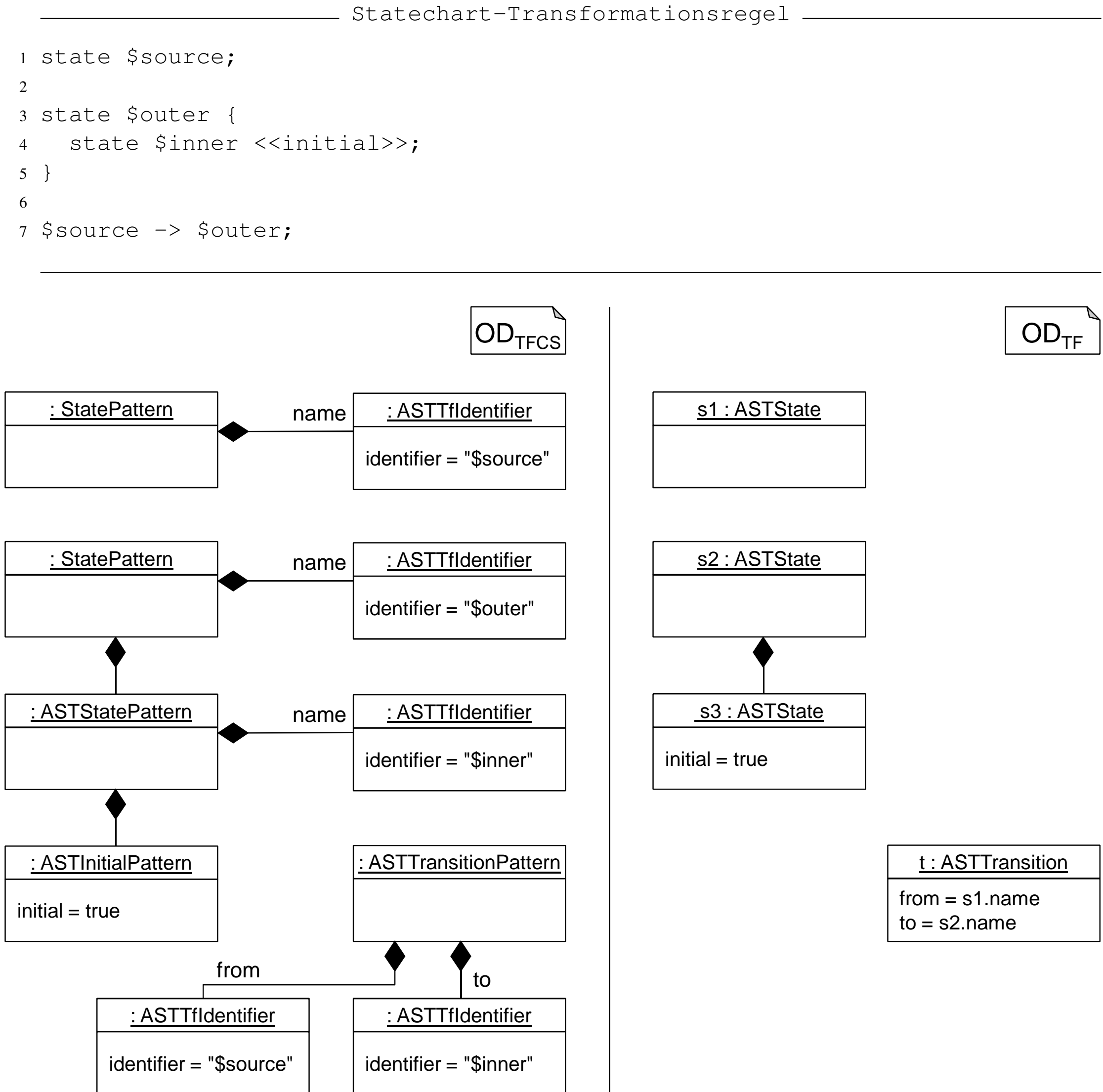


Abbildung 7.9.: Eine Regel ohne Ersetzung (oben), der AST dieser Regel (links) und das zu generierende Objektdiagramm (rechts)

Zunächst wird die Generierung der `visit`-Methode für `ASTState_Patterns` beschrieben, deren erste Zeilen in Abbildung 7.11 abgebildet sind.

Zu Beginn der Methode werden die Objekte für die linke und rechte Regelseite erzeugt (Zeilen 3 und 16). Zum Objektdiagramm hinzugefügt werden sie jedoch erst nach einer Prüfung, ob der Visitor tatsächlich auf einem Element der linken beziehungsweise rechten Regelseite operiert (Zeilen 4, 13, 16 und 18). Dadurch entsteht zwar ein gewisser Overhead, die Transformation für Attribute wird aber um einiges vereinfacht, wenn man dort die entsprechenden Prüfungen auslassen kann. Die übrigen Zeilen dienen entweder der Festelegung des Namens im Objektdiagramm, der gegebenenfalls aus der Regel in konkreter Syntax übernommen wird (Zeile 5) oder

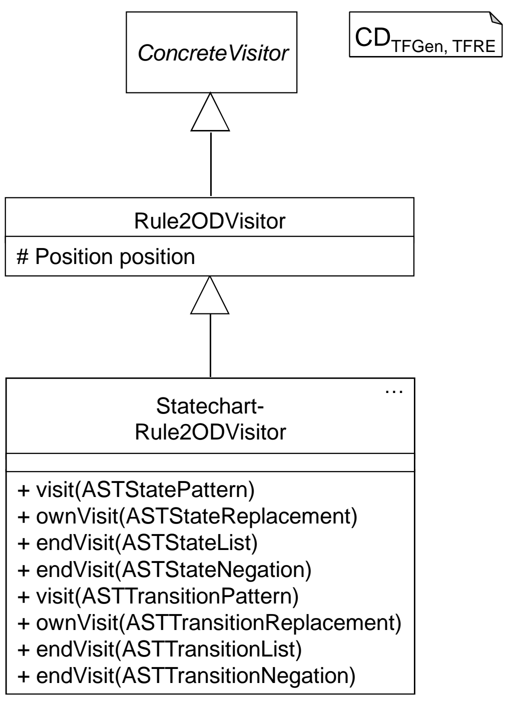


Abbildung 7.10.: Vererbungshierarchie und wichtige Methoden der Klasse `StatechartRule2ODVisitor`

Generierter Java-Code

```
1  public void visit(mc...rule._ast.ASTState_Pattern node) {
2    // create objects
3    ASTODObject o_lhs = ODNodeFactory.createASTODObject();
4    if (position == Position.LHS || position == Position.BOTH) {
5      o_lhs.setName(nameGen.getNameForElement(node));
6      List<String> lhs_qType =
7          NameHelper.createListFromDotSeparatedString(
8          node._getTFElementType().getName());
9      ASTReferenceType lhs_t =
10         TypesNodeFactory.createASTSimpleReferenceType(lhs_qType);
11     o_lhs.setType(lhs_t);
12
13     lhs.getODObjects().add(o_lhs);
14   }
15   ASTODObject o_rhs = ODNodeFactory.createASTODObject();
16   if (position == Position.RHS || position == Position.BOTH) {
17     ...
18     rhs.getODObjects().add(o_rhs);
19   }
20   ...
```

Abbildung 7.11.: Erzeugung von `ASTState_Pattern`-Objekten

der Festlegung des Typs des Objekts (Zeilen 6–11), wobei die Informationen zum Typ aus dem Quell-AST entnommen werden und nicht hart in die visit-Methode generiert werden müssen.

In diesem Ausschnitt ist daher lediglich die Parameterliste in Zeile 1 für die Statechartsprache spezifisch. Alle bisher gezeigten Codeteile des Methodenrumpfes sind für die Transformatoren zu anderen Nichtterminalen oder gar Sprachen identisch. Dies gilt jedoch nicht für die Verarbeitung der Attribute, die als Nächstes betrachtet wird.

Abbildung 7.12 beschreibt die Verarbeitung des Attributs `ASTState_Pattern.initial` durch den Visitor. Dabei werden zunächst der Wert der linken und der rechten Regelseite bestimmt (Zeilen 5–10). Der weitere Verlauf hängt von den konkreten Attributwerten ab (Zeilen 11 und 23). Ist der Modifier beispielweise auf beiden Seiten angegeben (Zeile 11), so wird auf der linken Regelseite ein boolesches Attribut mit dem Namen `initial` und dem Wert `true` angelegt (Zeilen 12–21). So wird der Wert nur für das Pattern-Matching verwendet, bei der Ersetzung wird dieses Attribut aber nicht berücksichtigt.

Generierter Java-Code

```
1  public void visit(mc...rule._ast.ASTState_Pattern node) {
2    ...
3    if (node.getInitial() != null) {
4
5      boolean attr_initial_lhs = node.getInitial().size() > 0 &&
6          ((mc.tfcs.ast.ITFAttribute)node.getInitial().get(0)).getLhs()
7          != null;
8      boolean attr_initial_rhs = node.getInitial().size() > 0 &&
9          ((mc.tfcs.ast.ITFAttribute)node.getInitial().get(0)).getRhs()
10         != null;
11     if (attr_initial_lhs && attr_initial_rhs) {
12       ASTODAttribute od_attr_initial_lhs =
13           ODNodeFactory.createASTODAttribute();
14       od_attr_initial_lhs.setName("initial");
15       od_attr_initial_lhs.setType(
16            TypesNodeFactory.createASTPrimitiveType(
17            ASTConstantsTypes.BOOLEAN));
18       od_attr_initial_lhs.setValue(
19       JavaDSLNodeFactory.createASTBooleanLiteral(
20           ASTConstantsLiterals.TRUE));
21       o_lhs.getODAttributes().add(od_attr_initial_lhs);
22     }
23     if (attr_initial_lhs && !attr_initial_rhs) {
24       ...
25     }
26     ...
27   }
```

Abbildung 7.12.: Verarbeitung des Attributs `initial`

Abbildung 7.13 zeigt einen Ausschnitt des Templates, aus dem der Code in den Zeilen 5–21 von Abbildung 7.12 generiert wurde. Dieses Template ist spezifisch für boolesche Attribute, für Attribute anderer Typen gibt es eigene Templates. Hier ist zu sehen, dass der Name des Attributs an mehreren Stellen als Variable `${name}` und `${Name}` verwendet wird. Des weiteren greift

dieses Template in Zeile 1 auf die Variable `isAttributeIterated` zu. Derartige Variablen werden von Hilfsklassen, den Template-Kalkulatoren, vorberechnet, was schwer verständliche Berechnungen innerhalb der Templates überflüssig macht.

FreeMarker-Template

```
1  <#if isAttributeIterated>
2    boolean attr_${name}_lhs = node.get${Name}().size() > 0 &&
3      ((mc.tfcs.ast.ITFAttribute)node.get${Name}().get(0)).getLhs() != null;
4    boolean attr_${name}_rhs = node.get${Name}().size() > 0 &&
5      ((mc.tfcs.ast.ITFAttribute)node.get${Name}().get(0)).getRhs() != null;
6  <#else>
7    boolean attr_${name}_lhs = node.get${Name}() != null &&
8      ((mc.tfcs.ast.ITFAttribute)node.get${Name}()).getLhs() != null;
9    boolean attr_${name}_rhs = node.get${Name}() != null &&
10     ((mc.tfcs.ast.ITFAttribute)node.get${Name}()).getRhs() != null;
11 </#if>
12   if (attr_${name}_lhs && attr_${name}_rhs) {
13     ASTODAttribute od_attr_${name}_lhs =
14         ODNodeFactory.createASTODAttribute();
15     od_attr_${name}_lhs.setName("${name}");
16     od_attr_${name}_lhs.setType(
17         TypesNodeFactory.createASTPrimitiveType(
18         ASTConstantsTypes.BOOLEAN));
19     od_attr_${name}_lhs.setValue(
20         JavaDSLNodeFactory.createASTBooleanLiteral(
21         ASTConstantsLiterals.TRUE));
22     o_lhs.getODAttributes().add(od_attr_${name}_lhs);
23   }
24   ...
```

Abbildung 7.13.: Template für die Verarbeitung boolescher Attribute

Im Template sind mehrere Fallunterscheidungen zu sehen, durch welche die verschiedenen Aufrufe berücksichtigt sind, die zum Methodenzugriff auf Attributwerte verwendet werden. Einige dieser Unterscheidungen werden bereits zur Codegenerierungszeit durchgeführt, beispielsweise die Unterscheidung, ob das Attribut in der DSL-Grammatik iteriert ist (Zeilen 1, 6 und 11). Andere können erst zur Ausführungszeit des generierten Transformators getroffen werden und sind daher im generierten Java-Code abgebildet, wie etwa die Unterscheidung, auf welchen Seiten der Transformationsregel das Attribut gesetzt ist (Zeile 12).

Für Kompositionen gelten diese Fallunterscheidungen nahezu analog, nur dass hier keine Attribute im Objektdiagramm, also `ASTODAttributes`, sondern Kompositionslinks, also `ASTODLinks`, die als Kompositionen markiert sind, erzeugt werden. Abbildung 7.14 zeigt die Templates zur Codegenerierung für Kompositionen.

Durch das Template im oberen Teil der Abbildung wird zunächst ein Kalkulator aufgerufen (Zeilen 1 und 2), der alle Attribute bestimmt, die Beziehungen zu Kindknoten im AST beschreiben. Hierbei wird unterschieden zwischen iterierten Attributen, die in der Variablen `component_lists` gespeichert werden, und anderen Attributen, die in `component_nodes` ab-

FreeMarker-Template

```
1 <#if op.callCalculator(
2     "mc.tfcs.ruletranslation.CollectCompositionsCalculator")>
3   ${op.includeTemplates(process_component_lists_pattern, component_lists)}
4   ${op.includeTemplates(process_component_nodes_pattern, component_nodes)}
5 </#if>
```

FreeMarker-Template

```
1 <#if op.callCalculator(
2     "mc.tfcs.ruletranslation.ComponentListNameCalculator")>
3   for (${package}.transformation.rule._ast.AST${Type} child:
4       node.get${Name}()) {
5     ASTODLink lhs_composition = ODNodeFactory.createASTODLink();
6     if ((position == Position.LHS || position == Position.BOTH)
7         && child.getLhs() != null) {
8       lhs_composition.setComposition(true);
9       lhs_composition.setRightRole("${name}");
10      List<String> lhs_compositeName =
11          NameHelper.createListFromDotSeparatedString(
12          nameGen.getNameForElement(node));
13      ASTQualifiedName lhs_compositeQName =
14          TypesNodeFactory.createASTQualifiedName(lhs_compositeName);
15      lhs_composition.getLeftReferenceNames().add(lhs_compositeQName);
16      List<String> lhs_componentName =
17          NameHelper.createListFromDotSeparatedString(
18          nameGen.getNameForElement((mc.tfcs.ast.ITFObject)child.getLhs()));
19      ASTQualifiedName lhs_componentQName =
20          TypesNodeFactory.createASTQualifiedName(lhs_componentName);
21      lhs_composition.getRightReferenceNames().add(lhs_componentQName);
22      lhs.getODLinks().add(lhs_composition);
23    }
24    ...
```

Abbildung 7.14.: Zwei Templates für die Verarbeitung von Kompositionen

gelegt sind. Für jedes Attribut wird dann, abhängig davon, in welcher Variable es gespeichert wurde, ein geeignetes Template aufgerufen (Zeilen 3 und 4).

Ein Ausschnitt aus dem Template für iterierte Kompositionsattribute ist im unteren Teil der Abbildung 7.14 dargestellt. Da dieses Attribut auf mehrere Kindknoten verweisen kann, wird über alle Einträge der entsprechenden Liste iteriert (Zeilen 3 und 4). Für jedes Element wird dann ein Link erzeugt (Zeile 5). Das weitere Vorgehen richtet sich wiederum nach dem Vorkommen auf der linken und rechten Regelseite (Zeile 6), wobei im gezeigten Fall zunächst das Kompositionsattribut und der Rollenname gesetzt werden (Zeilen 8 und 9). Anschließend werden die Namen des linken und rechten Objekts am Link bestimmt und gesetzt (Zeilen 10 bis 21) und der Link wird zum Objektdiagramm, im Beispiel dem Diagramm der linken Regelseite, hinzugefügt (Zeile 22).

Weitere Templates zur Generierung von Methoden des Visitors, auf deren Abbildung an dieser Stelle verzichtet wird, betreffen die negativen und Listenknoten sowie die Ersetzungen, also Subklassen von `IReplacement`.

Für die negativen Knoten und für Listenknoten wird eine `endVisit`-Methode generiert, die nach dem Erzeugen der Objekte in den Regeln, also beim Aufstieg aus dem Unter-AST der Regel in konkreter Syntax, in der Regel in Objektdiagramm-Notation den Stereotyp `<<not>>` beziehungsweise `<<list>>` dem Objekt hinzufügt.

Die `ownVisit`-Methoden für Ersetzungen setzen im Wesentlichen das Statusattribut `position` für die einzelnen Regelseiten auf den Wert `LHS` beziehungsweise `RHS` und starten dann die Traversierung der entsprechenden Regelseite. Nach dem Durchlauf beider Regelseiten wird dieses Statusattribut wieder auf den alten Wert zurückgesetzt.

7.4. Einbettung in die Kontrollflusssprache und generierte Werkzeuge

In den bisherigen Abschnitten dieses Kapitels wurde vor allem auf die Generierung der Sprache zur Beschreibung von Transformationsregeln eingegangen, insbesondere auf ihre Syntax und den Codegenerator. Die generierten Klassen zum Parsen der Transformationsregeln und zur Codegenerierung sind zwar prinzipiell aus beliebigen Java-Programmen aufrufbar. Jedoch erscheint die Verwendung der Regeln innerhalb eines Programms, das in der in Kapitel 6 vorgestellten Kontrollflusssprache geschrieben ist, für viele Anwendungen besonders nützlich. Diese Art der Verwendung wird deshalb durch die Generierung zusätzlicher Infrastruktur besonders unterstützt.

In diesem Abschnitt wird dabei lediglich auf die Generierung der Werkzeuge zur automatisierten Verarbeitung der Transformationsprogramme eingegangen. Die Nutzung dieser Werkzeuge hängt vom Verwendungszweck der Regeln ab. Sie wird am Beispiel der Integration der Transformationen in einen Codegenerierungsprozess in Kapitel 8 erläutert.

Das DSLTool-Framework in MontiCore gibt für die Verarbeitung von Modellen oder Programmen die in Abbildung 7.15 gezeigte Basisstruktur vor, die auch für die Verarbeitung der Transformationen genutzt wird. Die Basisklasse für Werkzeuge zur Sprachverarbeitung ist die Klasse `DSLTool`, und ein `DSLTool` kapselt mehrere `DSLWorkflows`, welche die einzelnen Verarbeitungsschritte darstellen. Die Verarbeitung der Modelle erfolgt dabei in Einheiten, die von der Klasse `DSLRoot` erben und typischerweise je eine Eingabedatei repräsentieren.

Die syntaktische Einbettung in die Kontrollflusssprache ist in einer Language-Datei [Kra10] definiert. Die Grammatik der Kontrollflusssprache enthält ein externes Nichtterminal `RuleDeclarationBody`, dessen Produktion in der Grammatik einer Regelsprache definiert ist. Welche Regelsprache hier verwendet wird, wird in der generierten Language-Datei bestimmt. Die entsprechende Festlegung besteht nur aus wenigen Zeilen, die aus dem Template-Fragment in Abbildung 7.16 generiert werden.

Die Verwendung einer Language-Datei ist im Vergleich zur Generierung von Java-Code, der den Parserwechsel steuert, wesentlich kompakter. Allerdings muss der entsprechende Java-Code in einem weiteren Schritt aus der Language-Datei generiert werden. Dies erfolgt durch das DSL-

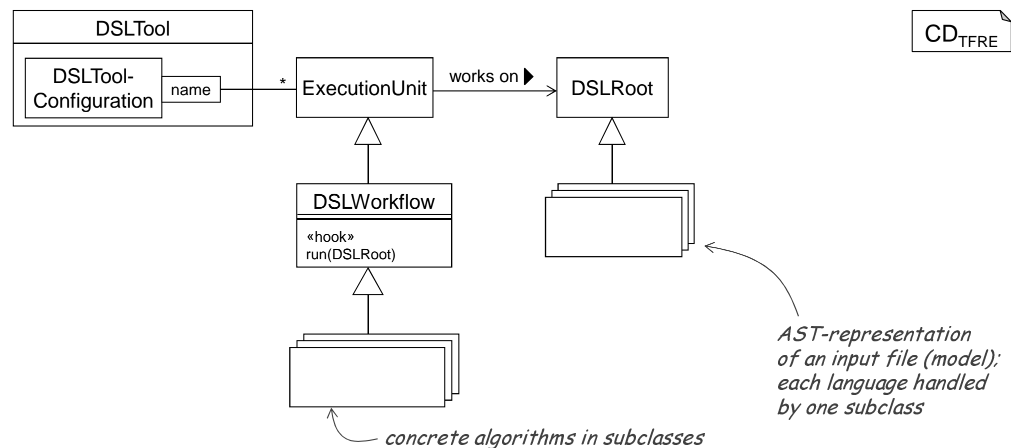


Abbildung 7.15.: Wichtige Klassen im DSLTool-Framework nach [Kra10]

FreeMarker-Template

```
1 rootfactory ${ast.getName()}TransformationRootFactory for
2     ${ast.getName()}TransformationRoot<mc.tf.cf._ast.MCCompilationUnit> {
3   mc.tf.cf.Controlflow.MCCompilationUnit module <<start>> ;
4   ...
5   ${package}.transformation.rule.${ast.getName()}TransformationRule.
6       TfObjects ruledeclarationbody in module.RuleDeclarationBody;
```

Abbildung 7.16.: Template für die Konfiguration der Spracheinbettung

Tool `MontiCore`, das ohnehin für die Verarbeitung der generierten Grammatik aufgerufen werden muss, sodass dieser Schritt kaum zusätzlichen Aufwand bedeutet.

Die Struktur des Codegenerators für Programme in der Kontrollflusssprache mit eingebetteten Regeln wurde am Beispiel der Transformationssprache für Statecharts bereits in Abbildung 6.11 gezeigt. Die Klassen in der oberen Hälfte dieses Diagramms sowie der `ControlflowTemplateOperator` sind nicht für die Statechartsprache spezifisch, sodass sie nicht generiert werden müssen. Von den in dieser Abbildung gezeigten Klassen sind also lediglich drei Unterklassen generiert.

Der `StatechartTransformationGenerationVisitor` enthält lediglich im Kontruktor einen sprachspezifischen Aufruf des TemplateOperators, im `StatechartTransformationGenerationWorkflow` ist der Aufruf eben dieses Visitors sprachspezifisch, und die `StatechartTransformationLanguage` erzeugt als Erweiterung der `ControlflowLanguage` den `StatechartTransformationGenerationWorkflow` als Workflow für die Codegenerierung. Diese Klassen verwenden wiederum einige generierte Hilfsklassen, die in der Abbildung 6.11 nicht aufgeführt sind. Diese Hilfsklassen erfüllen kleine Funktionen, wie etwa die für die Überprüfung der Parameterlisten erforderliche Bestimmung aller Objekte in einer Regel durch den `ObjectsCalculator`. Sie lassen sich wie die oben genannten Klassen aus den Informationen der DSL-Grammatik generieren.

Somit lässt sich zu einer gegebenen DSL nicht nur die Transformationssprache selbst, sondern auch grundlegende Infrastruktur zur Verarbeitung der Transformationen mit dem DSLTool-Framework generieren. Die Verwendung des DSLTool-Frameworks im Zusammenhang mit dieser Arbeit, also sowohl die Generierung der Transformationssprache und zugehöriger Infrastruktur durch den Sprachentwickler als auch die Nutzung dieser Infrastruktur durch den Anwender dieser Sprache, wird in Kapitel 8 beschrieben.

Kapitel 8.

Nutzung von Transformationen und Transformationssprachen

Damit die Transformationen, die sich mit den in den vorangegangenen Kapiteln beschriebenen Sprachen implementieren lassen, in Softwareentwicklungsprojekten sinnvoll genutzt werden können, ist es erforderlich, sie in die Entwicklungsprozesse zu integrieren. Dies setzt im Allgemeinen eine Integration in die bestehende Werkzeuglandschaft beziehungsweise in Toolketten voraus. Für Modelle in mit MontiCore definierten Sprachen findet die Verarbeitung der Modelle typischerweise im DSLTool-Framework statt. Dies gilt auch für die Verarbeitung von Transformationsmodulen mit Hilfe der für die entsprechenden Sprachen generierten Werkzeuge.

In diesem Kapitel wird zunächst erklärt, wie ein Sprachentwickler mit MontiCore zusätzlich zu den Werkzeugen und der Infrastruktur für die DSL gleichzeitig eine Transformationssprache sowie die zugehörigen Werkzeuge generieren kann. Anschließend wird die Nutzung dieser Sprache durch Transformationsentwickler erläutert, wobei zunächst der Aufruf der generierten Werkzeuge durch den Nutzer als technische Dokumentation beschrieben und anschließend die methodische Verwendung der Sprache am Beispiel der Vereinfachung von Statecharts erläutert wird.

8.1. Generierung der Transformationssprache durch den Sprachentwickler

Grundvoraussetzung für die Nutzbarkeit einer domänenspezifischen Transformationssprache ist selbstverständlich, dass die Sprache sowie die zugehörigen Werkzeuge überhaupt erstellt wurden. Dies wird in der Regel nicht von einem späteren Nutzer der Sprache, sondern von einem Sprachentwickler ausgeführt. Mit dem in Kapitel 7 vorgestellten Ansatz beschränkt sich der Aufwand hierfür allerdings auf den Aufruf einer Generators.

Dieser Aufruf ist am Beispiel eines Ant-Skripts [Ant11] in Abbildung 8.1 dargestellt. Dies ist die im MontiCore-Umfeld am weitesten verbreitete Art des Werkzeugaufrufs. Möglich sind aber auch der Aufruf aus anderen Build-Werkzeugen, der Kommandozeile oder aus Java-Programmen.

Die aufgerufene Klasse `DSL2TransformationLanguageTool` verfügt über eine Main-Methode. Sie ist in der Lage, Eingabeparameter zu verarbeiten, insbesondere die Angabe der auszuführenden Workflows in den Zeilen 5 bis 10. Der mit `generate_grammar` bezeichnete Synthese-Workflow erzeugt die Grammatik der Transformationssprache, `generate_trans-`

Ant-Build-Skript

```
1 <java
2   classname="mc.tfcs.grammartransformation.DSL2TransformationLanguageTool"
3   dir="." fork="true" failonerror="true">
4   <arg line="inputForTests/mc/testcases/statechart/Statechart.mc" />
5   <arg value="-synthesis" />
6   <arg value="ALL" /><arg value="generate_grammar" />
7   <arg value="-synthesis" />
8   <arg value="ALL" /><arg value="generate_translator" />
9   <arg value="-synthesis" />
10  <arg value="ALL" /><arg value="generate_tool" />
11  <arg value="-out" /><arg value="systemtest_gen/language" />
12  <classpath>
13    ...
14  </classpath>
15 </java>
```

Abbildung 8.1.: Aufruf des Transformationssprachengenerators aus Ant

`lator` erzeugt den Codegenerator für die Transformationssprache und `generate_tool` erzeugt die in Abschnitt 7.4 beschriebenen Werkzeuge zur Verarbeitung der Transformationsprogramme.

Über den Parameter `-out` in Zeile 11 lässt sich das Verzeichnis auswählen, in dem der generierte Code abgelegt wird. Innerhalb des angegebenen Verzeichnisses gilt die für MontiCore-Projekte übliche Konvention, dass die Grammatiken und Language-Dateien im Unterverzeichnis `def` und die generierten Java-Klassen im Verzeichnis `gen` liegen.

Zu beachten ist, dass der Generator der Transformationssprache nur die Grammatiken sowie Java-Dateien, die unmittelbar aus der DSL-Grammatik abgeleitet werden, generiert. Typischerweise ist deshalb nach dem Aufruf dieses Generators noch der Aufruf des DSLTools `MontiCore` sowie des Java-Compilers erforderlich, um die Sprache nutzen zu können. Die Aufrufparameter hierfür hängen jedoch vom Kontext ab, in dem die Transformationssprache verwendet wird, daher wird an dieser Stelle auf die Dokumentation der entsprechenden Werkzeuge verwiesen.

8.2. Aufruf der generierten Werkzeuge und Workflows

Nach der Erstellung der Transformationssprache ist der nächste Schritt die Entwicklung konkreter Transformationen. Damit diese Transformationen auf Modellen ausgeführt werden können, muss aus ihnen Java-Code generiert werden, der dann wiederum vom Java-Compiler in Bytecode übersetzt wird.

Die Generierung des Java-Codes erfolgt in der Regel im DSLTool-Framework von MontiCore, dessen Basisklassen und Funktionsweise bereits in Abschnitt 7.4 und in Abbildung 7.15 beschrieben wurden.

8.2.1. DSLTools für Transformationen

Das zentrale Werkzeug zur Verarbeitung von Transformationen ist ein DSLTool, dessen Namen sich aus dem Namen der zugrunde liegenden DSL und dem Suffix `TransformationTool` ergibt. Für die Statechartsprache heißt dieses Werkzeug also `StatechartTransformationTool`. Ein solches Werkzeug erwartet als Eingabe Dateien mit der Endung `.mtf` für *MontiCoreTransformation*.

Der Aufruf dieses Tools ist in Abbildung 8.2 für das Transformationsprogramm `SimplifyStatecharts.mtf` dargestellt. Dem Werkzeug müssen als Parameter die Grammatik- und Language-Datei der Statechartsprache (Zeilen 1 f. und 5 f.) sowie eine oder mehrere Eingabedateien (Zeile 7) übergeben werden. Die Angabe eines Ausgabeverzeichnisses (Zeile 8) ist optional. Standardmäßig wird hierfür das Verzeichnis `gen` verwendet. Relative Pfade beziehen sich dabei stets auf das Verzeichnis, in dem die Java-VM aufgerufen wird.

Java-Quellcode

```
1  String[] lngFiles = new String[]{"StatechartDSLTool.lng"};
2  String[] grammarFiles = new String[]{"Statechart.mc"};
3
4  Parameters parameters = new Parameters();
5  parameters.addOptionParameter(CFTool.PARAM_LANFILE, lngFiles);
6  parameters.addOptionParameter(CFTool.PARAM_GRAMMARFILE, grammarFiles);
7  parameters.addFile("inputForTests/mc/tf/sc/SimplifyStatecharts.mtf");
8  parameters.setGenOutputdir("test_output");
9
10 StatechartTransformationTool tool =
11     new StatechartTransformationTool(parameters);
12 tool.run();
```

Abbildung 8.2.: Aufruf eines `StatechartTransformationTools`

8.2.2. Workflows

Die generierten Werkzeuge zu den Transformationssprachen kapseln mehrere Workflows, die am Beispiel der Statechartsprache in Tabelle 8.3 aufgeführt sind. Bei Bedarf lässt sich auch nur eine Teilmenge der Workflows aufrufen, etwa indem vor dem Aufruf der `run()`-Methode einzelne Workflows aus den Parametern des Tools entfernt werden oder indem in einer Unterklasse von `StatechartTransformationTool` die Methode `initStandardParameters()` überschrieben wird. Für Details zur Konfiguration von Parametern der `DSLTools` wird auf [Kra10, Kap. 9] verwiesen. Die Namen und Funktionen der Workflows sind weitgehend durch die MontiCore-Komponenten *Extensible Type System* (ETS, [Völ11]) und *Generation and Language Infrastructure* (GLI, [Sch12]) vorgegeben.

Der Workflow `parse` verarbeitet die angegebenen Eingabedateien mit Hilfe der Parser für die Kontrollflusssprache und für Transformationsregeln in domänenspezifischer Notation. Anschließend liegt ein AST zu dieser Eingabedatei vor.

Durch `mapTypes` werden alle Typen, die in Grammatik der DSL definiert sind und in den Transformationen verwendet werden, auf die entsprechenden generierten Klassen der abstrakten

Name	Workflow-Klasse
parse	StatechartTransformationParsingWorkflow
mapTypes	ControlflowTypeMappingWorkflow
createExported	CreateExportedInterfaceWorkflow
prepareCheck	PrepareCheckWorkflow
check	CheckWorkflow
prepareGeneration	ControlflowPrepareGenerationWorkflow
_intern_codegen	StatechartTransformationGenerationWorkflow

Tabelle 8.3.: Workflows im StatechartTransformationTool

Syntax abgebildet, die für die spätere Codegenerierung benötigt werden. Diese Abbildung ergibt sich aus der Symboltabelle der Grammatik.

In den Workflows createExported und prepareCheck werden die Namensräume und Symboltabellen des Transformationsprogramms aufgebaut, die in check zur Überprüfung der Kontextbedingungen verwendet werden.

prepareGeneration führt eine Transformation auf dem AST aus. Alle undo-Annotationen für das Backtracking werden an die auf sie folgende Anweisung angehängt. Dies ist für die Codegenerierung vorteilhaft, da das Template, aus dem der Code für das Datenfluss-Backtracking generiert wird, auf dem AST-Knoten eben dieser Anweisung operiert.

Der Workflow _intern_codegen führt die eigentliche Codegenerierung durch. Der verwendete Templatesatz lässt sich wie in GLI-Generatoren üblich über den Parameter generator konfigurieren [Sch12, Kap. 7].

Beim Aufruf eines generierten Tools wird entsprechend dem durch das DSLTool-Framework vorgegebenen Ablauf zunächst der Workflow zum Parsen des Programms aufgerufen. Trifft der Parser auf eine eingebettete Transformationsregel, so wechselt er für die Verarbeitung der Regel auf den konkreten Parser der entsprechenden Regelsprache. Nach dem Parsen werden die Symboltabellen aufgebaut und die Kontextbedingungen überprüft. Ist das Programm syntaktisch korrekt und sind alle Kontextbedingungen eingehalten, so wird als letzter Schritt der generierte Codegenerierungsworkflow ausgeführt.

8.3. Integration von Transformationen in den Codegenerierungsprozess

Die Nutzung von Transformationen erfolgt in Modellierungsprojekten. Werden sie, wie für die Vereinfachung hierarchischer Statecharts beschrieben, zur Vorbereitung der Codegenerierung genutzt, so sind die Transformationsprogramme gemeinsam mit dem in MontiCore typischerweise templatebasierten Codegenerator und dem Compiler als Teil eines mehrstufigen Generators zu verstehen, der Modelle in ausführbaren Bytecode übersetzt. Aber auch Transformationen, die als Refactorings in Modellierungswerkzeugen eingesetzt werden, oder die Editier-Operationen beschreiben, sind als Bestandteil eines Modellierungswerkzeugs zu sehen.

8.3.1. Sprachen, Transformationen und Modelle

Da Transformationen also den Werkzeugen zuzuordnen sind, aber auch, weil viele Transformationen über die Grenzen eines Modellierungsprojektes hinaus sinnvoll anzuwenden sind, muss ihre Entwicklung nicht unbedingt in dem Projekt erfolgen, in dem die zu transformierenden Modelle entwickelt werden. Vielmehr sollten sie als eigenständige Artefakte betrachtet werden, sodass sich für die in dieser Arbeit beschriebenen Artefakte folgende Unterteilung ergibt:

- Die Modellierungssprache beziehungsweise die DSL wird von einem Sprachentwickler in einem Sprachentwicklungsprozess erstellt. Die zu dieser Sprache passende Transformationssprache wird aus den Quell-Artefakten, die für die Sprachentwicklung ohnehin zu erstellen sind, generiert, sodass sie als zusätzliches Produkt dieses Entwicklungsprozesses verstanden werden kann.
- Transformationen zu dieser DSL können entweder von Sprachnutzern oder von Werkzeugentwicklern erstellt werden. Dabei kann die Entwicklung der Transformationen in den Werkzeugentwicklungsprozess integriert sein. Sie können jedoch auch unabhängig von anderen Werkzeugen erstellt werden, wodurch die Wiederverwendung vereinfacht werden kann. Beispielsweise können die in Kapitel 3 beschriebenen semantikerhaltenden Transformationen nicht nur zur Vorbereitung der Codegenerierung, sondern auch als Refactoring-Operationen oder Makros in einem Statechart-Editor verwendet werden.
- Hiervon zu trennen ist wiederum die modellgetriebene Entwicklung eines Softwareproduktes, bei der sowohl die Modellierungssprache als auch Transformationen der Modelle eingesetzt werden. In diese Produktentwicklung sind typischerweise sowohl Softwareentwickler als auch Domänenexperten eingebunden. Neben den Modellen werden in diesen Produktentwicklungsprozessen oft auch Programme in GPLs als Quellartefakte verwendet. Codegeneratoren und Transformatoren werden in diesem Prozess lediglich eingesetzt, aber nicht entwickelt.

8.3.2. Transformationen in der Codegenerierung für Statecharts

Werden Transformationen als Bestandteile eines Werkzeugs eingesetzt, so sind sie meist darauf angewiesen, auf syntaktisch korrekten Modellen zu arbeiten[1], wobei syntaktische Korrektheit sowohl auf die kontextfreie Syntax als auch auf Kontextbedingungen zu beziehen ist. Die eingesetzten Transformationen sollten qualitätsgesichert sein, ebenso wie eingesetzte Codegeneratoren, Compiler oder sonstige Werkzeuge, und sie sollten die syntaktische Korrektheit des Modells erhalten.

Die Überprüfung von Kontextbedingungen auf dem Modell muss daher vor der Anwendung der Transformationen erfolgen. Für die Codegenerierung aus Statecharts ist die gesamte Workflowkette vom Parsen der Eingabemodelle bis zum Schreiben des übersetzen Codes in Abbil-

[1] Zu den Ausnahmen zählen beispielsweise Refactoring-Operationen in Entwicklungsumgebungen, die teilweise auch auf fehlerhaften Modellen oder Programmen ausgeführt werden können. Meist wird dann aber auch keine fehlerfreie Ausführung der Transformation oder Korrektheit des resultierenden Programms beziehungsweise Modells garantiert.

dung 8.4 dargestellt, wobei die Verarbeitung des handgeschriebenen GPL-Codes hier nicht abgebildet ist.

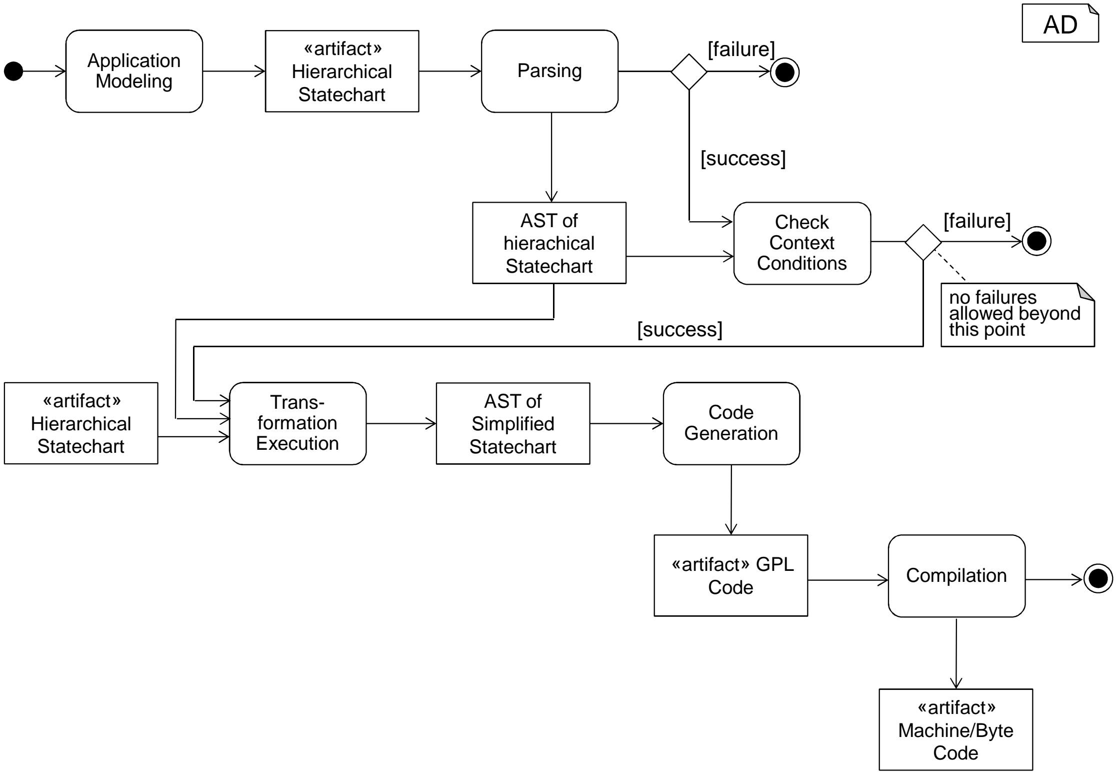


Abbildung 8.4.: Analyse, Transformationen und Codegenerierung

In dem dargestellten Arbeitsablauf legt ein Anwendungsmodellierer zunächst eines oder mehrere Modelle im Dateisystem persistent ab. Anschließend initiiert er den Codegenerierungsprozess.

Der folgende Verarbeitungsprozess vom Parsen bis hin zur Codegenerierung sollte vollständig automatisiert sein und ohne Benutzerinteraktion erfolgen. Insbesondere findet keine Modifikation der transformierten Modelle oder des generierte GPL- oder Bytecodes durch den Entwickler statt. Zur Implementierung eines solchen Prozesses bietet sich die Verwendung eines Build-Werkzeugs an, im Kontext von MontiCore ist vor allem Ant verbreitet.

Die Transformations- und Generierungsschritte einschließlich der Kompilierung sollten inkrementell sein in dem Sinne, dass nur auf Artefakten, die seit der letzten Ausführung der abgebildeten Aktivität verändert wurden, erneut die Transformations- und Generierungsschritte ausgeführt werden. Das hierfür erforderliche Abhängigkeitsmanagement kann entweder Bestandteil des Build-Werkzeugs oder der Generatoren sein. Die Transformationen und Generierungen sind nicht inkrementell in dem Sinne, dass sie manuelle Änderungen an den generierten Artefakten erhalten, da solche Änderungen in der beschriebenen Methodik nicht vorgesehen sind.

Nachdem der Entwickler den Codegenerierungsprozess angestoßen hat, müssen die Quelldateien zunächst geparst werden. Sind die Eingaben korrekt in Bezug auf die kontextfreie Syntax der Modellierungssprache, so liegen sie anschließend als abstrakter Syntaxbaum im Hauptspeicher vor. Fehlerhafte Eingaben führen dagegen zum Abbruch des Prozesses.

Auf dem Syntaxbaum werden im Folgenden die Kontextbedingungen aus der Sprache des Eingabemodells geprüft. Hierfür werden häufig Symboltabellen aufgebaut und Berechnungen mit Attributgrammatiken durchgeführt, was im gezeigten Diagramm aber nicht explizit abgebildet wurde. Wie bei der kontextfreien Analyse führen Fehler zum Abbruch des Prozesses. Werden bei der Überprüfung keine Verletzungen von Kontextbedingungen gefunden, so ist das Eingabemodell syntaktisch korrekt. Die Kontextbedingungen sollten garantieren, dass alle folgenden Schritte fehlerfrei durchgeführt werden können, das heißt dieser Schritt ist der letztmögliche Zeitpunkt, zu dem die Codegenerierung fehlschlagen darf. Folglich müssen auch nach dieser Aktion keine weiteren Überprüfungen von Kontextbedingungen auf transformierten Modellen oder auf dem generierten Code stattfinden, obwohl sich dies in der Praxis nicht mit allen Werkzeugen vermeiden lässt.

Durch die Transformation werden die Quellmodelle für die eigentliche Codegenerierung vorbereit. Für Statecharts können beispielsweise die in Kapitel 3 beschriebenen Vereinfachungen durchgeführt werden. Als Eingabe für diesen Schritt kann wiederum der beim Parsen entstandene AST verwendet werden, der durch die Überprüfung der Kontextbedingungen nicht modifiziert wurde. Als Ausgabe kann entweder ein neuer AST entstehen, oder der Eingabe-AST kann die durch In-Place-Transformationen entstandenen Modifikationen beinhalten. Je nach Implementierung können Ein- und Ausgabemodelle für diesen Schritt in einer Datei oder im Hauptspeicher vorliegen, wobei letztere Variante in Abbildung 8.4 dargestellt ist.

Die transformierten Modelle dienen als Eingabe für die eigentliche Codegenerierung. In MontiCore-basierten Sprachen kommt meist ein templatebasierter Generator zum Einsatz, aber auch API-basierte Codegeneratoren können hier verwendet werden. Üblicherweise wird das Resultat dieses Schrittes persistiert, da im folgenden Schritt der Compiler Dateien als Eingabe erwartet.

Für die Übersetzung des generierten Codes wird ein Compiler verwendet, etwa der Java-Compiler. Dieser erzeugt aus dem generierten Code Java-Code Bytecode, der in einer virtuellen Maschine ausgeführt werden kann. Damit ist der Transformations- und Codegenerierungsprozess beendet, und die Modelle können als Bestandteil des Softwareprodukts verwendet werden.

8.4. Fallbeispiele aus der Vereinfachung hierarchischer Statecharts

In diesem Abschnitt werden exemplarische Methoden aus der Vereinfachung hierarchischer Statecharts betrachtet, die typische Anwendungsfälle der generierten Transformationssprachen sind. Die vollständige Transformation ist im Anhang D zu finden.

Die Anwendungsfälle sind jeweils tabellarisch gelistet. Zunächst werden die Problemstellung und der Lösungsansatz vorgestellt. Anschließend wird die Lösung an einem konkreten Codebeispiel erläutert. Abschließend wird auf mögliche Varianten des Problems eingegangen.

8.4.1. Verschieben von Elementen

<table>
<tr><td>Problem</td><td>Ein Element muss innerhalb des Modells an eine andere Stelle verschoben werden. Im konkreten Beispiel soll eine Transition <code>$T</code> in einen Zustand <code>$S</code> verschoben werden. Beide Elemente werden als Parameter an die Regel übergeben.</td></tr>
<tr><td>Lösung</td><td>Die Transition muss an der Stelle, an der sie sich vor der Regelausführung im Modell befindet, unabhängig von ihrem Kontext und ihren Eigenschaften entfernt werden und innerhalb der Zustands wieder eingefügt werden.</td></tr>
<tr><td>Codebeispiel</td><td><pre>
1 transformation moveTransitionToState(Transition $T,
2   State $S) {
3   State $S [[state $s_name {
4   [[:- Transition $T;]]
5   }]]
6
7   [[Transition $T; :-]]
8 }
</pre></td></tr>
<tr><td>Erläuterung</td><td>Zu der Transition <code>$T</code> werden in Zeile 7 weder Kontext noch weitere Eigenschaften angegeben. Dadurch wird sie zunächst vorübergehend aus dem Modell entfernt. In Zeile 4 wird die selbe Transition, gekennzeichnet durch die Schemavariable <code>$T</code>, innerhalb der Zustands <code>$S</code> in das Modell eingefügt. Dabei bleiben alle Eigenschaften der Transition wie Quellzustand, Zielzustand oder Stimulus erhalten.</td></tr>
<tr><td>Varianten</td><td>Die Elemente müssen nicht als Parameter übergeben werden. Zu dem zu verschiebenden Element können ein Kontext und geforderte Eigenschaften (in konkreter Syntax oder durch Constraints) angegeben werden. Werden Elemente als Parameter übergeben, aber die geforderten Eigenschaften gelten nicht, so schlägt die Ausführung der Regel fehl.</td></tr>
</table>

Tabelle 8.5.: Verschieben von Elementen anhand der Regel `moveTransitionToState`

8.4.2. Kopieren von Elementen

Problem	Von einem Element muss an anderer Stelle im Modell eine exakte Kopie angelegt werden. Im konkreten Beispiel handelt es sich um eine Invariante, die in Unterzustände ihres aktuellen Zustands kopiert werden soll.
Lösung	Das Kopieren von Elementen erfolgt wie in MontiCore üblich über die Methode `deepClone`. Anschließend wird eine Transformationsregel zum Einhängen der Invariante an einer geeigneten Stelle im Modell aufgerufen.

<table>
<tr><td>Codebeispiel</td><td>

```
1 propagateInvariantsAndRemoveHierarchy() {
2   loop pullUpTransition();
3   State $s;
4   List<State> $substates;
5   loop if (findTopLevelStateWithSubstates($s,
6       $substates)) {
7     for (State $next : $substates) {
8       List<Expression> $invariants;
9       getInvariants($s, $invariants);
10      for (Expression $inv : $invariants) {
11        Expression $inv_copy =
12            (Expression) $inv.deepClone();
13        if (addInvariantToState($inv_copy, $next)) { }
14      }
15      pullUpState($next);
16    }
17    $s.delete();
18  }
19 }
```

</td></tr>
<tr><td>Erläuterung</td><td>Die Methode <code>deepClone</code> erstellt eine exakte Kopie eines Teil-ASTs, dessen Wurzel der Knoten ist, auf dem die Methode aufgerufen wird. Da die Methode in Schnittstellen wie <code>Expression</code> den Rückgabetyp <code>ASTNode</code> hat, ist es erforderlich, das Objekt zu casten. Die Methode zum Einhängen in den AST entspricht dem oben genannten Verschieben von Elementen, das auch für Knoten funktioniert, die noch nicht im AST hängen.</td></tr>
</table>

Tabelle 8.6.: Kopieren von Elementen anhand der Methode `propagateInvariants-AndRemoveHierarchy`

8.4.3. Veränderung gebundener Bezeichner

<table>
<tr><td>Problem</td><td>Ein referenzierender Bezeichner soll verändert werden. Im vorliegenden Fall soll der Quellzustand einer Transition verändert werden.</td></tr>
<tr><td>Lösung</td><td>Diese Änderung lässt sich über die konkrete Syntax durchführen, wobei der neue Quellzustand Teil des Matches sein muss. Dann kann die Änderung durch eine Ersetzung des entsprechenden <code>Name</code>-Elementes erfolgen.</td></tr>
<tr><td>Codebeispiel</td><td>

```
1 transformation startTransitionFromSubstate(Transition $T,
2     State $S) {
3   state $outer {
4     State $S [[ state $inner << [[ final :- ]] >>; ]]
5   }
6
7   Transition $T [[ [[ $outer :- $inner ]] -> $target; ]]
8 }
```

</td></tr>
<tr><td>Erläuterung</td><td>Während des Pattern-Matchings wird die Schemavariable <code>$inner</code> aus Zeile 4 gebunden. Ihr Wert ist jetzt eindeutig bestimmt, und sie kann auf der rechten Regelseite in Zeile 7 verwendet wenden.</td></tr>
</table>

Varianten	Anstatt der Ersetzung kann ein referenzierender Bezeichner auch neu erzeugt werden. In diesem Fall bleibt der linke Teil der Ersetzung leer.

Tabelle 8.7.: Veränderung gebundener Bezeichner anhand der Regel `startTransitionFromSubstate`

8.4.4. Verwendung von Kontrollstrukturen

Problem	Die Ausführung von Transformationsregeln erfolgt nicht in einer fixen Reihenfolge, sondern soll durch Kontrollstrukturen beeinflusst werden. Im gegebenen Beispiel werden zunächst ein Superzustand mit ausgehender Transition und all seine Subzustände ermittelt. Anschließend wird über alle Subzustände iteriert und je eine ausgehende Transition aus diesen Subzuständen angelegt. Danach wird die ursprüngliche Transition gelöscht. Dieses Verfahren wird iteriert, bis es nicht mehr anwendbar ist.
Lösung	Die Transformationssprachen in MontiCore bieten hierfür die Kontrollstrukturen `loop`, `if` und `for` an. Die `if`-Statements verhalten sich analog zu denen in Java. `loop`-Anweisungen iterieren über ihren Rumpf, bis er sich nicht mehr erfolgreich ausführen lässt. Durch `loop if` lässt sich eine Schleife mit Abbruchbedingung (analog zu `while`-Schleifen in Java) realisieren.

Codebeispiel

```
1  startFromFinal() {
2    State $superstate;
3    List<State> $substates;
4    Transition $t;
5    loop if (searchTransitionForFinalSubstates($t,
6        $superstate, $substates)) {
7      for (State $substate : $substates) {
8        Transition $t_new = $t.deepClone();
9        moveTransitionToState($t_new, $superstate);
10       startTransitionFromSubstate($t_new, $substate);
11     }
12     $t.delete();
13   }
14 }
```

Erläuterung	Die Wiederholung des gesamten Verfahrens wird durch die Kombination von `loop` und `if` im Block von Zeile 5 bis Zeile 13 gesteuert. Diese führt den Rumpf in den Zeilen 7 bis 12 so lange aus, bis die seiteneffektfreie Regel `searchTransitionForFinalSubstates` nicht mehr anwendbar ist. Die Iteration über alle Unterzustände erfolgt durch die `for`-Schleife in den Zeilen 7 bis 11.

Tabelle 8.8.: Verwendung von Kontrollstrukturen anhand der Methode `startFromFinal`

8.4.5. Erstellung eines ausführbaren Programms

Problem	Das Generat aus einer Transformation soll direkt in der virtuellen Maschine von Java ausführbar sein, ohne dass zusätzliche Java-Klassen händisch implementiert werden müssen.
Lösung	Enthält ein Transformationsmodul eine `main`-Methode ohne Parameter, so wird diese in eine Java-`main`-Methode übersetzt.

Codebeispiel	<pre>1 main() {
	2 Logger log = mc.MCG.getLogger();
	3 log.finer("createEmptyActionBlock");
	4 loop createEmptyActionBlock();
	5 log.finer("createEmptyEntryBlock");
	6 loop createEmptyEntryBlock();
	7 log.finer("createEmptyExitBlock");
	8 loop createEmptyExitBlock();
	9 log.finer("eliminateDo");
	10 loop eliminateDo();
	11 log.finer("expandInitial");
	12 loop expandInitial();
	13 log.finer("expandFinal");
	14 loop expandFinal();
	15 log.finer("forwardToInitial");
	16 forwardToInitial();
	17 log.finer("startFromFinal");
	18 startFromFinal();
	19 log.finer("eliminateEntryActions");
	20 eliminateEntryActions();
	21 log.finer("eliminateExitActions");
	22 eliminateExitActions();
	23 log.finer("resolveInternalTransitions");
	24 loop resolveInternalTransition();
	25 log.finer("propagateInvariantsAndRemoveHierarchy");
	26 propagateInvariantsAndRemoveHierarchy();
	27 }</pre>

Erläuterung	Der generierten Java-Klasse muss beim Aufruf genau ein Argument übergeben werden: Der Name der Eingabedatei, die von der MontiCore-Laufzeitumgebung eingelesen und durch die Transformation verarbeitet werden soll. Anschließend wird diese Datei mit dem Ergebnis der Transformation überschrieben.
Varianten	Der Aufruf kann nicht nur aus der Kommando-Zeile, sondern beispielsweise auch aus Build-Skripten erfolgen, sofern die entsprechenden Tools den Aufruf von Java-Programmen und Übergabe von Parametern an diese Programme unterstützen.

Tabelle 8.9.: Erstellung eines ausführbaren Programms in der `main`-Methode

Kapitel 9.

Die Entwicklung der Modelltransformationsengine

In den bisherigen Kapiteln wurden die einzelnen Komponenten der Transformationssprachen in MontiCore, die zugehörigen Werkzeuge, sowie die Erstellung und Nutzung der Sprachen und Werkzeuge durch Sprach- und Transformationsentwickler beschrieben. Die Erstellung all dieser Artefakte, sofern sie nicht generiert wurden, erfolgte in Teilprojekten innerhalb des MontiCore-Projekts.

In diesem Kapitel wird die Struktur der Teilprojekte innerhalb des MontiCore-Projekts, die sich mit Transformationen befassen, erläutert. Es wird außerdem beschrieben, wie bei der Entwicklung dieser Transformationsprojekte methodisch vorgegangen wurde, und wie sich die Methodik über die bisherige Projektlaufzeit hinweg verändert hat.

Dazu werden zunächst die Konventionen zur Struktur von MontiCore-Projekten und methodische Grundsätze bei der Entwicklung vorgestellt. Aufbauend darauf wird das initiale Setup der Transformationsprojekte beschrieben, das auf der testgetriebenen Entwicklung von Prototypen basiert. Dieses Vorgehen wurde später durch eine Methode ersetzt, bei der nur noch das Generat getestet wird; diese Methode wird ebenfalls erläutert, insbesondere auch das Testen der in Abschnitt 7.3 vorgestellten Generierung des Codegenerators. Abschließend folgt eine Gegenüberstellung der beiden Ansätze.

9.1. Projektstruktur und agile Entwicklung mit MontiCore

Die Transformationsengine in MontiCore ist in mehrere Projekte unterteilt, wobei Projekt hier im Sinne eines Projektes in der integrierten Entwicklungsumgebung Eclipse [Ecl] zu verstehen ist. Das MontiCore-Projekt als ganzes kapselt mehrere dieser Eclipse-Projekte, unter anderem auch diejenigen, die für Modelltransformationen von Bedeutung sind.

9.1.1. Innere Struktur der Transformationsprojekte

Viele dieser Projekte, unter anderem auch diejenigen, die im Rahmen dieser Arbeit entwickelt wurden, enthalten MontiCore-Grammatiken und Language-Dateien. Sie verwenden MontiCore – meist in einer älteren, bereits releaseten Version – als Generator für Lexer, Parser, Klassen der abstrakten Syntax und Werkzeuginfrastruktur.

Da diese Projekte also nicht nur Bestandteile des MontiCore-Projektes sind, sondern auch ihrerseits MontiCore als Werkzeug zur Sprachdefinition verwenden, folgen sie neben Empfehlun-

gen für ein agiles Vorgehen bei der Entwicklung von Sprachen auch der für MontiCore-Sprachen empfohlenen Konvention für die Projektstruktur. Diese sieht die Unterteilung des Projektes in die folgenden Unterverzeichnisse vor:

- Im Verzeichnis `def` liegen alle Dateien, die Quellartefakte des Projektes sind und von Codegeneratoren verarbeitet werden. Ausgenommen hiervon sind Artefakte, die als Eingabe eines Testfalls dienen, und auf die im Folgenden noch eingegangen wird. Generell liegen in diesem Verzeichnis beliebige Modelle, im Falle der Transformationsprojekte handelt es sich um MontiCore-Grammatiken, Language-Dateien, Bundle-Dateien [Kra10] und UML/P-Klassendiagramme [Sch12].

- Die aus diesen Artefakten generierten Dateien, zumeist Java-Dateien, werden im Verzeichnis `gen` abgelegt. Da diese Dateien automatisiert erstellt werden, müssen sie nicht versioniert werden. Sie dürfen auch nicht händisch verändert werden, da alle manuellen Änderungen bei der nächsten Generierung überschrieben werden würden.

- Das Verzeichnis `src` enthält handkodierte Java-Dateien. Diese machen gemeinsam mit den Artefakten im `def`-Verzeichnis die wesentlichen Quellartefakte des Projektes aus. Sie sind Bestandteil des im Projekt entwickelten Produktes; Testfälle werden in diesem Verzeichnis nicht abgelegt.

- In Projekten, die einen Codegenerator beinhalten, werden im Verzeichnis `templates` die FreeMarker-Templates abgelegt. Ein Codegenerator wird typischerweise mit einem oder mehreren Sätzen von Templates ausgeliefert, die für die Codegenerierung auf eine bestimmte Zielplattform geeignet sind. Für die Generierung zu anderen Zielplattformen können die Templates mit den in [Sch12] beschriebenen Mechanismen erweitert, angepasst oder ersetzt werden.

- Das Verzeichnis `test` enthält JUnit-Testfälle für Modul- und Integrationstests. Diese werden nur in der Entwicklung, jedoch nicht in der Anwendung des Projektes benötigt und daher separat von den Java-Klassen im `src`-Verzeichnis gehalten.

- Viele dieser Testfälle greifen auf Eingabedateien aus dem Verzeichnis `inputForTests` zu, die etwa zur Beschreibung der initialen Objektstruktur vor der Ausführung des Testfalls verwendet werden. Dazu können entweder Objektdiagramme oder – wie in den Transformationsprojekten häufiger anzutreffen – Modelle in einer geeigneten Sprache, etwa der Statechartsprache, verwendet werden. Diese sind dann meist das Quellmodell einer Transformation, die durch den Testfall ausgeführt wird.

- Die Verzeichnisse `build`, `dist` und `bin` enthalten kompilierte Java-Klassen. Wie die Dateien im Verzeichnis `gen` werden auch die Dateien in diesen Verzeichnissen automatisch erstellt und stehen daher nicht unter Versionskontrolle. Im Verzeichnis `build` legen die Build-Skripte die kompilierten Klassen zum generierten Code, zum handgeschrieben Code und zu den Testfällen sowie Kopien der Templates für die Codegenerierung ab. In `dist` fehlt der kompilierte Code für die Testfälle, sodass dieses Verzeichnis nur die Artefakte enthält, die an einen Anwender des Projektes ausgeliefert werden müssen. Das

bin-Verzeichnis wird von dem in Eclipse integrierten Java-Compiler als Zielverzeichnis verwendet. Es beinhaltet die kompilierten Klassen zu den .java-Dateien aus den Verzeichnissen src, gen und test. Inhaltlich ist es weitgehend redundant zum build-Verzeichnis, jedoch werden diese Artefakte separat voneinander gehalten, damit die IDE und das Build-Werkzeug störungsfrei und unabhängig voneinander arbeiten können.

Darüber hinaus existieren weitere Unterverzeichnisse, für die es bisher noch keine Konvention gibt, und in denen beispielsweise durch Testfälle generierte Dateien abgelegt werden oder die erwartete Ergebnisse von Tests enthalten. Des weiteren sind Dateien und Verzeichnisse vorhanden, die für die Konfiguration und Verwaltung der Projekte innerhalb von Eclipse verwendet werden und deren Funktion der Dokumentation zu Eclipse entnommen werden kann.

Die oben beschriebene Aufteilung sorgt zum einen dafür, dass Artefakte, die lediglich für Entwickler relevant sind, sauber von auszuliefernden Elementen getrennt sind. Zum anderen ermöglicht sie es, effizient, nämlich durch ein Standard-Build-Skript vorgegeben, einen inkrementellen Codegenerierungsprozess zu etablieren und häufig, wiederholbar und automatisiert ausführbare Modul- und Integrationstests zu erstellen. Beides sind wesentliche Voraussetzungen für agile Entwicklung, da sie den Entwicklern zum einen erlauben, schnell kleine Inkremente zu entwickeln und ihre Qualität zu sichern, und weil eine gute Überdeckung mit leicht und schnell ausführbaren Tests das Vertrauen der Entwickler in die Korrektheit des eigenen Codes und der Modifikationen an existierendem Code stärkt.

9.1.2. Testgetriebene Entwicklung der MontiCore-Transformationsengine

Die Entwicklung in den Transformationsprojekten erfolgt nahezu ausschließlich testgetrieben. Vor der Erstellung des Codes für das Produktivsystem werden also Testfälle erstellt, mit deren Hilfe sichergestellt wird, dass das erstellte Modul das geforderte Verhalten aufweist. Bei den getesteten Quellartefakten handelt es sich teilweise um Java-Code, teilweise aber auch um Grammatiken oder Codegenerierungs-Templates.

Zum Testen von Java-Code wird das Test-Framework JUnit [Bec04] verwendet, für die Templates und Grammatiken steht jedoch kein vergleichbares Werkzeug zur Verfügung. Daher werden für diese Artefakte ebenfalls JUnit-Tests erstellt, die entweder den generierten Code oder umgebende Werkzeuge, etwa die Template-Engine oder die Parse-Workflows, aufrufen.

Bei Codegeneratoren stellt sich darüber hinaus die Frage, ob durch einen Test das Verhalten des Generators oder das Verhalten des generierten Codes überprüft werden soll. Diese Problemstellungen sind über die Entwicklungszeit der Transformationsprojekte durch zwei verschiedene Projekt-Setups adressiert worden, die in den Abschnitten 9.2 und 9.3 vorgestellt werden.

Im Bereich der Qualitätssicherung von Codegeneratoren kann auf verschiedene Ansätze zurückgegriffen werden. Zum einen sind hier formale Beweise der Korrektheit des Generators zu nennen, wofür es im Compilerbau einige Vorarbeiten gibt [Nec00, Gle03]. Diese Techniken sind jedoch vergleichsweise aufwändig anzuwenden und für ein agiles Vorgehen mit einer in der Entwicklung befindlichen Sprache deshalb nicht besonders geeignet. Alternativ können ausgewählte, von Generatoren produzierte Ergebnisse validiert werden.

Bei der Validierung der generierten Artefakte lassen sich vollständige Überprüfungen von der Überprüfung ausgewählter Aspekte unterscheiden. Orthogonal hierzu kann entweder die syn-

taktische Struktur oder das Verhalten des Generats überprüft werden. Tabelle 9.1 zeigt Beispiele für die verschiedenen Ansätze.

Überprüfung	**Struktur**	**Verhalten**
vollständig	Isomorphietests, Vergleich mit Sollergebnis oder Vorgabe eines Testorakels [MBT08]	*unentscheidbar*
ausgewählt	Prüfung der Existenz und/oder Nichtexistenz von Elementen im Generat	Vergleich des Verhaltens mit dem einer Referenzimplementierung oder eines Interpreters [Stü06], Testfälle für generierten Code

Tabelle 9.1.: Möglichkeiten der Validierung von Generatoren

Die vollständige Strukturüberprüfung und ausgewählte Verhaltensüberprüfung werden im Folgenden noch näher betrachtet. Trotzdem sollen hier kurz die Vor- und Nachteile der einzelne Verfahren skizziert werden.

Eine vollständige strukturelle Überprüfung des Generats deckt sehr viele Fehler auf, ist aber vergleichsweise unflexibel. So kann sie – je nach Implementierung – auch White-Space-sensitiv sein, und sie führt zu Fehlermeldungen, wenn der generierte Code überflüssige Artefakte enthält, die sich auf das Verhalten nicht auswirken. Ausgewählte strukturelle Überprüfungen sind dagegen zu ungenau, sie garantieren unter Umständen nicht einmal syntaktische Korrektheit des Generats.

Verhaltensüberprüfungen setzen zunächst einmal syntaktische Korrektheit des Generats voraus. Sie sind auch robust gegenüber überflüssigen Bestandteilen im generierten Code. Vollständige Überprüfungen des Verhalten, etwa die semantische Äquivalenz zu einer Referenzimplementierung, sind für Turing-vollständige Zielsprachen allerdings nicht entscheidbar. Für unvollständige Analysen kann entweder eine umfangreiche Referenzimplementierung oder ein Interpreter erstellt werden, dessen Verhalten für ausgewählte Eingaben mit dem des generierten Code verglichen wird. Alternativ können einzelne Testfälle für den generierten Code erstellt werden. Diese Verfahren haben den Nachteil, dass die Fehlermeldung sich auf den generierten Code bezieht und der ursächliche Fehler im Generator dazu erst noch gesucht werden muss.

9.2. Prototypenbasierte Entwicklung der Generatoren

Zu Beginn der Entwicklung der MontiCore-Transformationsengine unterlag zum einen das Design der Sprachen noch häufigen Änderungen, sodass eine agile Entwicklung anzustreben war.

Des weiteren waren mit dem Codegenerator für die Regeln in Objektdiagramm-Notation und später auch der Kontrollflusssprache die zwei Generatoren qualitätszusichern, deren generierter Code vergleichsweise komplex ist. Bei den Regeln in Objektdiagramm-Notation liegt der Grund hierfür im Wesentlichen in der effizienten Mustersuche, aber auch in der abstrakten Syntax von MontiCore, die im Vergleich mit Ansätzen wie EMF [BSM+03] oder JMI [SM02] relativ viele Fallunterscheidungen beim Zugriff auf Assoziationen und Attribute erfordert. Bei der Kontrollflusssprache ist das Backtracking ausschlaggebend für die Komplexität des generierten Codes.

9.2.1. Prototypen- und Generatorprojekte mit Modultests

Die Komplexität der Generierung wurde durch ein geeignetes Setup der Projekte in zwei Teilprobleme zerlegt: Für jedes zu generierende Feature wurde zunächst ein handgeschriebener Prototyp des generierten Codes erstellt. Dieser wurde testgetrieben entwickelt, wobei darauf geachtet wurde, dass der Testfall die erwarteten Eigenschaften des später zu generierenden Codes abdeckt, und dass der handgeschriebene Code so systematisch erstellt wird, dass er sich später generieren lässt. In einem zweiten Schritt wurde der Codegenerator testgetrieben entwickelt. Die Tests des Codegenerators bestanden in einem syntaktischen Vergleich des generierten Codes mit dem Prototypen. Diese Tests sind zwar, wie im letzten Abschnitt bereits beschrieben wurde, vergleichsweise unflexibel. Sie zeigen jedoch die fehlerhaften Stellen präzise auf, in der Regel mit einer Angabe der Zeile und Spalte des fehlerherhaften AST-Knotens im Dokument.

Die Entwicklung des Generators und des Prototyps erfolgte in zwei verschiedenen Projekten, wobei das Projekt für Prototypen das Suffix „Prototype" im Namen trug. So wurde die Regelsprache in Objektdiagramm-Notation im Projekt `TFODRules` entwickelt, der entsprechende Prototyp im Projekt `TFODRulesPrototype`.

Abbildung 9.2 zeigt das Projekt-Setup für die Regelsprache in Objektdiagrammsyntax und das Generatorprojekt für Regelsprachen in konkreter Syntax. Die Projektgrenzen sind als gerundete Rechtecke dargestellt. Die Entwicklung eines Features, das in der Transformationsregel `EliminateDo` verwendet wird, kann in diesem Projekt-Setup folgendermaßem ablaufen.

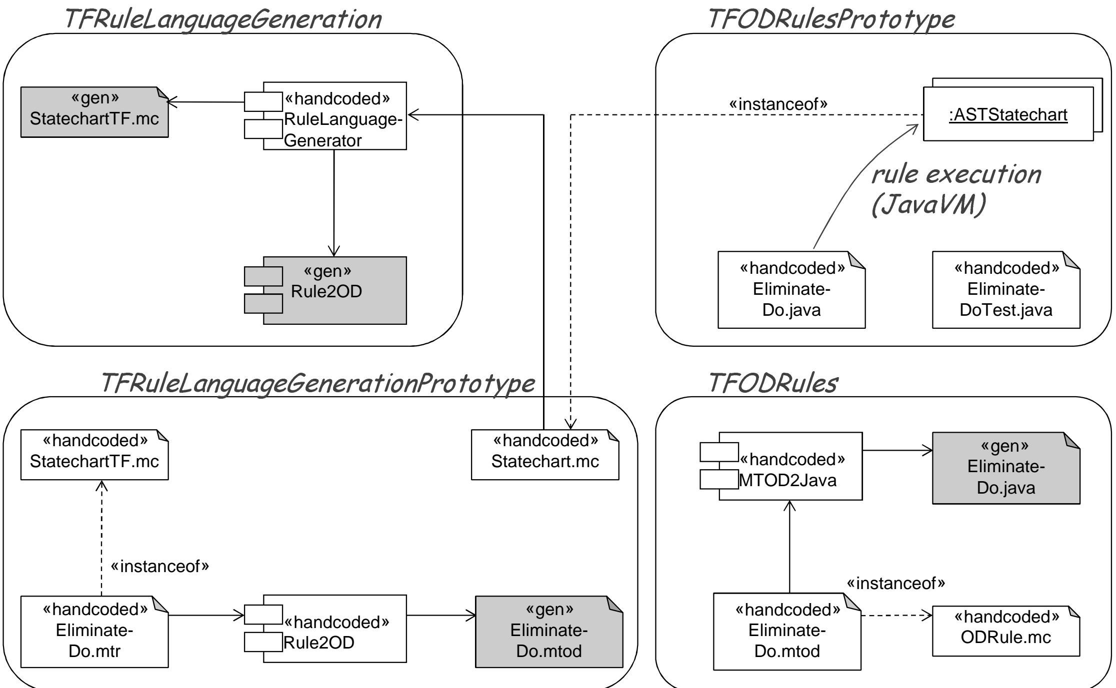


Abbildung 9.2.: Projektstruktur für die initiale Entwicklung der Transformationsprojekte

Die Grammatik der Statechartsprache sowie gegebenenfalls Beispielmodelle existieren typischerweise schon vor Beginn der Implementierung. Der Entwickler der Transformationsprojekte erstellt zunächst im Projekt `TFODRulesPrototype` einen Testfall für die Transformationsregel. Anschließend schreibt er im selben Projekt den Code der Java-Klassendatei `EliminateDo.java`, den er mit dem Testfall überprüft.

Danach erstellt er die Regel in Objektdiagramm-Notation `EliminateDo.mtod` und benutzt den Codegenerator für Regeln in Objektdiagramm-Notation, um hieraus Java-Code zu erstellen. Wenn in der Regel ein neues Feature verwendet wurde, zeigt eine Hilfsklasse zum Vergleich von Java-Dateien die genaue Position im Quellcode, an der sich die handgeschriebene Klasse und die generierte Klasse unterscheiden. Diese Hilfsklasse kann wahlweise auch nur einzelne Methoden analysieren, und sie ist in der Lage, die Reihenfolge von Elementen bei Bedarf zu ignorieren, etwa bei Methodendeklarationen oder Import-Statements. Anhand der Fehlermeldung ist der Entwickler in der Lage, das entsprechende Template zur Codegenerierung zu finden, und er kann das Template beziehungsweise weitere Komponenten des Codegenerators anpassen. Eine ähnliche Hilfsklasse existiert auch zum Vergleich von Grammatiken und ermöglicht dort ein analoges Vorgehen.

Ist die Regel in Objektdiagramm-Notation erstellt und erfüllt sie die Testfälle, so erstellt der Entwickler im Projekt `TFRuleLanguageGenerationPrototype` eine äquivalente Regel in konkreter Syntax. Für diese muss er gegebenenfalls die handgeschriebene Grammatik `StatechartTF.mc` anpassen, ebenso muss er den Übersetzer in Objektdiagramm-Notation `Rule2OD` erweitern. Die Grammatik testet er mit der neu erstellten Regel, gegebenenfalls auch mit weiteren Regeln. Den Codegenerator testet er, indem er die generierte Regel in Objektdiagramm-Notation `EliminateDo.mtod` mit der handgeschriebenen Regel aus dem Projekt `TFODRules` vergleicht.

Als letzten Schritt passt er im Projekt `TFRuleLanguageGeneration` den Generator der Regelsprachen an. Dessen Ausgabe vergleicht er mit den zuvor im Prototyp-Projekt erstellten Grammatik- und Transformator-Dateien. Laufen auch diese Tests durch, so ist das neue Feature vollständig umgesetzt und validiert und steht fortan in allen generierten Transformationssprachen zur Verfügung.

Für die Auslieferung an Nutzer kommen in diesem Projekt-Setup nur die Generator-Projekte, aber nicht die Prototyp-Projekte in Betracht. Letztere dienen lediglich der testgetriebenen Entwicklung und Qualitätssicherung und enthalten keine Artefakte, die zur Verwendung durch den Nutzer bestimmt sind.

9.2.2. Systemtest

Die einzelnen Projekte erlauben es in der beschriebenen Struktur noch nicht, das Zusammenspiel mehrerer Generatoren zu testen, oder in einem automatisierten Ablauf den Generator der Regelsprache zu generieren und dann zu verwenden. Diese Maßnahmen zur Validierung sind jedoch für die Gesamtheit der Transformationsprojekte unerlässlich, und diese Schritte stellen gleichzeitig einen zusätzlichen Regressionstest für die Sprachen und Werkzeuge dar.

Dieser Systemtest wurde innerhalb des Projektes `TFRuleLanguageGeneration` realisiert. Dieses Projekt ist für den Test am besten geeignet, da die generierten Sprachen und Werkzeuge sowohl den mehrstufigen Generierungsprozess mit den Regeln in Objektdiagramm-Nota-

tion als Zwischenprodukt verwenden (vgl. Abschnitt 7.3) als auch die Erweiterungen zur Einbettung in die Kontrollflusssprache enthalten (vgl. Abschnitt 7.4) sowie deren Codegenerator benutzen.

Unter Verwendung aller genannten Komponenten führt der Test, der als Ziel `systemtest` im Build-Skript des Projektes abgelegt wurde, die folgenden Schritte vollautomatisch, also gänzlich ohne Benutzerinteraktion, durch.

1. Die Transformationssprache für Statecharts inklusive der Einbettung in die Kontrollflusssprache sowie die zugehörigen Werkzeuge werden aus der Grammatik der Statechartsprache generiert. Dies schließt die Generierung des Java-Codes, der Grammatiken und Language-Dateien, Codegenerierung aus diesen Artefakten sowie Kompilierung des Codes ein.

2. Diese generierten Artefakte werden Tests unterzogen. So wird etwa die Parsebarkeit mit verschiedenen Transformationen überprüft.

3. Es wird aus mehreren Transformationsregeln und -modulen Java-Code generiert, unter anderem aus dem in Anhang D abgebildeten Modul zur Vereinfachung hierarchischer Statecharts. Dieser generierte Code wird anschließend mit dem Java-Compiler übersetzt.

4. Die aus den Transformationsmodulen generierten Klassen werden in JUnit-Testfällen ausgeführt. Anschließend werden strukturelle Integritätsbedingungen auf dem Resultat der Transformation überprüft. Diese Bedingungen sichern vor allem die Existenz beziehungsweise Nichtexistenz von Elementen, die von den Transformationen erzeugt oder gelöscht werden müssen. Unter anderem werden Zustände, Transitionen, Anweisungen und verschiedene Ausdrücke sowie deren Attribute überprüft.

Der erfolgreiche Durchlauf des Tests sichert die zumindest grundlegende Funktionalität aller für die Generierung der Transformationssprachen wesentlichen Komponenten, insbesondere aber deren Zusammenspiel. Eine bessere Überdeckung der Funktionen einzelner Komponenten findet sich in den Modul- und Integrationstests der jeweiligen Projekte beziehungsweise der zugehörigen Prototyp-Projekte.

9.3. Testgetriebene Weiterentwicklung und Wartung

Der im letzten Abschnitt beschriebene Ansatz zur testgetriebenen Entwicklung erwies sich zwar aufgrund der Möglichkeit, in den Prototypen Änderungen auszuprobieren und erst später den Generator anzupassen, als gut geeignet für den Beginn der Entwicklung der Transformationsprojekte. Die Forderung syntaktischer Äquivalenz zwischen den Prototypen und dem Generat erwies sich jedoch mit längerer Projektlaufzeit als unflexibel.

Weil der Code verschiedener Prototyp-Klassen in weiten Teilen redundant war, führten auch korrekte Änderungen am Generator dazu, dass eine Vielzahl von Testfällen angepasst werden musste, obwohl sich das Verhalten der Prototyp-Klassen im Bezug auf die für sie erstellten Testfälle nicht verändert hatte. Mit steigender Anzahl der Funktionen in den Generatoren stieg

auch die Anzahl der Prototyp-Klassen und Testfälle, sodass diese mangelnde Flexibilität eine agile Entwicklung zunehmend schwierig machte.

Letztlich führte dieser steigende Aufwand dazu, dass von einem vollständigen strukturellen Vergleich der generierten Artefakte mit den Prototypen zu einer Validierung ausgewählten Verhaltens des generierten Codes übergegangen wurde. Da die Erstellung einer Referenzimplementierung außerdem vergleichsweise aufwändig erschien, wurden die prototypenbasierte Qualitätssicherung durch Tests ersetzt, die gegen den generierten Code entwickelt wurden. Hierfür konnte im Wesentlichen auf die existierenden Tests für die Prototyp-Klassen zurückgegriffen werden.

9.3.1. Testen einfacher Generatoren

Das Vorgehen nach dieser Umstellung ist am Beispiel der Sprache für Transformationsregeln in Objektdiagramm-Notation in Abbildung 9.3 dargestellt. Für die Kontrollflusssprache ist das Vorgehen ähnlich, während das Testen des Generators für Regelsprachen wegen des generierten Codegenerators eine Erweiterung dieser Methode erfordert, die später noch erläutert wird.

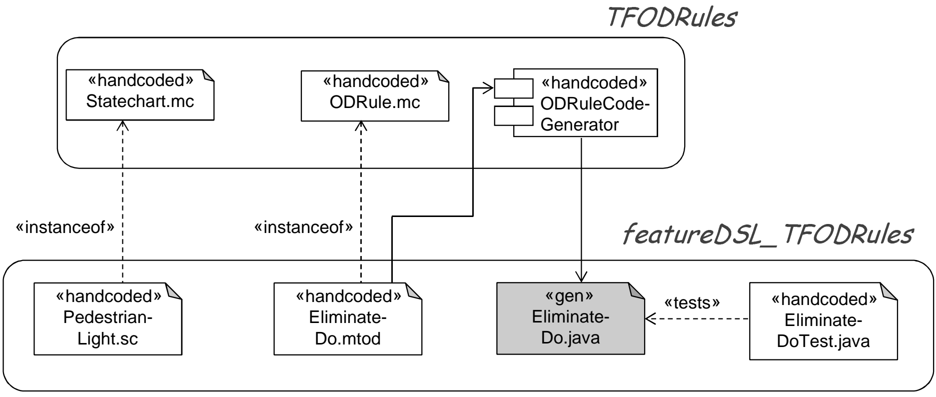


Abbildung 9.3.: Projektstruktur für die Weiterentwicklung und Wartung

Um eine Erweiterung der objektdiagrammbasierten Regelsprache vorzunehmen, schreibt der Entwickler zunächst einen Testfall, der die Änderungen, die durch die Ausführung der Regel an einem Modell vorgenommen werden müssen, überprüft. Diesen Testfall legt er im Projekt `featureDSL_TFODRules` ab. Zu diesem Zeitpunkt lässt sich der Testfall noch nicht übersetzen, da die Java-Klasse zur Regel, im Beispiel in der Datei `EliminateDo.java`, noch nicht existiert.

Anschließend erstellt er in der Datei `EliminateDo.mtod` die Regel in Objektdiagramm-Notation. Nachdem die notwendigen Erweiterungen für das neue Feature am Generator vorgenommen wurden, kann aus der Regel Code generiert werden.

Der generierte Code sollte sich übersetzen lassen, ansonsten liegt ein Fehler im Generator vor. Ebenfalls übersetzen lassen muss sich jetzt der Testfall für die Regel.

Lassen sich alle gezeigten Java-Klassen kompilieren, so kann der Testfall ausgeführt werden. Je nach Resultat des Tests sind dann weitere Änderungen am Generator, an der Regel oder am Test erforderlich, oder die Implementierung des neuen Features ist abgeschlossen.

Zur Ausführung des Tests muss der Entwickler lediglich das entsprechende Ziel im Build-Skript des Projekts `featureDSL_TFODRules` aufrufen. Die Generierung des Java-Codes aus der Regel und der Aufruf des Compilers werden durch das Abhängigkeitsmanagement automatisch in der richtigen Reihenfolge ausgelöst.

9.3.2. Testen des Generators für generierte Regelsprachen

Mit dem zuvor beschriebenen Projekt-Setup werden Codegeneratoren dadurch validiert, dass ihr generierter Code übersetzt und ausgeführt und sein Verhalten geprüft wird. In den bisher betrachteten Fällen erfolgt die Validierung des generierten Codes durch Testfälle, die eben diesen Code ausführen.

Handelt es sich beim generierten Code aber ebenfalls um einen Codegenerator, so kann dessen Validierung ebenfalls nach dem oben beschriebenen Ansatz erfolgen. Für das Beispiel der generierten Transformationssprachen führt dies zu folgender Argumentation für die Korrektheit des Regelsprachengenerators: Der Regelsprachengenerator arbeitet nur dann korrekt, wenn der generierte Codegenerator, der Transformationen in Java-Code übersetzen kann, korrekt arbeitet. Dieser Codegenerator arbeitet wiederum nur dann korrekt, wenn sich der aus Transformationsregeln generierte Code korrekt verhält.

Die Korrektheit des generierten Java-Codes aus Transformationsregeln ist also ein plausibles Argument und eine notwendige Bedingung für die Korrektheit des Regelsprachengenerators. Sie ist jedoch keine hinreichende Bedingung; dies liegt aber in der Natur von Software-Tests, die in der Regel keine vollständige Korrektheit des Systems nachweisen können, sondern es nur unter Betrachtung ausgewählter Funktionen und Eingaben prüfen.

Das Projekt-Setup zum Testen des Regelsprachengenerators zeigt Abbildung 9.4. Wie bereits bekannt sind die Projektgrenzen durch gerundete Rechtecke dargestellt.

Zur Erweiterung des Generators spielen drei Projekte eine Rolle: Im Projekt `TFRuleLanguageGeneration` liegt der Generator für die Regelsprachen. Das Projekt `featureDSL_TFRuleLanguageGeneration` enthält die handgeschriebenen Grammatiken von Beispiel-DSLs, im gezeigten Fall die Statechart-Grammatiken. Es enthält außerdem die aus diesen Grammatiken generierten Transformationssprachen, also deren Grammatiken, Klassen der abstrakten Syntax, Parser, Werkzeuge und Codegeneratoren. Für Transformationen in Statechart-Notation ist dieser Generator in Abbildung 9.4 als Komponente `Rule2Java` dargestellt. In dem Projekt `featureDSL_GeneratedTFLanguages` werden konkrete Transformationsregeln abgelegt. Auch das Generat aus diesen Regeln und die Testfälle hierfür liegen in diesem Projekt.

Zur testgetriebenen Entwicklung einer Änderung am Generator erstellt der Entwickler zunächst einen Testfall `EliminateDoTest` für den generierten Code aus einer konkreten Transformationsregel im Projekt `featureDSL_GeneratedTFLanguages`. Anschließend passt er die Implementierung des Regelsprachengenerators in `TFRuleLanguageGeneration` an.

Zum Testen der Änderungen führt er das Ziel `test` im Build-Skript zu `featureDSL_GeneratedTFLanguages` aus. Das Abhängigkeitsmanagement sorgt dafür, dass zunächst das Projekt `TFRuleLanguageGeneration` übersetzt wird. Anschließend wird der aktualisierte Generator verwendet, um in `featureDSL_TFRuleLanguageGeneration` die Grammatik beziehungsweise den Parser sowie den Codegenerator der Transformationssprache zu erstellen. Diese werden ebenfalls kompiliert und als Codegenerator für die Transformationsregeln in

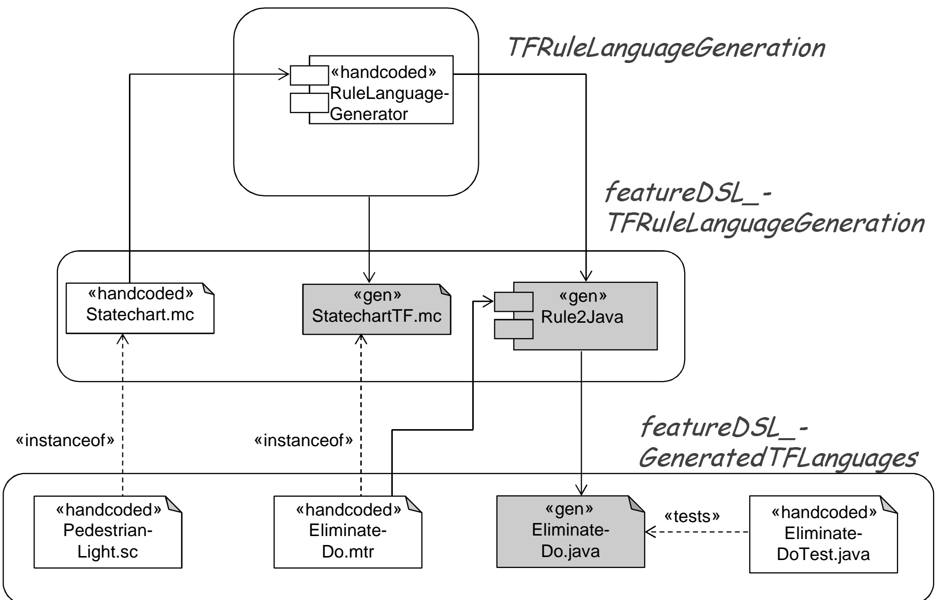


Abbildung 9.4.: Projektstruktur für den Generator des Codegenerators

`featureDSL_GeneratedTFLanguages` verwendet. Der aus den hier abgelegten Transformationsregeln generierte Code sowie der neue Testfall werden ebenfalls kompiliert. Als letzter Schritt wird der Testfall mit Hilfe des JUnit-Frameworks ausgeführt.

Dieses Verfahren wird so lange iteriert, bis kein Testfall fehlschlägt. Damit ist die testgetriebene Entwicklung eines Features für den Regelsprachengenerator erfolgreich beendet.

Zusätzlich kann es erforderlich sein, den Codegenerator oder die Sprache für Regeln in Objektdiagramm-Notation anzupassen oder zu erweitern. Ist dies der Fall, so wird dieser Schritt typischerweise vor dem oben beschriebenen Ablauf durchgeführt. Das Vorgehen dabei entspricht der in Abschnitt 9.3.1 präsentierten Methodik.

9.4. Vor- und Nachteile der Ansätze

Bei der Entwicklung der Modelltransformationsengine für MontiCore kamen beide bisher in diesem Kapitel vorgestellten Methoden zur testgetriebenen Entwicklung zum Einsatz. In diesem Abschnitt sollen ihre Vor -und Nachteile noch einmal gegenübergestellt werden und die Gründe für den Wechsel zum FeatureDSL-basierten Ansatz dargelegt werden.

Der in Abschnitt 9.2 vorgestellte Ansatz ermöglichte und erforderte es, den zu generierenden Code zunächst vollständig und testgetrieben von Hand zu schreiben. Solange nicht exakt abzusehen war, wie dieser Code auszusehen hat, bot dieses Verfahren folgende Vorteile:

- Änderungen am Code konnten im Java-Code mit Werkzeugunterstützung durch die integrierte Entwicklungsumgebung ausgeführt werden. Diese bietet Komfortfunktionen wie

Syntax-Highlighting, sofortige Anzeige von Fehlern ohne expliziten Aufruf eines Generators oder Compilers und vorgefertigte Refactoring-Operationen.

- Fehler im Generator waren von Fehlern beim Entwurf des generierten Codes unterscheidbar, da es hierfür unterschiedliche Testfälle gab.
- Fehlermeldungen von Testfällen für den Generator waren vergleichsweise präzise und gaben die Fehlerstelle im generierten Code beziehungsweise die entsprechende Stelle im Prototypen an.
- Nachdem der Prototyp getestet wurde, war der Aufwand zur Anpassung des Generators erfahrungsgemäß gering: In den meisten Fällen konnte der Code aus den Prototypen unter Einfügung von Variablen an einigen Stellen in die Templates kopiert werden.

Allerdings zogen die Änderungen am Generator oft die Notwendigkeit mit sich, existierenden Prototypen anzupassen. Zwar hatte sich das getestete Verhalten dieser Prototypen nicht geändert, wohl aber die Struktur des generierten Codes, die gegen die Prototypen vergleichen wurde. Der Aufwand dieser Anpassungen stieg mit zunehmender Anzahl von Testfällen und Prototypen, sodass auf die in Abschnitt 9.3 vorgestellte Methodik mit folgenden Vorteilen gewechselt wurde:

- Die Testfälle nach dieser Methode sind weniger redundant. Bei Ergänzungen in den Templates müssen nicht alle aus diesem Template generierten Prototypen angepasst werden, sondern nur die Testfälle, die sich auf das von dieser Ergänzung betroffene Verhalten des generierten Codes beziehen.
- Modifikationen, die das beobachtbare Verhalten des generierten Codes nicht ändern, führen nicht zu fehlgeschlagenen Tests.
- Die händische Kodierung der Prototypen entfällt.

Die Umstellung auf das neue Verfahren war nur mit moderatem Aufwand verbunden, da die existierenden Testfälle für die Prototypen im neuen Verfahren als Tests für den generierten Code genutzt werden können.

Rückblickend lautet das Fazit, das basierend auf der Entwicklung der Transformationsprojekte in MontiCore zu diesen Vorgehensweisen gezogen werden kann, dass die Entwicklung mit dem prototypbasierten Ansatz nur für die Initiierungsphase komplexer Codegeneratoren, bei denen der Aufbau des generierten Codes nicht vorab klar ist, empfehlenswert ist. Für die Entwicklung einfacher Generatoren oder die Wartung laufender Projekte empfiehlt sich dagegen eher die FeatureDSL-basierte Methode, wobei ein Wechsel hin zu dieser Methode mit vertretbarem Aufwand erfolgen kann.

Kapitel 10.

Epilog

In der vorliegenden Arbeit wurde eine Engine zur Transformation MontiCore-basierter Modelle geschaffen. Die wichtigsten Eigenschaften dieser Engine sowie die Ergebnisse der Arbeit werden als Zusammenfassung der vorherigen Kapitel in diesem Kapitel aggregiert. Anschließend wird ein Ausblick auf mögliche Erweiterungen und Anschlussarbeiten gegeben. Dabei werden sowohl eng verwandte Forschungsfelder betrachtet, die im Rahmen dieser Arbeit nicht adressiert wurden, als auch neue Fragestellungen erläutert, die sich während der Anfertigung ergeben haben, die Erweiterungen des hier vorgestellten Ansatzes betreffen, oder deren Untersuchung durch die Ergebnisse dieser Arbeit überhaupt erst ermöglicht wird.

10.1. Zusammenfassung

Das primäre Ziel dieser Arbeit war es, Experten für eine gegebene Modellierungssprache eine Transformationssprache an die Hand zu geben, die für Nutzer der Modellierungssprache ohne tiefgehende Kenntnisse in der Sprachentwicklung verständlich ist. Dadurch sollte die Nützlichkeit solcher Sprachen für Domänenexperten verbessert und die Einbeziehung solcher Experten in Softwareentwicklungsprozesse erleichtert werden. Dieses Ziel wurde durch die weitgehende Verwendung der konkreten Syntax der zugrunde liegenden Modellierungssprache in Transformationen erreicht.

Die Arbeit fand innerhalb des MontiCore-Projektes statt: Zum einen werden mit den Transformationen Modelle modifiziert, deren Sprache mit MontiCore erstellt wurde. Zum anderen wurde das Werkzeug MontiCore auch zur Definition von Sprachen innerhalb der Transformationsengine verwendet, und das DSLTool-Framework von MontiCore kann zur Ausführung von Transformationen genutzt werden.

Ein weiteres Ziel war es, den Aufwand zur Erstellung solcher Sprachen gegenüber der Durchführung eines vollständigen Sprachentwicklungsprozesses signifikant zu reduzieren. Dieses Ziel wurde durch eine Kombination generierter Komponenten, die aus Artefakten der zugrunde liegenden Sprache abgeleitet werden, und generischen Komponenten, die für Transformationen auf beliebigen Modellen wiederverwendbar sind, erreicht.

Die verbesserte Nutzbarkeit von Transformationen ist neben der Steigerung der Effizienz bei der Ausführung, der Qualitätssicherung und weiteren Themen eines der spannendsten und aktuellen Forschungsfelder im Bereich der Modelltransformationen und modellgetriebener Softwareentwicklung. Mit einem geringen Aufwand zur Erstellung von Transformationssprachen nach

dem hier vorgestellten Ansatz wird auch die Kosteneffizienz bei der Einführung von Transformationen innerhalb eines Softwareentwicklungsprozesses berücksichtigt.

Als durchgängiges Beispiel zum Nachweis der Nutzbarkeit der Arbeitsergebnisse wurden eine an das UML-Profil UML/P angelehnte Statechartsprache sowie basierend auf dieser Sprache ein Verfahren zur Vereinfachung hierarchischer Statecharts verwendet. Dieses durchgängige Beispiel ist keiner bestimmten fachlichen Domäne eindeutig zuzuordnen, jedoch kann die zustandsbasierte Implementierung von Systemen als technische Domäne betrachtet werden, und die Wahl einer vielfältig anwendbaren Modellierungssprache mindert nicht die Aussagekraft in Bezug auf die Anwendbarkeit des Ansatzes.

Als Zielplattform für die Codegenerierung aus Transformationsregeln in konkreter Syntax wurde in dieser Arbeit zunächst eine Sprache entwickelt, mit der Graphersetzungsregeln durch zwei Objektdiagramme als linke und rechte Regelseite beschrieben werden können. Zusätzlich können weitere Constraints für die Anwendbarkeit der Regeln angegeben werden oder nichtisomorphe Matches zugelassen werden. Durch die Abbildung auf Graphersetzungsregeln besteht eine solide und formale Basis für die Modelltransformationen in MontiCore. Des weiteren zeigt die Abbildung auf eine Sprache, die konzeptionell vielen verfügbaren Transformationssprachen ähnelt, die Anwendbarkeit des in dieser Arbeit entwickelten Ansatzes auch auf andere Werkzeuglandschaften.

Die Regelsprachen in domänenspezifischer Syntax verwenden neben den aus der zugrunde liegenden Modellierungssprache bekannten Elementen auch transformationsspezifische Konstrukte, etwa zur Auszeichnung spezieller Arten von Objekten in den Regeln oder zur Kennzeichnung von Ersetzungen. Von zentraler Bedeutung ist dabei die Verwendung von Schemavariablen, die als Platzhalter für variable Elemente im zu transformierenden Modell eingesetzt werden. Neben den Elementen, die sich aus der konkreten Syntax der zugrunde liegenden Sprache ergeben, kann auch auf die abstrakte Syntax des zu transformierenden Modells Bezug genommen werden. Dies kann zwar die Verständlichkeit für Domänenexperten beeinträchtigen, erhöht jedoch die Ausdrucksmächtigkeit der Sprache, indem es beispielsweise die Verwendung abstrakter Typen oder die Einführung bestimmter Constraints überhaupt erst möglich macht. Eine integrierte Notation der linken und rechten Regelseite macht zum einen die Regeln kompakter, zum anderen macht sie eindeutige Identifier für Objekte in Transformationsregeln überflüssig, die sonst in bestimmten Fällen für die Zuordnung zwischen den Elementen beider Regelseiten erforderlich wären.

Regeln aller generierten Sprachen, aber auch in der auf Objektdiagrammen basierenden Sprache, können in eine generische Kontrollflusssprache eingebettet werden. Dadurch können mehrere Regeln, die kleine Transformationsschritte beschreiben, zu komplexen Programmen verbunden werden. Die vorgestellte Kontrollflusssprache ist syntaktisch an Java angelehnt. Neben den wichtigsten aus Java bekannten Elementen bietet sie auch eine Kontrollstruktur für Fixpunktiterationen an. Als herausragender Unterschied zu Java ist vor allem das Backtracking nach fehlgeschlagenen Anweisungen auszuführen. Darüber hinaus unterscheiden sich Methodenaufrufe insofern, als dass für die Parameterübergabe das Verfahren Call-by-Reference anstatt des in Java üblichen Call-by-Value zum Einsatz kommt.

Zum Nachweis der Generierbarkeit domänenspezifischer Transformationssprachen wurde ein Regelsprachengenerator erstellt. Am Beispiel der Statechartsprache wurde gezeigt, wie ein sol-

cher Generator aus der Grammatik einer gegebenen Sprache eine dazu passende Transformationssprache erstellen kann. Neben der eigentlichen Sprache können auch Basiswerkzeuge zur Verarbeitung der Transformationen, insbesondere zur Ausführung der Codegenerierung, verwendet werden. Neben den generierten Anteilen, allen voran der Regelsprache, enthalten die Transformationssprachen und die zugehörigen Werkzeuge auch generische Komponenten, etwa die Kontrollflusssprache im Frontend oder den Codegenerator für Ersetzungsregeln in Objektdiagramm-Notation als Teil eines mehrstufigen Codegenerators im Backend.

Sowohl der Erstellungsprozess der Transformationssprache als auch die Codegenerierung aus Transformationsprogrammen sind innerhalb der Softwareentwicklung mit MontiCore gut nutzbar, da sie in entsprechende Werkzeuglandschaften leicht eingebunden werden können. Im Spracherstellungsprozess laufen die Generierung der Transformationssprache und der zugehörigen Werkzeuge innerhalb von Workflows, die von einem DSLTool gekapselt werden. Im Entwicklungsprozess konkreter Transformationen ist die Generierung von Java-Code aus dem Transformationen ebenfalls in Workflows und DSLTools gekapselt, die gemeinsam mit der Transformationssprache generiert werden. Die Verwendung dieses generierten Codes ist dann wiederum vom Einsatzzweck der Transformationen abhängig, sodass die Definition und Implementierung eines entsprechenden Nutzungsprozesses dem Anwender der Transformationen überlassen bleibt.

Die Entwicklung der Transformationsprojekte innerhalb von MontiCore erfolgte testgetrieben und agil. In der Initiierungsphase wurde ein Projekt-Setup gewählt, bei dem zunächst Prototypen des zu generierenden Codes von Hand erstellt wurden. Anschließend wurde der Generator angepasst oder erweitert, sodass das Resultat der Codegenerierung dem Prototypen entsprach. Dieser Ansatz erlaubte es vor allem, Änderungen am zu generierenden Code schnell und einfach zu testen und ein Grundverständnis für das gewünschte Verhalten der Transformationsregeln aufzubauen. Im Verlauf des Entwicklungsprozesses wurde auf einen FeatureDSL-basierten Ansatz umgestellt, bei dem die Qualitätssicherung durch Testen des generierten Codes erfolgt. Diese neue Entwicklungsmethode erlaubt durch geringere Redundanz der Tests in Zukunft eine agilere und effizientere Weiterentwicklung der Transformationsprojekte. Der aus MontiCore bereits bekannte Ansatz der FeatureDSLs wurde für die Validierung generierter Generatoren erweitert. Bei der Umstellung konnte der Großteil der bestehenden Testfälle weiter verwendet werden, sodass hierdurch kein großer Zusatzaufwand entstand.

10.2. Ausblick

Über die umgesetzten Features hinaus sind noch einige Anschlussarbeiten an dieses Forschungsvorhaben möglich. Diese lassen sich unterteilen in Anwendungen des hier vorgestellten Ansatzes, in konzeptuell neue oder verschiedene, aber verwandte Transformationsansätze, sowie in technische Erweiterungen der Transformationsengine.

10.2.1. Anwendungen der Ergebnisse dieser Arbeit

Mit den in den vorangegangenen Kapiteln präsentierten Ergebnissen lässt sich die Beantwortung verschiedener wissenschaftlicher Fragestellungen angehen, von denen an dieser Stelle eine Auswahl vorgestellt wird.

Akzeptanz domänenspezifischer Transformationssprachen

Durch die Ergebnisse dieser Arbeit wurden die Voraussetzungen geschaffen, für Domänenexperten verständliche Transformationssprachen mit geringem Aufwand zu erstellen. In welchem Umfang eine Nutzung solcher Sprachen durch Domänenexperten tatsächlich erfolgt, inwiefern hierdurch die Einbeziehung der Experten in die Softwareentwicklung verbessert werden kann, oder ob die Sprachen auch für Werkzeug- und Sprachentwickler einen Mehrwert bieten, kann aufbauend auf dieser Arbeit in zukünftigen Projekten untersucht werden.

So ist beispielsweise durch Domänenexperten die Beschreibung von Editor-Makros in Transformationen denkbar. Eine Kombination aus textuellen Modellen und solchen Transformationen könnte als domänenspezifischer Ersatz für relationale Datenbanken oder Tabellenkalkulationen und zugehörige Makros etabliert werden.

Sprach- und Werkzeugentwickler könnten die Transformationssprachen nutzen, unter anderem, weil auch für diese Nutzergruppe die Verständlichkeit gegenüber generischen Transformationssprachen verbessert wurde, aber auch, weil die Transformationsregeln in konkreter Syntax nach bisherigen Erfahrungen meist kompakter sind.

Ob und in welchem Umfang diese Vorteile von den Nutzern ebenso empfunden werden, ist bisher nicht untersucht worden. Jedoch sind mit den Ergebnissen dieser Arbeit die Voraussetzungen zur Durchführung einer solchen Untersuchung, welche die Nützlichkeit der hier präsentierten Sprachen untermauern oder widerlegen kann, erfüllt.

Bibliotheken von Transformationen

Die Transformationsregeln stellen zunächst kleine Modifikationsschritte dar, die durch die Kontrollflusssprache zu sinnvollen Einheiten verbunden werden können. Eine herausragend wichtige Klasse sinnvoller Einheiten sind semantikerhaltende oder -verfeinernde Transformationen.

Um die Wiederverwendung komplexer Transformationen zu erhöhen, können qualitätsgesicherte Transformationen in Bibliotheken abgelegt werden, die für weitere Anwender einer Sprache zugänglich sind. Der Aufwand zur Nutzung von Transformationen könnte so zusätzlich reduziert werden.

Die Einführung von Transformationsbibliotheken ist grundsätzlich zwar nicht an die Ergebnisse dieser Arbeit gebunden. Jedoch ist mit diesen Ansätzen zum einen die Einbindung von Domänenexperten in die Entwicklung der Bibliothek möglich. Zum anderen ist die in dieser Ausarbeitung vorgestellte Engine Voraussetzung, um Graphersetzungsregeln und auf solchen Regeln basierende Transformationsmodule auf MontiCore-basierten Modellen beschreiben zu können.

10.2.2. Konzeptuell verwandte Problemstellungen

Neben den in dieser Arbeit beschriebenen In-Place-Transformationen wird in einer Vielzahl von Forschungsprojekten auch die Behandlung von Modell-zu-Modell-Transformationen beschrieben. Oft handelt es sich um exogene Transformationen, die ein Modell einer Ausgangssprache in ein Modell einer anderen Sprache überführen. Die Beschreibung solcher Transformationen

ist mit dem in dieser Arbeit vorgestellten Ansatz zunächst nicht möglich, da die generierten Transformationssprachen nur Modelle einer Sprache verarbeiten können.

Eine mögliche Erweiterung hin zu exogenen Transformationen wäre die Erstellung einer Vereinigungssprache aus Quell- und Zielsprache. Das Quellmodell einer Transformation könnte dann als Instanz dieser Vereinigungssprache interpretiert werden. Mit einer aus der Vereinigungssprache abgeleiteten Transformationssprache könnten Transformationen beschrieben werden, die Elemente der Quellsprache durch korrespondierende Elemente der Zielsprache ersetzen. Zum Ende der Transformation enthält das Modell nur noch Elemente der Zielsprache. Es kann dann als eine Instanz dieser Sprache interpretiert und mit den entsprechenden Werkzeugen weiter verarbeitet werden. Einschränkungen für dieses Ansatz ergeben sich unter anderem bei der Parsebarkeit der Vereinigungssprache, da die Klasse der LL (k)-Sprachen unter direkt vorgenommener Vereinigung nicht abgeschlossen ist [RS70].

Eine weitere Möglichkeit zur Beschreibung von Modell-zu-Modell-Transformationen sind die Paar-Grammatiken, auf die sich einige der in dieser Arbeit vorgestellten Ansätze übertragen lassen. Anstatt der in [Pra71] vorgestellten String-zu-Graph-Übersetzung ist für MontiCore-basierte Sprachen aber auch eine String-zu-String-Übersetzung denkbar.

Erweitert man die Paar-Grammatiken um einen Korrespondenzgraphen, so gelangt man zu Tripel-Graph-Grammatiken (TGGs). Um mit TGGs [Sch94] Transformationen in konkreter Syntax spezifizieren zu können, wäre vor allem zu untersuchen, wie die Angabe des Korrespondenzgraphen ohne Bezug auf die abstrakte Syntax des Quell- und Ziel-Modells erfolgen kann. Ähnliches gilt auch für den OMG-Standard für deklarative Modell-zu-Modell-Transformationen, QVT relational [OMG08b].

10.2.3. Technische Erweiterungen

Die folgenden Erweiterungen stellen Ergänzungen der im Rahmen dieser Arbeit entwickelten Projekte dar. Sie dienen größtenteils der Komfortverbesserung bei der Entwicklung von Transformationen, teilweise aber auch der Erweiterung der Funktionalität oder der Anwendbarkeit der Ansätze auf weitere Modellierungssprachen mit von den in dieser Arbeit vorausgesetzten Eigenschaften abweichenden Charakteristika.

Expression-Sprache für primitive Datentypen

In den Transformationsregeln können Werte für primitive Datentypen bisher nur aus existierenden Modellelementen übernommen werden. Komplexe Wertberechnungen müssen jedoch außerhalb der Regeln in der Kontrollflusssprache erfolgen. Eine Expression-Sprache für solche Werte würde es erlauben, Werte auch mit einer vorgegebenen Menge an Operationen in den Regeln zu berechnen. Solche Operationen könnten beispielsweise die Konkatenation von Strings sowie boolesche oder arithmetische Operationen sein.

Bezeichnerbindung für qualifizierte Namen

Der Vergleich von Bezeichnern beim Pattern-Matching ist derzeit auf einfache Bezeichner in einem globalen, flachen Namensraum, die nach dem Parsen im AST vorhanden sind, beschränkt.

Für Sprachen mit hierarchischen Namensräumen wäre aber ein Vergleich der qualifizierten Namen, die nach dem Aufbau der Symboltabellen dort zur Verfügung stehen, zweckmäßiger.

Parsebarkeit der Regeln, syntaktische/semantische Prädikate

Die generierten Grammatiken der Transformationssprachen haben sich bisher insoweit als parsebar erwiesen, als dass auch größere Programme, wie etwa die in Anhang D abgebildete Transformation auf Statecharts, verarbeitet werden können. Jedoch wurde noch nicht untersucht, unter welchen Bedingungen das Parsen problematisch ist. Schwierigkeiten sind hier beispielsweise zu erwarten, wenn die Namen von Typen der Schemavariablen und die Schlüsselwörter der Modellierungssprache identisch oder sehr ähnlich sind.

Mögliche Ansätze zur Verbesserung der Parsebarkeit wären die gezielte Erhöhung des Parser-Lookaheads und die Einführung syntaktischer oder semantischer Prädikate. Details hierzu finden sich in [Par07, Kap. 12-14] sowie speziell für MontiCore in [Kra10, Kap. 3.5.2]. Konflikte zwischen den Namen von Typen der Transformationssprache und exisitierende Schlüsselwörtern ließen sich durch konfigurierbare Präfixe der Namen entschärfen. Auch durch den Umstieg auf eine neue ANTLR-Version im Parsergenerator von MontiCore wäre eine Verbesserung der Parsebarkeit denkbar, da neuere Versionen einen echten k-Lookahead anstelle des derzeit verwendeten linear approximierten Lookaheads [Par93] verwenden.

Kontextbedingungen für Transformationsregeln in konkreter Syntax

Für die generierten Transformationssprachen findet derzeit noch keine vollständige Überprüfung der Kontextbedingungen in den Regeln statt. Diese Überprüfung könnte in zukünftigen Projekten umgesetzt werden.

Denkbar ist, dass es eine Reihe generischer Kontextbedingungen gibt, wie etwa Eindeutigkeit von Objektbezeichnern, und dass weitere Bedingungen aus den DSL-Grammatiken generiert werden können. In die letzte Kategorie würden beispielsweise Kontextbedingungen zum Typsystem fallen, da dies von den Typen der abstrakten Syntax der DSL abhängt, die ebenfalls aus der Grammatik abgeleitet werden.

Zusätzlich zu den regelinternen Kontextbedingungen existieren auch Bedingungen, die sich aus der Einbettung in die Kontrollflusssprache ergeben. Hier ist vor allem die Typsicherheit im Zusammenhang mit an die Regeln übergebenen Parametern zu erwähnen.

Generierung zusätzlicher Infrastruktur für MontiCore-Projekte

Infrastruktur zur Verarbeitung von Nutzern erstellter Eingabedateien existiert bisher für MontiCore-Grammatiken zur Generierung der Transformationssprachen sowie als generierte Infrastruktur zur Codegenerierung aus Transformationen.

Darüber hinaus ließe sich auch Basis-Infrastruktur zur Verwendung des generierten Codes aus Transformationen automatisch erzeugen, sofern die Umgebung, in der dieser Code eingesetzt wird, bekannt ist. Handelt es sich beispielsweise um eine MontiCore-basierte Werkzeuglandschaft, so könnten zu jeder Transformation mit einer `main`-Methode ein Workflow generiert werden, der die Transformation auf dem aktuellen `DSLRoot`-Objekt ausführt, und der leicht in

DSLTools eingebunden werden kann. Diese Workflows müssten vom Codegenerator der jeweiligen Transformationssprache erstellt werden, sodass hierfür eine Erweiterung des Generators der Codegeneratoren vorzunehmen wäre.

10.3. Abschließende Bemerkungen

Durch die vorliegende Arbeit wurde der Grundstein gelegt, um Modelltransformationen für ein breiteres Nutzerspektrum zugänglich und beherrschbar zu gestalten. Ob und inwiefern die Implementierung von Modelltransformationen tatsächlich in Zukunft durch Domänenexperten durchgeführt oder intensiver begleitet werden kann, ist eine noch zu klärende Frage, für deren Beantwortung hier eine Voraussetzung geschaffen wurde.

Die wichtigsten Ergebnisse der Arbeit und mögliche Projekte, die sich an dieses Dissertationsvorhaben anschließen können, wurden in diesem Kapitel vorgestellt. Dabei wurde vor allem im Ausblick eine Reduktion auf ausgewählte Aspekte vorgenommen. Viele weitere Anknüpfungspunkte sind denkbar, und die Wissenschaft lebt davon, dass sowohl das Stellen als auch die Beantwortung nachfolgender Fragen nicht nur vom Verfasser der ursprünglichen Arbeit ausgehen kann. Ebenso lebt sie von der Diskussion und von der Möglichkeit, ähnliche Ziele mit anderen Mitteln und Wegen zu erreiche, sodass sich zeigen wird, ob und wann in nachfolgenden Arbeiten weitergehende Konzepte präsentiert werden, die einen zusätzlichen Mehrwert zu den hier vorgestellten Ergebnissen darstellen, sowie wann und in welchen Domänen die Ergebnisse dieser Arbeit in Nutzungsprojekten aufgegriffen werden.

Neben den in dieser Arbeit in den Fokus gestellten Aspekten der Sprachen und Werkzeuge spielen bei der Durchführung solcher Projekte immer auch die an dieser Stelle weniger intensiv betrachteten Prozesse und Methoden sowie natürlich am allermeisten die beteiligten Personen eine herausragend wichtige Rolle. Die Akzeptanz dieser Arbeit und die Nützlichkeit für den Anwender sind unter Berücksichtigung dieser Aspekte sicherlich möglich, aber auch im Rahmen zukünftiger Projekte und Forschungsvorhaben zu untersuchen.

Literaturverzeichnis

[ABK07] Kyriakos Anastasakis, Behzad Bordbar, Jochen M. Küster. Analysis of Model Transformations via Alloy. In *Proc. of MoDeVVa 2007*, 2007.

[AK05] Colin Atkinson, Thomas Kühne. A Generalized Notion of Platforms for Model-Driven Development. In Sami Beydeda, Matthias Book, Volker Gruhn (Editoren), *Model-Driven Software Development*, Seiten 119–136. Springer, Berlin/Heidelberg, 2005.

[AK08] Malte Appeltauer, Günter Kniesel. Towards Concrete Syntax Patterns for Logic-based Transformation Rules. *Electron. Notes Theor. Comput. Sci.*, 219:113–132, November 2008.

[AKRS06] Carsten Amelunxen, Alexander Königs, Tobias Rötschke, Andy Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In *Proc. of European Conference on Model-Driven Architecture - Foundations and Applications (ECMDA-FA) 2006*, Lecture Notes in Computer Science (LNCS) 4066. Springer, 2006.

[ANT11] ANTLR Website `http://www.antlr.org`. Stand: 28.11.2011.

[Ant11] Apache Ant Website `http://ant.apache.org/`. Stand: 06.12.2011.

[ASU86] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[BBKR08] Jörg Bauer, Iovka Boneva, Marcos E. Kurbán, Arend Rensink. A Modal-Logic Based Graph Abstraction. In *ICGT '08: Proceedings of the 4th international conference on Graph Transformations*, Seiten 321–335. Springer, 2008.

[BCP+03] Margaret Burnett, Curtis Cook, Omkar Pendse, Gregg Rothermel, Jay Summet, Chris Wallace. End-User Software Engineering with Assertions in the Spreadsheet Paradigm. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, Seiten 93–103, Washington, DC, USA, 2003. IEEE Computer Society.

[Bec97] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1997.

[Bec04] Kent Beck. *JUnit Pocket Guide*. O'Reilly, 1 Auflage, 2004.

[BEH+87] Friedrich L. Bauer, Herbert Ehler, A. Horsch, Bernhard Möller, Helmuth Partsch, O. Paukner, Peter Pepper. *The Munich Project CIP, Volume II: The Program Transformation System CIP-S*, volume 292 of *Lecture Notes in Computer Science*. Springer, 1987.

[BEJ10] Enrico Biermann, Claudia Ermel, Stefan Jurack. Modeling the „Ecore to GenModel“ Transformation with EMF Henshin. In Pieter Van Gorp, Steffen Mazanek, Arend Rensink (Editoren), *Transformation Tool Contest*, Malaga, 2010.

[BEW04] Roswitha Bardohl, Claudia Ermel, Ingo Weinhold. GenGED – A Visual Definition Tool for Visual Modeling Environments. In John L. Pfaltz, Manfred Nagl, Boris Böhlen (Editoren), *Applications of Graph Transformations with Industrial Relevance(AGTIVE 2003), Revised Selected and Invited Papers*, volume 3062 of *Lecture Notes in Computer Science*, Seiten 413–419. Springer, 2004.

[BGF+10] Benoit Baudry, Sudipto Ghosh, Franck Fleurey, Robert France, Yves Le Traon, Jean-Marie Mottu. Barriers to Systematic Model Transformation Testing. *Commun. ACM*, 53:139–143, June 2010.

[Bis11] Bison – GNU Parser Generator.
`http://www.gnu.org/software/bison/`. Stand: 28.11.2011.

[BKG08] Gernot Veit Batz, Moritz Kroll, Rubino Geiß. A First Experimental Evaluation of Search Plan Driven Graph Pattern Matching. In *Applications of Graph Transformation with Industrial Relevance, Third International Symposium, AGTIVE 2007, Kassel*, Seiten 471–486. Springer, 2008.

[BKVV08] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, Eelco Visser. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Science of Computer Programming*, 72:52–70, 2008.

[BLS+09] Petra Brosch, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, Gerti Kappel, Werner Retschitzegger, Wieland Schwinger. An Example Is Worth a Thousand Words: Composite Operation Modeling By-Example. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, MODELS ’09, Seiten 271–285, Berlin, Heidelberg, 2009. Springer-Verlag.

[BSM+03] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, Timothy J. Grose. *Eclipse Modeling Framework*. Addison-Wesley, 2003.

[BW06] Thomas Baar, Jon Whittle. On the Usage of Concrete Syntax in Model Transformation Rules. In *Proc. of Sixth International Andrei Ershov Memorial Conference, Perspectives of System Informatics (PSI)*, Lecture Notes in Computer Science, Seiten 84–97, 2006.

[CB07] Ivica Crnkovic, Antonia Bertolino (Editoren). *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*. ACM, 2007.

[CGM06] Tal Cohen, Joseph (Yossi) Gil, Itay Maman. JTL: The Java Tools Language. *SIGPLAN Not.*, 41:89–108, October 2006.

[CGR08] María V. Cengarle, Hans Grönniger, Bernhard Rumpe. System Model Semantics of Statecharts. Informatik-Bericht 2008-04, Technische Universität Braunschweig, 2008.

[CH06] Krzysztof Czarnecki, Simon Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[CHK+93] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, Alan Turransky (Editoren). *Watch what I do: Programming by Demonstration*. MIT Press, Cambridge, MA, USA, 1993.

[CLSAT09] Daniel Calegari, Carlos Luna, Nora Szasz, Álvaro Tasistro. Experiment with a Type-Theoretic Approach to the Verification of Model Transformations. In *ChWFM'09: Proceedings del II Workshop Chileno de Métodos Formales*, 2009.

[Coo71] Stephen A. Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of the third Annual ACM Symposium on Theory of Computing*, STOC '71, Seiten 151–158, New York, NY, USA, 1971. ACM.

[CYD+08] Jiefeng Cheng, Jeffrey Xu Yu, Bolin Ding, Philip S. Yu, Haixun Wang. Fast Graph Pattern Matching. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, Seiten 913–922, Washington, DC, USA, 2008. IEEE Computer Society.

[DDFV09] Marcos Didonet Del Fabro, Patrick Valduriez. Towards the Efficient Development of Model Transformations Using Model Weaving and Matching Transformations. *Software and Systems Modeling*, 8(3):305–324, July 2009.

[Dij68] Edsger W. Dijkstra. Go To Statement Considered Harmful. *Communications of the ACM archive*, 11(3):147–148, 1968. letter to the Editor.

[Dör95] Heiko Dörr. *Efficient Graph Rewriting and Its Implementation*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1995.

[Ecl] Eclipse Foundation, Inc. Eclipse - The Eclipse Foundation Open Source Community Website. `http://www.eclipse.org`.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[EHK+97] Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner, Andrea Corradini. Algebraic Approaches to Graph Transformation - Part II: Single Pushout Approach and Comparison with Double Pushout Approach. In Grzegorz Rozenberg (Editor), *Handbook of Graph Grammars*, Seiten 247–312. World Scientific, 1997.

[EKT09] Karsten Ehrig, Jochen Küster, Gabriele Taentzer. Generating Instance Models from Meta Models. *Software and Systems Modeling*, 8(4):479–500, 2009.

[EM85] Hartmut Ehrig, B. Mahr. *Fundamentals of Algebraic Specification I*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1985.

[ERRS10] Hartmut Ehrig, Arend Rensink, Grzegorz Rozenberg, Andy Schürr (Editoren). *Graph Transformations – 5th International Conference, ICGT 2010, Enschede, The Netherlands, September 27 - October 2, 2010. Proceedings*, volume 6372 of *Lecture Notes in Computer Science*. Springer, 2010.

[FBMLT07] Franck Fleurey, Benoit Baudry, Pierre-Alain Muller, Yves Le Traon. Towards Dependable Model Transformations: Qualifying Input Test Data. *Journal of Software and Systems Modeling (SoSyM)*, 2007.

[Fla05] David Flanagan. *Java in a Nutshell - A Desktop Quick Reference*. O'Reilly, 2005.

[FNTZ00] Thorsten Fischer, Jörg Niere, Lars Torunski, Albert Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In *Selected Papers from the 6th International Workshop on Theory and Application of Graph Transformations*, Seiten 296–309, London, UK, 2000. Springer-Verlag.

[Fow10] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley, 2010.

[Fre11] FreeMarker Website `http://freemarker.org/`. Stand: 15.11.2011.

[GB03] Fredrik Gustafsson, Niclas Bergman. *MATLAB for Engineers Explained*. Springer, 2003.

[GBG+06] Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack, Adam Szalkowski. GrGen: A Fast SPO-Based Graph Rewriting Tool. In *Proc. of ICGT '06 (International Conference on Graph Transformation)*, Lecture Notes in Computer Science (LNCS) 4178, Seiten 383–397. Springer, 2006.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[GHR+03] Jens Grabowski, Dieter Hogrefe, György Réthy, Ina Schieferdecker, Anthony Wiles, Colin Willcock. *An Introduction to the Testing and Test Control Notation (TTCN-3)*, volume 42. Elsevier North-Holland, Inc., New York, NY, USA, 2003.

[GK03] Jeff Gray, Gabor Karsai. An Examination of DSLs for Concisely Representing Model Traversals and Transformations. In *Proceedings of HICSS*, Seite 325, 2003.

[Gle03] Sabine Glesner. Using Program Checking to Ensure the Correctness of Compiler Implementations. *Journal of Universal Computer Science (J.UCS)*, 9(3):191–222, March 2003.

[GMF11] Graphical Modeling Framework Website. `http://www.eclipse.org/gmf/`. Stand: 16.05.2011.

[GMP09] R. Grønmo, Birger Møller-Pedersen. Concrete Syntax-Based Graph Transformation. Research Report 389, Department of Informatics, University of Oslo, 2009.

[GMW97] David Garlan, Robert T. Monroe, David Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, Seiten 169–183, Toronto, Ontario, November 1997.

[Gol84] Robert Goldblatt. *Topoi, the Categorical Analysis of Logic*, volume 98 of *Studies in Logic and the foundations of mathematics*. North Holland, New York, 1984. `http://historical.library.cornell.edu/cgi-bin/cul.math/docviewer?did=Gold010&seq=3`.

[Grø09] R. Grønmo. *Using Concrete Syntax in Graph-based Model Transformations*. Doktorarbeit, Dept. of Informatics, University of Oslo, 2009.

[HCJ+97] Cheng-Hsueh A. Hsieh, Marie T. Conte, Teresa L. Johnson, John C. Gyllenhaal, Wen-Mei W. Hwu. Optimizing NET Compilers for Improved Java Performance. *Computer*, 30:67–75, June 1997.

[HL07] Reiko Heckel, Marc Lohmann. Model-Driven Development of Reactive Information Systems: From Graph Transformation Rules to JML Contracts. *Int. J. Softw. Tools Technol. Transf.*, 9(2):193–207, 2007.

[Höl10] Katrin Hölldobler. Effiziente Mustersuche in MontiCore-Modellen. Bachelorarbeit, RWTH Aachen, 2010.

[HR04] David Harel, Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *Computer*, 37(10):64–72, 2004.

[HRR10] Arne Haber, Jan Oliver Ringert, Bernhard Rumpe. Towards Architectural Programming of Embedded Systems. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VI*, Seiten 13 – 22, Munich, Germany, February 2010. fortiss GmbH.

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VII*, Seiten 1 – 10, Munich, Germany, February 2011. fortiss GmbH.

[II96] ISO, IEC. EBNF Spezifikation ISO/IEC 14977:1996, 1996. `http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf`, Stand: 09.01.2012.

[Jac02] Daniel Jackson. Alloy: a Lightweight Object Modelling Notation. *Software Engineering and Methodology*, 11(2):256–290, 2002.

[JBSM08] Constantin Jucovschi, Peter Baumann, Sorin Stancu-Mara. Speeding up Array Query Processing by Just-In-Time Compilation. In *Proceedings of the 2008 IEEE International Conference on Data Mining Workshops*, Seiten 408–413, Washington, DC, USA, 2008. IEEE Computer Society.

[JH02] J. Ullman J. Hopcroft, R. Motwani. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. Addison-Wesley, 2002.

[JK05] Frédéric Jouault, Ivan Kurtev. Transforming Models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2005.

[KCS05] Audris Kalnins, Edgars Celms, Agris Sostaks. Model Transformation Approach Based on MOLA. In *ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, Workshop: Model Transformations in Practice (MTIP), Montego*. Springer, 2005.

[Kle07] Anneke Kleppe. A Language Description is More than a Metamodel. In *Proceedings of Fourth International Workshop on Software Language Engineering, 1 Oct 2007, Nashville, USA.*, 2007.

[KMB+96] Richard B. Kieburtz, Laura McKinney, Jeffrey M. Bell, James Hook, Alex Kotov, Jeffrey Lewis, Dino P. Oliva, Tim Sheard, Ira Smith, Lisa Walton. A Software Engineering Experiment in Software Component Generation. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, Seiten 542–552, Washington, DC, USA, 1996. IEEE Computer Society.

[KMS+10] Thomas Kühne, Gergely Mezei, Eugene Syriani, Hans Vangheluwe, Manuel Wimmer. Explicit Transformation Modeling. In Sudipto Ghosh (Editor), *Models in Software Engineering*, volume 6002 of *Lecture Notes in Computer Science*, Seiten 240–255. Springer Berlin / Heidelberg, 2010.

[Kön09] A. Königs. *Model Integration and Transformation - A Triple Graph Grammar-based QVT Implementation*. Doktorarbeit, Technische Universität Darmstadt, January 2009.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*, volume 1 of *Aachener Informatik-Berichte, Software Engineering*. Shaker Verlag, 2010.

[Küs06] Jochen M. Küster. Definition and Validation of Model Transformations. *Software and Systems Modeling*, V5(3):233–259, 2006.

[KW07] Ekkart Kindler, Robert Wagner. Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios. Technischer Bericht tr-ri-07-284, Software Engineering Group, Department of Computer Science, University of Paderborn, June 2007.

[LE91] M. Löwe, H. Ehrig. Algebraic Approach to Graph Transformation Based on Single Pushout Derivations. In Rolf Möhring (Editor), *Graph-Theoretic Concepts in Computer Science*, volume 484 of *Lecture Notes in Computer Science*, Seiten 338–353. Springer Berlin / Heidelberg, 1991.

[LEG11] LEGO Group. LEGO.com MINDSTORMS : Home, 2011. `http://mindstorms.lego.com/`.

[Lie01] H. Lieberman. *Your Wish is my Command: Programming by Example*. Morgan Kaufmann Series in Interactive Technologies. Morgan Kaufmann Publishers, 2001.

[LKAS09] E. Legros, F. Klar, C. Amelunxen, A. Schürr. Generic and Reflective Graph Transformations for Checking and Enforcement of Modeling Guidelines. *Journal of Visual Languages and Computing*, 20(4):252–268, August 2009. Special Issue on Best Papers from VL/HCC2008.

[LMB92] John Levine, Tony Mason, Doug Brown. *lex & yacc, 2nd Edition (A Nutshell Handbook)*. O'Reilly, October 1992.

[LPKW06] Henry Lieberman, Fabio Paternò, Markus Klann, Volker Wulf. End-User Development: An Emerging Paradigm. In Henry Lieberman, Fabio Paternò, Volker Wulf (Editoren), *End User Development*, volume 9 of *Human-Computer Interaction Series*, Seiten 1–8. Springer Netherlands, 2006.

[LV02] Javier Larrosa, Gabriel Valiente. Constraint Satisfaction Algorithms for Graph Pattern Matching. *Mathematical Structures in Comp. Sci.*, 12:403–422, August 2002.

[LWK10] Philip Langer, Manuel Wimmer, Gerti Kappel. Model-to-Model Transformations By Demonstration. In Laurence Tratt, Martin Gogolla (Editoren), *Theory and Practice of Model Transformations*, volume 6142 of *Lecture Notes in Computer Science*, Seiten 153–167. Springer Berlin / Heidelberg, 2010.

[MB03] Frank Marschall, Peter Braun. Model Transformations for the MDA with BOTL. Technischer Bericht, University of Twente, 2003.

[MBT08] Jean-Marie Mottu, Benoit Baudry, Yves Le Traon. Model Transformation Testing: Oracle Issue. In *ICSTW '08: Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, Seiten 105–112, Washington, DC, USA, 2008. IEEE Computer Society.

[MD11] No Magic, Inc. MagicDraw Macro Engine User Guide. Stand: 16.12.2011, http://www.magicdraw.com/files/manuals/MagicDraw%20MacroEngine%20UserGuide.pdf.

[Mea55] G. H. Mealy. A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[Met11] MetaCase Website http://www.metacase.com. Stand: 16.05.2011.

[Met07] MetaCase. Nokia Case Study http://www.metacase.com/papers/metaedit_in_nokia.pdf, 2007.

[MLSL] Simulink Website http://www.mathworks.com/products/simulink/.

[Nag79] Manfred Nagl. *Graph-Grammatiken: Theorie, Anwendungen, Implementierung*. Vieweg, 1979.

[Nec00] George C. Necula. Translation Validation for an Optimizing Compiler. *SIGPLAN Not.*, 35(5):83–94, 2000.

[NMSS09] Mangala Gowri Nanda, Senthil Mani, Vibha Singhal Sinha, Saurabh Sinha. Demystifying Model Transformations: An Approach Based on Automated Rule Inference. *SIGPLAN Not.*, 44:341–360, October 2009.

[NR+10] Manfred Nagl, Bernhard Rumpe, et al. *Software Engineering*, volume 2010-1 of *Aachener Informatik-Berichte*, Seiten 118–148. Fachgruppe Informatik der RWTH, Aachen, 2010. http://www.informatik.rwth-aachen.de/FGI/Forschung/Jabe/jb2011.pdf, Stand: 09.01.2012.

[NS91] Manfred Nagl, Andy Schürr. A Specification Environment for Graph Grammars. In *Proceedings of the 4th International Workshop on Graph-Grammars and Their Application to Computer Science*, Seiten 599–609. Springer-Verlag, 1991.

[OMG03] Object Management Group. MDA Guide Version 1.0.1 (2003-06-01), June 2003. http://www.omg.org/docs/omg/03-06-01.pdf.

[OMG08a] Object Management Group. A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems (2008-06-08), August 2008. http://www.omg.org/cgi-bin/apps/doc?ptc/08-06-08.pdf.

[OMG08b] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification (2008-04-03), April 2008. http://www.omg.org/spec/QVT/1.0/.

[OMG10a] Object Management Group. Object Constraint Language Version 2.2 (OMG Standard 2010-02-01), 2010. http://www.omg.org/spec/OCL/2.2/PDF.

[OMG10b] Object Management Group. OMG Unified Modeling Language (OMG UML), Infrastructure Version 2.3 (10-05-03), May 2010. http://www.omg.org/spec/UML/2.3/Infrastructure/PDF/.

[OMG10c] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure Version 2.3 (10-05-05), May 2010. http://www.omg.org/spec/UML/2.3/Superstructure/PDF/.

[Opt06] OptXware Research and Development LLC. The Viatra-I Model Transformation Framework Pattern Language Specification, August 2006. `http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/doc/ViatraSpecification.pdf`.

[Par93] Terence John Parr. *Obtaining Practical Variants of LL (K) and LR (K) for K Greater than 1 by Splitting the Atomic K-Tuple*. Doktorarbeit, Purdue University, West Lafayette, IN, USA, 1993.

[Par07] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first Auflage, May 2007.

[Pet95] Marian Petre. Why Looking isn't always Seeing: Readership Skills and Graphical Programming. *Commun. ACM*, 38(6):33–44, 1995.

[PR69] John L. Pfaltz, Azriel Rosenfeld. Web Grammars. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, Seiten 609–619, San Francisco, CA, USA, 1969. Morgan Kaufmann Publishers Inc.

[Pra71] Terrence W. Pratt. Pair Grammars, Graph Languages and String-to-Graph Translations. *Journal of Computer and System Sciences*, 5:560–595, December 1971.

[PRH10a] Dorina C. Petriu, Nicolas Rouquette, Øystein Haugen (Editoren). *Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I*, volume 6394 of *Lecture Notes in Computer Science*. Springer, 2010.

[PRH10b] Dorina C. Petriu, Nicolas Rouquette, Øystein Haugen (Editoren). *Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part II*, volume 6395 of *Lecture Notes in Computer Science*. Springer, 2010.

[Rau96] Wolfgang Rautenberg. *Einführung in die Mathematische Logik*. Vieweg, 1996.

[RN03] Stuart J. Russell, Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[RS70] D.J. Rosenkrantz, R.E. Stearns. Properties of Deterministic Top-Down Grammars. *Information and Control*, 17(3):226–256, 1970.

[Rud00] Michael Rudolf. Utilizing Constraint Satisfaction Techniques for Efficient Graph Pattern Matching. In *Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations*, TAGT'98, Seiten 238–251, London, UK, 2000. Springer-Verlag.

[Rum98] Bernhard Rumpe. A Note on Semantics (with an Emphasis on UML). In Haim Kilov, Bernhard Rumpe (Editoren), *Second ECOOP Workshop on Precise Behavioral Semantics*. Technische Universität München, TUM-I9813, 1998.

[Rum04] Bernhard Rumpe. *Agile Modellierung mit UML : Codegenerierung, Testfälle, Refactoring*. Springer, 2004.

[Rum11] Bernhard Rumpe. *Modellierung mit UML*. Springer, 2. Auflage, 2011.

[RW08] Ulrike Ranger, Erhard Weinell. The Graph Rewriting Language and Environment PROGRES. In *AGTIVE 2007*, volume 5088 of *Lecture Notes in Computer Science*, Seiten 575–576. Springer, 2008.

[Sch70] Hans Jürgen Schneider. Chomsky-Systeme für partielle Ordnungen. Arbeitsbericht 3, 3, Institut für Mathematische Maschinen und Datenverarbeitung, Universität Erlangen, 1970.

[Sch88] Andy Schürr. Modellierung und Simulation komplexer Systeme mit PROGRES. In W. Ameling (Editor), *Proc. 5. Symp. Simulationstechnik, Aachen, Germany*, volume 179 of *Informatik-Fachberichte*, Seiten 84–91, Heidelberg, 1988. Springer Verlag.

[Sch91] A. Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*. Deutscher Universitätsverlag, Wiesbaden, 1991.

[Sch94] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In *Proc. of the 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science (WG '94), Herrsching (D)*. Springer, 1994.

[Sch06] Markus Schmidt. Transformations of UML 2 Models Using Concrete Syntax Patterns. In *RISE 2006 International Workshop on Rapid Integration of Software Engineering techniques*, volume 4401 of *Lecture Notes in Computer Science (LNCS)*, Seiten 130–143, Heidelberg, 2006. Springer Verlag.

[Sch11] Christian Schreder. Kontrollflusssprache für Modelltransformationen in MontiCore. Master's thesis, RWTH Aachen, 2011.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*, volume 11 of *Aachener Informatik-Berichte, Software Engineering*. Shaker Verlag, 2012.

[SD11] Jim Steel, Robin Drogemuller. Domain-Specific Model Transformation in Building Quantity Take-Off. In Jon Whittle, Tony Clark, Thomas Kühne (Editoren), *Model Driven Engineering Languages and Systems*, volume 6981 of *Lecture Notes in Computer Science*, Seiten 198–212. Springer Berlin / Heidelberg, 2011.

[Shn82] Ben Shneiderman. The Future of Interactive Systems and the Emergence of Direct Manipulation. *Behaviour & Information Technology*, 1(3):237–256, 1982.

[SK95] Kenneth Slonneger, Barry L. Kurtz. *Formal Syntax and Semantics of Programming Languages - a Laboratory Based Approach*. Addison-Wesley, 1995.

[SK03] Shane Sendall, Wojtek Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45, 2003.

[SKSS07] I. Stürmer, I. Kreuz, W. Schäfer, A. Schürr. Enhanced Simulink/Stateflow Model Transformation: The MATE Approach. In *Proc. of MathWorks Automotive Conference (MAC 2007), Dearborn (MI), USA*, 6 2007.

[SM02] Sun Microsystems. Java™ Metadata Interface (JMI) Specification, June 2002.

[SNZ08] Andy Schürr, Manfred Nagl, Albert Zündorf (Editoren). *Applications of Graph Transformations with Industrial Relevance, Third International Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and Invited Papers*, volume 5088 of *Lecture Notes in Computer Science*. Springer, 2008.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer, 1973.

[Sta11] Stagecast Software, Inc. Stagecast - Make Your Own Interactive Games, Simulations, and Stories, 2011.
`http://www.stagecast.com/`.

[Stü06] Ingo Stürmer. *Systematic Testing of Code Generation Tools - A Test Suite-oriented Approach for Safeguarding Model-based Code Generation*. Doktorarbeit, Technische Universität Berlin, 2006.

[Sun99] Sun Microsystems. Code Conventions for the Java Programming Language, 1999. `http://www.oracle.com/technetwork/java/codeconv-138413.html`.

[SWG09] Yu Sun, Jules White, Jeff Gray. Model Transformation by Demonstration. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, MODELS ’09, Seiten 712–726, Berlin, Heidelberg, 2009. Springer-Verlag.

[SWZ99] A. Schürr, A. Winter, A. Zündorf. The PROGRES Approach: Language and Environment. In H. Ehrig, G. Engels, H. Kreowski, G. Rozenberg (Editoren), *Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools*, volume 2, Seiten 487–550. World Scientific, Singapur, 1999.

[SZ92] Andy Schürr, Albert Zündorf. Nondeterministic Control Structures for Graph Rewriting Systems. In *Proc. WG'91: Workshop in Graph - Theoretic Concepts in Computer Science (LNCS 570)*, Seiten 48–62. Springer, 1992.

[Tae04] Gabriele Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *Applications of Graph Transformations with Industrial Relevance*, Seiten 446–453, 2004.

[TBB+08] Gabriele Taentzer, Enrico Biermann, Dénes Bisztray, Bernd Bohnet, Iovka Boneva, Artur Boronat, Leif Geiger, Rubino Geiß, Ákos Horvath, Ole Kniemeyer, Tom Mens, Benjamin Ness, Detlef Plump, Tamás Vajk. Generation of Sierpinski Triangles: A Case Study for Graph Transformation Tools. In *Applications of Graph Transformations with Industrial Relevance: Third International Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and Invited Papers*, Lecture Notes in Computer Science (LNCS) 5088, Seiten 514–539, Berlin, Heidelberg, 2008. Springer-Verlag.

[TG10] Laurence Tratt, Martin Gogolla (Editoren). *Theory and Practice of Model Transformations, Third International Conference, ICMT 2010, Malaga, Spain, June 28-July 2, 2010*, volume 6142 of *Lecture Notes in Computer Science*. Springer, 2010.

[Var06] Dániel Varró. Model Transformation by Example. In *Proc. 9th International Conference on Model Driven Engineering Languages and Systems (MODELS 2006)*, LNCS, Genova, October 2006. Springer.

[VB07] Dániel Varró, András Balogh. The Model Transformation Language of the VIATRA2 Framework. *Sci. Comput. Program.*, 68(3):187–207, 2007.

[vE08] Frank J. van Es. Type Inference for Graph Transformation Systems. Master's thesis, University of Twente, Enschede, The Netherlands, 2008.

[Vis02] Eelco Visser. Meta-Programming with Concrete Object Syntax. In Don Batory, Charles Consel, Walid Taha (Editoren), *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *Lecture Notes in Computer Science*, Seiten 299–315, Pittsburgh, PA, USA, October 2002. Springer-Verlag.

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*, volume 9 of *Aachener Informatik-Berichte, Software Engineering*. Shaker Verlag, 2011.

[Vos00] G. Vossen. *Datenmodelle, Datenbanksprachen und Datenbankmanagementsysteme*. Oldenbourg, 2000.

[VSV05] G. Varró, A. Schürr, D. Varró. Benchmarking for Graph Transformation. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, Seiten 79–88, September 2005.

[Wag09] Robert Wagner. *Inkrementelle Modellsynchronsiation*. Doktorarbeit, Universität Paderborn, 2009.

[WKS10] I. Weisemöller, F. Klar, A. Schürr. Development of Tool Extensions with MOFLON. In H. Giese, G. Karsai, E. Lee, B. Rumpe, B. Schätz (Editoren), *Model-Based Engineering of Embedded Real-Time Systems*, volume 6100 of *Lecture Notes in Computer Science (LNCS)*, Seiten 337–343. Springer Verlag, Heidelberg, 2010.

[ZCO09] Lei Zou, Lei Chen, M. Tamer Özsu. Distance-Join: Pattern Match Query In a Large Graph Database. *Proc. VLDB Endow.*, 2:886–897, August 2009.

[ZHV09] U. Zoltán, Á. Horváth, D. Varró. Static Type Checking of Model Transformations by Constraint Satisfaction Programming. Technischer Bericht, Budapest University of Technology & Economics, 2009.

[Zün92] Albert Zündorf. Implementation of the Imperative/Rule Based Language PROGRES. Technischer Bericht AIB-38-1992, RWTH Aachen, 1992.

[Zün96a] Albert Zündorf. *Eine Entwicklungsumgebung für PROgrammierte GRaphErsetzungsSysteme - Spezifikation, Implementierung und Verwendung*. Doktorarbeit, RWTH Aachen, 1996.

[Zün96b] Albert Zündorf. Graph Pattern Matching in PROGRES. In Janice E. Cuny, Hartmut Ehrig, Gregor Engels, Grzegorz Rozenberg (Editoren), *Graph Grammars and Their Application to Computer Science, 5th International Workshop, Williamsburg, VA, USA, November 13-18, 1994, Selected Papers*, volume 1073 of *Lecture Notes in Computer Science*, Seiten 454–468. Springer, 1996.

Anhang A.
Glossar und Abkürzungsverzeichnis

Die folgenden Begriffe werden im Kontext dieser Arbeit wie angegeben verwendet.

Abstrakte Syntax
: Die abstrakte Syntax ist die recherinterne Darstellung der Instanzen einer Programmier- oder Modellierungssprache. Siehe auch →konkrete Syntax.

Abstrakter Syntaxbaum
: Ein abstrakter Syntaxbaum (Englisch: abstract syntax tree, kurz AST) ist ein attributierter und gelegentlich auch getypter Baum, der ein geparstes Eingabeprogramm oder -modell repräsentiert.

Argument
: Argumente, auch als *tatsächliche Parameter* bezeichnet, sind die Werte, die zu einem →Formalparameter beim Methodenaufruf übergeben werden.

AST
: →abstrakter Syntaxbaum

Backtracking
: Backtracking-Algorithmen sind Verfahren, die mögliche Lösungen zu einem Problem inkrementell berechnen. Wird bei einem Inkrement festgestellt, dass im Folgenden keine Lösung mehr gefunden werden kann, so wird zum vorherigen Schritt zurückgesprungen. Dort wird, falls dies möglich ist, eine alternative Fortsetzung des Verfahrens gewählt. Dieser Rücksprung wird als Backtracking bezeichnet.

Bezeichner
: Ein Bezeichner ist eine Zeichenkette, die ein Element eines Modells oder Programms eindeutig kennzeichnet. Bezeichner können auch als Namen dieser Elemente betrachtet werden. Oft wird zwischen einfachen Bezeichnern, in Java etwa `String` für die entsprechende Klasse, und qualifizierten Bezeichnern, in diesem Fall `java.lang.String`, unterschieden. Qualifizierte Bezeichner sind über das gesamte Modell beziehungsweise Programm global eindeutig.

Codegenerierung
: →Generierung

Domänenexperte
Im Kontext dieser Arbeit sind Domänenexperten Personen, die an der Softwareentwicklung beitragen und besonderes Wissen in der fachlichen Anwendungsdomäne oder in einer technischen Domäne haben. Sie haben aber nicht notwendigerweise umfangreiche Kenntnisse in der Programmierung oder Softwaretechnik.

Domänenspezifische Sprache
Eine domänenspezifische Sprache ist für die Verwendung in einer technischen oder fachlichen Domäne besonders geeignet, oft auch auf diese eingeschränkt.

DSL
Abkürzung für domain specific language, siehe →domänenspezifische Sprache.

DSLTool
Ein DSLTool ist ein Werkzeug, dass die Verarbeitung von Modellen in mehreren →Workflows kapselt und die Ausführung der Workflows steuert.

ETS
→Extensible Type System

Extensible Type System
Das Extensible Type System in MontiCore ist ein Framework zum Aufbau von →Symboltabellen und zur Analyse von →Kontextbedingungen. Eine wesentliche Eigenschaft ist die Unterstützung kompositionaler Entwicklung von Sprachen.

Formalparameter
Die Formalparameter eine Methode legen fest, wie viele →Argumente dieder Methode beim Aufruf übergeben müssen. Oft ist durch die Formalparameter auch definiert, welchen Typ die Argumente haben müssen.

Generation and Language Infrastructure
Die MontiCore Generation and Language Infrastructure, kurz GLI, ist ein Framework zur templatebasierten Codegenerierung aus Modellen. Sie ist in [Sch12] beschrieben.

General Purpose Language
General Purpose Languages, kurz GPLs sind Programmier- oder Modellierungssprachen, die nicht auf eine bestimmte Anwendungsdomäne zugeschnitten wurden, und die in vielen oder beliebigen Domänen einsetzbar sind. Siehe auch →domänenspezifische Sprache.

Generator
Ein Generator ist eine Komponente, die für die →Generierung zuständig ist.

Generierung
Die Generierung ist der Spezialfall einer →Transformation, der dadurch gekennzeichnet ist, dass sie ein neues Modell oder Programm erzeugt.

GLI
→Generation and Language Infrastructure

GPL

→General Purpose Language

Grammatik

Grammatiken sind ein Beschreibungsmittel für die Syntax von Sprachen. Sofern nicht anders angegeben, sind in dieser Arbeit damit kontextfreie String-Grammatiken gemeint, die aus →Terminalen, →Nichtterminalen und →Produktionen bestehen. Darüber hinaus existieren beispielsweise auch kontextsensitive Grammatiken und Graphgrammatiken.

Hostgraph

Der Graph, auf den eine →Transformationsregel angewendet wird, wird als Hostgraph bezeichnet. Im Bezug auf Modelle ist hiermit der →abstrakte Syntaxbaum oder -graph des Modells gemeint.

Konkrete Syntax

Die konkrete Syntax ist die Darstellung der Instanzen einer Programmier- oder Modellierungssprache gegenüber dem Benutzer. Siehe auch →abstrakte Syntax.

Kontextbedingung

Kontextbedingungen sind Bedingungen für die syntaktische Korrektheit eines Programms oder Modell, die nicht vom Parser überprüft werden. Oft sind diese Bedingungen kontextsensitiv. Eine wesentliche Teilmenge der Kontextbedingungen betrifft meist die Korrektheit bezüglich eines Typsystems.

Kontrollstruktur

Als Kontrollstrukturen werden Elemente von Programmiersprachen bezeichnet, die den Ablauf von Programmen beeinflussen. Sie können beispielsweise alternative Abläufe oder Schleifen beschreiben.

LHS

Left Hand Side, linke Regelseite. Siehe auch →Transformationsregel.

MDA

→Model Driven Architecture

Model Driven Architecture

Die Model Driven Architecture oder MDA ist eine vom Standardisierungskonsortium Object Managenment Group (OMG) vorgeschlagene und verabschiedete Methode zur →modellgetriebenen Entwicklung.

Modell

Ein Modell ist nach [Sta73] eine abstrahierende Darstellung eines Originals, welches sie im Hinblick auf einen pragmatischen Zweck beschreibt. Im Kontext des Software Engineering ist das Original meistens ein Softwaresystem, das sich möglicherweise noch in der Entwicklung befindet.

Modellgetriebene Entwicklung
In der modellgetriebenen Entwicklung wird lauffähige Software aus Modellen erzeugt, sei es durch Generierung oder Interpretation.

Modelltransformation
Eine (Modell-)transformation ist ein Programm, das ein Modell erzeugt oder manipuliert.

Nichtterminal
Ein Nichtterminalsymbol oder kurz Nichtterminal ist ein Bestandteil einer →Grammatik, dass durch →Produktionen in →Terminale zerlegt werden kann.

Objekt
Ein Objekt ist eine Ausprägung einer Klasse, welche Werte für die durch die Klasse festgelegten Eigenschaften enthält. Objekte können von einem Programm erzeugt, verändert oder gelöscht werden. In →abstrakten Syntaxbäumen korrespondiert ein Objekt zu genau einem Knoten im AST.

Produktion
Eine Produktion innerhalb einer kontextfreien →Grammatik beschreibt die Zerlegung eines →Nichtterminals auf der linken Seite der Produktion in eine Sequenz aus →Terminalen und Nichtterminalen auf ihrer rechten Seite.

Schemavariable
Eine Schemavariable ist innerhalb einer →Transformationsregel ein Platzhalter, der für variable Elemente im durch die Regel transformierten Modell steht, zu denen nicht alle Details angegeben sind. Bei der Ausführung der Regel werden die Schemavariablen mit konkreten Elementen aus dem →Hostgraphen belegt.

Symboltabelle
Symboltabellen sind Datenstrukturen, die Bezeichner auf weiterführende Informationen abbilden, beispielsweise auf den qualifizierten Bezeichner deklarierende Auftreten des Bezeichners.

RHS
right hand side, rechte Regelseite. Siehe auch →Transformationsregel.

Term
Ein Term beschreibt innerhalb einer →Transformationsregel einen zusammenhängenden Ausschnitt eines Modells. Bei der Mustersuche wird jeder Term auf einen Teilbaum des →ASTs gematcht.

Terminal
Ein Terminalsymbol oder kurz Terminal ist in einer kontextfreien →Grammatik ein nicht weiter zerlegbares Symbol, also ein atomarer Bestandteil der durch die Grammatik beschriebenen Sprache.

Transformation
→Modelltransformation

Transformationsregel

Transformationsregeln sind atomare Bestandteile von →Transformationen. Sie werden in dieser Arbeit durch Graphersetzungsregeln beschrieben, deren linke Seite ein im →Hostgraph zu suchendes Graphmuster beschreibt, und deren rechte Seite einen Teilgraphen zeigt durch den dieses Muster ersetzt wird.

Unified Modeling Language / programmier-geeignet

Die UML/P ist ein Profil der Unified Modeling Language, das in [Rum11] beschrieben wird, und für das eine textuelle Notation sowie dazu passende Werkzeuge in [Sch12] entwickelt wurden.

UML/P

→Unified Modeling Language / programmier-geeignet

Workflow

In MontiCore sind Workflows zustandslose Verarbeitungsschritte für Modelle einer bestimmten Sprache.

Anhang B.

Notationen

In dieser Arbeit wurden an Abbildungen die folgenden Markierungen angebracht, die jeweils den Typ der entsprechenden Abbildung angeben.

AD **Aktivitätsdiagramm**

Aktivitätsdiagramme [OMG10b, OMG10c] beschreiben Abläufe von Aktivitäten, die in einzelne Aktionen unterteilt sind. Kontroll- und Datenflüsse zwischen diesen Aktionen können ebenfalls modelliert werden.

Aut **Endlicher Automat**

Endliche Automaten [JH02] sind zustandsbasierte Modelle für Systemverhalten. Übergänge zwischen Zuständen werden durch Transitionen beschrieben. Das Schalten einer Transition wird durch ein Eingabesymbol ausgelöst.

CD_T **Klassendiagramm**

UML-Klassendiagramme [OMG10b, OMG10c] beschreiben objektorientierte Klassen, ihre Eigenschaften, Methoden und Beziehungen untereinander. Innerhalb der Vererbungshierarchie werden Eigenschaften und Verhalten an spezialisierende Klassifizierer weitergegeben. Zur Bedeutung der tiefgestellten Kürzel siehe unten.

OD_T **Objektdiagramm**

Objektdiagramme [OMG10b, OMG10c] beschreiben exemplarische Systemzustände. Zu den Objekten können unter anderem Typen, Attribute, sowie Referenzen auf weitere Objekte als Links angegeben werden. Zur Bedeutung der tiefgestellten Kürzel siehe unten.

PL **Aussagenlogische Formel**

Notation für Formeln in der Aussagenlogik (Englisch: propositional logic) [Rau96].

SC **Statechart**

Statecharts sind zustandsbasierte Modelle zur Beschreibung von Objektverhalten. Eine informelle Einführung wird in Abschnitt 3.2 gegeben, detaillierte Beschreibungen finden sich in [Rum11, Sch12].

SD **Sequenzdiagramm**

Ein Sequenzdiagramm beschreibt exemplarisch einen Programmablauf oder einen Teil hiervon. Dabei werden sowohl die beteiligten Objekte als auch die zwischen den Objekten ausgetauschten Nachrichten modelliert. Weitere Informationen finden sich ebenfalls in [Rum11, Sch12].

Zusätzlich können die Markierungen der Klassen- und Objektdiagramme tiefgestellte Kürzel enthalten (in den oben stehenden Grafiken jeweils T), die auf den Verwendungszweck hinweisen. Hier werden die folgenden Abkürzungen verwendet. Kombinationen aus den Abkürzungen sind möglich, wenn sich das Diagramm auf mehrere der genannten Aspekte bezieht.

D_{AS} **Diagramm der abstrakten Syntax**

Das Diagramm bezieht sich auf die abstrakte Syntax eines Modells, zeigt also einen abstrakten Syntaxbaum beziehungsweise -graphen oder einen Ausschnitt daraus.

D_{TF} **Transformationsregel in Objektdiagramm-Notation**

Das Diagramm beschreibt eine Transformationsregel oder eine Regelseite in Objektdiagramm-Notation.

D_{TFCS} **Syntaxdiagramm zu Regeln in konkreter Syntax**

Das Diagramm beschreibt die abstrakte Syntax zu einer Transformationsregel in konkreter Syntax, also den AST nach dem Parsen einer solchen Regel oder einen Ausschnitt daraus.

D_{TFGen} **Generierter Code**

Das Diagramm zeigt Teile des aus einer Transformation generierten Java-Codes.

D_{TFRE} **Diagramm der Laufzeitumgebung**

Das Diagramm beschreibt Teile der Laufzeitumgebung (Englisch: runtime environment, abgekürzt RE), die zur Ausführung des aus Transformationen generierten Codes benötigt wird. Dies kann auch Teile der außerhalb dieser Arbeit entwickelten MontiCore-Laufzeitumgebung [Kra10] mit einschließen.

Anhang C.

Grammatiken

Diese Grammatiken beschreibt die Syntax der Statechartsprache (Abschnitt C.1), die als durchgängiges Beispiel in der vorliegenden Arbeit verwendet wurde, sowie die generierte Grammatik der Sprache für Transformationsregeln auf Statecharts (Abschnitt C.2).

C.1. Grammatik der Statechartsprache

MontiCore-Grammatik

```
1 package mc.testcases.statechart;
2
3 grammar Statechart {
4
5   options {
6     parser lookahead = 3
7     lexer lookahead = 7
8     compilationunit Statechart
9   }
10
11  concept globalnaming {
12    define Statechart.Name;
13    define State.Name;
14    usage Transition.From;
15    usage Transition.To;
16  }
17
18  Statechart implements SCStructure= "statechart" Name "{"
19     (States:State | Transitions:Transition )* "}";
20
21
22  EntryAction= "entry" ":" Block:BlockStatement;
23
24  ExitAction= "exit" ":" Block:BlockStatement;
25
26  DoAction= "do" ":" Block:BlockStatement;
27
28  InternTransition = "-intern>"
29     ( ":"
30       (Event:Name (
31         "(" (Arguments:Argument ( "," Arguments:Argument)*) ")" )?
32       )?
```

```
33      ("[" PreCondition: Expression "]")?
34      ("/" Action: BlockStatement ("[" PostCondition: Expression "]")?)? ";"
35    | ";");
36
37  State implements SCStructure =
38    "state" Name ("<<" (Initial:["initial"] | Final:["final"]) ">>")*
39    ( ("{" ("[" Invariant:Expression ( "&&" Invariant:Expression)* "]")?
40      (EntryAction:EntryAction)?
41      (DoAction)?
42      (ExitAction:ExitAction)?
43      (States:State
44        | Transitions:Transition
45        | InternTransitions:InternTransition
46      )* "}") | ";")
47  ;
48
49  Transition = From:Name "->" To:Name
50    (":" (Event:Name (
51      "(" ((Arguments:Argument ( "," Arguments:Argument)*)?) ")" )? )?
52    ("[" PreCondition: Expression "]")?
53    ("/" Action: BlockStatement ("[" PostCondition: Expression "]")?)? ";"
54  | ";");
55
56  Argument= ParamType:Name ParamName:Name;
57
58  interface SCStructure;
59
60  ast SCStructure =
61    Name
62    States:State*
63    Transitions:Transition*
64    method public String toString() {{return name;}}
65  ;
66
67  ast State =
68    method public String toString() {
69        return name;
70    }
71    method public mc.ast.ASTNode getParentElement() {
72      if (get_Parent() != null && get_Parent() instanceof mc.ast.ASTList) {
73        return get_Parent().get_Parent();
74      }
75      return get_Parent();
76    }
77  ;
78
79  ast Transition =
80    method public mc.ast.ASTNode getParentElement() {
81      if (get_Parent() != null && get_Parent() instanceof mc.ast.ASTList) {
82        return get_Parent().get_Parent();
83      }
84      return get_Parent();
85    }
```

```
86   ;
87
88   interface Statement;
89
90   BlockStatement implements ("{")=>Statement =
91     "{" (Statements:Statement)* "}"
92   ;
93
94   interface Expression;
95
96   ExpressionStatement implements (Expression ";")=>Statement =
97     Expression:Expression ";"
98   ;
99
100  EqualityExpression implements (Name ("=="|"!=") )=>Expression =
101        LeftOperand:Name
102          Operator:["!=" | "=="] RightOperand:Name
103    ;
104
105  MethodInvocationWithQualifiedName
106    implements (Name (options{greedy=true;}: "." Name)* "(")=>Expression =
107      Name
108      (options{greedy=true;}: "." Name)*
109      "("
110        (Arguments: Expression ("," Arguments: Expression)*)?
111      ")"
112  ;
113
114  FieldAccess implements (Name)=>Expression =
115    Name
116  ;
117
118 }
```

C.2. Grammatik der Transformationsregeln auf Statecharts

MontiCore-Grammatik

```
1 package mc.testcases.statechart.transformation.rule;
2
3 grammar StatechartTransformationRule extends mc.tf.commons.TFCommons {
4   ast Expression_Replacement =
5     method public Class _getTFElementType(){
6       return mc.testcases.statechart._ast.ASTExpression.class;
7     }
8   ;
9
10  ast Expression_Optional =
11    method public ASTExpression getRhs(){return getExpression();}
12    method public ASTExpression getLhs(){return getExpression();}
13    method public Class _getTFElementType(){
```

```
14        return mc.testcases.statechart._ast.ASTExpression.class;
15      }
16    ;
17
18    ast Expression_Pattern =
19      method public Class _getTFElementType(){
20        return mc.testcases.statechart._ast.ASTExpression.class;
21      }
22      method public ASTExpression_Pattern getLhs(){return this;}
23      method public ASTExpression_Pattern getRhs(){return this;}
24    ;
25
26    ast Expression_Negation =
27      method public ASTExpression getRhs(){return getExpression();}
28      method public ASTExpression getLhs(){return getExpression();}
29      method public Class _getTFElementType(){
30        return mc.testcases.statechart._ast.ASTExpression.class;
31      }
32    ;
33
34    ast Expression_List =
35      method public ASTExpression getRhs(){return getExpression();}
36      method public ASTExpression getLhs(){return getExpression();}
37      method public Class _getTFElementType(){
38        return mc.testcases.statechart._ast.ASTExpression.class;
39      }
40    ;
41
42    ast Statement_Replacement =
43      method public Class _getTFElementType(){
44        return mc.testcases.statechart._ast.ASTStatement.class;
45      }
46    ;
47
48    ast Statement_Optional =
49      method public ASTStatement getRhs(){return getStatement();}
50      method public ASTStatement getLhs(){return getStatement();}
51      method public Class _getTFElementType(){
52        return mc.testcases.statechart._ast.ASTStatement.class;
53      }
54    ;
55
56    ast Statement_Pattern =
57      method public Class _getTFElementType(){
58        return mc.testcases.statechart._ast.ASTStatement.class;
59      }
60      method public ASTStatement_Pattern getLhs(){return this;}
61      method public ASTStatement_Pattern getRhs(){return this;}
62    ;
63
64    ast Statement_Negation =
65      method public ASTStatement getRhs(){return getStatement();}
66      method public ASTStatement getLhs(){return getStatement();}
```

```
67     method public Class _getTFElementType(){
68       return mc.testcases.statechart._ast.ASTStatement.class;
69     }
70   ;
71
72   ast Statement_List =
73     method public ASTStatement getRhs(){return getStatement();}
74     method public ASTStatement getLhs(){return getStatement();}
75     method public Class _getTFElementType(){
76       return mc.testcases.statechart._ast.ASTStatement.class;
77     }
78   ;
79
80   ast SCStructure_Replacement =
81     method public Class _getTFElementType(){
82       return mc.testcases.statechart._ast.ASTSCStructure.class;
83     }
84   ;
85
86   ast SCStructure_Optional =
87     method public ASTSCStructure getRhs(){return getSCStructure();}
88     method public ASTSCStructure getLhs(){return getSCStructure();}
89     method public Class _getTFElementType(){
90       return mc.testcases.statechart._ast.ASTSCStructure.class;
91     }
92   ;
93
94   ast SCStructure_Pattern =
95     method public Class _getTFElementType(){
96       return mc.testcases.statechart._ast.ASTSCStructure.class;
97     }
98     method public ASTSCStructure_Pattern getLhs(){return this;}
99     method public ASTSCStructure_Pattern getRhs(){return this;}
100  ;
101
102  ast SCStructure_Negation =
103    method public ASTSCStructure getRhs(){return getSCStructure();}
104    method public ASTSCStructure getLhs(){return getSCStructure();}
105    method public Class _getTFElementType(){
106      return mc.testcases.statechart._ast.ASTSCStructure.class;
107    }
108  ;
109
110  ast SCStructure_List =
111    method public ASTSCStructure getRhs(){return getSCStructure();}
112    method public ASTSCStructure getLhs(){return getSCStructure();}
113    method public Class _getTFElementType(){
114      return mc.testcases.statechart._ast.ASTSCStructure.class;
115    }
116  ;
117
118  ast FieldAccess_Replacement =
119    method public Class _getTFElementType(){
```

```
120       return mc.testcases.statechart._ast.ASTFieldAccess.class;
121     }
122   ;
123
124   ast FieldAccess_Optional =
125     method public ASTFieldAccess getRhs(){return getFieldAccess();}
126     method public ASTFieldAccess getLhs(){return getFieldAccess();}
127     method public Class _getTFElementType(){
128       return mc.testcases.statechart._ast.ASTFieldAccess.class;
129     }
130   ;
131
132   ast FieldAccess_Pattern =
133     method public Class _getTFElementType(){
134       return mc.testcases.statechart._ast.ASTFieldAccess.class;
135     }
136     method public ASTFieldAccess_Pattern getLhs(){return this;}
137     method public ASTFieldAccess_Pattern getRhs(){return this;}
138   ;
139
140   ast FieldAccess_Negation =
141     method public ASTFieldAccess getRhs(){return getFieldAccess();}
142     method public ASTFieldAccess getLhs(){return getFieldAccess();}
143     method public Class _getTFElementType(){
144       return mc.testcases.statechart._ast.ASTFieldAccess.class;
145     }
146   ;
147
148   ast FieldAccess_List =
149     method public ASTFieldAccess getRhs(){return getFieldAccess();}
150     method public ASTFieldAccess getLhs(){return getFieldAccess();}
151     method public Class _getTFElementType(){
152       return mc.testcases.statechart._ast.ASTFieldAccess.class;
153     }
154   ;
155
156   ast MethodInvocationWithQualifiedName_Replacement =
157     method public Class _getTFElementType(){
158     return mc.testcases.statechart._ast.
159         ASTMethodInvocationWithQualifiedName.class;
160     }
161   ;
162
163   ast MethodInvocationWithQualifiedName_Optional =
164     method public ASTMethodInvocationWithQualifiedName getRhs(){
165       return getMethodInvocationWithQualifiedName();
166     }
167     method public ASTMethodInvocationWithQualifiedName getLhs(){
168       return getMethodInvocationWithQualifiedName();
169     }
170     method public Class _getTFElementType(){
171       return mc.testcases.statechart._ast.
172         ASTMethodInvocationWithQualifiedName.class;
```

```
173     }
174   ;
175
176   ast MethodInvocationWithQualifiedName_Pattern =
177     method public Class _getTFElementType(){
178       return mc.testcases.statechart._ast.
179           ASTMethodInvocationWithQualifiedName.class;}
180     method public ASTMethodInvocationWithQualifiedName_Pattern getLhs(){
181       return this;
182     }
183     method public ASTMethodInvocationWithQualifiedName_Pattern getRhs(){
184       return this;
185     }
186   ;
187
188   ast MethodInvocationWithQualifiedName_Negation =
189     method public ASTMethodInvocationWithQualifiedName getRhs(){
190       return getMethodInvocationWithQualifiedName();
191     }
192     method public ASTMethodInvocationWithQualifiedName getLhs(){
193       return getMethodInvocationWithQualifiedName();
194     }
195     method public Class _getTFElementType(){
196       return mc.testcases.statechart._ast.
197           ASTMethodInvocationWithQualifiedName.class;
198     }
199   ;
200
201   ast MethodInvocationWithQualifiedName_List =
202     method public ASTMethodInvocationWithQualifiedName getRhs(){
203       return getMethodInvocationWithQualifiedName();
204     }
205     method public ASTMethodInvocationWithQualifiedName getLhs(){
206       return getMethodInvocationWithQualifiedName();
207     }
208     method public Class _getTFElementType(){
209       return mc.testcases.statechart._ast.
210           ASTMethodInvocationWithQualifiedName.class;
211     }
212   ;
213
214   ast Operator_Pattern =
215     method public ASTOperator_Pattern getLhs(){return this;}
216     method public ASTOperator_Pattern getRhs(){return this;}
217   ;
218
219   ast Operator_Negation =
220     method public ASTOperator_Pattern getLhs(){
221       return getOperator_Pattern();
222     }
223     method public ASTOperator_Pattern getRhs(){
224       return getOperator_Pattern();
225     }
```

```
226 ;
227
228 ast EqualityExpression_Replacement =
229   method public Class _getTFElementType(){
230     return mc.testcases.statechart._ast.ASTEqualityExpression.class;
231   }
232 ;
233
234 ast EqualityExpression_Optional =
235   method public ASTEqualityExpression getRhs(){
236     return getEqualityExpression();
237   }
238   method public ASTEqualityExpression getLhs(){
239     return getEqualityExpression();
240   }
241   method public Class _getTFElementType(){
242     return mc.testcases.statechart._ast.ASTEqualityExpression.class;
243   }
244 ;
245
246 ast EqualityExpression_Pattern =
247   method public Class _getTFElementType(){
248     return mc.testcases.statechart._ast.ASTEqualityExpression.class;
249   }
250   method public ASTEqualityExpression_Pattern getLhs(){return this;}
251   method public ASTEqualityExpression_Pattern getRhs(){return this;}
252 ;
253
254 ast EqualityExpression_Negation =
255   method public ASTEqualityExpression getRhs(){
256     return getEqualityExpression();
257   }
258   method public ASTEqualityExpression getLhs(){
259     return getEqualityExpression();
260   }
261   method public Class _getTFElementType(){
262     return mc.testcases.statechart._ast.ASTEqualityExpression.class;
263   }
264 ;
265
266 ast EqualityExpression_List =
267   method public ASTEqualityExpression getRhs(){
268     return getEqualityExpression();
269   }
270   method public ASTEqualityExpression getLhs(){
271     return getEqualityExpression();
272   }
273   method public Class _getTFElementType(){
274     return mc.testcases.statechart._ast.ASTEqualityExpression.class;
275   }
276 ;
277
278 ast ExpressionStatement_Replacement =
```

```
279     method public Class _getTFElementType(){
280       return mc.testcases.statechart._ast.ASTExpressionStatement.class;
281     }
282   ;
283
284   ast ExpressionStatement_Optional =
285     method public ASTExpressionStatement getRhs(){
286       return getExpressionStatement();
287     }
288     method public ASTExpressionStatement getLhs(){
289       return getExpressionStatement();
290     }
291     method public Class _getTFElementType(){
292       return mc.testcases.statechart._ast.ASTExpressionStatement.class;
293     }
294   ;
295
296   ast ExpressionStatement_Pattern =
297     method public Class _getTFElementType(){
298       return mc.testcases.statechart._ast.ASTExpressionStatement.class;
299     }
300     method public ASTExpressionStatement_Pattern getLhs(){return this;}
301     method public ASTExpressionStatement_Pattern getRhs(){return this;}
302   ;
303
304   ast ExpressionStatement_Negation =
305     method public ASTExpressionStatement getRhs(){
306       return getExpressionStatement();
307     }
308     method public ASTExpressionStatement getLhs(){
309       return getExpressionStatement();
310     }
311     method public Class _getTFElementType(){
312       return mc.testcases.statechart._ast.ASTExpressionStatement.class;
313     }
314   ;
315
316   ast ExpressionStatement_List =
317     method public ASTExpressionStatement getRhs(){
318       return getExpressionStatement();
319     }
320     method public ASTExpressionStatement getLhs(){
321       return getExpressionStatement();
322     }
323     method public Class _getTFElementType(){
324       return mc.testcases.statechart._ast.ASTExpressionStatement.class;
325     }
326   ;
327
328   ast BlockStatement_Replacement =
329     method public Class _getTFElementType(){
330       return mc.testcases.statechart._ast.ASTBlockStatement.class;
331     }
```

```
332  ;
333
334  ast BlockStatement_Optional =
335    method public ASTBlockStatement getRhs(){
336      return getBlockStatement();
337    }
338    method public ASTBlockStatement getLhs(){
339      return getBlockStatement();
340    }
341    method public Class _getTFElementType(){
342      return mc.testcases.statechart._ast.ASTBlockStatement.class;
343    }
344  ;
345
346  ast BlockStatement_Pattern =
347    method public Class _getTFElementType(){
348      return mc.testcases.statechart._ast.ASTBlockStatement.class;
349    }
350    method public ASTBlockStatement_Pattern getLhs(){
351      return this;
352    }
353    method public ASTBlockStatement_Pattern getRhs(){
354      return this;
355    }
356  ;
357
358  ast BlockStatement_Negation =
359    method public ASTBlockStatement getRhs(){
360      return getBlockStatement();
361    }
362    method public ASTBlockStatement getLhs(){
363      return getBlockStatement();
364    }
365    method public Class _getTFElementType(){
366      return mc.testcases.statechart._ast.ASTBlockStatement.class;
367    }
368  ;
369
370  ast BlockStatement_List =
371    method public ASTBlockStatement getRhs(){
372      return getBlockStatement();
373    }
374    method public ASTBlockStatement getLhs(){
375      return getBlockStatement();
376    }
377    method public Class _getTFElementType(){
378      return mc.testcases.statechart._ast.ASTBlockStatement.class;
379    }
380  ;
381
382  ast Argument_Replacement =
383    method public Class _getTFElementType(){
384      return mc.testcases.statechart._ast.ASTArgument.class;
```

```
385     }
386   ;
387
388   ast Argument_Optional =
389     method public ASTArgument getRhs(){return getArgument();}
390     method public ASTArgument getLhs(){return getArgument();}
391     method public Class _getTFElementType(){
392       return mc.testcases.statechart._ast.ASTArgument.class;
393     }
394   ;
395
396   ast Argument_Pattern =
397     method public Class _getTFElementType(){
398       return mc.testcases.statechart._ast.ASTArgument.class;
399     }
400     method public ASTArgument_Pattern getLhs(){return this;}
401     method public ASTArgument_Pattern getRhs(){return this;}
402   ;
403
404   ast Argument_Negation =
405     method public ASTArgument getRhs(){return getArgument();}
406     method public ASTArgument getLhs(){return getArgument();}
407     method public Class _getTFElementType(){
408       return mc.testcases.statechart._ast.ASTArgument.class;
409     }
410   ;
411
412   ast Argument_List =
413     method public ASTArgument getRhs(){return getArgument();}
414     method public ASTArgument getLhs(){return getArgument();}
415     method public Class _getTFElementType(){
416       return mc.testcases.statechart._ast.ASTArgument.class;
417     }
418   ;
419
420   ast Transition_Replacement =
421     method public Class _getTFElementType(){
422     return mc.testcases.statechart._ast.ASTTransition.class;
423     }
424   ;
425
426   ast Transition_Optional =
427     method public ASTTransition getRhs(){return getTransition();}
428     method public ASTTransition getLhs(){return getTransition();}
429     method public Class _getTFElementType(){
430       return mc.testcases.statechart._ast.ASTTransition.class;
431     }
432   ;
433
434   ast Transition_Pattern =
435     method public Class _getTFElementType(){
436       return mc.testcases.statechart._ast.ASTTransition.class;
437     }
```

```
438     method public ASTTransition_Pattern getLhs(){return this;}
439     method public ASTTransition_Pattern getRhs(){return this;}
440   ;
441
442   ast Transition_Negation =
443     method public ASTTransition getRhs(){return getTransition();}
444     method public ASTTransition getLhs(){return getTransition();}
445     method public Class _getTFElementType(){
446       return mc.testcases.statechart._ast.ASTTransition.class;
447     }
448   ;
449
450   ast Transition_List =
451     method public ASTTransition getRhs(){return getTransition();}
452     method public ASTTransition getLhs(){return getTransition();}
453     method public Class _getTFElementType(){
454       return mc.testcases.statechart._ast.ASTTransition.class;
455     }
456   ;
457
458   ast Final_Pattern =
459     method public ASTFinal_Pattern getLhs(){return this;}
460     method public ASTFinal_Pattern getRhs(){return this;}
461   ;
462
463   ast Final_Negation =
464     method public ASTFinal_Pattern getLhs(){return getFinal_Pattern();}
465     method public ASTFinal_Pattern getRhs(){return getFinal_Pattern();}
466   ;
467
468   ast Initial_Pattern =
469     method public ASTInitial_Pattern getLhs(){return this;}
470     method public ASTInitial_Pattern getRhs(){return this;}
471   ;
472
473   ast Initial_Negation =
474     method public ASTInitial_Pattern getLhs(){
475       return getInitial_Pattern();
476     }
477     method public ASTInitial_Pattern getRhs(){
478       return getInitial_Pattern();
479     }
480   ;
481
482   ast State_Replacement =
483     method public Class _getTFElementType(){
484       return mc.testcases.statechart._ast.ASTState.class;
485     }
486   ;
487
488   ast State_Optional =
489     method public ASTState getRhs(){return getState();}
490     method public ASTState getLhs(){return getState();}
```

```
491     method public Class _getTFElementType(){
492       return mc.testcases.statechart._ast.ASTState.class;
493     }
494   ;
495
496   ast State_Pattern =
497     method public Class _getTFElementType(){
498       return mc.testcases.statechart._ast.ASTState.class;
499     }
500     method public ASTState_Pattern getLhs(){return this;}
501     method public ASTState_Pattern getRhs(){return this;}
502   ;
503
504   ast State_Negation =
505     method public ASTState getRhs(){return getState();}
506     method public ASTState getLhs(){return getState();}
507     method public Class _getTFElementType(){
508       return mc.testcases.statechart._ast.ASTState.class;
509     }
510   ;
511
512   ast State_List =
513     method public ASTState getRhs(){return getState();}
514     method public ASTState getLhs(){return getState();}
515     method public Class _getTFElementType(){
516       return mc.testcases.statechart._ast.ASTState.class;
517     }
518   ;
519
520   ast InternTransition_Replacement =
521     method public Class _getTFElementType(){
522       return mc.testcases.statechart._ast.ASTInternTransition.class;
523     }
524   ;
525
526   ast InternTransition_Optional =
527     method public ASTInternTransition getRhs(){
528       return getInternTransition();
529     }
530     method public ASTInternTransition getLhs(){
531       return getInternTransition();
532     }
533     method public Class _getTFElementType(){
534       return mc.testcases.statechart._ast.ASTInternTransition.class;
535     }
536   ;
537
538   ast InternTransition_Pattern =
539     method public Class _getTFElementType(){
540       return mc.testcases.statechart._ast.ASTInternTransition.class;
541     }
542     method public ASTInternTransition_Pattern getLhs(){return this;}
543     method public ASTInternTransition_Pattern getRhs(){return this;}
```

```
544  ;
545
546  ast InternTransition_Negation =
547    method public ASTInternTransition getRhs(){
548      return getInternTransition();
549    }
550    method public ASTInternTransition getLhs(){
551      return getInternTransition();
552    }
553    method public Class _getTFElementType(){
554      return mc.testcases.statechart._ast.ASTInternTransition.class;
555    }
556  ;
557
558  ast InternTransition_List =
559    method public ASTInternTransition getRhs(){
560      return getInternTransition();
561    }
562    method public ASTInternTransition getLhs(){
563      return getInternTransition();
564    }
565    method public Class _getTFElementType(){
566      return mc.testcases.statechart._ast.ASTInternTransition.class;
567    }
568  ;
569
570  ast DoAction_Replacement =
571    method public Class _getTFElementType(){
572      return mc.testcases.statechart._ast.ASTDoAction.class;
573    }
574  ;
575
576  ast DoAction_Optional =
577    method public ASTDoAction getRhs(){return getDoAction();}
578    method public ASTDoAction getLhs(){return getDoAction();}
579    method public Class _getTFElementType(){
580      return mc.testcases.statechart._ast.ASTDoAction.class;
581    }
582  ;
583
584  ast DoAction_Pattern =
585    method public Class _getTFElementType(){
586      return mc.testcases.statechart._ast.ASTDoAction.class;
587    }
588    method public ASTDoAction_Pattern getLhs(){return this;}
589    method public ASTDoAction_Pattern getRhs(){return this;}
590  ;
591
592  ast DoAction_Negation =
593    method public ASTDoAction getRhs(){
594      return getDoAction();
595    }
596    method public ASTDoAction getLhs(){
```

```
597       return getDoAction();
598     }
599     method public Class _getTFElementType(){
600       return mc.testcases.statechart._ast.ASTDoAction.class;
601     }
602   ;
603
604   ast DoAction_List =
605     method public ASTDoAction getRhs(){
606       return getDoAction();
607     }
608     method public ASTDoAction getLhs(){
609       return getDoAction();
610     }
611     method public Class _getTFElementType(){
612       return mc.testcases.statechart._ast.ASTDoAction.class;
613     }
614   ;
615
616   ast ExitAction_Replacement =
617     method public Class _getTFElementType(){
618       return mc.testcases.statechart._ast.ASTExitAction.class;
619     }
620   ;
621
622   ast ExitAction_Optional =
623     method public ASTExitAction getRhs(){return getExitAction();}
624     method public ASTExitAction getLhs(){return getExitAction();}
625     method public Class _getTFElementType(){
626       return mc.testcases.statechart._ast.ASTExitAction.class;
627     }
628   ;
629
630   ast ExitAction_Pattern =
631     method public Class _getTFElementType(){
632       return mc.testcases.statechart._ast.ASTExitAction.class;
633     }
634     method public ASTExitAction_Pattern getLhs(){return this;}
635     method public ASTExitAction_Pattern getRhs(){return this;}
636   ;
637
638   ast ExitAction_Negation =
639     method public ASTExitAction getRhs(){return getExitAction();}
640     method public ASTExitAction getLhs(){return getExitAction();}
641     method public Class _getTFElementType(){
642       return mc.testcases.statechart._ast.ASTExitAction.class;
643     }
644   ;
645
646   ast ExitAction_List =
647     method public ASTExitAction getRhs(){return getExitAction();}
648     method public ASTExitAction getLhs(){return getExitAction();}
649     method public Class _getTFElementType(){
```

```
650       return mc.testcases.statechart._ast.ASTExitAction.class;
651     }
652   ;
653
654   ast EntryAction_Replacement =
655     method public Class _getTFElementType(){
656       return mc.testcases.statechart._ast.ASTEntryAction.class;
657     }
658   ;
659
660   ast EntryAction_Optional =
661     method public ASTEntryAction getRhs(){
662       return getEntryAction();
663     }
664     method public ASTEntryAction getLhs(){
665       return getEntryAction();
666     }
667     method public Class _getTFElementType(){
668       return mc.testcases.statechart._ast.ASTEntryAction.class;
669     }
670   ;
671
672   ast EntryAction_Pattern =
673     method public Class _getTFElementType(){
674       return mc.testcases.statechart._ast.ASTEntryAction.class;
675     }
676     method public ASTEntryAction_Pattern getLhs(){return this;}
677     method public ASTEntryAction_Pattern getRhs(){return this;}
678   ;
679
680   ast EntryAction_Negation =
681     method public ASTEntryAction getRhs(){return getEntryAction();}
682     method public ASTEntryAction getLhs(){return getEntryAction();}
683     method public Class _getTFElementType(){
684       return mc.testcases.statechart._ast.ASTEntryAction.class;
685     }
686   ;
687
688   ast EntryAction_List =
689     method public ASTEntryAction getRhs(){return getEntryAction();}
690     method public ASTEntryAction getLhs(){return getEntryAction();}
691     method public Class _getTFElementType(){
692       return mc.testcases.statechart._ast.ASTEntryAction.class;
693     }
694   ;
695
696   ast Statechart_Replacement =
697     method public Class _getTFElementType(){
698       return mc.testcases.statechart._ast.ASTStatechart.class;
699     }
700   ;
701
702   ast Statechart_Optional =
```

```
703     method public ASTStatechart getRhs(){return getStatechart();}
704     method public ASTStatechart getLhs(){return getStatechart();}
705     method public Class _getTFElementType(){
706       return mc.testcases.statechart._ast.ASTStatechart.class;
707     }
708   ;
709
710   ast Statechart_Pattern =
711     method public Class _getTFElementType(){
712       return mc.testcases.statechart._ast.ASTStatechart.class;
713     }
714     method public ASTStatechart_Pattern getLhs(){return this;}
715     method public ASTStatechart_Pattern getRhs(){return this;}
716   ;
717
718   ast Statechart_Negation =
719     method public ASTStatechart getRhs(){return getStatechart();}
720     method public ASTStatechart getLhs(){return getStatechart();}
721     method public Class _getTFElementType(){
722       return mc.testcases.statechart._ast.ASTStatechart.class;
723     }
724   ;
725
726   ast Statechart_List =
727     method public ASTStatechart getRhs(){return getStatechart();}
728     method public ASTStatechart getLhs(){return getStatechart();}
729     method public Class _getTFElementType(){
730       return mc.testcases.statechart._ast.ASTStatechart.class;
731     }
732   ;
733
734   ast TfIdentifier =
735     method public boolean isNewIdentifierFix(){
736       return !getNewIdentifier().startsWith("$");
737     }
738     method public boolean isIdentifierFix(){
739       return !getIdentifier().startsWith("$");
740     }
741   ;
742
743   interface Expression astextends /mc.tfcs.ast.ITFElement;
744
745   interface Statement astextends /mc.tfcs.ast.ITFElement;
746
747   interface SCStructure astextends /mc.tfcs.ast.ITFElement;
748
749   interface FieldAccess extends Expression
750       astextends /mc.tfcs.ast.ITFElement;
751
752   interface MethodInvocationWithQualifiedName extends Expression
753       astextends /mc.tfcs.ast.ITFElement;
754
755   interface Operator;
```

```
756
757  interface EqualityExpression extends Expression
758      astextends /mc.tfcs.ast.ITFElement;
759
760  interface ExpressionStatement extends Statement
761      astextends /mc.tfcs.ast.ITFElement;
762
763  interface BlockStatement extends Statement
764      astextends /mc.tfcs.ast.ITFElement;
765
766  interface Argument astextends /mc.tfcs.ast.ITFElement;
767
768  interface Transition astextends /mc.tfcs.ast.ITFElement;
769
770  interface Final;
771
772  interface Initial;
773
774  interface State extends SCStructure
775      astextends /mc.tfcs.ast.ITFElement;
776
777  interface InternTransition astextends /mc.tfcs.ast.ITFElement;
778
779  interface DoAction astextends /mc.tfcs.ast.ITFElement;
780
781  interface ExitAction astextends /mc.tfcs.ast.ITFElement;
782
783  interface EntryAction astextends /mc.tfcs.ast.ITFElement;
784
785  interface Statechart extends SCStructure
786      astextends /mc.tfcs.ast.ITFElement;
787
788  Expression_Optional implements Expression
789      astimplements /mc.tfcs.ast.IOptional =
790    "opt" ("<" "Expression" ">")?
791    SchemaVarName:IDENTVAR? "[[" Expression "]]";
792
793  Expression_List implements Expression
794      astimplements /mc.tfcs.ast.IList =
795    "list" ("<" "Expression" ">")? SchemaVarName:IDENTVAR?
796    "[[" Expression "]]";
797
798  Expression_Negation implements Expression
799      astimplements /mc.tfcs.ast.INegation =
800    "not" ("<" "Expression" ">")? SchemaVarName:IDENTVAR?
801    "[[" Expression "]]";
802
803  Expression_Replacement implements Expression
804      astimplements /mc.tfcs.ast.IReplacement =
805    ("[[" lhs:Expression? ":-" rhs:Expression? "]]");
806
807  Statement_Optional implements Statement
808      astimplements /mc.tfcs.ast.IOptional =
```

```
809    "opt" ("<" "Statement" ">")? SchemaVarName:IDENTVAR?
810    "[[" Statement "]]";
811
812  Statement_List implements Statement astimplements /mc.tfcs.ast.IList =
813    "list" ("<" "Statement" ">")? SchemaVarName:IDENTVAR?
814    "[[" Statement "]]";
815
816  Statement_Negation implements Statement
817      astimplements /mc.tfcs.ast.INegation =
818    "not" ("<" "Statement" ">")? SchemaVarName:IDENTVAR?
819    "[[" Statement "]]";
820
821  Statement_Replacement implements Statement
822      astimplements /mc.tfcs.ast.IReplacement =
823    ("[[" lhs:Statement? ":-" rhs:Statement? "]]");
824
825  SCStructure_Optional implements SCStructure
826      astimplements /mc.tfcs.ast.IOptional =
827    "opt" ("<" "SCStructure" ">")? SchemaVarName:IDENTVAR?
828    "[[" SCStructure "]]";
829
830  SCStructure_List implements SCStructure
831      astimplements /mc.tfcs.ast.IList =
832    "list" ("<" "SCStructure" ">")? SchemaVarName:IDENTVAR?
833    "[[" SCStructure "]]";
834
835  SCStructure_Negation implements SCStructure
836      astimplements /mc.tfcs.ast.INegation =
837    "not" ("<" "SCStructure" ">")? SchemaVarName:IDENTVAR?
838    "[[" SCStructure "]]";
839
840  SCStructure_Replacement implements SCStructure
841      astimplements /mc.tfcs.ast.IReplacement =
842    ("[[" lhs:SCStructure? ":-" rhs:SCStructure? "]]");
843
844  FieldAccess_Pattern implements FieldAccess
845      astimplements /mc.tfcs.ast.IPattern =
846    (Name:TfIdentifier) | "FieldAccess" SchemaVarName:IDENTVAR?
847    "[[" (Name:TfIdentifier) "]]" | "FieldAccess" SchemaVarName:IDENTVAR
848    ";";
849
850  FieldAccess_Replacement implements FieldAccess
851      astimplements /mc.tfcs.ast.IReplacement =
852    ("[[" lhs:FieldAccess? ":-" rhs:FieldAccess? "]]");
853
854  FieldAccess_Negation implements FieldAccess
855      astimplements /mc.tfcs.ast.INegation =
856    "not" ("<" "FieldAccess" ">")? SchemaVarName:IDENTVAR?
857    "[[" FieldAccess "]]";
858
859  FieldAccess_Optional implements FieldAccess
860      astimplements /mc.tfcs.ast.IOptional =
861    "opt" ("<" "FieldAccess" ">")? SchemaVarName:IDENTVAR?
```

```
862   "[[" FieldAccess "]]";
863
864 FieldAccess_List implements FieldAccess
865     astimplements /mc.tfcs.ast.IList =
866   "list" ("<" "FieldAccess" ">")? SchemaVarName:IDENTVAR?
867   "[[" FieldAccess "]]";
868
869 MethodInvocationWithQualifiedName_Pattern
870     implements MethodInvocationWithQualifiedName
871     astimplements /mc.tfcs.ast.IPattern =
872   (Name:TfIdentifier (options {greedy=true;} : "." Name:TfIdentifier)*
873   "(" (Arguments:Expression ("," Arguments:Expression)*)? ")")
874   | "MethodInvocationWithQualifiedName" SchemaVarName:IDENTVAR?
875   "[[" (Name:TfIdentifier (options {greedy=true;} : "."
876   Name:TfIdentifier)* "(" (Arguments:Expression
877   ("," Arguments:Expression)*)? ")") "]]"
878   | "MethodInvocationWithQualifiedName" SchemaVarName:IDENTVAR ";";
879
880 MethodInvocationWithQualifiedName_Replacement
881     implements MethodInvocationWithQualifiedName
882     astimplements /mc.tfcs.ast.IReplacement =
883   ("[[" lhs:MethodInvocationWithQualifiedName? ":-"
884   rhs:MethodInvocationWithQualifiedName? "]]");
885
886 MethodInvocationWithQualifiedName_Negation
887     implements MethodInvocationWithQualifiedName
888     astimplements /mc.tfcs.ast.INegation =
889   "not" ("<" "MethodInvocationWithQualifiedName" ">")?
890   SchemaVarName:IDENTVAR? "[[" MethodInvocationWithQualifiedName "]]";
891
892 MethodInvocationWithQualifiedName_Optional
893     implements MethodInvocationWithQualifiedName
894     astimplements /mc.tfcs.ast.IOptional =
895   "opt" ("<" "MethodInvocationWithQualifiedName" ">")?
896   SchemaVarName:IDENTVAR? "[[" MethodInvocationWithQualifiedName "]]";
897
898 MethodInvocationWithQualifiedName_List
899     implements MethodInvocationWithQualifiedName
900     astimplements /mc.tfcs.ast.IList =
901   "list" ("<" "MethodInvocationWithQualifiedName" ">")?
902   SchemaVarName:IDENTVAR? "[[" MethodInvocationWithQualifiedName "]]";
903
904 Operator_Negation implements Operator
905     astimplements /mc.tfcs.ast.IAttributeNegation =
906   "not" "[[" Operator_Pattern "]]";
907
908 Operator_Replacement implements Operator
909     astimplements /mc.tfcs.ast.IAttributeReplacement =
910   ("[[" lhs:Operator_Pattern ":-" "]]")
911   | ("[[" ":-" rhs:Operator_Pattern "]]");
912
913 Operator_Pattern implements Operator
914     astimplements /mc.tfcs.ast.IAttributePattern =
```

```
915    Operator:["!=" | "=="];
916
917  EqualityExpression_Pattern implements EqualityExpression
918      astimplements /mc.tfcs.ast.IPattern =
919    (LeftOperand:TfIdentifier Operator RightOperand:TfIdentifier)
920    | "EqualityExpression" SchemaVarName:IDENTVAR? "[["
921    (LeftOperand:TfIdentifier Operator RightOperand:TfIdentifier) "]]"
922    | "EqualityExpression" SchemaVarName:IDENTVAR ";";
923
924  EqualityExpression_Replacement implements EqualityExpression
925      astimplements /mc.tfcs.ast.IReplacement =
926    ("[[" lhs:EqualityExpression? ":-" rhs:EqualityExpression? "]]");
927
928  EqualityExpression_Negation implements EqualityExpression
929      astimplements /mc.tfcs.ast.INegation =
930    "not" ("<" "EqualityExpression" ">")? SchemaVarName:IDENTVAR?
931    "[[" EqualityExpression "]]";
932
933  EqualityExpression_Optional implements EqualityExpression
934      astimplements /mc.tfcs.ast.IOptional =
935    "opt" ("<" "EqualityExpression" ">")? SchemaVarName:IDENTVAR?
936    "[[" EqualityExpression "]]";
937
938  EqualityExpression_List implements EqualityExpression
939      astimplements /mc.tfcs.ast.IList =
940    "list" ("<" "EqualityExpression" ">")? SchemaVarName:IDENTVAR?
941    "[[" EqualityExpression "]]";
942
943  ExpressionStatement_Pattern implements ExpressionStatement
944      astimplements /mc.tfcs.ast.IPattern =
945    (Expression:Expression ";")
946    | "ExpressionStatement" SchemaVarName:IDENTVAR?
947    "[[" (Expression:Expression ";") "]]"
948    | "ExpressionStatement" SchemaVarName:IDENTVAR ";";
949
950  ExpressionStatement_Replacement implements ExpressionStatement
951      astimplements /mc.tfcs.ast.IReplacement =
952    ("[[" lhs:ExpressionStatement? ":-" rhs:ExpressionStatement? "]]");
953
954  ExpressionStatement_Negation implements ExpressionStatement
955      stimplements /mc.tfcs.ast.INegation =
956    "not" ("<" "ExpressionStatement" ">")? SchemaVarName:IDENTVAR?
957    "[[" ExpressionStatement "]]";
958
959  ExpressionStatement_Optional implements ExpressionStatement
960      astimplements /mc.tfcs.ast.IOptional =
961    "opt" ("<" "ExpressionStatement" ">")? SchemaVarName:IDENTVAR?
962    "[[" ExpressionStatement "]]";
963
964  ExpressionStatement_List implements ExpressionStatement
965      astimplements /mc.tfcs.ast.IList =
966    "list" ("<" "ExpressionStatement" ">")? SchemaVarName:IDENTVAR?
967    "[[" ExpressionStatement "]]";
```

```
968
969  BlockStatement_Pattern implements BlockStatement
970      astimplements /mc.tfcs.ast.IPattern =
971    ("{" (Statements:Statement)* "}")
972    | "BlockStatement" SchemaVarName:IDENTVAR?
973    "[[" ("{" (Statements:Statement)* "}") "]]"
974    | "BlockStatement" SchemaVarName:IDENTVAR ";";
975
976  BlockStatement_Replacement implements BlockStatement
977      astimplements /mc.tfcs.ast.IReplacement =
978    ("[[" lhs:BlockStatement? ":-" rhs:BlockStatement? "]]");
979
980  BlockStatement_Negation implements BlockStatement
981      astimplements /mc.tfcs.ast.INegation =
982    "not" ("<" "BlockStatement" ">")? SchemaVarName:IDENTVAR?
983    "[[" BlockStatement "]]";
984
985  BlockStatement_Optional implements BlockStatement
986      astimplements /mc.tfcs.ast.IOptional =
987    "opt" ("<" "BlockStatement" ">")? SchemaVarName:IDENTVAR?
988    "[[" BlockStatement "]]";
989
990  BlockStatement_List implements BlockStatement
991      astimplements /mc.tfcs.ast.IList =
992    "list" ("<" "BlockStatement" ">")? SchemaVarName:IDENTVAR?
993    "[[" BlockStatement "]]";
994
995  Argument_Pattern implements Argument
996      astimplements /mc.tfcs.ast.IPattern =
997    (ParamType:TfIdentifier ParamName:TfIdentifier) |
998    "Argument" SchemaVarName:IDENTVAR?
999    "[[" (ParamType:TfIdentifier ParamName:TfIdentifier) "]]"
1000   | "Argument" SchemaVarName:IDENTVAR ";";
1001
1002 Argument_Replacement implements Argument
1003     astimplements /mc.tfcs.ast.IReplacement =
1004   ("[[" lhs:Argument? ":-" rhs:Argument? "]]");
1005
1006 Argument_Negation implements Argument
1007     astimplements /mc.tfcs.ast.INegation =
1008   "not" ("<" "Argument" ">")? SchemaVarName:IDENTVAR?
1009   "[[" Argument "]]";
1010
1011 Argument_Optional implements Argument
1012     astimplements /mc.tfcs.ast.IOptional =
1013   "opt" ("<" "Argument" ">")? SchemaVarName:IDENTVAR?
1014   "[[" Argument "]]";
1015
1016 Argument_List implements Argument
1017     astimplements /mc.tfcs.ast.IList =
1018   "list" ("<" "Argument" ">")? SchemaVarName:IDENTVAR?
1019   "[[" Argument "]]";
1020
```

```
1021 Transition_Pattern implements Transition
1022     astimplements /mc.tfcs.ast.IPattern =
1023   (From:TfIdentifier "->" To:TfIdentifier
1024   (":" (Event:TfIdentifier ("("
1025   ((Arguments:Argument ("," Arguments:Argument)*)?) ")")?)?
1026   ("[" PreCondition:Expression "]")?
1027   ("/" Action:BlockStatement ("[" PostCondition:Expression "]")?)? ";"
1028   | ";")) | "Transition" SchemaVarName:IDENTVAR?
1029   "[[" (From:TfIdentifier "->" To:TfIdentifier
1030   (":" (Event:TfIdentifier
1031   ("(" ((Arguments:Argument ("," Arguments:Argument)*)?) ")")?)?
1032   ("[" PreCondition:Expression "]")?
1033   ("/" Action:BlockStatement ("[" PostCondition:Expression "]")?)? ";"
1034   | ";")) "]]" | "Transition" SchemaVarName:IDENTVAR ";";
1035
1036 Transition_Replacement implements Transition
1037     astimplements /mc.tfcs.ast.IReplacement =
1038   ("[[" lhs:Transition? ":-" rhs:Transition? "]]");
1039
1040 Transition_Negation implements Transition
1041     astimplements /mc.tfcs.ast.INegation =
1042   "not" ("<" "Transition" ">")? SchemaVarName:IDENTVAR?
1043   "[[" Transition "]]";
1044
1045 Transition_Optional implements Transition
1046     astimplements /mc.tfcs.ast.IOptional =
1047   "opt" ("<" "Transition" ">")? SchemaVarName:IDENTVAR?
1048   "[[" Transition "]]";
1049
1050 Transition_List implements Transition
1051     astimplements /mc.tfcs.ast.IList =
1052   "list" ("<" "Transition" ">")? SchemaVarName:IDENTVAR?
1053   "[[" Transition "]]";
1054
1055 Final_Negation implements Final
1056     astimplements /mc.tfcs.ast.IAttributeNegation =
1057   "not" "[[" Final_Pattern "]]";
1058
1059 Final_Replacement implements Final
1060     astimplements /mc.tfcs.ast.IAttributeReplacement =
1061   ("[[" lhs:Final_Pattern ":-" "]]")
1062   | ("[[" ":-" rhs:Final_Pattern "]]");
1063
1064 Final_Pattern implements Final
1065     astimplements /mc.tfcs.ast.IAttributePattern =
1066   Final:["final"];
1067
1068 Initial_Negation implements Initial
1069     astimplements /mc.tfcs.ast.IAttributeNegation =
1070   "not" "[[" Initial_Pattern "]]";
1071
1072 Initial_Replacement implements Initial
1073     astimplements /mc.tfcs.ast.IAttributeReplacement =
```

```
1074     ("[[" lhs:Initial_Pattern ":-" "]]")
1075     | ("[[" ":-" rhs:Initial_Pattern "]]");
1076
1077   Initial_Pattern implements Initial
1078       astimplements /mc.tfcs.ast.IAttributePattern =
1079     Initial:["initial"];
1080
1081   State_Pattern implements State
1082       astimplements /mc.tfcs.ast.IPattern =
1083     ("state" Name:TfIdentifier ("<<" (Initial | Final) ">>")* (("{"
1084     ("[" Invariant:Expression ("&&" Invariant:Expression)* "]")?
1085     (EntryAction:EntryAction)? (DoAction)? (ExitAction:ExitAction)?
1086     (States:State | Transitions:Transition
1087     | InternTransitions:InternTransition)* "}") | ";"))
1088     | "State" SchemaVarName:IDENTVAR? "[[" ("state" Name:TfIdentifier
1089     ("<<" (Initial | Final) ">>")* (("{" ("[" Invariant:Expression
1090     ("&&" Invariant:Expression)* "]")?
1091     (EntryAction:EntryAction)? (DoAction)? (ExitAction:ExitAction)?
1092     (States:State | Transitions:Transition
1093     | InternTransitions:InternTransition)* "}") | ";")) "]]"
1094     | "State" SchemaVarName:IDENTVAR ";";
1095
1096   State_Replacement implements State
1097       astimplements /mc.tfcs.ast.IReplacement =
1098     ("[[" lhs:State? ":-" rhs:State? "]]");
1099
1100   State_Negation implements State
1101       astimplements /mc.tfcs.ast.INegation =
1102     "not" ("<" "State" ">")? SchemaVarName:IDENTVAR? "[[" State "]]";
1103
1104   State_Optional implements State astimplements /mc.tfcs.ast.IOptional =
1105     "opt" ("<" "State" ">")? SchemaVarName:IDENTVAR? "[[" State "]]";
1106
1107   State_List implements State astimplements /mc.tfcs.ast.IList =
1108     "list" ("<" "State" ">")? SchemaVarName:IDENTVAR? "[[" State "]]";
1109
1110   InternTransition_Pattern implements InternTransition
1111       astimplements /mc.tfcs.ast.IPattern =
1112     ("-intern>" (":" (Event:TfIdentifier ("("
1113     (Arguments:Argument ("," Arguments:Argument)*) ")")?)?
1114     ("[" PreCondition:Expression "]")? ("/" Action:BlockStatement
1115     ("[" PostCondition:Expression "]")?)? ";" | ";"))
1116     | "InternTransition" SchemaVarName:IDENTVAR? "[[" ("-intern>"
1117     (":" (Event:TfIdentifier ("("
1118     (Arguments:Argument ("," Arguments:Argument)*) ")")?)?
1119     ("[" PreCondition:Expression "]")? ("/" Action:BlockStatement
1120     ("[" PostCondition:Expression "]")?)? ";" | ";")) "]]"
1121     | "InternTransition" SchemaVarName:IDENTVAR ";";
1122
1123   InternTransition_Replacement implements InternTransition
1124       astimplements /mc.tfcs.ast.IReplacement =
1125     ("[[" lhs:InternTransition? ":-" rhs:InternTransition? "]]");
1126
```

```
1127  InternTransition_Negation implements InternTransition
1128      astimplements /mc.tfcs.ast.INegation =
1129    "not" ("<" "InternTransition" ">")? SchemaVarName:IDENTVAR?
1130    "[[" InternTransition "]]";
1131
1132  InternTransition_Optional implements InternTransition
1133      astimplements /mc.tfcs.ast.IOptional =
1134    "opt" ("<" "InternTransition" ">")? SchemaVarName:IDENTVAR?
1135    "[[" InternTransition "]]";
1136
1137  InternTransition_List implements InternTransition
1138      astimplements /mc.tfcs.ast.IList =
1139    "list" ("<" "InternTransition" ">")? SchemaVarName:IDENTVAR?
1140    "[[" InternTransition "]]";
1141
1142  DoAction_Pattern implements DoAction
1143      astimplements /mc.tfcs.ast.IPattern =
1144    ("do" ":" Block:BlockStatement) | "DoAction" SchemaVarName:IDENTVAR?
1145    "[[" ("do" ":" Block:BlockStatement) "]]"
1146    | "DoAction" SchemaVarName:IDENTVAR ";";
1147
1148  DoAction_Replacement implements DoAction
1149      astimplements /mc.tfcs.ast.IReplacement =
1150    ("[[" lhs:DoAction? ":-" rhs:DoAction? "]]");
1151
1152  DoAction_Negation implements DoAction
1153      astimplements /mc.tfcs.ast.INegation =
1154    "not" ("<" "DoAction" ">")? SchemaVarName:IDENTVAR?
1155    "[[" DoAction "]]";
1156
1157  DoAction_Optional implements DoAction
1158      astimplements /mc.tfcs.ast.IOptional =
1159    "opt" ("<" "DoAction" ">")? SchemaVarName:IDENTVAR?
1160    "[[" DoAction "]]";
1161
1162  DoAction_List implements DoAction
1163      astimplements /mc.tfcs.ast.IList =
1164    "list" ("<" "DoAction" ">")? SchemaVarName:IDENTVAR?
1165    "[[" DoAction "]]";
1166
1167  ExitAction_Pattern implements ExitAction
1168      astimplements /mc.tfcs.ast.IPattern =
1169    ("exit" ":" Block:BlockStatement)
1170    | "ExitAction" SchemaVarName:IDENTVAR?
1171    "[[" ("exit" ":" Block:BlockStatement) "]]"
1172    | "ExitAction" SchemaVarName:IDENTVAR ";";
1173
1174  ExitAction_Replacement implements ExitAction
1175      astimplements /mc.tfcs.ast.IReplacement =
1176    ("[[" lhs:ExitAction? ":-" rhs:ExitAction? "]]");
1177
1178  ExitAction_Negation implements ExitAction
1179      astimplements /mc.tfcs.ast.INegation =
```

```
1180    "not" ("<" "ExitAction" ">")? SchemaVarName:IDENTVAR?
1181    "[[" ExitAction "]]";
1182
1183  ExitAction_Optional implements ExitAction
1184      astimplements /mc.tfcs.ast.IOptional =
1185    "opt" ("<" "ExitAction" ">")? SchemaVarName:IDENTVAR?
1186    "[[" ExitAction "]]";
1187
1188  ExitAction_List implements ExitAction
1189      astimplements /mc.tfcs.ast.IList =
1190    "list" ("<" "ExitAction" ">")? SchemaVarName:IDENTVAR?
1191    "[[" ExitAction "]]";
1192
1193  EntryAction_Pattern implements EntryAction
1194      astimplements /mc.tfcs.ast.IPattern =
1195    ("entry" ":" Block:BlockStatement)
1196    | "EntryAction" SchemaVarName:IDENTVAR?
1197    "[[" ("entry" ":" Block:BlockStatement) "]]"
1198    | "EntryAction" SchemaVarName:IDENTVAR ";";
1199
1200  EntryAction_Replacement implements EntryAction
1201      astimplements /mc.tfcs.ast.IReplacement =
1202    ("[[" lhs:EntryAction? ":-" rhs:EntryAction? "]]");
1203
1204  EntryAction_Negation implements EntryAction
1205      astimplements /mc.tfcs.ast.INegation =
1206    "not" ("<" "EntryAction" ">")? SchemaVarName:IDENTVAR?
1207    "[[" EntryAction "]]";
1208
1209  EntryAction_Optional implements EntryAction
1210      astimplements /mc.tfcs.ast.IOptional =
1211    "opt" ("<" "EntryAction" ">")? SchemaVarName:IDENTVAR?
1212    "[[" EntryAction "]]";
1213
1214  EntryAction_List implements EntryAction
1215      astimplements /mc.tfcs.ast.IList =
1216    "list" ("<" "EntryAction" ">")? SchemaVarName:IDENTVAR?
1217    "[[" EntryAction "]]";
1218
1219  Statechart_Pattern implements Statechart
1220      astimplements /mc.tfcs.ast.IPattern =
1221    ("statechart" Name:TfIdentifier "{"
1222    (States:State | Transitions:Transition)* "}")
1223    | "Statechart" SchemaVarName:IDENTVAR?
1224    "[[" ("statechart" Name:TfIdentifier
1225    "{" (States:State | Transitions:Transition)* "}") "]]"
1226    | "Statechart" SchemaVarName:IDENTVAR ";";
1227
1228  Statechart_Replacement implements Statechart
1229      astimplements /mc.tfcs.ast.IReplacement =
1230    ("[[" lhs:Statechart? ":-" rhs:Statechart? "]]");
1231
1232  Statechart_Negation implements Statechart
```

```
1233      astimplements /mc.tfcs.ast.INegation =
1234    "not" ("<" "Statechart" ">")? SchemaVarName:IDENTVAR?
1235    "[[" Statechart "]]";
1236
1237  Statechart_Optional implements Statechart
1238      astimplements /mc.tfcs.ast.IOptional =
1239    "opt" ("<" "Statechart" ">")? SchemaVarName:IDENTVAR?
1240    "[[" Statechart "]]";
1241
1242  Statechart_List implements Statechart
1243      astimplements /mc.tfcs.ast.IList =
1244    "list" ("<" "Statechart" ">")? SchemaVarName:IDENTVAR?
1245    "[[" Statechart "]]";
1246
1247  TfIdentifier =
1248    identifier:IDENTVAR
1249    | ("[[" identifier:IDENTVAR ":-" newIdentifier:IDENTVAR "]]")
1250    | "Name" identifier:IDENTVAR ";";
1251
1252  TfObjects =
1253    "{" ((Statechart_Pattern | Statechart_List | Statechart_Replacement)
1254    => Statechart
1255    | (EntryAction_Pattern | EntryAction_List | EntryAction_Replacement)
1256    => EntryAction
1257    | (ExitAction_Pattern | ExitAction_List | ExitAction_Replacement)
1258    => ExitAction
1259    | (DoAction_Pattern | DoAction_List | DoAction_Replacement)
1260    => DoAction
1261    | (InternTransition_Pattern | InternTransition_List
1262    | InternTransition_Replacement)=> InternTransition
1263    | (State_Pattern | State_List | State_Replacement)=> State
1264    | (Transition_Pattern | Transition_List | Transition_Replacement)
1265    => Transition
1266    | (Argument_Pattern | Argument_List | Argument_Replacement)
1267    => Argument
1268    | (BlockStatement_Pattern | BlockStatement_List
1269    | BlockStatement_Replacement)=> BlockStatement
1270    | (ExpressionStatement_Pattern | ExpressionStatement_List
1271    | ExpressionStatement_Replacement)=> ExpressionStatement
1272    | (EqualityExpression_Pattern | EqualityExpression_List
1273    | EqualityExpression_Replacement)=> EqualityExpression
1274    | (MethodInvocationWithQualifiedName_Pattern
1275    | MethodInvocationWithQualifiedName_List
1276    | MethodInvocationWithQualifiedName_Replacement)
1277    => MethodInvocationWithQualifiedName
1278    | (FieldAccess_Pattern | FieldAccess_List | FieldAccess_Replacement)
1279    => FieldAccess
1280    | (SCStructure_Pattern | SCStructure_List | SCStructure_Replacement)
1281    => SCStructure
1282    | (Statement_Pattern | Statement_List | Statement_Replacement)
1283    => Statement
1284    | (Expression_Pattern | Expression_List | Expression_Replacement)
1285    => Expression)*
```

```
1286     ("folding" "{" FoldingSet* "}")?
1287     ("where" "{" Constraint:BooleanExpression "}")? "}";
1288
1289   options {
1290     parser lookahead=4
1291     lexer lookahead=7
1292   }
1293
1294   concept attributes {
1295     syn Objects : /java.util.Map {
1296   }}
1297
1298   Expression_Pattern implements Expression
1299       astimplements /mc.tfcs.ast.IPattern =
1300     "Expression" SchemaVarName:IDENTVAR ";";
1301
1302   Statement_Pattern implements Statement
1303       astimplements /mc.tfcs.ast.IPattern =
1304     "Statement" SchemaVarName:IDENTVAR ";";
1305
1306   SCStructure_Pattern implements SCStructure
1307       astimplements /mc.tfcs.ast.IPattern =
1308     "SCStructure" SchemaVarName:IDENTVAR ";";
1309
1310 }
```

Anhang D.

Transformation zur Vereinfachung hierarchischer Statecharts

Diese Transformation vereinfacht Statecharts nach dem Verfahren, das in den Abschnitten 3.3 und 3.4 vorgestellt wurde.

Statechart-Transformation

```
1 package mc.tf.sc;
2
3 import java.util.logging.Logger;
4
5 module SimplifyStatecharts {
6
7   main() {
8     Logger log = mc.MCG.getLogger();
9     log.finer("createEmptyActionBlock");
10    loop createEmptyActionBlock();
11    log.finer("createEmptyEntryBlock");
12    loop createEmptyEntryBlock();
13    log.finer("createEmptyExitBlock");
14    loop createEmptyExitBlock();
15    log.finer("eliminateDo");
16    loop eliminateDo();
17    log.finer("expandInitial");
18    loop expandInitial();
19    log.finer("expandFinal");
20    loop expandFinal();
21    log.finer("forwardToInitial");
22    forwardToInitial();
23    log.finer("startFromFinal");
24    startFromFinal();
25    log.finer("eliminateEntryActions");
26    eliminateEntryActions();
27    log.finer("eliminateExitActions");
28    eliminateExitActions();
29    log.finer("resolveInternalTransitions");
30    loop resolveInternalTransition();
31    log.finer("propagateInvariantsAndRemoveHierarchy");
32    propagateInvariantsAndRemoveHierarchy();
33  }
34
35  /*====================================================================*/
```

```
36  /*======================== HELPER METHODS ============================*/
37  /*====================================================================*/
38
39  transformation moveTransitionToState(Transition $T, State $S) {
40    State $S [[ state $s_name {
41      [[ :- Transition $T; ]]
42    } ]]
43
44    [[ Transition $T; :- ]]
45  }
46
47  transformation collectStates(List<State> result) {
48    list<State> result [[ state $_; ]]
49  }
50
51  /*====================================================================*/
52  /*========================== COMPLETION ==============================*/
53  /*====================================================================*/
54
55  transformation createEmptyActionBlock() {
56    $src -> $target : $stimulus / [[ not [[ { } ]] :- { } ]] ;
57  }
58
59  transformation createEmptyEntryBlock() {
60    State $S [[ state $s_name {
61      [[ :- entry : { } ]]
62    } ]]
63    where { match.$S.getEntryAction() == null }
64  }
65
66  transformation createEmptyExitBlock() {
67    State $S [[ state $s_name {
68      [[ :- exit : { } ]]
69    } ]]
70    where { match.$S.getExitAction() == null }
71  }
72
73  transformation expandInitial() {
74    state $outer {
75      not [[ state $inner <<initial>>; ]]
76      list $L [[ state $_ << [[ :- initial ]] >> ; ]]
77    }
78    where {
79      !match.$L.isEmpty()
80    }
81  }
82
83  transformation expandFinal() {
84    state $outer {
85      not [[ state $inner <<final>>; ]]
86      list $L [[ state $_ << [[ :- final ]] >> ; ]]
87    }
88    where {
```

```
89        !match.$L.isEmpty()
90      }
91    }
92
93    /*==================================================================*/
94    /*======================== ELIMINATE DO ============================*/
95    /*==================================================================*/
96
97    transformation eliminateDo() {
98      state $s_name {
99        [[
100         entry : BlockStatement $B_ENTRY;
101       :-
102         entry : { BlockStatement $B_ENTRY; timer.set(this, delay); }
103       ]]
104
105       [[ do : BlockStatement $B_DO; :- ]]
106
107       [[
108         exit : BlockStatement $B_EXIT;
109       :-
110         exit : { timer.stop(this); BlockStatement $B_EXIT; }
111       ]]
112
113       [[ :- -intern>: timeout / {
114               timer.set(this, delay);
115               BlockStatement $B_DO;
116             }; ]]
117     }
118   }
119
120   /*==================================================================*/
121   /*===================== FORWARD TO INITIAL =========================*/
122   /*==================================================================*/
123
124   forwardToInitial() {
125       State $superstate;
126       List<State> $substates;
127       Transition $t;
128       loop if (searchTransitionForInitialSubstates($t,
129           $superstate, $substates)) {
130         for (State $substate : $substates) {
131           Transition $t_new = $t.deepClone();
132           moveTransitionToState($t_new, $superstate);
133           redirectTransitionToSubstate($t_new, $substate);
134         }
135         $t.delete();
136       }
137   }
138
139   transformation searchTransitionForInitialSubstates(Transition $T,
140       State $SUPER, List<State> $SUB) {
141     State $SUPER [[ state $outer {
```

```
142      list<State> $SUB [[ state $_ << initial >>; ]]
143    } ]]
144
145    Transition $T [[ $_ -> $outer; ]]
146
147    where { !match.$SUB.isEmpty() }
148  }
149
150  transformation redirectTransitionToSubstate(Transition $T,
151      State $S) {
152    state $outer {
153      State $S [[ state $inner << [[ initial :- ]] >>; ]]
154    }
155    Transition $T [[ $source -> [[ $outer :- $inner ]]; ]]
156  }
157
158  /*=================================================================*/
159  /*====================== START FROM FINAL ==========================*/
160  /*=================================================================*/
161
162  startFromFinal() {
163      State $superstate;
164      List<State> $substates;
165      Transition $t;
166      loop if (searchTransitionForFinalSubstates($t, $superstate,
167          $substates)) {
168        for (State $substate : $substates) {
169          Transition $t_new = $t.deepClone();
170          moveTransitionToState($t_new, $superstate);
171          startTransitionFromSubstate($t_new, $substate);
172        }
173        $t.delete();
174      }
175  }
176
177  transformation searchTransitionForFinalSubstates(Transition $T,
178      State $SUPER, List<State> $SUB) {
179    State $SUPER [[ state $outer {
180      list<State> $SUB [[ state $_ << final >>; ]]
181    } ]]
182
183    Transition $T [[ $outer -> $_; ]]
184
185    where { !match.$SUB.isEmpty() }
186  }
187
188
189  transformation startTransitionFromSubstate(Transition $T, State $S) {
190    state $outer {
191      State $S [[ state $inner << [[ final :- ]] >>; ]]
192    }
193    Transition $T [[ [[ $outer :- $inner ]] -> $target; ]]
194  }
```

```
195
196  /*=================================================================*/
197  /*==================== ELIMINATE ENTRY ACTIONS =======================*/
198  /*=================================================================*/
199
200  eliminateEntryActions() {
201    List<State> $allStates;
202    collectStates($allStates);
203    for (State $s : $allStates) {
204      List<Transition> $incoming_transitions;
205      BlockStatement $action;
206      if(hasEntryAction($s, $incoming_transitions, $action)) {
207        for (Transition $t : $incoming_transitions) {
208          Statement $statement = $action.deepClone();
209          if (addLastAction($t, $statement)) {}
210        }
211        if (deleteEntryAction($s)){ }
212      }
213    }
214  }
215
216  transformation hasEntryAction(State $S, List<Transition> $INCOMING,
217      BlockStatement $ACTION) {
218    State $S [[ state $s_name {
219        entry : BlockStatement $ACTION;
220    } ]]
221    list<Transition> $INCOMING [[ $_ -> $s_name; ]]
222  }
223
224  transformation deleteEntryAction(State $S) {
225    State $S [[ state $s_name {
226        [[ entry : { }  :- ]]
227    } ]]
228  }
229
230  transformation addLastAction(Transition $T, Statement $S) {
231    Transition $T [[
232      $source -> $target : $stimulus /
233        [[ BlockStatement $B; :- { BlockStatement $B; Statement $S; } ]];
234    ]]
235
236    [[ Statement $S; :- ]]
237  }
238
239  /*=================================================================*/
240  /*===================== ELIMINATE EXIT ACTIONS =======================*/
241  /*=================================================================*/
242
243  eliminateExitActions() {
244    List<State> $allStates;
245    collectStates($allStates);
246    for (State $s : $allStates) {
247      List<Transition> $outgoing_transitions;
```

```
248       BlockStatement $action;
249       if(hasExitAction($s, $outgoing_transitions, $action)) {
250         for (Transition $t : $outgoing_transitions) {
251           Statement $statement = $action.deepClone();
252           if (addFirstAction($t, $statement)) {}
253         }
254         if (deleteExitAction($s)){ }
255       }
256     }
257   }
258 
259   transformation hasExitAction(State $S, List<Transition> $OUTGOING,
260       BlockStatement $ACTION) {
261     State $S [[ state $s_name {
262         exit : BlockStatement $ACTION;
263     } ]]
264     list<Transition> $OUTGOING [[ $s_name -> $_; ]]
265   }
266 
267   transformation deleteExitAction(State $S) {
268     State $S [[ state $s_name {
269         [[ exit : { }  :- ]]
270     } ]]
271   }
272 
273   transformation addFirstAction(Transition $T, Statement $S) {
274     Transition $T [[
275       $source -> $target : $stimulus /
276         [[ BlockStatement $B; :- { Statement $S; BlockStatement $B; } ]];
277     ]]
278 
279     [[ Statement $S; :- ]]
280   }
281 
282   /*===================================================================*/
283   /*================== RESOLVE INTERNAL TRANSITIONS ====================*/
284   /*===================================================================*/
285 
286   transformation resolveInternalTransition() {
287     state $_ {
288       state $s {
289         [[ -intern>:$stimulus / BlockStatement $B; ; :- ]]
290       }
291 
292       [[ :- $s->$s : $stimulus / BlockStatement $B; ; ]]
293     }
294   }
295 
296   /*===================================================================*/
297   /*=========== PROPAGATE INVARIANTS AND REMOVE HIERARCHY ==============*/
298   /*===================================================================*/
299 
300   propagateInvariantsAndRemoveHierarchy() {
```

```
301     loop pullUpTransition();
302     State $s;
303     List<State> $substates;
304     loop if (findTopLevelStateWithSubstates($s, $substates)) {
305       for (State $next : $substates) {
306         List<Expression> $invariants;
307         getInvariants($s, $invariants);
308         for (Expression $inv : $invariants) {
309           Expression $inv_copy = (Expression) $inv.deepClone();
310           if (addInvariantToState($inv_copy, $next)) { }
311         }
312         pullUpState($next);
313       }
314       $s.delete();
315     }
316   }
317
318   transformation getInvariants(State $S, List<Expression> $E) {
319     State $S [[ state $s_name {
320       [ list<Expression>  [[ Expression $E; ]] ]
321     } ]]
322   }
323
324   transformation pullUpTransition() {
325       [[ Transition $T; :- ]]
326       Statechart $SC [[ statechart $sc_name {
327         [[ :- Transition $T; ]]
328       } ]]
329       where { ! match.$SC.getTransitions().contains(match.$T) }
330   }
331
332   transformation pullUpState(State $S) {
333       [[ State $S; :- ]]
334       statechart $sc_name {
335         [[ :- State $S; ]]
336       }
337   }
338
339   transformation findTopLevelStateWithSubstates(State $S, List<State> $SUB) {
340     statechart $sc_name {
341       State $S[[ state $s_name {
342         list<State> $SUB [[ state $_ ; ]]
343       } ]]
344     } where { !match.$SUB.isEmpty() }
345   }
346
347   transformation addInvariantToState(Expression $INV, State $S) {
348        [[ Expression $INV; :- ]]
349        State $S [[ state $s_name {
350         [ [[ :- Expression $INV; ]] ]
351       } ]]
352   }
```

```
353
354 }
```

Anhang E.

Lebenslauf

Name	Weisemöller
Vorname	Ingo
Geburtstag	10.11.1978
Geburtsort	Osnabrück
Staatsangehörigkeit	deutsch

2009 - 2011	Wissenschaftlicher Mitarbeiter am Lehrstuhl Software Engineering, RWTH Aachen
2006 - 2009	Wissenschaftlicher Mitarbeiter am Fachgebiet Echtzeitsysteme, TU Darmstadt
2006	Beginn der Promotion
2006	Abschluss als Diplom-Informatiker an der RWTH Aachen
2000 - 2006	Studium der Informatik an der RWTH Aachen
1999 - 2000	Studium der Eletrotechnik an der Ruhr-Universität Bochum
1998 - 1999	Zivildienst
1998	Abitur am Gymnasium Carolinum, Osnabrück
1985 - 1998	Grundschule, Orientierungsstufe und Gymnasium