SE | RWTH AACHEN UNIVERSITY

Katrin Hölldobler

MontiTrans: Agile, modellgetriebene Entwicklung von und mit domänenspezifischen, kompositionalen Transformationssprachen

Aachener Informatik-Berichte,
Software Engineering

Band 36

Hrsg: Prof. Dr. rer. nat. Bernhard Rumpe

MontiTrans: Agile, modellgetriebene Entwicklung von und mit domänenspezifischen, kompositionalen 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

Katrin Hölldobler,
M.Sc.RWTH
aus Haan

Berichter: Universitätsprofessor Dr. Bernhard Rumpe
Universitätsprofessor Dr. Ralf Lämmel

Tag der mündlichen Prüfung: 11. September 2018

[Hoe18] K. Hölldobler:
MontiTrans: Agile, modellgetriebene Entwicklung von und mit domänenspezifischen, kompositionalen Transformationssprachen.
Shaker Verlag, ISBN 978-3-8440-6322-6. Aachener Informatik-Berichte, Software Engineering, Band 36. Dezember 2018.
www.se-rwth.de/publications/

Kurzfassung

Modelle sind die zentralen Entwicklungsartefakte der modellgetriebenen Softwareentwicklung [CE00] und müssen entsprechend überarbeitet, weiterentwickelt und gewartet werden. Daher sind Modelltransformationen ebenfalls essenziell für die modellgetriebene Softwareentwicklung [SK03]. Während domänenspezifische Sprachen (DSLs) zur Modellierung mittlerweile weitverbreitet sind, sind spezifische Transformationssprachen rar. Stattdessen werden General Purpose Transformationssprachen (GPTLs) verwendet, die Transformationen basierend auf der internen Repräsentation der Modelle formulieren. Diese interne Repräsentation ist Modellierern und Domänenexperten in der Regel unbekannt, was deren Einbindung in den Entwicklungsprozess deutlich erschwert. Domänenspezifische Transformationssprachen (DSTLs) basieren auf der den Modellierern bekannten konkreten Syntax der DSL [BW07], wodurch sie spezifisch für die zugehörige DSL sind. Dies verringert den initialen Aufwand zum Erlernen der Transformationssprache, da der größte Teil der Syntax bereits bekannt ist und zusätzlich nur die Syntax der Transformationsoperatoren erlernt werden muss. Andererseits haben DSTLs durch ihre Zugehörigkeit zu einer DSL den Nachteil, dass für jede neu entwickelte DSL gleichzeitig oder für existierende DSLs nachträglich eine DSTL entwickelt werden muss. Aus diesem Grund erhöht sich der Aufwand der DSL-Entwicklung deutlich. Darüber hinaus erhöht sich der Aufwand weiter, wenn DSTLs zur Übersetzung zwischen verschiedenen Sprachen benötigt werden und entwickelt werden müssen. Zur Reduzierung dieses Aufwands wurde in [Wei12] ein erster DSTL-Generator vorgestellt. Dieser wurde mithilfe der Language Workbench MontiCore 2 entwickelt. Besondere Merkmale von MontiCore 4 sind die Möglichkeiten zur modularen Sprachdefinition durch Sprachkomposition [HLMSN+15b] sowie die an die Objektorientierung angelehnten Konzepte der Nichtterminalerweiterung als Erweiterung des Alternativenkonzepts von Grammatiken [Kra10]. Der bisherige DSTL-Generator ermöglicht eine Generierung von DSTLs aus MontiCore-Grammatiken und unterstützt ausschließlich monolithische Sprachdefinitionen. Die Mehrzahl der mit MontiCore entwickelten Modellierungssprachen sind hingegen kompositional definiert. Die bisher generierten DSTLs unterstützen ausschließlich Transformationen von Modellen innerhalb einer Modellierungssprache. Übersetzungen zwischen verschiedenen Sprachen oder die Migration von Modellen zwischen Sprachversionen ist somit nicht möglich. Des Weiteren wird bisher nur ein Teil der möglichen Konzepte von MontiCore-Grammatiken unterstützt, auf den sich die restlichen Konzepte abbilden lassen. Dadurch sind Sprachentwickler in der Sprachdefinition eingeschränkt.

Im Rahmen dieser Dissertation wird die generative Entwicklung und Verwendung von DSTLs in der modellgetriebenen Softwareentwicklung durch MontiTrans unterstützt. MontiTrans ermöglicht die Entwicklung neuer DSTLs und der zugehörigen Infrastruktur zur Spezifikation und Ausführung von Modelltransformationen. Für die Entwicklung von MontiTrans wurden die zuvor beschriebenen offenen Punkte aufgegriffen und die Generierung von DSTLs basierend auf den Ergebnissen aus [Wei12] weiterentwickelt.

Die wichtigsten Ergebnisse dieser Arbeit sind:

- Eine Systematik zur Ableitung von domänenspezifischen Transformationssprachen aus domänenspezifischen Sprachen.
- Ein Generator für domänenspezifische Transformationssprachen, der diese Systematik umsetzt, und alle Features des MontiCore 4-Grammatikformats unterstützt.
- Speziell angepasste DSTLs für die Sprachen CD4Analysis und CD4Code zur Klassendiagrammodellierung und die Architekturbeschreibungssprache MontiArc.
- Bibliotheken von wiederverwendbaren Transformationen für Klassendiagramme und Modelle der Architekturbeschreibungssprache MontiArc.
- Methodiken zum Einsatz von MontiTrans, DSTLs und Transformationen in der modellgetriebenen Softwareentwicklung.
- Eine Methodik zur handgeschriebenen Erweiterung der generierten DSTLs.

MontiTrans wurde sowohl zur Entwicklung der DSTLs als auch zur Entwicklung der Bibliotheken von wiederverwendbaren Transformationen verwendet. Hierdurch konnte gezeigt werden, dass MontiTrans ein umfassendes Werkzeug zur Entwicklung von DSTLs sowie für die Entwicklung von Modelltransformationen innerhalb modellgetriebener Softwareentwicklungsprojekte ist. MontiTrans erleichtert sowohl Sprachentwicklern die Entwicklung neuer DSLs und zugehöriger DSTLs als auch Transformationsentwicklern die Definition und Anwendung neuer Transformationen.

Abstract

Models are the central development artifact in model-driven software development [CE00]. These models need to be refactored, evolved and maintained. Thus, model transformations are indispensable for model-driven software development [SK03]. While using domain-specific languages (DSLs) has become common practice, tailored transformation languages are still uncommon. Instead, general-purpose transformation languages (GPTLs) are used that specify transformation based on the internal representation of models. Modelers and domain experts are typically unfamiliar with this representation, which hampers their integration into the development process. In contrast, DSTLs are based on the concrete syntax of the modeling language that is known by modelers [BW07]. Therefor DSTLs are specific for their corresponding modeling language. This reduces the initial effort to learn using the transformation language as the majority of the syntax is already familiar and only the transformation operators need to be understood in addition. However, a downside of DSTLs is that one DSTL need to be developed for every DSL which results in significantly increased development effort for DSLs. In addition, this effort further increases in case DSTLs to translate between different modeling languages are needed. In [Wei12] a DSTL generator was developed to reduce this effort. This generator was developed using the language workbench MontiCore in version 2. MontiCore in version 4 provides modular language definitions via language composition [HLMSN+15b] as well as nonterminal extension features inspired by object orientation [Kra10]. The previous DSTL generator generates DSTLs from MontiCore grammars but only supports monolithic language definitions. However, the majority of languages developed using MontiCore is defined compositionally. Furthermore, the generated DSTLs only supported transformations of models within one modeling language. Thus, translation or migration between two languages was not possible. In addition, only a subset of the MontiCore grammar features were supported by the DSTL generator, which requires language developer to only use the supported features.

In this dissertation, MontiTrans supports the generative development of DSTLs and their usage in model-driven software development projects. MontiTrans facilitates developing new DSTLs and their infrastructure to specify and apply model transformations. For developing MontiTrans the open issues described above were considered and the generation of DSTLs based on the results of [Wei12] improved.

The main contributions of this thesis are:

- A systematics to derive domain-specific transformation languages from domain-specific languages.
- A generator for domain-specific transformation languages that implements this systematics and supports all features of the MontiCore grammar format.
- Specially tailored DSTLs for the modeling languages CD4Analysis and CD4Code to model class diagrams and the architecture description language MontiArc.
- Libraries of reusable transformations for class diagrams and MontiArc models.

- Methodologies to use MontiTrans, DSTLs and transformations in model-driven software development.
- A methodology to manually adapt and extend generated DSTLs.

MontiTrans was used to develop the DSTLs as well as the libraries of reusable transformations, which demonstrates that MontiTrans is a comprehensive tool to develop DSTLs and transformations within model-driven software development projects. MontiTrans facilitates developing DSLs and corresponding DSTLs for language engineers and specifying and applying transformations for transformation developers.

Danksagung

Ich möchte diese Worte nutzen, um mich bei den lieben Menschen zu bedanken, die mich auf dem Weg zur Promotion begleitet, unterstützt sowie motiviert oder an passender Stelle abgelenkt haben. Sie alle haben zum Gelingen dieses Vorhaben beigetragen.

Mein erster Dank gilt meinem Doktorvater Prof. Dr. Bernhard Rumpe für die Möglichkeit der Promotion am Lehrstuhl für Software Engineering. Durch konstruktive Diskussionen, Tipps und Ideen hat er diese Arbeit mitgeformt. Neben der spannenden, akademischen Arbeit durfte ich auch praktische Erfahrungen in verschiedenen Modellierungsprojekten sowie durch die Leitung der Arbeitsgruppe *Modellierung* und des MontiCore-Projekts sammeln. Auch für diese Erfahrungen bin ich dankbar.

Als nächstes möchte ich Prof. Dr. Ralf Lämmel für die Bereitschaft, das Zweitgutachten dieser Arbeit zu übernehmen, danken. Darüber hinaus bedanke ich mich bei Prof. Dr. Martin Grohe für die Leitung meines Prüfungskomitees und Prof. Dr. Thomas Noll für die Mitarbeit in diesem Komitee.

Bedanken möchte ich mich auch bei Dr. Ingo Weisemöller, der mich schon während meines Studiums für das Thema Modelltransformationen begeistern konnte und auf dessen Arbeit ich aufbauen konnte. Herzlich bedanken möchte ich mich auch bei allen Kollegen, die mich während und teils auch außerhalb meiner Zeit am Lehrstuhl begleitet haben: Kai Adam, Vincent Bertram, Marita Breuer, Lennart Bucher, Arvid Butting, Gereon Bürvenich, Manuela Dalibor, Imke Drave, Florian Donath, Robert Eikermann, Angelika Fleck, Sylvia Gunder, Timo Greifenberg, Dr. Arne Haber, Robert Heim, Lars Hermerschmidt, Dr. Christoph Herrmann, Gabriele Heuschen, Steffen Hillemacher, Andreas Horst, Oliver Kautz, Thomas Kurpick, Evgeny Kusmenko, Achim Lindt, Dr. Markus Look, Ben Mainz, Matthias Markthaler, Dr. Judith Michael, Dr. Pedram Mir Seyed Nazari, Dr. Klaus Müller, Antonio Navarro Pérez, Lukas Netz, Sebastian Oberhoff, Jerome Pfeiffer, Nina Pichler, Dr. Claas Pinkernell, Dr. Dimitri Plotnikov, Manuel Pützer, Deni Raco, Dr. Dirk Reiss, Dr. Jan Oliver Ringert, Dr. Alexander Roth, Dr. Martin Schindler, Steffi Schrader, Christoph Schulze, Brian Sinkovec, Galina Volkova, Max Voß, Michael von Wenckstern und Dr. Andreas Wortmann. Weiterhin möchte ich mich bei Achim, Andreas, Arvid, David, Dimitri, Heinz, Hille, Markus, Oliver, Robert, Steffen und Timo für das Sichten und Kommentieren früher Fassungen dieser Arbeit bedanken.

Auch möchte ich mich bei den Kollegen und Freunden bedanken, die immer ein offenes Ohr für mich hatten und mich mit Rat und Tat unterstützt haben. Darüber hinaus bedanke ich mich bei allen beteiligten Hiwis, Studenten und Abschlussarbeitern, die die Entwicklung von MontiTrans unterstützt haben. Vielen Dank für eure Unterstützung, sie hat mir sehr geholfen.

An dieser Stelle möchte ich außerdem meiner Familie und insbesondere meiner Mutter Ute meinen besonderen Dank aussprechen. Sie hat mir diesen Weg überhaupt erst ermöglicht, mich auf diesem Weg unterstützt und war jederzeit für mich da. Zusätzlich möchte ich mich auch bei Mechthild und Heinz, Hille und Detlef sowie Ralf bedanken. Auch auf ihre Unterstützung konnte ich mich immer verlassen.

Schließlich möchte ich mich noch ganz besonders bei meiner Partnerin Isa bedanken.

Sie hat mich stets unterstützt, mich motiviert und mir den Rücken freigehalten, sodass ich mich auf meine Promotion konzentrieren konnte, selbst wenn darunter die gemeinsame Zeit leiden musste. Ich danke dir!

Aachen, September 2018
Katrin Hölldobler

Inhaltsverzeichnis

Teil I

Grundlagen

Kapitel 1

Einführung

In der modellgetriebenen Softwareentwicklung sind Modelle die zentralen Entwicklungsartefakte [CE00]. Zur Entwicklung dieser Modelle werden unterschiedlichste Modellierungssprachen verwendet. Die Unified Modeling Language (UML) [OMG15] ist als universell einsetzbare Sammlung von Modellierungssprachen hierbei eine häufige Wahl. Demgegenüber stehen domänenspezifische Sprachen (DSLs). DSLs sind an eine konkrete Domäne oder einen Anwendungsfall angepasst, wie beispielsweise die Structured Query Language (SQL) für Datenbankabfragen. DSLs werden innerhalb der modellgetriebenen Softwareentwicklung aus vielen Gründen eingesetzt. Wichtige Gründe sind beispielsweise die geringere Komplexität von Modellen und die leichtere Kommunikation mit Domänenexperten [MHS05, Fow10].

Werden Modelle als zentrale Entwicklungsartefakte verwendet, müssen diese überarbeitet, weiterentwickelt und gewartet werden. Aus diesem Grund bilden neben Modellen auch Modelltransformationen einen zentralen Bestandteil der modellgetriebenen Softwareentwicklung [SK03, Läm04]. Um die verschiedenen Arten von Modelltransformationen wie Refactorings, Migration oder Übersetzungen zu formulieren [MVG06, CH06], wurden mit der Zeit zahlreiche Modelltransformationssprachen entwickelt.

Während im Bereich der Modellierungssprachen DSLs zur Modellierung inzwischen verbreitet sind, sind spezifische Transformationssprachen weiterhin rar. Stattdessen werden häufig General Purpose Transformationssprachen (GPTLs) wie beispielsweise die Atlas Transformation Language (ATL) [JABK08] verwendet. GPTLs verwenden zur Beschreibung von Transformationen eine generische Notation, die in der Regel auf der abstrakten Syntax der Modelle basiert. Damit basieren sie auf der internen Repräsentation der Modelle, die Modellierern und Domänenexperten in der Regel unbekannt ist. Modellierer und Domänenexperten kennen als Nutzer einer DSL meist lediglich die zur Modellierung verwendete, konkrete Syntax. Ähnlich wie General Purpose Languages (GPLs) haben GPTLs damit den Nachteil, dass sie nicht die Terminologie der Domäne verwenden, sondern eine generische, allgemein verwendbare. Dem gegenüber stehen domänenspezifische Transformationssprachen [BW07, GMP09, RW11].

Domänenspezifische Transformationssprachen (DSTLs) verwenden zur Beschreibung von Transformationen die konkrete Syntax der Modellierungssprache [BW07, GMP09, RW11]. Durch die Verwendung der konkreten Syntax sind DSTLs spezifisch für die zugehörige Modellierungssprache. Durch diese Zugehörigkeit kann die DSTL jedoch nur für genau diese DSL verwendet werden. Auch wenn sich in der Literatur einige Beispiele für DSTLs finden lassen, wie beispielsweise die DSTL für Gebäudemodelle in [SD11]

oder für Aktivitätsdiagramme in [GMPO09b], gibt es weiterhin eine Vielzahl von DSLs ohne passende DSTLs. DSTLs haben den Nachteil, dass sie für jede DSL entwickelt werden müssen. Damit muss für jede neu entwickelte DSL gleichzeitig oder für existierende DSLs nachträglich eine DSTL entwickelt werden, wodurch sich der Aufwand der DSL-Entwicklung deutlich erhöht. Darüber hinaus erhöht sich der Aufwand weiter, wenn DSTLs zur Übersetzung zwischen verschiedenen Sprachen benötigt werden und entwickelt werden müssen.

Um diesen Aufwand zu reduzieren, wurde in [Wei12] ein Generator für DSTLs vorgestellt. Dieser Generator wurde mithilfe der Language Workbench MontiCore [KRV08, GKR+08, Kra10, KRV10] entwickelt. MontiCore erlaubt sowohl die Entwicklung textueller Modellierungssprachen als auch die Entwicklung vornehmlich Template-basierter Codegeneratoren. Hierbei setzt MontiCore auf ein Grammatikformat, dass an die erweiterte Backus-Naur Form (EBNF) angelehnt ist. Basierend auf einer solchen Grammatik generiert MontiCore die nötige Infrastruktur zur Verarbeitung von Modellen bestehend unter anderem aus Lexer und Parser, Klassen der internen Repräsentation (abstrakten Syntax) von Modellen und einer Infrastruktur zur Definition und Überprüfung von Wohlgeformtheitsprüfungen (Kontextbedingungen). Besondere Merkmale von MontiCore in Bezug auf die Sprachentwicklung sind die Möglichkeiten zur modularen Sprachdefinition durch Sprachkomposition [LNPR+13, HLMSN+15b, HLMSN+15a] sowie die an die Objektorientierung angelehnten Konzepte der Nichtterminalerweiterung als Erweiterung des Alternativenkonzepts von Grammatiken [Kra10].

Basierend auf MontiCore wurde eine Reihe textueller Modellierungssprachen entwickelt. Im Bereich der UML wurde hierzu das UML Profil UML/P [Sch12, Rum16, Rum17] umgesetzt. UML/P besteht aus zur Programmierung geeigneten Varianten der Modellierungssprachen für Klassendiagramme (CDs), Statechart (SCs), Objektdiagramm (ODs), Sequenzdiagramm (SDs) und der Object Constraint Language (OCL). In [Rot17] wurden basierend auf der UML/P Klassendiagrammsprache zwei Klassendiagrammsprachvarianten entwickelt: *CD4Analysis* und *CD4Code*. Hierbei ist CD4Analysis zur Modellierung von Analysemodellen und CD4Code zur Modellierung von implentierungsnahen Modellen geeignet. Einen weiteren Bereich der Modellierungssprachen, die auf MontiCore basieren, bilden die Architekturmodellierungssprachen. In diesem Bereich wurden verschiedene Varianten von Architekturbeschreibungssprachen (ADLs) entwickelt. Die Basis dieser ADLs stellt hierbei MontiArc [HRR10, HRR12, Hab16] dar. Diese erlaubt die Definition von Component & Connector Architekturen [MT00]. Ausgehend von MontiArc wurden weitere Varianten, wie MontiArcAutomaton [RRW15, HKR+16, BRW16, Wor16] mit eingebettetem Verhalten oder MontiSecArc [HHRW15] zur Security-Modellierung, entwickelt. Bereits dieser kleine Auszug aus der Menge der mit MontiCore entwickelten Modellierungssprachen verdeutlicht den Bedarf eines Generators zur Erstellung passender DSTLs.

Der in [Wei12] entwickelte DSTL-Generator hat zum Ziel für MontiCore-Grammatiken eine Generierung von DSTLs zu ermöglichen. Der Fokus lag hierbei auf der Unterstützung von Refactorings und semantikerhaltenden Transformationen [Wei12]. Dieser Generator unterstützt lediglich monolithische Modellierungssprachdefinitionen, während die

Mehrzahl der Modellierungssprachen, die mit MontiCore entwickelt werden, kompositionale Sprachdefinitionen hat. Durch den Fokus auf Refactorings und semantikerhaltende Transformationen unterstützten die generierten DSTLs ausschließlich Transformationen von Modellen innerhalb einer Modellierungssprache. Das bedeutet, dass weder Übersetzungen zwischen verschiedenen Sprachen noch die Migration von Modellen zwischen Sprachversionen möglich ist. Des Weiteren wird nur ein Teil der möglichen Konzepte von MontiCore-Grammatiken unterstützt. Zwar stellt dies keine echte Einschränkung dar, da die fehlenden Konzepte auf die unterstützten abbildbar sind, dennoch bedeutet es, dass Sprachentwickler in der Sprachdefinition eingeschränkt sind. Eine Kombination von Transformationen und templatebasierter Generierung ist durch die generierten DSTLs nicht explizit vorgesehen. Existierende Generatoren wie der MontiCore-Generator [Kra10], der MontiArc-Generator [Hab16] oder der Data Explorer-Generator [MSNRR15, Rot17], die mit MontiCore entwickelt wurden, verwenden jedoch eine Kombination aus handgeschriebenen Transformationen und templatebasierter Generierung. Diese Generatoren verdeutlichen, dass eine Integration der mithilfe von DSTLs entwickelten Transformationen und templatebasierter Generierung notwendig ist.

Die aufgezeigten Anknüpfungspunkte wurden im Rahmen dieser Arbeit für die Entwicklung von MontiTrans aufgegriffen und die Generierung von DSTLs weiterentwickelt. MontiTrans basiert auf den Ergebnissen aus [Wei12] und ermöglicht die Generierung von DSTLs für kompositionale Sprachdefinitionen. Außerdem kann MontiTrans dazu genutzt werden DSTLs zur Übersetzung von Modellen zwischen verschiedenen Sprachen zu entwickeln. Zusätzlich können auch DSTLs zur gleichzeitigen Transformation mehrerer Modelle verschiedener Modellierungssprachen mit MontiTrans erstellt werden. Schließlich wurden ergänzend zu den Ergebnissen aus [Wei12] auch die Nutzung der DSTL-Generierung für realistische, existierende und in Verwendung befindliche DSLs untersucht. Hierzu wurden DSTLs für die Modellierungssprachen CD4Analysis, CD4-Code sowie MontiArc entwickelt und speziell an diese angepasst. Außerdem wurden Bibliotheken von wiederverwendbaren Transformationen entwickelt, die aufzeigen, dass und wie diese DSTLs zur Modelltransformation genutzt werden können. Der Begriff der Wiederverwendbarkeit zielt hierbei auf die Wiederverwendung für verschiedene Modelle der gleichen Modellierungssprache ab. Damit ist der Begriff in dieser Arbeit enger gefasst als beispielsweise in [Läm02], da in [Läm02] auch die Verwendung für Modelle verschiedener Modellierungssprachen betrachtet wird. Abschließend wurden auf Basis dieser Erfahrungen Methodiken zur Entwicklung von DSTLs und Modelltransformationen entwickelt.

Im Folgenden werden die Ziele dieser Arbeit detailliert. Anschließend werden die Ergebnisse dieser Arbeit vorgestellt und abschließend der Aufbau dieser Arbeit beschrieben.

1.1 Ziele dieser Arbeit

Im Rahmen dieser Arbeit wird MontiTrans als Generator für DSTLs und Framework zur modellgetriebenen Entwicklung von Transformationen vorgestellt. Die zentrale Fragestellung, die in dieser Arbeit untersucht wurde, ist:

Wie lässt sich die agile, modellgetriebene Softwareentwicklung durch generierte domänenspezifische Transformationssprachen sowie durch Modelltransformationen, die zu diesen Transformationssprachen konform sind, unterstützen?

Ziel dieser Arbeit ist die Weiterentwicklung des in [Wei12] entwickelten Generators für DSTLs, um Methodiken zur Entwicklung von DSTLs sowie Modelltransformationen zu schaffen. Die Forschungsfrage deutet bereits an, dass hier verschiedene Metaebenen (Transformationssprachen und Modelltransformationen) und Nutzer involviert sind. Die DSTLs werden in der Regel von einer anderen Person entwickelt als die zu diesen konformen Transformationen. Darüber hinaus können sich auch die Entwickler von den Nutzern von Transformationen unterscheiden.

DSTLs haben den Vorteil, dass sie die konkrete Syntax der Modellierungssprache zur Spezifikation von Transformationen verwenden. Damit verwenden sie ein Vokabular, das dem Modellierer vertraut ist und das auch für Domänenexperten geeignet ist. Durch diesen Vorteil sind DSTLs jedoch spezifisch für eine zugehörige Modellierungssprache, wodurch für jede Modellierungssprache eine eigene DSTLs entwickelt werden muss. Das Ziel ist daher den Aufwand für Sprachentwickler möglichst gering zu halten, ohne hierbei die Nutzbarkeit aus Sicht der Transformationsentwickler zu beeinträchtigen. Eine DSTL soll hierbei für Modellierer einfach zu erlernen sein, das heißt sich in Syntax und Struktur an der Modellierungssprache orientieren. Hierbei soll sie jedoch gleichzeitig nicht in ihrer Mächtigkeit eingeschränkt werden. Transformationen müssen generalisierbar sein und die typischen Create, Read, Update, Delete (CRUD) Operationen für einzelne DSL-Sprachelemente unterstützen. Für Nutzer der Transformationen – im Kontext der modellgetriebenen Entwicklung also Modellierer sowie Generatorentwickler – sind wiederverwendbare Transformationen von Interesse, außerdem müssen sie sich in die Entwicklung von Generatoren integrieren lassen.

Die obige Forschungsfrage lässt sich entsprechend der vorgestellten Ebenen und Nutzern daher in die folgenden Teilforschungsfragen unterteilen. Eine Beantwortung dieser Teilforschungsfragen ermöglicht die Beantwortung der übergeordneten Forschungsfrage.

RQ1 Wie kann die Generierung von DSTLs aus DSLs basierend auf dem in [Wei12] vorgestellten Generator verbessert und komplettiert werden?

- **RQ1.1** Welche (weiteren) Grammatikkonzepte müssen durch die Generierung von DSTLs berücksichtigt werden?
- **RQ1.2** Welche Features muss eine generierte DSTLs bieten und welche zusätzlichen Konzepte können die Verwendung der abgeleiteten DSTLs weiter verbessern?
- **RQ1.3** Wie können generierte DSTLs und dazu zugehörige Generatoren an die Bedürfnisse der Nutzer angepasst werden?

RQ2 Wie kann eine generierte DSTL und deren Infrastruktur handgeschrieben erweitert werden und wie muss die generierte DSTL und die dazu passende Infrastruktur dafür strukturiert sein?

RQ3 Wie kann eine DSTL zum Beispiel für Sprachen wie CD4Code und CD4-Analysis generativ entwickelt werden, welche handgeschriebenen Erweiterungen sind für deren Benutzbarkeit notwendig und wie kann die Verwendung der DSTL durch wiederverwendbare Modelltransformationen unterstützt werden?

RQ4 Wie kann eine DSTL für MontiArc generativ entwickelt werden, welche handgeschriebenen Erweiterungen sind für deren Benutzbarkeit notwendig und wie kann die Verwendung der DSTL durch wiederverwendbare Modelltransformationen unterstützt werden?

RQ5 Wie lassen sich Transformationen und templatebasierte Generierung kombinieren und welche Erweiterungen der DSTLs sind hierzu notwendig?

Diese Teilforschungsfragen beschäftigen sich mit den verschiedenen Sichten der involvierten Personen auf die DSTL- und Transformationsentwicklung. Die zuvor beschriebene Dreiteilung der DSTL-Entwicklung, Transformationsentwicklung und Transformationsnutzung spiegelt sich in den Teilforschungsfragen wider und wird auch durch die Aufteilung der Kapitel verdeutlicht. Der Fokus dieser Arbeit liegt auf der Strukturmodellierung und damit der Transformation von Strukturmodellen wie Klassendiagrammen und Architekturdiagrammen. Dies zeigt sich insbesondere in den Teilforschungsfragen RQ3 und RQ4. In den einzelnen Kapiteln liegt der Fokus auf der Beantwortung der Teilforschungsfragen, um die übergeordnete Forschungsfrage beantworten zu können. Als nächstes wird ein Überblick über die Ergebnisse dieser Arbeit gegeben und abschließend wird der Aufbau dieser Arbeit erläutert.

1.2 Wichtigste Ergebnisse der Arbeit

MontiTrans umfasst sowohl einen Generator für DSTLs als auch eine Infrastruktur zur Generierung von Java-Implementierungen aus Transformationen, die mithilfe der generierten DSTLs spezifiziert wurden. Sowohl die Infrastruktur als auch der DSTL-Generator, der eine Weiterentwicklung des Generators aus [Wei12] ist, beantworten Teilfragestellungen der Forschungsfrage.

Ausgehend von der globalen Forschungsfrage sowie den Teilforschungsfragen wurden im Rahmen dieser Arbeit die nachfolgend zusammengefassten Ergebnisse erreicht. Die wichtigsten Ergebnisse dieser Arbeit sind:

- Eine Systematik zur Ableitung von domänenspezifischen Transformationssprachen aus beliebigen, durch Grammatiken definierten DSLs. DSTLs erlauben es Modelltransformationen in der konkreten Syntax der Modellierungssprache in Kombination mit Transformationsoperatoren auszudrücken. Dies ermöglicht Modellieren und Domänenexperten einen besseren Zugang zur Transformationsentwicklung.

- Die Umsetzung der Systematik zur Ableitung von DSTLs in Form eines DSTL-Generators in MontiTrans auf Basis der Ergebnisse aus [Wei12].

- Die Verbesserung der vorhandenen Operatoren zur Spezifikation negativer und listenwertiger Elemente innerhalb von Modelltransformationen.
- Die Erweiterung der DSTLs um Stringmanipulation, Zuweisungen an Schemavariablen, direkte Manipulation durch einen Anweisungsblock sowie Built-Ins zur Vereinfachung der Application Constraints, Zuweisungen und Anweisungsblöcke.
- Die Erweiterung der zur Generierung verwendeten Zwischenrepräsentation von Transformationsregeln und Anpassung des generierten Generators zur Abbildung von Transformationsregeln in DSTL-Notation in diese Repräsentation.
- Die mit MontiTrans entwickelten DSTLs CDTrans, MATrans und MACDTrans zur Transformation von Klassendiagrammen und MontiArc-Modellen. Hierbei erlauben CDTrans und MATrans die Transformation von Klassendiagrammen bzw. MontiArc-Modellen. MACDTrans erlaubt sowohl die Übersetzung zwischen Klassendiagrammen und MontiArc-Modellen als auch deren Koevolution.
- Sammlungen von wiederverwendbaren Modelltransformationen zum Refactoring von Klassendiagrammen sowie zur Normalisierung von MontiArc-Modellen. Außerdem umfasst die Sammlung Transformationen, die die Interoperabilität mit dem Robot Operation System (ROS) [QGC+09] verbessern.
- Die Performanceverbesserung des Pattern Matching Algorithmus durch automatische Modularisierung der Application Constraints und früherer Überprüfung dieser Constraints durch die Ersetzung der zentralisierten Überprüfung des Constraint durch eine dezentralisierte Überprüfung.
- Methodiken zum Einsatz von MontiTrans, DSTLs und Transformationen in der modellgetriebenen Softwareentwicklung.
- Eine Methodik zur handgeschriebenen Erweiterung der generierten DSTLs.

Die Ergebnisse dieser Arbeit unterstützen die agile, modellgetriebene Entwicklung sowohl im Bereich der DSTL-Entwicklung als auch der Entwicklung und Verwendung von Modelltransformationen. Nachdem in diesem Abschnitt die Ergebnisse dieser Arbeit zusammengefasst wurden, gibt der folgende Abschnitt einen Überblick über den Aufbau dieser Arbeit.

1.3 Aufbau der Arbeit

Eine Übersicht über die verschiedenen Ebenen der DSTL- und Transformationsentwicklung mit MontiTrans inklusive einer Zuordnung, in welchen Kapiteln welche Aspekte der DSTL- bzw. Transformationsentwicklung thematisiert werden, zeigt Abbildung 1.1. Auf der oberen Ebene befindet sich der Sprachentwickler, der der DSTL-Entwicklung zugeordnet werden kann. Die Ergebnisse der DSTL-Entwicklung, das heißt die DSTL und die zugehörige Infrastruktur, werden durch den Transformationsentwickler auf der

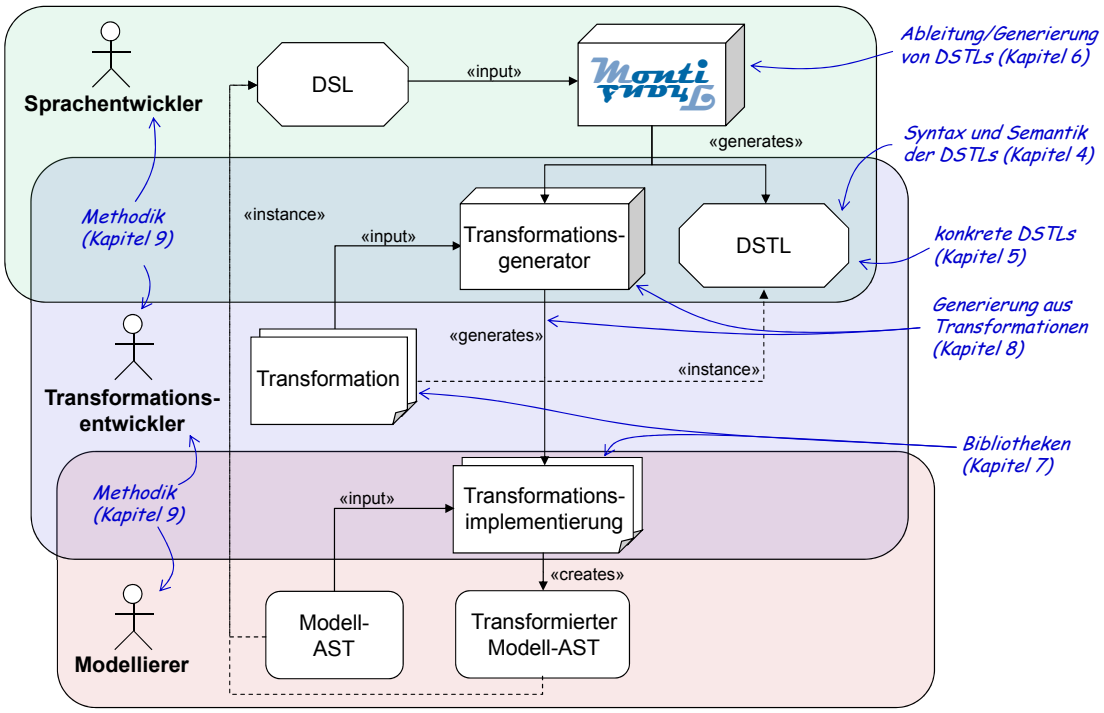


Abbildung 1.1: Entwicklung von Transformationssprachen und Transformationen.

darunterliegenden Ebene verwendet. Dieser nutzt die Ergebnisse zur Entwicklung von Transformationen, zu denen er die passenden Java-Implementierungen generieren lässt. Diese nutzt der Modellierer auf der darunterliegenden und zugleich untersten Ebene, um die modellierten Modelle transformieren zu können. Die Arbeit ist wie folgt strukturiert:

Kapitel 2 stellt die Grundlagen, die zum Verständnis dieser Arbeit notwendig sind, vor. Dazu werden die modellgetriebene Softwareentwicklung im Allgemeinen sowie im im Speziellen mit MontiCore erläutert. Des Weiteren werden hier die Begrifflichkeiten aus dem Bereich der Modelltransformation erklärt und die Verwendung im Rahmen dieser Arbeit festgelegt. Schließlich wird noch die Basis, auf der diese Arbeit aufsetzt, erläutert.

Kapitel 3 stellt die in die Transformationssprachen- und Transformationsentwicklung involvierten Rollen sowie drei Szenarien vor, anhand derer die Anforderungen sowohl an DSTLs als auch an MontiTrans im Speziellen vorgestellt werden. Darüber hinaus wird in diesem Kapitel die Analyse bestehender Generatoren im Hinblick auf Transformationssprachenfeatures vorgestellt und die aus den Szenarien und der Analyse ermittelten Anforderungen erläutert.

Kapitel 4 stellt Syntax und Semantik der Operatoren und Features der abgeleiteten bzw. mit MontiTrans generierten DSTLs vor. Hierbei werden sowohl die in dieser Arbeit neu entwickelten und die weiterentwickelten Operatoren und Features als auch die bereits in [Wei12] vorgestellten Features erläutert. Zusätzlich werden hier auch die im Rahmen dieser Arbeit entwickelten Kontextbedingungen für DSTLs beschrieben.

Kapitel 5 stellt die mithilfe von MontiTrans entwickelten DSTLs CDTrans, MATrans und MACDTrans vor. CDTrans ist hierbei speziell an die Modellierungssprachen CD4-Analysis und CD4Code, die über die gleiche Grammatik definiert sind, angepasst. CD4-Analysis erlaubt hierbei nur eine Teilmenge der Modellelemente, die in CD4Code erlaubt sind. MATrans ist an die ADL MontiArc angepasst. Außerdem wird hier auch die Kombination der DSTLs CDTrans und MATrans zur Entwicklung von MACDTrans beschrieben sowie erläutert wie analog weitere exogene DSTLs entwickelt werden können.

Kapitel 6 stellt die Ableitung von DSTLs aus Grammatiken von DSLs sowie die Umsetzung in Form eines DSTL-Generators vor. Der Generator wurde auf Basis des Generators aus [Wei12] entwickelt. Sowohl die abgeleiteten als auch die generierten DSTLs unterstützen die in Kapitel 4 vorgestellten Operatoren und Features.

Kapitel 7 stellt die wiederverwendbaren Transformationen zum Refactoring von Klassendiagramme und zur Normalisierung von MontiArc-Modellen sowie zur Unterstützung der Interoperabilität mit ROS vor. Diese wurden mit den in Kapitel 5 vorgestellten DSTLs CDTrans bzw. MATrans entwickelt. Es wird jeweils die Ausgangssituation und der Effekt erklärt. Zusätzlich wird die Realisierung beschrieben.

Kapitel 8 stellt die Generierung von Java-Implementierungen aus Transformationsregeln vor. Dazu wird sowohl die Abbildung in die Zwischenrepräsentation auf Objektdiagrammbasis als auch die resultierende Java-Implementierung und hiervon insbesondere der Pattern Matching Algorithmus erläutert.

Kapitel 9 stellt die im Rahmen dieser Arbeit entwickelte Methodik zu modellgetriebenen Entwicklung mit MontiTrans vor. Dabei wird zunächst die DSTL-Entwicklung mit MontiTrans erläutert. Anschließend wird auf die Transformationsregelentwicklung und abschließend auf die Kombination von Transformationsregeln zur Formung komplexer Transformationen und deren Verwendung eingegangen.

Kapitel 10 fasst diese Arbeit abschließend zusammen und gibt einen Überblick über mögliche Anknüpfungspunkte, die aufbauend auf den Ergebnissen dieser Arbeit bearbeitet werden können.

1.4 Verwandte Publikationen

Die Ergebnisse dieser Arbeit sind in mehreren Jahren Forschung entstanden. Aus diesem Grund sind einige Teile bereits vor Veröffentlichung dieser Arbeit veröffentlicht worden. Im Folgenden werden diese Veröffentlichungen kurz erläutert.

- *Ableitung von Sprachen:* Die Ableitung von Sprachen aus anderen Sprachen wurde in [HHK+13, HHK+15] für Deltasprachen und in [HRW15] für DSTLs vorgestellt. Hierbei wurden in [HRW15] Ableitungsregeln definiert, deren Anwendung eine DSTL ableiten. Die Ableitung wurde am Beispiel von CDTrans erklärt. In [HRRW17] wird die Ableitung von Sprachen als eine mögliche Methodik zur Sprachentwicklung erläutert.
- *Die Generierung von DSTLs:* Die Generierung von DSTLs wurde in [HHRW15] erklärt. Der Aufbau des DSTLs-Generators wurde in [HHRW15] und [HRW15] erläutert. Die Generierung wurde in [HRW15] am Beispiel Klassendiagrammen und in [HHRW15] am Beispiel von MontiArc erklärt.
- *Mit MontiTrans entwickelte DSTLs:* MontiTrans setzt die Systematik zur Ableitung von DSTLs als Generator um und wurde bereits zur Entwicklung von DSTLs genutzt. Die DSTL MATrans wurde in [HHRW15, AHRW17a, AHRW17b] und CDTrans in [HRW15, HRRW17] vorgestellt.
- *Wiederverwendbare Transformationen:* Eine Reihe von wiederverwendbaren Transformationen für MontiArc-Modelle, die mit MATrans entwickelt wurden, wurden in [AHRW17a, AHRW17b] vorgestellt. Diese Transformationen wurden für die Realisierung eines Generators für MontiArc-Modelle entwickelt und verwendet.
- *Erweiterung Mechanismen für generierten Code:* Für die Erweiterung von generiertem Code durch handgeschriebenen Code gibt es verschiedene Ansätze. Eine Übersicht der verschiedenen Ansätze wurde in [GHK+15a, GHK+15b] vorgestellt. Eine Variante des *Generation Gap Pattern* wurde im Rahmen dieser Arbeit zur Erweiterung der generierten DSTLs verwendet.

Kapitel 2

Grundlagen der modellgetriebenen Softwareentwicklung

Im vorangegangenen Kapitel wurden die Motivation sowie die Ziele und Ergebnisse dieser Arbeit beschrieben. Außerdem wurde der Aufbau der Arbeit vorgestellt. In diesem Kapitel werden die zum Verständnis der Arbeit notwendigen Grundlagen erläutert. Dies umfasst sowohl einen Überblick über und eine Einordnung in die modellbasierte Softwareentwicklung, eine Erläuterung der relevanten Eigenschaften des verwendeten Werkzeugs MontiCore als auch Erläuterungen und Definitionen verwendeter Begriffe wie beispielsweise Modell und Transformation. Diese Arbeit basiert auf Techniken der agilen Entwicklung [BBB+01] sowie der modellgetriebenen Entwicklung [SVC06] und der generativen Programmierung [CE00]. Eine Gemeinsamkeit dieser Techniken ist der Begriff des Modells. Eine einzige, gemeinsame Definition dieses Begriffs existiert hingegen nicht [Sei03, Küh06]. In dieser Arbeit wird der Begriff nach Stachowiaks Modelldefinition [Sta73] verwendet. Ein Modell ist demnach durch die folgenden drei Eigenschaften charakterisiert. Ein Modell

- basiert auf einem Original,
- ist eine Abstraktion dieses Originals und
- erfüllt einen Zweck im Bezug auf dieses Original.

In vielen Disziplinen wird ein Modell nach dem Original erstellt. Ein Beispiel hierfür ist die Chemie, in der Modelle zum Verständnis der Molekülstruktur erstellt werden. Im Gegensatz dazu dient ein Modell in der Softwareentwicklung als Grundlage zur Entwicklung des Originals, also der Software. Darüber hinaus haben Modelle weitere Einsatzzwecke. So dienen sie etwa der Durchführung von Analysen oder zur Kommunikation und können sogar als Grundlage zur Generierung der Software verwendet werden.

Die Einsatzzwecke unterscheiden sich je nach Entwicklungsparadigma. Der Literatur entsprechend lassen sich *modellbasierte* und *modellgetriebene* Ansätze unterscheiden [SVC06, BCW12, Sch12]. In der modellbasierten Entwicklung dienen Modelle neben Dokumentationszwecken auch zur Entwicklung der eigentlichen Software. Die modellgetriebene Entwicklung ist eine Teilmenge der modellbasierten, die formale Modelle verwendet, um diese automatisiert verarbeiten zu können. Hierzu zählt auch die generative Verwendung von Modellen. Den Erfolg von modellgetriebener Entwicklung untermauern verschiedene Studien [KR05, Sta06, WWM+07, HRW11, HWRK11, WHR14]. Diese Arbeit lässt sich der modellgetriebenen Entwicklung zuordnen.

Modellierungssprachen ermöglichen die Definition von formalen, zur automatisierten Verarbeitung geeigneten Modellen. Hierbei können Modellierungssprachen in *General Purpose Modeling Languages (GPMLs)* und *Domänenspezifische Modellierungssprachen (DSMLs)* unterteilt werden. Dabei dienen GPMLs der allgemeinen Modellierung und sind nicht auf eine Domäne spezialisiert. Dem gegenüber stehen DSMLs, die – wie DSLs [Fow10] – für eine bestimmte Domäne oder einen bestimmten Zweck entwickelt wurden und für diesen besonders geeignet sind. Ein klassisches Beispiel für eine GPML ist die UML [OMG15]. Da auch GPMLs wie beispielsweise die UML durch Bildung von Profilen für einen Bereich speziell angepasst werden können, ist der Übergang zwischen GPML und DSML oft fließend. Ein Beispiel hierfür ist die UML/P [Rum16, Sch12], die auf Konzepte reduziert ist, die zur Programmierung geeignet sind. In [Fow10] nennt Fowler als Beispiele für DSLs unter anderem GraphViz [EGK+02], die Hibernate Query Language (HQL) [ML05], die Extensible Application Markup Language (XAML) [Mac06] sowie die SQL [Mol06] und die Hypertext Markup Language (HTML) [SS11]. Im Folgenden werden daher Modellierungssprache, DSL und DSML synonym verwendet.

Die UML [OMG15] bietet Modellierungssprachen sowohl zur Struktur- als auch zur Verhaltensmodellierung. In dieser Arbeit liegt der Fokus auf der Strukturmodellierung. Von besonderer Relevanz für diese Arbeit sind Kassendiagramme und hierbei insbesondere die Varianten CD4Analysis und CD4Code [Rot17], die in Abschnitt 2.2 beschrieben werden. Im Allgemeinen sind Klassendiagramme zur implementierungsnahen Strukturmodellierung geeignet. Sie beschreiben die interne Struktur eines Systems. Zur Erläuterung von Abläufen werden im Rahmen dieser Arbeit Aktivitätsdiagramme (ADs) verwendet. Dies ist insbesondere in den Kapiteln 6, 8 und 9 der Fall. Aktivitätsdiagramme erlauben die Modellierung von Aktivitäten und Übergängen zwischen diesen Aktivitäten.

Als weitere Strukturmodellierungssprache wird in dieser Arbeit die Architekturbeschreibungssprache (ADL) MontiArc [HRR10, HRR12, Hab16] betrachtet. Hierbei handelt es sich um eine ADL zur Beschreibung von Component & Connector (C&C) Architekturen [MT00]. Die für diese Arbeit relevanten Konzepte und die Syntax von MontiArc wird in Abschnitt 2.3 beschrieben.

Bei der Definition und Implementierung von Modellierungssprachen gibt es zwei verschiedene Herangehensweisen [SLH+17]. Eine Modellierungssprache kann entweder durch ein Metamodell [Küh05] oder durch eine Grammatik [GKR+08] definiert werden. Die im Rahmen dieser Arbeit verwendeten sowie entwickelten Modellierungssprachen sind grammatikbasiert. Außerdem geht der im Rahmen dieser Arbeit vorgestellte Ansatz zur Ableitung von domänenspezifischen Transformationssprachen (DSTLs) von einer grammatikbasierten Sprachdefinition aus. Unabhängig von der Definition per Metamodell oder Grammatik werden in der Regel vier Modellebenen M0–M4 unterschieden [BCW12]. Die verschiedenen Ebenen sind in Abbildung 2.1 illustriert. Hierbei entspricht M0 der untersten Ebene, dem Original nach Stachoviak [Sta73]. Dieses Original wird durch das Modell auf der darüberliegenden Ebene M1 abstrahiert. In Abbildung 2.1 sind dies exemplarisch eine Klasse `Person` (Modell) und das Profil einer Person in einem sozialen Netzwerk (Original). Auf der nächsthöheren Ebene M2 befindet sich die Grammatik, die definiert wie Modelle der Sprache aussehen. Im Beispiel in Abbildung 2.1 definiert

die Grammatik wie eine Klasse modelliert wird. Schließlich findet sich auf der obersten Ebene die Grammatik, die beschreibt wie Grammatiken aufgebaut sind.

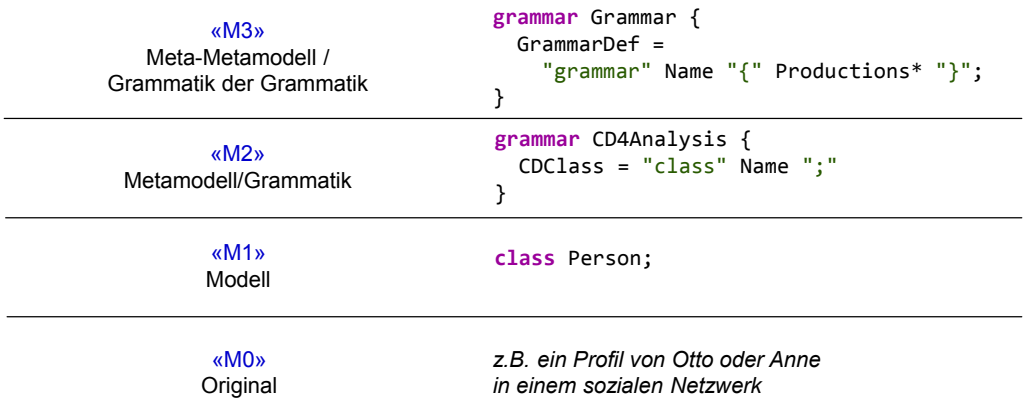


Abbildung 2.1: Darstellung der verschiedenen Modellebenen.

Übertragen auf diese Arbeit sind die Implementierungen von Transformationen das Original, die Transformationsbeschreibung das Modell und die Grammatik der DSTLs die Grammatik der M2-Ebene. Die Ableitung einer neuen DSTL für eine Modellierungssprache sowie die Generierung solcher DSTLs durch MontiTrans basiert auf der Grammatik einer Modellierungssprache. Aus diesem Grund wird ein grundlegendes Verständnis der Struktur einer Grammatik benötigt. Diese Arbeit basiert auf dem MontiCore-Grammatikformat, das in Abschnitt 2.1.1 erläutert wird.

Bei der Syntax einer Modellierungssprache wird außerdem zwischen der konkreten und der abstrakten Syntax unterschieden [HR04]. Hierbei stellt die konkrete Syntax die dem Modellierer bekannte – in MontiCore textuelle – Syntax dar, während die abstrakte Syntax die interne Repräsentation des Modells darstellt. Diese wird auch als abstrakter Syntaxbaum (Abstract Syntax Tree, kurz AST) bezeichnet. Mehr Details zur Definition der abstrakten und konkreten Syntax einer Modellierungssprache mittels MontiCore ist in Abschnitt 2.1 zu finden. Diese Unterscheidung ist für diese Arbeit von besonderer Bedeutung, da eines der Ziele dieser Arbeit ist Transformationen Modellierern ohne tiefgehende Kenntnis der abstrakten Syntax zu ermöglichen und hierbei auf der konkreten Syntax der Modellierungssprache aufzusetzen.

Ein zentraler Bestandteil der modellgetriebenen Entwicklung sind Modelltransformationen [SK03]. Im Bereich der Modelltransformation kann zwischen Model-to-Text- und Model-to-Model-Transformationen (kurz M2T- und M2M-Transformationen) unterschieden werden [CH06]. M2M-Transformationen verändern ein (oder mehrere) Eingabemodell(e) zu einem (oder mehreren) Ausgabemodell(en). Die im Rahmen dieser Arbeit vorgestellten sowie die von MontiTrans generierten Transformationssprachen erlauben die Definition von M2M-Transformationen. Auf das Thema M2M-Transformationen wird in Abschnitt 2.4 detaillierter eingegangen. Bei der generativen Programmierung werden Modelle mittels Codegeneratoren in ausführbare Implementierungen übersetzt [CE00]. Diese Übersetzung entspricht einer M2T-Transformation. Hierbei wird eine vollständi-

ge Generierung des Zielsystems angestrebt, was jedoch in vielen Fällen nicht möglich ist. In diesen Fällen kann das Generat um handgeschriebenen Code ergänzt werden [GHK+15a, GHK+15b]. M2T-Transformationen sind jedoch nicht auf die Generierung von Quellcode beschränkt, sondern können beliebige Arten von Textdateien erzeugen, beispielsweise auch Konfigurationsdateien. Innerhalb der M2T-Transformationen können visitor- von templatebasierten Ansätzen unterschieden werden, wobei diese nicht zwangsläufig disjunkt sind. MontiTrans nutzt M2T-Transformationen sowohl in templatebasierter Form zur Generierung des Quellcodes von Transformationen sowie der Infrastruktur zur Übersetzung von Transformationen in ihre ausführbaren Java-Implementierungen als auch die visitorbasierte Form für die Generierung der Grammatiken von DSTLs. Die entsprechenden Generatoren sind in den Kapiteln 6 und 8 beschrieben.

Nachdem in diesem Abschnitt die allgemeinen Grundlagen der modellgetriebenen Softwareentwicklung kurz erläutert wurden, werden in den folgenden Abschnitten einzelne Themen vertieft. In Abschnitt 2.1 wird zunächst die Entwicklung von Modellierungssprachen und Codegeneratoren mit der Language Workbench MontiCore erklärt. Im Anschluss werden die Modellierungssprachen CD4Analysis und CD4Code in Abschnitt 2.2 und 2.3 beschrieben, für die in dieser Arbeit Transformationssprachen entwickelt wurden. Danach werden die Grundlagen und Begrifflichkeiten aus dem Bereich der Modelltransformation in Abschnitt 2.4 erklärt. In Abschnitt 2.5 wird eine Einordnung der domänenspezifischen Transformationssprachen basierend auf MontiCore auf Basis der Begrifflichkeiten aus Abschnitt 2.4 gemacht und die Unterstützung solcher Transformationen durch MontiCore vor der Entwicklung von MontiTrans zusammengefasst.

2.1 Die Language Workbench MontiCore

MontiTrans wurde mit und für MontiCore in Version 4 entwickelt. MontiCore ist eine Language Workbench zur Entwicklung von Modellierungssprachen und Codegeneratoren. Mit MontiCore entwickelte Modellierungssprachen sind textuelle Sprachen, die durch eine Grammatik definiert werden. MontiCore bietet hierfür ein EBNF-ähnliches Grammatikformat zur Definition von Modellierungssprachen. Ausgehend von einer zu diesem Format konformen Grammatik generiert MontiCore Modellverarbeitungswerkzeuge wie Parser und Lexer, Infrastruktur zur Definition und Überprüfung von Kontextbedingungen, zur Definition von Symboltabellen und Visitoren. Im Folgenden werden in Abschnitt 2.1.1 zunächst die zum Verständnis dieser Arbeit notwendigen Grammatikkonzepte erläutert. Anschließend wird die generierte Modellinfrastruktur mit Fokus auf der Generierung der Klassen für die abstrakte Syntax in Abschnitt 2.1.2 vorgestellt. Danach wird in Abschnitt 2.1.3 auf die Codegenerierung mit MontiCore eingegangen, da die hier vorgestellten Generatoren mit MontiCore entwickelt wurden. Zusätzlich ist eine gute Integration von Transformationen, die mit MontiTrans entwickelt wurden, in die Codegenerierung mit MontiCore Ziel dieser Arbeit. Für eine umfassendere Erläuterung der Language Workbench MontiCore sei auf [GKR+08, KRV08, KRV10, Kra10] verwiesen. Abschließend werden in Abschnitt 2.1.4 die Möglichkeiten zur modularen Definition von Modellierungssprachen mit MontiCore erläutert.

2.1.1 MontiCore-Grammatiken

MontiCore bietet ein eigenes, an die erweiterte Backus-Naur Form (EBNF) angelehntes Grammatikformat. Auf Basis einer Grammatik generiert MontiCore die komplette Infrastruktur für die durch die Grammatik definierte Sprache. Die Grammatik ist daher essentiell zur Definition einer Modellierungssprache mit MontiCore. Eine Besonderheit des MontiCore-Grammatikformats ist, dass über dieses Artefakt sowohl die abstrakte als auch die konkrete Syntax der Modellierungssprache definiert werden. Die konkrete Syntax ergibt sich aus den Terminalen der Grammatik, wohingegen sich die abstrakte Syntax der Modellierungssprache aus der Struktur der Nichtterminale der Grammatik ergibt. Auf die abstrakte Syntax und die Generierung der entsprechenden Klassen zur internen Repräsentation der Modelle wird in Abschnitt 2.1.2 eingegangen. Eine MontiCore-Grammatik wird durch das Schlüsselwort `grammar` eingeleitet und definiert anschließend den Namen der Grammatik.

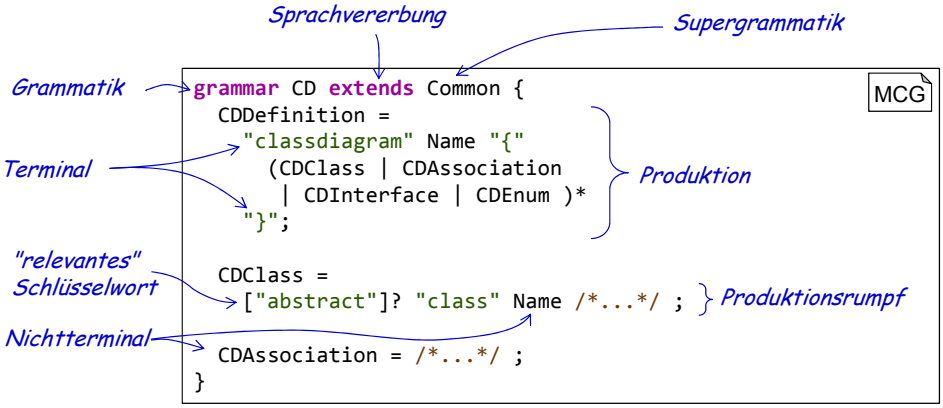
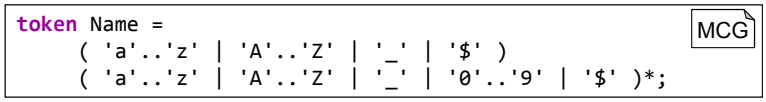


Abbildung 2.2: Auszug[1] aus der Grammatik für CD4Code.

Abbildung 2.2 zeigt einen vereinfachten Ausschnitt der CD4Code-Grammatik, wie sie im späteren Verlauf dieser Arbeit verwendet wird. Nach dem Namen der Grammatik kann das Schlüsselwort `extends` gefolgt von einer kommaseparierten Liste von Grammatiknamen verwendet werden. Dieses Schlüsselwort dient der Sprachvererbung und drückt aus, dass die Grammatik von den angegebenen Grammatiken erbt. Sprachvererbung wird in Abschnitt 2.1.4 detailliert. In den nachfolgenden, geschwungenen Klammern wird die Syntax der durch die Grammatik definierten Sprache spezifiziert. Hierfür werden Produktionsregeln innerhalb der Grammatik definiert. Eine Produktionsregel besteht aus einem Kopf und einem Rumpf getrennt durch ein Gleichheitszeichen. Auf der linken Seite steht der Kopf der Produktionsregel und auf der rechten Seite der Rumpf. Der Kopf besteht aus einem Nichtterminal, das durch den Rumpf definiert wird. Der Rumpf der Produktion besteht aus Terminalen und Nichtterminalen. Terminale sind atomare Elemente einer Grammatik und definieren die konkrete Syntax der Sprache. Sie werden in Anführungszeichen eingefasst und, falls sie für die interne Repräsentation

[1] Der Auszug wurde zu Demonstrationszwecken angepasst.

von Interesse sind, zusätzlich in eckige Klammern eingeschlossen. In Abbildung 2.2 ist dies für das Schlüsselwort `abstract` der Produktionsregeln für Klassen gemacht. Im Folgenden werden diese Terminale als relevante Schlüsselworter bezeichnet, da sie Teil der internen Repräsentation und somit per Transformation veränderbar sind. Nichtterminale sind Verweise auf Produktionsregeln. Der Rumpf einer Produktionsregel erlaubt es, Alternativen zu definieren durch `|` und Kardinalitäten festzulegen mittels `*` für beliebig viele Vorkommen, `+` für beliebig viele, aber mindestens ein Vorkommen und `?` für maximal ein Vorkommen. Als Strukturierungselement sind runde Klammern möglich.




```
token Name =
     ( 'a'..'z' | 'A'..'Z' | '_' | '$' )
     ( 'a'..'z' | 'A'..'Z' | '_' | '0'..'9' | '$' )*;
```

Abbildung 2.3: Lexikalische Produktion für das `Name`-Nichtterminal.

Eine Sonderform der Produktionsregeln sind lexikalische Produktionsregeln. Diese definieren Token für den Lexer anhand von regulären Ausdrücken. Lexikalische Produktionsregeln werden durch das Schlüsselwort `token` eingeleitet. Ein Beispiel für eine solche Produktionsregel ist die Produktionsregel des Nichtterminals `Name`. Diese ist in Abbildung 2.3 abgebildet. Die Produktionsregel erlaubt Namen, die den gleichen Regeln unterliegen wie Namen in Java.

Neben den einfachen Nichtterminalen gibt es außerdem Interface-Nichtterminale, abstrakte Nichtterminale und externe Nichtterminale. Interface- und abstrakte Nichtterminale stellen bezüglich der konkreten Syntax eine Erweiterung von Alternativen dar. Die Abbildung in der abstrakten Syntax wird in Abschnitt 2.1.2 beschrieben. Interface-Nichtterminale werden durch das Schlüsselwort `interface` eingeleitet. Angelehnt an Java können Interface-Nichtterminale andere Interface-Nichtterminale mithilfe des Schlüsselworts `extends` erweitern und von (abstrakten) Nichtterminalen mithilfe des Schlüsselworts `implements` implementiert werden. Nichtterminale können hierbei beliebig viele Interface-Nichtterminale implementieren. Alle Implementierungen eines Interface-Nichtterminals bilden zusammengenommen eine Alternative an den Stellen, an denen das Interface-Nichtterminal im Rumpf einer Produktionsregel verwendet wird. Abbildung 2.4 zeigt erneut die Produktionsregeln `CDDefinition`, `CDClass` und `CDAssociation` aus Abbildung 2.2, wobei diesmal zusätzlich ein Interface-Nichtterminal `CDElement` existiert. Dieses Interface-Nichtterminal wird mithilfe des `implements` Schlüsselworts von `CDClass` und `CDAssociation` implementiert. `CDElement` wird anstelle der zuvor verwendeten Alternative im Rumpf der Produktionsregel `CDDefinition` verwendet. Abstrakte Nichtterminale funktionieren analog zu Interface-Nichtterminalen, unterscheiden sich jedoch in der generierten Klassenstruktur für den AST. Außerdem können abstrakte Nichtterminale nur durch andere (abstrakte) Nichtterminale erweitert und nicht implementiert werden. Abstrakte Nichtterminale werden durch das Schlüsselwort `abstract` eingeleitet.

Nichtterminale können andere Nichtterminale erweitern. Auch dies stellt eine weitere Möglichkeit zur Definition von Alternativen dar. Wie bei Interface-Nichtterminalen und abstrakten Nichtterminalen wird hierzu das Schlüsselwort `extends` verwendet. Erwei-

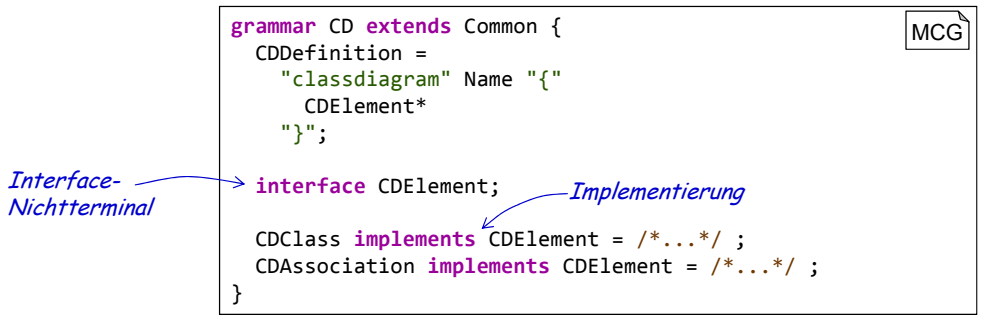


Abbildung 2.4: Beispielmodell in MontiCore-Grammar.

tert ein Nichtterminal ein anderes Nichtterminal, dann fügt das erweiternde Nichtterminal dem erweiterten Nichtterminal eine Alternative zu dessen Rumpf hinzu. Das heißt `A = "a";` und `B extends A = "b"` ist gleichbedeutend mit `A = "a" | "b";`.

Sowohl für die Ableitung neuer DSTLs als auch für die Suche von Mustern innerhalb eines Modells ist der Begriff *Modellelement* von Bedeutung. Im Folgenden wird ein *Modellelement* als ein durch ein Nichtterminal definierter Teil eines Modells betrachtet. Hierbei kann ein Modellelement in einer Kompositionsbeziehung zu anderen Modellelementen stehen, das heißt „angehängte“ weitere Modellelemente haben. Beispielweise wäre `CDDefinition` ein Modellelement. An dieses können `CDClass`-Modellelemente angehängt sein. Damit einspricht ein Modellelement einer Instanz einer AST-Klasse.

Externe Nichtterminale definieren Erweiterungspunkte innerhalb einer Grammatik. Sie werden durch das Schlüsselwort `external` eingeleitet und haben keinen Produktionsrumpf. Damit handelt es sich um Nichtterminale, die ähnlich wie *bottom nonterminals* [Läm01], nur verwendet werden, aber keine Syntax definieren. Ein Beispiel wäre `external CDElement;`. Stattdessen definieren sie ein „Loch“ innerhalb der Grammatik, welches per Spracheinbettung gefüllt werden kann. Spracheinbettung wird in Abschnitt 2.1.4 erklärt.

Neben normalen Grammatiken können mittels MontiCore auch Grammatikkomponenten – ähnlich zu den Grammatikfragmenten aus [Läm01, KLV05] – definiert werden. Eine Grammatikkomponente ist eine Grammatik, die nicht eigenständig verwendet werden soll und kann. Dazu wird vor das `grammar` Schlüsselwort zusätzlich das Schlüsselwort `component` geschrieben. Dies markiert die Grammatik als Komponente und damit als unvollständig und zur Erweiterung vorgesehen. In diesem Fall wird von MontiCore kein Parser generiert. Enthält eine Grammatik nicht implementierte Interface-Nichtterminale, nicht erweiterte abstrakte Nichtterminale oder externe Nichtterminal, so muss sie als Komponente markiert werden.

2.1.2 Generierung der abstrakten Syntax

Bisher wurde erklärt, wie die konkrete Syntax und somit der Parser durch eine MontiCore-Grammatik definiert wird. In diesem Abschnitt wird ergänzend erläutert, wie MontiCore anhand der Nichtterminalstruktur die Klassen der abstrakten Syntax generiert.

Zu einer gegebenen Grammatik generiert MontiCore neben dem Lexer und Parser, eine Infrastruktur zur Definition und Überprüfung von Kontextbedingungen und Entwicklung von Symboltabellen insbesondere die Klassen der abstrakten Syntax. Lexer und Parser werden hierbei mithilfe des ANTLR Parser Generators [Par14] generiert. Bei der generierten AST-Klassenstruktur orientiert sich MontiCore an der Struktur der Nichtterminale, die die Sprache definieren. Für jedes normale Nichtterminal wird eine Klasse generiert, wobei sich deren Name aus dem Namen des Nichtterminals plus das Präfix `AST` zusammensetzt. Für das Beispiel aus Abbildung 2.2 also `ASTCDDefinition`, `ASTCDClass` und `ASTCDAssociation`. Die Attribute der AST-Klassen ergeben sich aus dem Rumpf der Produktionsregeln. Nichtterminale im Produktionsregelrumpf werden zu Attributen des Typs der zugehörigen AST-Klasse des Nichtterminals. Für die Kardinalitäten `*` und `+` werden Listen der Java Collections dieser AST-Klassen verwendet, also beispielsweise `List<ASTCDClass>`. Für die `?` Kardinalität wird das Java `Optional` verwendet. Im AST gespeicherte Schlüsselworte werden als boolesche Werte gespeichert, die angeben, ob diese im Modell vorhanden waren. Das lexikalische Nichtterminal `Name` wird als `String` in die AST-Klassen abgebildet.

Interface-Nichtterminale bilden sich in der abstrakten Syntax als Java Interfaces ab. Abstrakte Klassen werden auf abstrakte Java-Klassen abgebildet. Die `implements` und `extends` Beziehungen zwischen Nichtterminalen werden entsprechend als `implements` und `extends` Beziehungen zwischen den AST-Klassen abgebildet. Für externe Nichtterminale werden Java Interfaces generiert, die im Zuge der Sprachvererbung von AST-Klassen der eingesetzten Nichtterminale implementiert werden.

2.1.3 Codegenerierung mit MontiCore

Neben der Definition neuer Modellierungssprachen unterstützt MontiCore auch die Generatorentwicklung. Ein Generator kann als eine Kombination von Transformationen und Templates inklusive Hilfsklassen zur Generierung von Code entwickelt werden. Die Unterstützung von Transformationen durch MontiCore wird in Abschnitt 2.5 detailliert, während in diesem Abschnitt auf die templatebasierte Generierung eingegangen wird. Hierbei wird lediglich ein Überblick über die templatebasierte Codegenerierung gegeben. Für eine umfassendere Erläuterung sei auf [Kra10, Sch12, Rot17] verwiesen.

Bei der templatebasierten Generierung sind Templates die zentralen Artefakte. Diese bestehen aus Code der Zielsprache sowie Anweisungen der Templatesprache wie Platzhaltern und Kontrollstrukturen. Diese Templates werden anschließend durch eine Templateengine verarbeitet und produzieren so generierten Code. Zusätzlich benötigten Templates eine Datenstruktur, auf der sie arbeiten. MontiCore setzt zur templatebasierten Generierung auf die Templateengine Freemarker [For13, Fre17]. Abbildung 2.5 zeigt einen Überblick der templatebasierten Generierung mittels MontiCore.

Ein Parser liest das Modell ein und überführt es so in seine interne Repräsentation. Das heißt, dass Instanzen der AST-Klassen erzeugt werden. Anschließend führt die Templateengine eine Menge von Templates auf diesem AST aus, wobei für jeden Knoten innerhalb des ASTs ein oder mehrere Templates ausgeführt werden können. Dies geschieht unter Zuhilfenahme von Hilfsklassen, die Berechnungen auf dem AST durch-

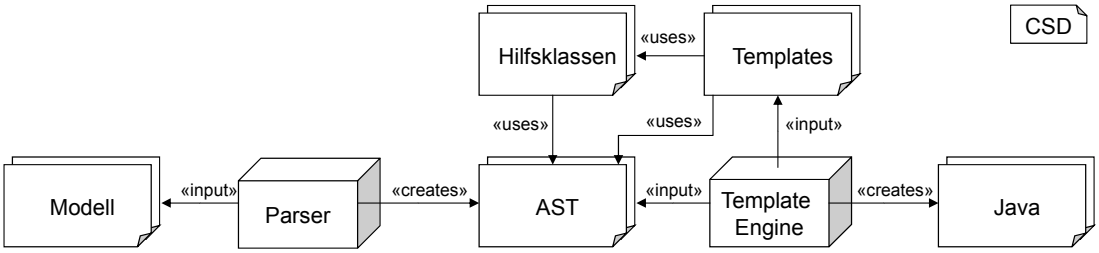


Abbildung 2.5: Überblick über die templatebasierte Generierung mit MontiCore angelehnt an [Kra10, Sch12].

führen. Hilfsklassen sind Instanzen von Javaklassen, die als Unterstützung der Templates dienen. Außerdem können Templates weitere Subtemplates aufrufen. Schließlich wird das Ergebnis der Templates in Dateien geschrieben, beispielsweise Java-Klassen.

Template Hookpoints und Templateerweiterung

In [Rot17] wurden außerdem Mechanismen zur Erweiterung bzw. Anpassung von templatebasierten Codegeneratoren vorgestellt. Hierbei wurde zwischen *Template Hookpoints* und *Templateerweiterung* unterschieden. Beide Mechanismen sind durch die Klassen `GobalExtensionManagment` und `TemplateController`, abgebildet in Abbildung 2.6, realisiert. Innerhalb der Templates ist durch MontiCore immer jeweils eine Instanz dieser Klassen verfügbar, wodurch diese Mechanismen genutzt werden können [Rot17].

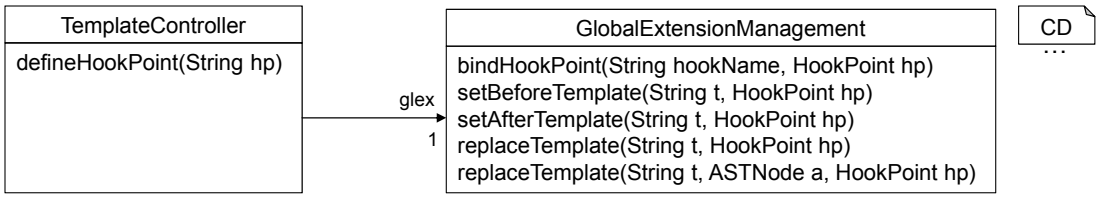


Abbildung 2.6: Auszug aus der Signatur der Klasse `GobalExtensionManagment`.

Template Hookpoints sind hierbei Stellen innerhalb von Templates, die zur Erweiterung vorgesehen sind. Template Hookpoints werden über die Methode `defineHookPoint(...)` der Klasse `TemplateController` gesetzt. Über die Methode `bindHookPoint(...)` der Klasse `GlobalExtensionManagement` können an diesen Stellen Strings oder Templateaufrufe integriert werden.

Templateerweiterung erlaubt die Ersetzung von Templateaufrufen und das Hinzufügen eigener Templates zu existierenden Templateaufrufen. Dazu stellt die Klasse `GlobalExtensionManagement` die Methoden `setBeforeTemplate(...)`, `setAfterTemplate(...)` und `replaceTemplate(...)` zur Verfügung. Hier wurde nur ein Überblick dieser Mechanismen gegeben. Eine detaillierte Erläuterung findet sich in [Rot17].

Für eine einfache Integration der transformations- und templatebasierten Codegenerierung ist daher eine Möglichkeit zur Verwendung der Klasse `GlobalExtensionMa-`

`nagement` notwendig. Im Rahmen dieser Arbeit wurde eine Integration dieser Klasse in die mit MontiTrans entwickelten Transformationen geschaffen (vgl. Kapitel 4 und 9).

2.1.4 Modulare Sprachdefinition und Sprachintegration mit MontiCore

Neben eigenständigen, monolithischen Sprachen erlaubt MontiCore auch eine modulare Sprachdefinition, bei der eine neue Sprache basierend auf existierenden Grammatiken definiert wird [Völ11, MSN17]. Zudem bietet MontiCore Mechanismen zur Sprachintegration [Völ11, MSN17]. Dies ermöglicht die Wiederverwendung von existierenden Sprachen sowie eine zielgerichtete Definition von einzelnen Sprachen für verschiedene Sachverhalte. Abbildung 2.7 zeigt schematisch die drei Möglichkeiten der Sprachintegration mit MontiCore. Im Folgenden werden die verschiedenen Mechanismen kurz erläutert. Eine umfassende Beschreibung dieser Mechanismen findet sich in [LNPR+13, HLMSN+15b, HLMSN+15a].

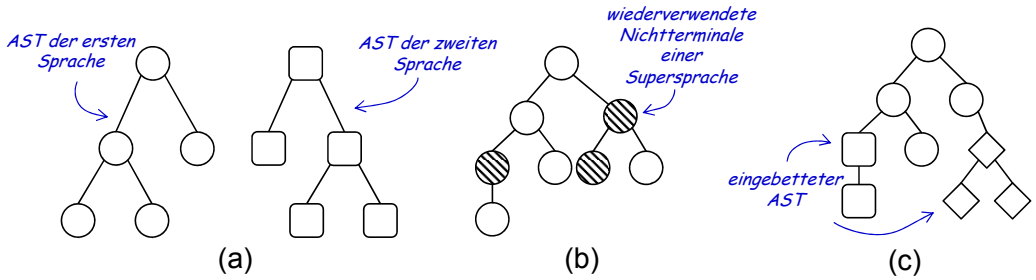


Abbildung 2.7: Möglichkeiten der modularen Sprachdefinition: (a) Sprachaggregation, (b) Sprachvererbung und (c) Spracheinbettung. Abbildung abgeändert aus [LNPR+13, HLMSN+15b, HLMSN+15a, Loo17].

Sprachaggregation

Die erste der drei Möglichkeiten ist die *Sprachaggregation*. Hierbei werden Modelle mehrerer Sprachen aggregiert, das heißt es werden mehrere Modelle „nebeneinander“ gelegt. Eine Verbindung wird über Namen geschaffen, indem innerhalb des einen Modells Namen auf Modellelemente eines anderen Modells verweisen. Die Grammatiken der beiden Sprachen bleiben hierbei unabhängig, was sich auch in deren ASTs widerspiegelt. Die AST-Klassen der beiden Sprachen sind unabhängig voneinander, wodurch auch die Instanzen der AST-Klassen als interne Repräsentation der Modelle voneinander unabhängig sind. Dies verdeutlicht Abbildung 2.7 (a). Ein Zusammenhang zwischen den Modellen ergibt sich lediglich aus der Definition und Referenz von Namen innerhalb der Modelle, welche über die Symboltabelle der Modelle aufgelöst werden können [Völ11, MSN17]. Im Rahmen dieser Arbeit wurde MontiTrans so erweitert, dass mit MontiTrans entwickelte DSTLs sowohl zur Transformation der einzelnen Modelle als auch zur gemeinsamen Koevolution der Modelle genutzt werden können.

Sprachvererbung

Die zweite der drei Möglichkeiten ist die *Sprachvererbung*. Bei der Sprachvererbung wird eine (Sub)Sprache basierend auf existierenden Grammatiken (Supersprachen) definiert (vgl. Abschnitt 2.1.1). Die Nichtterminale der erweiterten Sprache können genauso wie lokal definierte Nichtterminale im Rumpf von Produktionsregeln verwendet werden. Auch die Mechanismen zur Erweiterung von Nichtterminalen oder Implementierung von Interface-Nichtterminalen können über mehrere Grammatiken hinweg verwendet werden. Zusätzlich ist es möglich in der Subsprache Nichtterminale der Supersprache zu redefinieren, indem sie überschrieben werden. Dazu wird in der Subsprache das Nichtterminal erneut durch eine Produktionsregel definiert. Diese Definition ersetzt die Definition aus der Supersprache. In Abbildung 2.7 (b) ist Sprachvererbung durch die schraffierten Elemente veranschaulicht. Innerhalb von MontiTrans wird Sprachvererbung genutzt, um für die verschiedenen generierten DSTLs eine gemeinsame Basissprache bereitstellen zu können. Außerdem wurde MontiTrans dahingehend erweitert, dass auch für Sprachen, die Sprachvererbung nutzen, DSTLs generiert werden können.

Spracheinbettung

Die letzte der drei Möglichkeiten ist die *Spracheinbettung*. Bei der Spracheinbettung werden die Erweiterungspunkte einer Grammatikkomponente ausgefüllt und somit eine vollständige Sprache aus dieser gebildet. Hierzu wird eine Subsprache mittels Sprachvererbung gebildet und die externen Nichtterminale überschrieben. In Abbildung 2.7 (c) ist dies verdeutlicht. Hier sind Nichtterminale der Basissprache in rund dargestellt. Die eingebetteten Nichtterminale sind durch kasten- bzw. rautenförmige Elemente repräsentiert. MontiTrans wurde im Rahmen dieser Arbeit so erweitert, dass auch für Sprachen, die Spracheinbettung nutzen, DSTLs generiert werden können.

2.2 Die Modellierungssprachen CD4Analysis und CD4Code

CD4Analysis und CD4Code sind zwei auf MontiCore basierende Modellierungssprachen für Klassendiagramme, die aus den UML/P Klassendiagrammen [Sch12, Rum16] entstanden sind. In diesem Abschnitt werden die zum Verständnis dieser Arbeit erforderlichen Aspekte dieser Modellierungssprachen vorgestellt. Eine detaillierte Beschreibung findet in [Rot17] statt. Außerdem werden weitere Details in Abschnitt 5.1.1 erläutert. Die Besonderheit der beiden Modellierungssprachen ist, dass sie durch die gleiche MontiCore-Grammatik definiert werden, sich jedoch durch die geltenden Kontextbedingungen unterscheiden. Hierbei ist die Menge der erlaubten Modelle von CD4Analysis eine Teilmenge der in CD4Code erlaubten Modelle. CD4Analysis ist zur Erstellung von Analysemodellen geeignet. CD4Code ist hierbei implementierungsnäher als CD4Analysis. Der Unterschied der beiden Sprachen liegt darin, dass in CD4Code-Modellen im Vergleich zu CD4Analysis-Modellen Methoden erlaubt sind. Außerdem können in CD4Code-Modellen Sichtbarkeiten für Attribute, Klassen, Interfaces und Methoden spezifiziert werden. Ein

CD4Analysis-Modell besteht aus einer Klassendiagrammdefinition, die durch das Schlüsselwort `classdiagram` eingeleitet wird, den Namen des Klassendiagramms bestimmt und (abstrakte) Klassen, Enums, Interfaces und Assoziationen enthalten kann. Abbildung 2.8 zeigt ein Beispiel für ein CD4Analysis Klassendiagramm mit dem Namen `SocNet` und drei Klassen `Profile`, `Person` und `Group`, das einen Teil eines sozialen Netzwerks beschreibt.

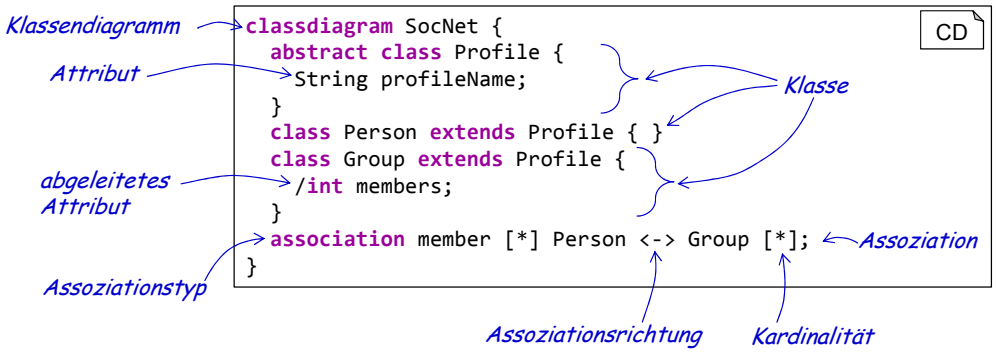


Abbildung 2.8: Beispielmodell in CD4Analysis.

Klassen haben einen Modifikator, der innerhalb von CD4Analysis erlaubt, dass Klassen abstrakt sein können und in CD4Code zusätzlich die Sichtbarkeit einer Klasse spezifizierbar macht. Klassen können andere Klassen erweitern (Schlüsselwort `extends`). Im Beispiel ist die Klasse `Profile` abstrakt. Klassen und Interfaces können außerdem Attribute enthalten. Im Beispiel enthält die Klasse `Profile` ein Attribut `profileName` und die Klasse `Group` ein Attribut `members`. Attribute haben einen Typ und einen Namen. Außerdem haben Attribute einen Modifier. In CD4Analysis können Attribute hierdurch als abgeleitet (vgl. Attribut `members`) markiert werden. Dies bedeutet, dass sich ihr Wert aus anderen Werten berechnet. In CD4Code kann der Modifier zusätzlich die Sichtbarkeit (`public`, `private`, `protected`) der Attribute angeben. Assoziationen können entweder „einfache“ Assoziationen (Schlüsselwort `association`) oder Kompositionen (Schlüsselwort `composition`) sein. Beide Typen von Assoziationen haben eine Assoziationsrichtung (`->`, `<-`, `--`, `<->`) und verbinden zwei Klassen, Interfaces oder Enums miteinander. Außerdem können Assoziationen an beiden Seiten optional einen Rollennamen und eine Kardinalität haben.

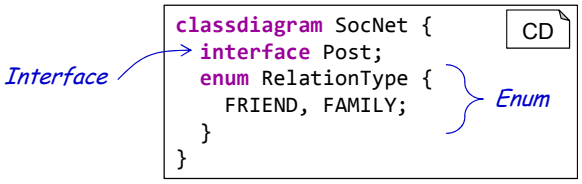


Abbildung 2.9: Beispielmodell in CD4Analysis.

Abbildung 2.9 zeigt einen weiteren Ausschnitt des `SocNet` Klassendiagramms. Das Modell zeigt ein Enum inklusive zwei Enumkonstanten `FRIEND` und `FAMILY` und ein Interface. Interfaces können genau wie Klassen Attribute und Methoden enthalten. Klassen können Interfaces über das Schlüsselwort `implements` implementieren. Ein Beispiel wäre `class Message implements Post;`. Interfaces können andere Interfaces über das Schlüsselwort `extends` erweitern (z.B. `interface Message extends Post;`).

2.3 Die Modellierungssprache MontiArc

Neben CD4Code und CD4Analysis wird als weitere Strukturmodellierungssprache in dieser Arbeit MontiArc verwendet [HRR10, HRR12, Hab16]. MontiArc ist eine auf MontiCore basierende Architekturbeschreibungssprache (Architecture Desciption Language, kurz ADL) zur Beschreibung von Component und Connector Architekturen (C&C Architekturen) [MT00] und insbesondere verteilter interaktiver Systeme [HRR12]. In diesem Abschnitt werden die zum Verständnis erforderlichen Aspekte von MontiArc erläutert, eine umfassende Erläuterung zu MontiArc findet sich in [HRR12, Hab16]. Außerdem werden weitere Details in Abschnitt 5.2.1 erläutert. MontiArc unterstützt die Kernelemente einer C&C Architektur [MT00]. Zu diesen Kernelementen gehören Komponenten, Ports und Konnektoren. Komponenten haben eine Schnittstelle bestehend aus getypten, gerichteten Ports. Ports haben einen Typ, einen Namen und eine Richtung (eingehend oder ausgehend). Konnektoren sind gerichtet und verbinden Ports miteinander, wobei sie einen Quellport und mindestens einen Zielport haben. MontiArc erlaubt hierarchische Dekomposition von Komponenten. Das bedeutet, dass Komponenten atomar oder dekomponiert sein können, also aus Subkomponenten bestehen.

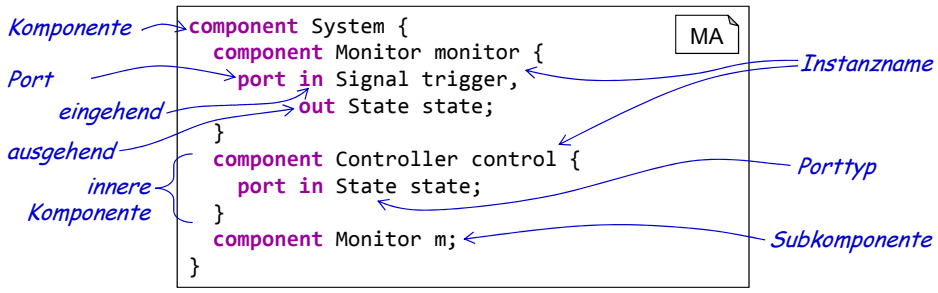


Abbildung 2.10: Beispielmodell in MontiArc.

Abbildung 2.10 zeigt ein Beispiel für ein MontiArc-Modell. Dieses besteht aus einer Komponente `System`. Bei dieser Komponente handelt es sich um eine dekomponierte Komponente, die zwei innere Komponenten `Monitor` und `Controller` hat. Innere Komponenten sind ähnlich wie innere Klassen in Java lokal definierte Komponenten. Die Systemkomponente hat außerdem eine Subkomponente vom Typ `Monitor` mit dem Instanznamen `m`. Innere Komponenten können mit Instanznamen versehen werden. Dies entspricht einer impliziten Subkomponentendefinition. Die Systemkomponente in

Abbildung 2.10 umfasst daher zwei implizit definierte Subkomponenten `monitor` und `control` als Instanzen der beiden inneren Komponenten `Monitor` und `Controller`. Weitere Features und Eigenschaften von MontiArc-Modellen werden nach Bedarf an den Stellen, an denen sie benötigt werden, erläutert.

MontiArc verfügt über eine Reihe von Kontextbedingungen, die die Wohlgeformtheit der Modelle überprüfen wie beispielsweise Eindeutigkeitsüberprüfungen von Namen. Diese können in [Hab16] nachgelesen werden. Die Kommunikation der Komponenten basiert auf Focus [BS01, BR07, RR11]. Der Nachrichtenaustausch der Komponenten erfolgt über unidirektionale Kanäle, die Konnektoren, welche erhaltene Nachrichten in der Reihenfolge des Erhalts transportieren. Die Semantik über die Zeit ist hierbei durch geordnete Ströme von Nachrichten formalisiert. Hierbei wird jedoch keine Aussage über die zeitlichen Abstände zweier Nachtrichten gemacht. Ein Voranschreiten der Zeit kann über spezielle Tick-Nachrichten (√) simuliert werden. Eine detaillierte Erläuterung findet sich in [Hab16]. Außerdem wird in [Hab16] auch ein Simulationsframework für MontiArc vorgestellt, das eine Simulation der MontiArc-Modelle ermöglicht.

2.4 Grundlagen der Modelltransformation

Bislang wurden die Grundlagen der modellgetriebenen Entwicklung im Allgemeinen sowie im Speziellen mit der Language Workbench MontiCore erläutert. Hierbei wurde insbesondere die Sprachentwicklung sowie M2T-Transformationen in Form von templatebasierter Codegenerierung beschrieben. Darauf aufbauend wird in diesem Abschnitt der zweite Bereich der Modelltransformation – M2M-Transformation – detailliert. Hierbei werden insbesondere die Begrifflichkeiten aus diesem Bereich, wie sie in dieser Arbeit verwendet werden, erläutert. Anschließend werden hierauf aufbauend die für diese Arbeit grundlegenden Ergebnisse aus [Wei12] zusammengefasst.

Wie zuvor erwähnt, sind Modelltransformationen in der modellgetriebenen Softwareentwicklung ein zentraler Bestandteil. In den vorangegangenen Abschnitten wurde zwischen M2T- und M2M-Transformationen unterschieden. Der Fokus dieser Arbeit liegt auf M2M-Transformationen. Aus diesem Grund bezieht sich im Folgenden der Begriff Transformation und auch Modelltransformation – sofern nicht explizit anders angegeben – grundsätzlich auf M2M-Transformationen.

Angelehnt an [KWB03] und [MVG06] wird eine *(komplexe) Transformation* als die automatische Generierung eines oder mehrerer Zielmodelle aus einem oder mehreren Quellmodellen mithilfe einer Menge von Transformationsregeln und einer Anwendungsstrategie der Regeln definiert. Diese Idee ist ähnlich zur Idee der Komposition von komplexen Refactorings aus simplen Basisoperationen in [KK04]. Eine *Transformationsregel* beschreibt wie ein Eingabemodell in ein Zielmodell überführt wird. Eine komplexe Transformation ist beispielsweise die einmalige Anwendung einer Transformationsregel auf ein Eingabemodell. Die Begriffe Transformation und Modelltransformation werden nachfolgend synonym verwendet.

Die Verwendbarkeit einer Transformationssprache hängt nicht allein von den Möglichkeiten und Features der Transformationssprache ab [SK03, Wei12]. Stattdessen haben

auch die Fähigkeiten der Nutzer sowie deren Präferenzen Einfluss auf die Eignung der Transformationssprache für den jeweiligen Nutzer. Je besser die Transformationssprache an die Nutzer angepasst ist, desto besser empfinden diese deren Nutzbarkeit.

Für Transformationssprachen lassen sich ähnlich wie für Modellierungssprachen zwei Arten unterscheiden: *General Purpose Transformationssprachen* (GPTLs) und *domänenspezifische Transformationssprachen* (DSTLs). GPTLs sind hierbei für eine Vielzahl von Modellierungssprachen verwendbar. Sie basieren auf einer modellunabhängigen, generischen Notation, die in der Regel der internen Repräsentation der Modelle dient. GPTLs beschreiben Modelltransformationen dadurch basierend auf der abstrakten Syntax. Dazu basieren GPTLs zum Beispiel auf dem Ecore-Metamodell innerhalb der Eclipse Modeling Frameworks (EMF) [SBPM09]. Beispiele für GPTLs sind ATL [JK06, JABK08], VIATRA [VB07, VIA17], PROGRES [Sch91], eMoflon [LAS14] und Henshin [ABJ+10, SBG+17]. GPTLs haben den Nachteil, dass die interne Repräsentation Domänenexperten und Modellierern in der Regel unbekannt ist und damit eine Einbindung von Domänenexperten und Modellierern in den Transformationsentwicklungsprozess erschwert wird. Demgegenüber steht der Vorteil, dass diese Sprachen Modellierungssprachen übergreifend einsetzbar und somit wiederverwendbar sind. DSTLs hingegen sind spezifisch für eine Modellierungssprache [RW11, Wei12]. Im Gegensatz zu GPTLs verwenden sie eine Notation, die an die Modellsyntax, die dem Modellierer vertraut ist, angelehnt ist. Das bedeutet, sie verwenden die konkrete Syntax [BW07, RW11]. Aufgrund der vertrauten Syntax lassen sich somit Modellierer und im Fall von DSLs als Modellierungssprachen auch Domänenexperten besser in die Transformationsentwicklung einbinden. Diesem Vorteil der DSTLs steht der Nachteil gegenüber, dass aufgrund ihrer Spezialisierung auf eine Modellierungssprache zu jeder DSL eine eigene DSTL entwickelt werden muss. Genau wie Modellierungssprachen gibt es GPTLs und DSTLs mit textueller oder graphischer Notation.

Transformationsansätze lassen sich außerdem an der Art der internen Repräsentation sowie der Art der Definition von Pattern (Deutsch: Muster) unterscheiden [CH06]. Hierbei lassen sich drei Ansätze unterscheiden: graphbasierte, termbasierte und stringbasierte Ansätze. Für stringbasierte Ansätze dienen Zeichenketten als Eingabe. Diese Zeichenketten werden modifiziert, ohne dass hierfür eine Datenstruktur im Speicher erstellt wird. Templatebasierte Transformationen sind ein Beispiel für einen stringbasierten Ansatz und FreeMarker [For13, Fre17] ein entsprechendes Tool. Für M2M-Transformationen werden in der Regel term- oder graphbasierte Ansätze verwendet [CH06], welche die interne Struktur der Modelle im Speicher aufbauen und transformieren. Termbasierte Ansätze arbeiten auf Bäumen der abstrakten Syntax und für graphbasierte Ansätze werden getypte, attributierte Graphen verwendet [Wei12]. Als Beispiele für Tools, die auf Termersetzung setzen, sind Rascal [KvdSV09, HKV12] und Stratego/XT [Vis04, BKVV08], das mittlerweile Teil des Spoofax Language Workbenchs [KV10] ist, zu nennen. Beispiele für Graphtransformationstools sind Henshin [ABJ+10, SBG+17], GReAT [BNvBK06], eMoflon [LAS14] und PROGRES [SWZ95, SWZ99]. Da für MontiCore auf einen Graphtransformationansatz gesetzt wird [Wei12], werden string- und termbasierte Ansätze im Folgenden nicht weiter betrachtet.

Graphtransformationsansätze verwenden zur Transformation von Graphen Graphersetzungsregeln [ERK99]. Eine Graphersetzungregel lässt sich in einen Graphen für die linke Regelseite (left-hand side, kurz LHS) und einen Graphen für die rechte Regelseite (right-hand side, kurz RHS) aufteilen. Hierbei beschreibt die LHS das Pattern der Transformationsregel. Der durch das Pattern beschriebene Modellteil wird durch die Transformationsregel durch die rechte Regelseite ersetzt. Das heißt, die RHS beschreibt den gleichen Ausschnitt nach Anwendung der Transformationsregel. Zusätzlich können Graphersetzungregeln Anwendungsbedingungen (Englisch: Application Conditions) haben [EH86, HHT96]. Diese schränken die Anwendbarkeit der Graphersetzungsregel weiter ein. Man kann hierbei positive und negative Anwendungsbedingungen unterscheiden. Positive fordern bestimmte Eigenschaften des Graphen, während negative die Abwesenheit bestimmter Graphteile fordern [EH86, HHT96]. Zur Darstellung einer Graphersetzungsregel existieren sowohl Notationen, die linke Regelseite (LHS) und rechte Regelseite (RHS) getrennt angeben, als auch solche, die beides integriert angeben [RW11]. In diesem Fall werden hinzugefügte und gelöschte Elemente entsprechend markiert. Eine separate Notation hat den Nachteil, dass es für den Entwickler schwierig ist den Unterschied zwischen den beiden Graphen schnell zu erkennen und dass alle Teile des Graphen, die erhalten bleiben sollen, auf der rechten Regelseite wiederholt werden müssen. Eine integrierte Notation verwenden beispielsweise Henshin [ABJ$^+$10, SBG$^+$17] und eMoflon [LAS14]. PROGRES [SWZ95, SWZ99] hingegen verwendet eine separierte Notation. Die Anwendbarkeit einer Graphersetzungsregel kann durch eine Bedingung, im Folgenden als Application Constraint bezeichnet, eingeschränkt werden [Wei12, HRW15].

Ein *Pattern* ist der Teil einer Transformation, der den zu transformierenden Teil eines Modells beschreibt. Ein Pattern beschreibt Modellelemente hierbei in der Regel nicht vollständig, sondern beschränkt sich auf die Eigenschaften der Modellelemente, die für die Transformation von Bedeutung sind. Über nicht beschriebene Eigenschaften von Modellelementen ist keine Aussage getroffen. Ein Pattern besteht aus Patternelementen.

Ein *Patternelement* ist das Pendant zu einem Modellelement innerhalb eines Pattern. Ein Patternelement stellt die Beschreibung eines Modellelements dar. Für jedes Patternelement wird während des Pattern Matching ein entsprechendes Modellelement gesucht.

Pattern Matching ist der Prozess, bei dem für ein gegebenes Pattern ein (gültiger) Match innerhalb eines Modells (oder innerhalb mehrerer Modelle) gesucht wird. Hierzu gibt es unterschiedliche Ansätze wie die Berechnung eines Suchplans oder eine Abbildung auf das Constraint Satisfaction Problem [GBA14]. Kapitel 8 detailliert diese Ansätze.

Ein *Match* eines Patternelements ist ein Modellelement, das alle Eigenschaften des Patternelements erfüllt. Ein *gültiger Match* eines Patternelements erfüllt zusätzlich den Application Constraint. Entsprechend ist ein Match eines Pattern eine Menge von Modellelementen, die zu jeweils einem Patternelement ein Match sind. Ein *gültiger Match* eines Pattern erfüllt zusätzlich den Application Constraint. Diese Unterscheidung ist für die Erklärung des Pattern Matching Algorithmus in Kapitel 8 relevant.

Ein weiteres Unterscheidungsmerkmal ist die Richtung in der Transformationsregeln ausgeführt werden können [CH06, MVG06, CFH$^+$09]. *Unidirektionale* Transformationen können ausschließlich das Quellmodell in das Zielmodell überführen. *Bidirektionale*

Transformationen unterstützen zusätzlich die Rückrichtung. Unidirektionale Transformationen werden zum Beispiel von PROGRES [SWZ95, SWZ99] und Fujaba [NNZ00, GZ06, SZG06] unterstützt, bidirektionale von TGG [Sch95] und eMoflon [LAS14].

Es kann außerdem zwischen *endogenen* und *exogenen* Transformationen unterschieden werden [MVG06, CH06, KC15]. *Endogene* Transformationen transformieren Modelle innerhalb der gleichen Modellierungssprache. Das heißt, die Ausgabemodelle sind konform zu derselben Modellierungssprache wie die Eingabemodelle. Bei *exogenen* Modelltransformationen hingegen unterscheiden sich die Modellierungssprachen, zu denen die Ein- und Ausgabemodelle konform sind. Typische Beispiele für endogene Transformationen sind Refactorings und Normalisierungen [MVG06]. Beispiele für exogene Transformationen sind Übersetzungen zwischen zwei Sprachen wie zwischen Klassendiagrammen und Entity-Relationship-Diagrammen, die Migration zwischen zwei Sprachversionen oder auch die Generierung von Code. Beispiele für Tools, die endogene Transformationen unterstützen, sind PROGRES [SWZ95, SWZ99] und Fujaba [NNZ00, GZ06, SZG06]. Ein Beispiel für exogene Transformationen ist TGG [Sch95].

Darüber hinaus können Modelltransformationen *in-place* oder *out-place* stattfinden [CH06, MVG06, KC15]. *In-place* Transformationen arbeiten direkt auf dem Eingabemodell. Das heißt, das Eingabemodell wird während der Transformation verändert. Im Gegensatz dazu produzieren *out-place* Transformationen ein neues Modell basierend auf dem Eingabemodell. Endogene Transformationen können in-place oder out-place realisiert werden. Exogene Transformationen finden normalerweise out-place statt [MVG06].

Des Weiteren lässt sich unterscheiden wie viele Ein- und Ausgabemodelle eine Transformation haben kann [CH06]. Mögliche Kombinationen sind hierbei 1–zu–1, 1–zu–n, n–zu–1 und n–zu–m. Hierbei stellt 1–zu–1 die restriktivste Variante und n–zu–m die flexibelste dar. Viele Tools unterstützen alle Varianten [KC15], beispielsweise UML-RSDS [LKR10] oder ATL [JABK08]. Es gibt jedoch auch eingeschränktere Tools wie PROGRES [SWZ95, SWZ99] und AToMPM [EHR+13, CSE16], die lediglich 1–zu–1 Transformationen unterstützen.

Zudem kann die Art der Transformationsspezifikation unterschieden werden [KC15]. Hierbei werden deklarative, imperative und hybride Ansätze unterschieden. In imperativen Ansätzen wird mittels Kontrollstrukturen der Ablauf der Transformation beschrieben. Deklarative Ansätze beschreiben stattdessen den Effekt der Transformation, jedoch nicht den notwendigen Ablauf, um diesen zu erreichen. Hybride Ansätze verwenden eine Mischung aus beiden Strategien, also beispielsweise eine deklarative Angabe des Pattern und eine imperative Beschreibung der Anwendungsstrategie der Transformationsregeln. Umple [FLB09] nutzt zum Beispiel einen imperativen Ansatz. UML-RSDS [LKR10] verwendet einen deklarativen Ansatz. Einen hybriden Ansatz unterstützt ATL [JABK08].

2.5 Domänenspezifische Transformationssprachen in MontiCore

In diesem Abschnitt werden die Vorarbeiten im Bereich der Modelltransformation innerhalb von MontiCore erläutert. Dazu werden die Ergebnisse aus [Wei12] zusammen-

gefasst, entsprechend der im vorherigen Abschnitt vorgestellten Merkmale eingeordnet und die im Rahmen dieser Arbeit geschaffenen Erweiterungen genannt. In [Wei12] wurde eine Transformationsengine entwickelt, die die Generierung von DSTLs aus monolithisch definierten Modellierungssprachen erlaubt. Die Generierung kompensiert hierbei den Nachteil, dass DSTLs für jede DSL individuell entwickelt werden müssen. MontiCore 4 unterstützt jedoch auch modulare Sprachdefinitionen. Im Rahmen dieser Arbeit wurde die Generierung dahingehend erweitert, dass auch modulare Definitionen von Modellierungssprachen unterstützt werden. Dies wird in Kapitel 6 beschrieben.

Die nach [Wei12] generierten DSTLs erlauben es unidirektionale, in-place Transformationen zu beschreiben. Als Transformationsansatz wurde ein graphbasierter Ansatz gewählt, da hierdurch neben Baumstrukturen zusätzlich auch Graphstrukturen durch die Transformationsengine unterstützt werden. Es wurde außerdem eine integrierte Notation von Transformationsregeln entwickelt. Dies wurde im Rahmen dieser Arbeit beibehalten. Der hybride Ansatz aus [Wei12], bei dem die Definition von Transformationsregeln deklarativ und die Ablaufsteuerung imperativ beschrieben wird, wurde ebenfalls beibehalten und ausgebaut. Neben der Möglichkeit, die in [Wei12] beschriebene Kontrollflusssprache zur Spezifikation der Anwendungsstrategie der Transformationsregeln zu nutzen, wurde die Möglichkeit geschaffen die Anwendungsstrategie in Java zu spezifizieren. Dies wird in Kapitel 9 beschrieben. Zudem wurden die Operatoren der DSTLs erweitert sowie zusätzliche Operatoren für die DSTLs entwickelt. Schließlich wurde auch die Performance des Pattern Matching Algorithmus verbessert. Dies ist in Kapitel 4 und 8 beschrieben.

Der Fokus für die Transformationsengine aus [Wei12] lag auf Refactorings und semantikerhaltenden Transformationen, weshalb endogene Transformationen ausreichend waren. Im Rahmen dieser Arbeit wurde der Fokus erweitert und sowohl Übersetzung [Vis01a, SAL+03, MVG06] als auch Koevolution [CH05, CV07, SKSL11, Läm16] anvisiert. MontiTrans unterstützt weiterhin endogene Transformationen und wurde erweitert, sodass zusätzlich exogene Transformationen möglich sind. Dies wird insbesondere in Kapitel 5 und 6 beschrieben.

Mit den Transformationen, die mit der Transformationsengine aus [Wei12] entwickelt wurden, kann ein Eingabemodell zu einem Ausgabemodell transformiert werden. Mit MontiTrans ist es nun möglich, beliebig viele Eingabemodelle in beliebig viele Ausgabemodelle zu transformieren. Erläuterungen hierzu finden sich in Kapitel 5, 8 und 9.

Als Ergänzung der Ergebnisse aus [Wei12] sowie aufbauend auf diesen und den in dieser Arbeit entwickelten Erweiterungen wurden zudem mithilfe von MontiTrans spezielle DSTLs für die Modellierungssprachen CD4Code, CD4Analysis und MontiArc entwickelt. Zusätzlich wurden mithilfe dieser DSTLs wiederverwendbare Transformationen zum Refactoring und zur Normalisierung von Klassendiagrammen und MontiArc-Modellen entwickelt. Dies ist in Kapitel 5 bzw. 7 beschrieben.

Kapitel 3

Rahmen und Anforderungen

Im vorangegangenen Kapitel wurden die Grundlagen und damit die wichtigsten Begrifflichkeiten dieser Arbeit erläutert. In diesem Kapitel werden darauf aufbauend Szenarien und Anforderungen vorgestellt. Zunächst werden die Szenarien zur Transformationssprachenentwicklung, Transformationsentwicklung sowie -nutzung und die damit verbundenen Rollen in Abschnitt 3.1 erklärt. Basierend auf den Szenarien wurden Highlevel-Anforderungen abgeleitet. Diese wurden durch eine Analyse existierender Modellierungssprachen, die mögliche Ausgangssprachen für die Entwicklung neuer DSTLs sein können, und in Java programmierter Transformationen, die durch modellierte Transformationen ersetzt werden können, im Kontext von MontiCore verfeinert. Die Ergebnisse der Analyse werden in Abschnitt 3.2 vorgestellt, während die resultierenden Anforderungen in Abschnitt 3.3 beschrieben werden.

3.1 Szenarien und Rollen

In diesem Abschnitt werden Szenarien vorgestellt, auf Basis derer die in Abschnitt 3.3 vorgestellten Anforderungen abgeleitet werden. Dazu werden zunächst verschiedene Rollen im Kontext der Entwicklung von Transformationen und Transformationssprachen erläutert und anschließend zu diesen Rollen drei grundlegende Szenarien beschrieben. Die Szenarien ergeben sich hierbei aus den Forschungsfragen.

3.1.1 Rollen

Die Ergebnisse dieser Arbeit sind auf verschiedenen Ebenen der Sprach- und Generatorentwicklung zu nutzen (vgl. Metalevel in Abbildung 2.1). Abbildung 3.1 illustriert diese verschiedenen Ebenen. Die oberste Ebene stellt hierbei die Ebene der Entwicklung der MontiTrans Transformationsengine dar, welche auf dem MontiCore Grammatikformat basiert. Auf der nächsten Ebene befindet sich die Entwicklung von Modellierungssprachen mit zugehörigen Transformationssprachen. Eine Ebene darunter findet die Entwicklung von Transformationen und wiederum darunter die reine Nutzung von Transformationen statt. Zu diesen Ebenen lassen sich verschiedene Rollen identifizieren, die mit den Ergebnissen dieser Arbeit interagieren und für die die Methodik in Kapitel 9 entwickelt wurde. Diese Rollen sind ebenfalls in Abbildung 3.1 dargestellt und werden im Folgenden beschrieben.

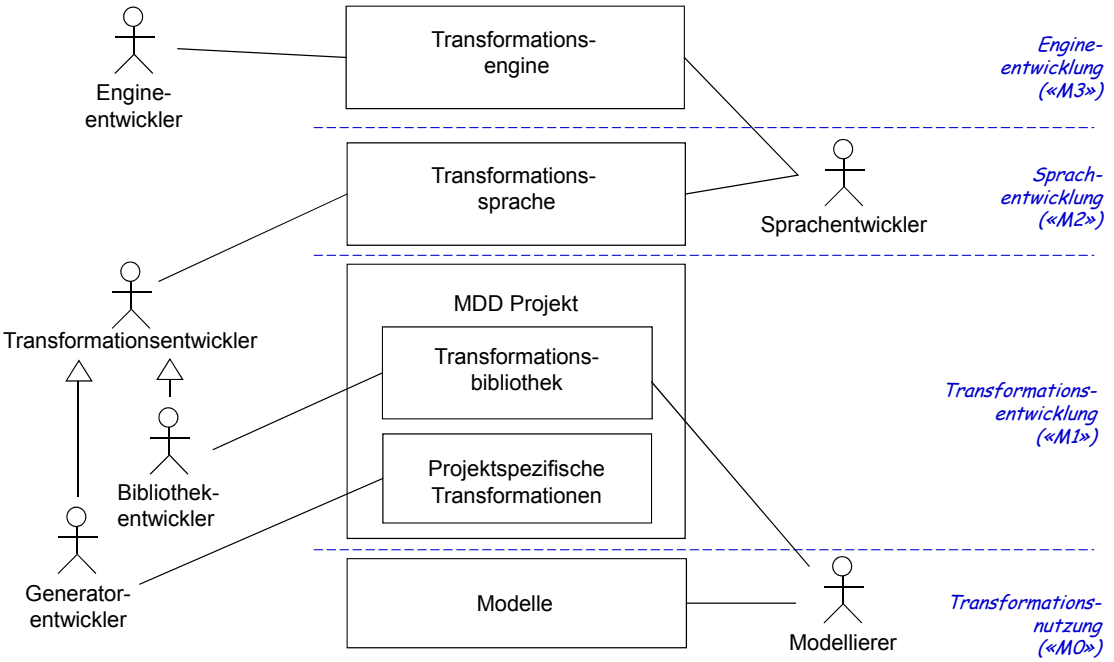


Abbildung 3.1: Rollen und deren Interaktion mit der Transformationsengine, Transformationssprachen und konkreten Transformationen.

Engineentwickler: Der Engineentwickler ist auf der obersten der definierten Ebenen einzuordnen. Er ist ausschließlich für die Entwicklung der MontiTrans Transformationsengine zuständig, die aus einer durch eine MontiCore-Grammatik definierten Sprache eine dazugehörige Transformationssprache inklusive Infrastruktur zur Übersetzung und Anwendung von Transformationen generiert.

Sprachentwickler: Der Sprachentwickler ist auf der zweiten Ebene einzuordnen. Er ist ein Nutzer der Transformationsengine und entwickelt generativ neue Transformationssprachen inklusive Infrastruktur zur Übersetzung und Anwendung von Transformationen für gegebene oder selbst entwickelte Modellierungssprachen.

Transformationsentwickler: Auf der dritten Ebene befindet sich der Transformationsentwickler. Er ist Nutzer der generativ entwickelten Transformationssprachen. Die Rolle kann weiter unterteilt werden in Bibliotheks- und Generatorentwickler.

Bibliotheksentwickler: Ein Bibliotheksentwickler entwickelt Transformationen mit dem primären Ziel diese zur Wiederverwendung bereitzustellen.

Generatorentwickler: Der Generatorentwickler kann sowohl auf der Ebene der Transformationsentwicklung als auch auf der Ebene der Transformationsnutzung eingeordnet werden. Er kann während der Umsetzung von MDD Projekten sowohl neue Transformationen spezifisch für das aktuelle Projekt entwickeln als

auch Gebrauch von bereits entwickelten und bereitgestellten Transformationen machen, um neue Transformationen zu entwickeln.

Modellierer: Der Modellierer wird der vierten und damit untersten Ebene zugeordnet. Er entwickelt neue Modelle und kann hierzu bereits entwickelte Transformationen nutzen, um Modelle weiter zu entwickeln. Damit ist er nicht mehr in direktem Sinne Nutzer der Transformationssprachen, kann die Notation zumindest zur Dokumentation von Transformationen jedoch kennen.

3.1.2 Szenario 1: Neuentwicklung einer Transformationssprache

Sven ist Sprachentwickler und hat eine neue Modellierungssprache zur Beschreibung von Automaten entwickelt. Zur Entwicklung hat er die Language Workbench MontiCore verwendet und baut auf der dort bereitgestellten Basissprache *Literals* auf. Sven möchte den Nutzern seiner Sprache – vorwiegend Domänenexperten – eine Transformationssprache zur Verfügung stellen, die möglichst ähnlich zu der verwendeten Automatensprache ist und somit wenig Einarbeitungsaufwand erfordert. Um den eigenen Aufwand gering zu halten, möchte er diese automatisch generieren lassen und daher die DSTL-Generierung von MontiCore [Wei12] nutzen. Sven möchte frei in der Verwendung der Grammatikfeatures sein und insbesondere seine Grammatik nicht umschreiben müssen, um den DSTL Generator verwenden zu können.

Daraus ergeben sich die nachfolgenden Anforderungen. Jede Anforderung hat ein Kürzel der Form GR, LR oder TR plus eine Nummerierung. Hierbei steht G für Anforderungen an den DSTL-Generator, L für Anforderungen an Transformationssprachen und T für Anforderungen an Transformationen.

- Das MontiCore-Grammatikformat muss unterstützt werden (GR 1). Der in [Wei12] entwickelte Generator unterstützt bereits normale Nichtterminale. Zusätzlich müssen folgende Features des Grammatikformats unterstützt werden:
  - Interface-Nichtterminale müssen unterstützt werden (GR 1.1)
  - Abstrakte Nichtterminale müssen unterstützt werden (GR 1.2)
  - Externe Nichtterminale müssen unterstützt werden (GR 1.3)
  - Nichtterminalerweiterung muss unterstützt werden (GR 1.4)
  - Redefinition von Nichtterminale muss unterstützt werden (GR 1.5)
- Der DSTL-Generator muss die Modularitätskonzepte des MontiCore-Grammatikformats unterstützen (GR 2). Der in [Wei12] entwickelte Generator unterstützt bereits monolithische Sprachen. Es müssen folgende Features des Grammatikformats unterstützt werden:
  - Sprachvererbung muss unterstützt werden (GR 2.1)
  - Spracheinbettung muss unterstützt werden (GR 2.2)

- Für die Basissprachen von MontiCore muss es DSTLs geben (LR 3), die als Bausteine für die Entwicklung neuer DSTLs dienen können.

Damit die Nutzer seiner Transformationssprache nicht nur Transformationen aufschreiben, sondern diese auch ausführen können, braucht Sven außerdem eine Infrastruktur, die dies ermöglicht. Gleichzeitig sollen die Nutzer seiner DSTL keine syntaktisch korrekten, aber semantisch invaliden Transformationen aufschreiben können. Daher soll es für die Transformationen Prüfungen geben, ob diese wohlgeformt sind.

Daraus ergeben sich die folgenden Anforderungen:

- Es muss generierte Wohlgeformtheitschecks für generierte DSTLs geben (GR 3.1)

Schließlich möchte Sven noch in der Lage sein, die DSTL bzw. deren Infrastruktur an die Bedürfnisse seiner Nutzer anzupassen. Ein geeigneter Konfigurations- bzw. Erweiterungsmechanismus muss daher durch den DSTL Generator gegeben sein.

Daraus ergeben sich die folgenden Anforderungen:

- Es muss einen Erweiterungsmechanismus für die generierte DSTL bzw. deren Infrastruktur geben (GR 13)

3.1.3 Szenario 2: Entwicklung von Transformationen

Für die Entwicklung von Transformationen ergeben sich drei Szenarien für die exemplarisch betrachteten Modellierungssprachen Klassendiagramme und MontiArc. Das erste Szenario betrachtet die Klassendiagrammtransformationsentwicklung, das zweite die Entwicklung von MontiArc-Transformationen und das dritte die Entwicklung von Transformationen für die gemeinsame Behandlung von MontiArc-Modellen und Klassendiagrammen.

Szenario 2a: Entwicklung von Klassendiagrammtransformationen

Karla ist eine Softwareentwicklerin. Sie nutzt zur Strukturmodellierung Klassendiagramme und beherrscht die Programmiersprache Java. Sie nutzt einen Generator, der aus Klassendiagrammen Java-Code generiert. Um die Modelle vor der Generierung anzupassen, möchte sie Transformationen schreiben, die die Klassendiagramme automatisiert anpassen können.

Daraus ergeben sich die folgenden Anforderungen:

- Es muss eine DSTL für Klassendiagrammtransformationen geben (LR 1)

Szenario 2b: Entwicklung von MontiArc-Transformationen

Marc ist ebenfalls ein Softwareentwickler. Er nutzt zur Modellierung die Architekturbeschreibungssprache MontiArc. Er entwickelt eine Sammlung von Transformationen, die im Kontext der Architekturmodellierung relevant sind.

Daraus ergeben sich die folgenden Anforderungen:

- Es muss eine DSTL für MontiArc-Transformationen geben (LR 2)

Zudem möchte er die Transformationsentwicklung modular gestalten. Ist ein Teil einer Transformation für eine andere wiederverwendbar, dann will er diesen für beide Transformationen nutzen können und diesen nicht kopieren müssen.

Daraus ergeben sich die folgenden Anforderungen:

- Es muss möglich sein, Transformationsregeln in verschiedenen Transformationen wieder zu verwenden (GR 6)

Szenario 2c: Entwicklung von kombinierten Klassendiagramm-MontiArc-Transformationen

Mirko arbeitet mit einer Kombination aus MontiArc-Modellen und Klassendiagrammen. Die Klassendiagramme dienen ihm sowohl zur implementierungsnahen Modellierung von Softwaresystemen als auch zur Definition von Typen, die er innerhalb der MontiArc-Modelle verwendet. Er möchte eine Transformationssprache, die sowohl die ihm vertraute Syntax der Modellierungssprachen aufgreift als auch erlaubt Transformationen auf einer Kombination aus MontiArc-Modellen und Klassendiagrammen auszudrücken.

Daraus ergeben sich die folgenden Anforderungen:

- Es muss eine MontiArc-Klassendiagramm-DSTL geben (LR 4)
  - Es muss möglich sein, DSTLs für eine Kombination von Modellierungssprachen zu erstellen (GR 5)
- Transformationen müssen auf mehreren Modellen arbeiten können (GR 4)
  - Es muss möglich sein, Pattern über Modellgrenzen hinweg zu finden (GR 4.1)
  - Es muss möglich sein, Modifikation mehrerer Modelle innerhalb einer Transformation auszudrücken (GR 4.2)
  - Mehrere Ausgabemodelle müssen möglich sein (GR 4.3)
- Die Transformationen müssen auf mehreren Modellen verschiedener Modellierungssprachen arbeiten können (GR 5, GR 4)

Gemeinsamkeiten der Szenarien 2a, 2b, und 2c

Alle drei Entwickler haben gemeinsam, dass sie sich so wenig wie möglich von der abstrakten Syntax der Modellierungssprache aneignen wollen. Zudem möchten sie nur so viele Transformationen entwickeln, wie unbedingt notwendig.

Daraus ergeben sich die folgenden Anforderungen:

- Die DSTLs müssen sich an der konkreten, den Entwicklern bekannten Syntax der Modellierungssprache orientieren. Gleichzeitig müssen die Constraints der Sprache gelockert werden, da ein Auslassen von Modellteilen für Pattern wichtig ist, um Transformationen allgemeingültig und nicht modellspezifisch formulieren zu

können. Es muss Konzepte zur Generalisierung von Transformationsregeln geben (GR 7). Dazu ist insbesondere wichtig:

- Die DSTLs erlauben es, Patternelemente an Schemavariablen zu binden. Der Typ einer Schemavariable soll aus dem Kontext der Schemavariable inferiert statt durch den Nutzer angegeben werden, um das notwendige Wissen über die abstrakte Syntax zu minimieren. (GR 3.2)
- Zur besseren Generalisierung von Transformationsregeln müssen die Constraints der Sprache zur Angabe von Pattern gelockert werden (Relaxation), z.B. Weglassen von Elementen (GR 7.1)
- Es muss möglich sein, optional Elemente anzugeben, die durch die Transformation behandelt werden, falls sie im Modell vorhanden sind (GR 7.2). Dadurch lässt sich die Anzahl der zu entwickelnden Transformationen reduzieren, da mehr Transformationen zusammengefasst werden können.

3.1.4 Szenario 3: Verwendung von Transformationen

Ursula ist Modelliererin und Domänenexpertin im Bereich Robotik. Sie verwendet zur Modellierung der Domäne Klassendiagramme und MontiArc-Modelle. Die Modelle, mit denen sie arbeitet, sind in der Regel handgeschrieben und über einen längeren Zeitraum hinweg entstanden. Aus diesem Grund gibt es regelmäßig Bedarf für Qualitätsverbesserungen. Um diese oft schematischen Probleme zu lösen, möchte Ursula Refactoring in Form von vorgefertigten Transformationen nutzen. Da Ursula für verschiedene Zielplattformen wie beispielsweise das Robot Operating System [QGC+09] oder Java modelliert, benötigt sie außerdem Transformationen, die MontiArc-Modelle normalisieren und damit auf ein reduziertes Featureset von MontiArc abbilden.

Daraus ergeben sich die folgenden Anforderungen:

- Es muss eine Bibliothek für Klassendiagrammtransformationen geben (TR 1)
- Es muss eine Bibliothek für MontiArc-Transformationen geben (TR 2)
- Falls es Varianten einer Transformation gibt, soll dies erkennbar sein, beispielsweise durch geeignete Schnittstellen (TR 3)

3.2 Analyse von Transformationen

In Abschnitt 3.1 wurden verschiedene Szenarien für MontiTrans vorgestellt und die daraus resultierenden Anforderungen an MontiTrans erläutert. Zusätzlich wurden im Rahmen dieser Arbeit verschiedene existierende Generatoren, die zur Generierung auf Modelltransformationen in Kombination mit templatebasierter Generierung zurückgreifen, untersucht: der Data Explorer Generator [MSNRR15, Rot17], der MontiCore-Generator [GKR+08, KRV08, KRV10, Kra10] und der MontiArc-Generator [HRR10, HRR12, Hab16]. Neben den Anforderungen, die sich aus den Szenarien ergeben, wurden hierbei zusätzlich die folgenden Anforderungen identifiziert:

- Es muss eine Möglichkeit zur Stringkonkatenation geben, um Namen basierend auf im Modell vorhandenen Namen berechnen zu können (GR 8)
- Es muss möglich sein, die Anwendbarkeit einer Transformation an die Abwesenheit ganzer Teilstrukturen zu binden (GR 9)
- Es muss möglich sein, beliebig viele gleiche Strukturen innerhalb des Modells zu matchen (GR 10)
- Es muss eine Integration mit dem Tempateerweiterungsmechanismus geben, um erstellte Modellelemente mit Templates assoziieren zu können (GR 14)

Darüber hinaus wurden bei der Analyse spezifische Anforderungen an die Klassendiagrammtransformationssprachen bzw. die MontiArc-Transformationssprache identifiziert. Für die Klassediagrammtransformationssprache ergeben sich durch die Anforderung des Zusammenfassens ähnlicher Transformationen (GR 7) die folgenden Anforderungen für komfortablere Schreibweisen:

- Attribute und Methoden sollen durch einander ersetzt werden können (LR 1.1)
- Interfaces, Klassen und Enums sollen mit dem selben Ersetzungsoperator durch einander ersetzt werden können (LR 1.2)
- Modifikatoren (z.B. `public`, `protected`) sollen durch einander ersetzt werden können (LR 1.3)
- Es muss möglich sein, bei der Angabe von Pattern den Typ einer Assoziation zu ändern (Komposition oder Assoziation) (LR 1.4)
- Es muss möglich sein, bei der Angabe von Pattern von der Assoziationsrichtung zu abstrahieren (LR 1.5)
- Es muss möglich sein, bei der Angabe von Pattern von dem Assoziationstyp (Komposition oder Assoziation) zu abstrahieren (LR 1.6)

Für die MontiArc-Transformationssprache ergeben sich durch die Anforderung des Zusammenfassens ähnlicher Transformationen (GR 7) die folgenden Anforderungen für komfortablere Schreibweisen:

- Es muss eine Abstraktionsmöglichkeit von der Portrichtung geben (LR 2.1)
- Innere Komponenten sollen durch Subkomponenten ersetzbar sein (LR 2.2)

Im Folgenden werden die Ergebnisse der Analyse der einzelnen Generatoren vorgestellt. Es wird jeweils auf die Anzahl bzw. den Anteil der Transformationen innerhalb der Generatoren eingegangen, die ein bestimmtes Feature benötigen.

3.2.1 Analyse des Data Explorer Generator

Der Data Explorer Generator ist ein Generator, der aus einem gegebenen Klassendiagramm eine datenzentrische Businessapplikation generiert [MSNRR15, Rot17]. Die Generierung geschieht hierbei durch eine Kombination aus Modelltransformationen des als Eingabe verwendeten Klassendiagramms und templatebasierter Codegenerierung mithilfe von FreeMarker [Fre17].

	#T	#T mit Stringkon-katenation	#T mit Negation	#T mit Collection	#T mit GlobalExtension-Management
Daten-klassen	14	8	11	7	9
DB	14	14	13	2	14
GUI	15	15	15	0	15
Gesamt	43	37	39	9	38

Tabelle 3.2: Übersicht über die Data Explorer Transformationen.

Zum Zeitpunkt der Analyse verwendete der Data Explorer Generator 43 Modelltransformationen, die sich in die drei Bereiche *Graphische Benutzeroberfläche* (GUI), *Datenverwaltung* und *Persistenz* einordnen lassen (vgl. Abbildung 3.2). Hierbei verteilen sich die Transformationen gleichmäßig auf die verschiedenen Kategorien (15, 14 und 14). Die Abbildung stellt die Anzahl der Transformationen (#T) dar, die die einzelnen Features (Stringkonkatenation, Negation komplexer Strukturen, Collections komplexer Strukturen, Verwendung des GlobalExtensionManagements) benötigen. Die Zeilen repräsentieren die Kategorien Datenhaltungsklassen (Datenklassen), Benutzeroberfläche (GUI) und Persistenz, sowie die Gesamtheit der Transformationen (Gesamt). Zusammengefasst zielen die Transformationen stark auf die anschließende templatebasierte Generierung ab. Dies spiegelt sich darin wider, dass 38 der 43 Transformationen – gut 88 % der Transformationen – den Templateerweiterungsmechanismus über die Klasse *GlobalExtensionManagement* von MontiCore verwenden, um Templates an die erstellten Modellelemente anzuhängen. Weitere stark genutzte Features sind die Negation beliebiger Modellstrukturen, welche für 39 der 43 Transformationen benötigt wird (ca. 91 %), und die Möglichkeit Strings zu manipulieren bzw. basierend auf vorhandenen Namen neue Namen zu berechnen. Dies benötigen immerhin 85 % der Transformationen (d.h. 37 der 43 Transformationen). Ein im Vergleich zu den anderen Features weniger genutztes Feature ist die Möglichkeit zum Matchen beliebig vieler gleicher Strukturen (Collections). Lediglich 9 der 43 Transformationen (21 %) benötigen dieses Feature. Die Verteilung auf die verschiedenen Arten der Transformationen wird im Folgenden erläutert. Außerdem werden weitere als nützlich identifizierte Features beschrieben.

Datenklassen

14 der insgesamt 43 Transformationen, die innerhalb des Data Explorer Generators verwendet werden, dienen der Generierung von Datenhaltungsklassen. Der Anteil der Transformationen, die Stringkonkatenation benötigen, ist hier im Vergleich zum gesamten Generator vergleichsweise gering. Nur 8 der 14 Transformationen (57 % statt 91 %) benötigen die Möglichkeit zur Stringkonkatenation. Im Gegensatz dazu ist der Anteil der Transformationen, die viele gleichartige Strukturen matchen, deutlich erhöht (7 von 14, d.h. 50 %). Der Grund hierfür ist, dass viele Transformationen in dieser Kategorie Normalisierungen und Refactorings sind, während die Transformationen zur Generierung der GUI oder Persistenzklassen einen additiven Charakter haben und damit häufiger neue Elemente basierend auf bestehenden erstellen. Neben diesen allgemeinen Features ist während der Analyse außerdem der Bedarf zum Zusammenfassen von Assoziationsrichtungen, die direkte Ersetzung von Methoden durch Attribute, Klassen, Interfaces und Enums und der verschiedenen Sichtbarkeiten der Modifier durch einander aufgefallen.

Persistenz

14 der insgesamt 43 Transformationen, die innerhalb des Data Explorer Generators verwendet werden, dienen der Generierung der Persistenz. Durch den additiven Charakter der Transformationen benötigen alle 14 Transformationen Stringkonkatenation sowie Template Attachment mittels GlobalExtensionManagement. Im Vergleich zu den Transformationen zur Generierung von Datenhaltungsklassen ist der Bedarf für einen Collection Operator für hierarchische Strukturen eher gering (2 der 14 Transformationen, ca. 14 %). Negation komplexer Strukturen wird hingegen von den meisten Transformationen dieser Kategorie benötigt (ca. 93 %, 13 Transformationen). Ein Bedarf an klassendiagrammspezifischen Features konnte in dieser Kategorie nicht festgestellt werden.

Benutzeroberfläche (GUI)

15 der insgesamt 43 Transformationen, die innerhalb des Data Explorer Generator verwendet werden, dienen der Generierung der graphischen Benutzeroberfläche (GUI). Durch den additiven Charakter der Transformationen benötigen alle 15 Transformationen Stringkonkatenation, Negation komplexer Strukturen sowie den Templateerweiterungsmechanismus. Im Vergleich zu den Transformationen zur Generierung von Datenhaltungsklassen ist der Bedarf für einen Collection Operator für hierarchische Strukturen nicht gegeben. Keine der betrachteten Transformationen zeigte hierfür Bedarf. Ein Bedarf an zusätzlichen klassendiagrammspezifischen Features konnte in dieser Kategorie nicht festgestellt werden.

3.2.2 Analyse des MontiCore-Generator

Der MontiCore-Generator [Kra10] generiert zu einer gegebenen Grammatik den Parser und Lexer, die AST-Klassen [KRV10, KRV08, GKR+08], sowie eine Infrastruktur für die Erstellung von Symboltabellen [Völ11, MSN17], Kontextbedingungen und Visitoren

[HMSNRW16]. Die Generierung erfolgt – ähnlich wie die des Data Explorer Generators – zunächst mithilfe von Modelltransformationen und geht anschließend zu einer templatebasierten Generierung über. In einem ersten Schritt wird während der Generierung der Grammatik-AST in einen AST der Klassendiagrammsprache überführt. Anschließend wird das entstandene Klassendiagramm mittels Klassendiagrammtransformationen mit weiteren, für die templatebasierte Generierung relevanten Informationen angereichert. Schließlich wird dieses angereicherte Klassendiagramm als Eingabe für die templatebasierte Generierung verwendet.

	#T	#T mit Stringkon-katenation	#T mit Negation	#T mit Collection	#T mit GlobalExtension-Management
Gesamt	11	6	10	1	8

Tabelle 3.3: Übersicht über die MontiCore Generator Transformationen.

Der MontiCore-Generator verwendet zum Zeitpunkt der Analyse 11 Transformationen (vgl. Abbildung 3.3), welche händisch in Java implementiert wurden. Die Transformationen haben im wesentlichen additiven Charakter. Dies zeigt sich beispielsweise durch den geringen Bedarf für einen Collection Operator für komplexe Strukturen. Lediglich 1 der 11 Transformationen zeigte hierfür Bedarf (9%). Den Templateerweiterungsmechanismus benötigen 8 der 11 Transformationen (ca. 73 %). Negation komplexer Strukturen wird von fast allen Transformationen benötigt (10 der 11 Transformationen, 91 %). Konkatenation von Strings wird ebenfalls von einer Mehrheit der Transformationen benötigt (6 von 11, 55 %). Ein Bedarf an zusätzlichen klassendiagrammspezifischen Features konnte in dieser Kategorie nicht festgestellt werden.

3.2.3 Analyse des MontiArc-Generator

Der MontiArc-Generator wurde in [Hab16] vorgestellt. Basierend auf einem MontiArc-Modell generiert er eine Simulationsinfrastruktur für das gegebene Modell. Auch dieser Generator arbeitet mit einer Kombination aus Transformationen und templatebasierter Codegenerierung. Der Generator verwendet Transformationen zu zwei verschiedenen Zwecken. Der erste Teil der Transformationen wird angewendet, um anschließend die Kontextbedingungen (leichter) prüfen zu können. Der zweite Teil der Transformationen bereitet das Modell auf die templatebasierte Codegenerierung vor. Insgesamt verwendet der Generator 10 Transformationen, die sich auf diese beiden Kategorien (Pre CoCo und Pre CodeGen) aufteilen (vgl. Abbildung 3.4).

Insgesamt hat die Analyse ergeben, dass 40 % der Transformationen Stringkonkatenation benötigt (4 von 10) und 60 % der Transformationen Bedarf für die Negation komplexer Strukturen haben. Collections komplexer Strukturen sowie die Möglichkeit den Templateerweiterungsmechanismus zu nutzen, benötigt hingegen keine der betrachteten Transformationen. Hierbei ist zu beachten, dass der Codegenerator insgesamt kei-

	#T	#T mit Stringkon-katenation	#T mit Negation	#T mit Collection
Pre CoCo	4	0	4	0
Pre CodeGen	6	4	2	0
Gesamt	10	4	6	0

Tabelle 3.4: Übersicht über die MontiArc Generator Transformationen.

nen Gebrauch von Templateerweiterung macht, was sich auch in den Transformationen widerspiegelt. Im Folgenden wird der Bedarf innerhalb der zwei Kategorien detailliert.

Transformationen zur Vorbereitung der Prüfung von Kontextbedingungen

4 der 10 Transformationen dienen der Vorbereitung des Modells auf die Prüfung der Kontextbedingungen. Bei diesen Transformationen handelt es sich um normalisierende Transformationen [MVG06]. Aus diesem Grund benötigt keine dieser Transformationen die Möglichkeit der Stringkonkatenation. Ebenfalls konnte für diese Transformationen kein Bedarf für Collections komplexer Strukturen ermittelt werden. Jede der Transformationen in dieser Kategorie benötigt hingegen Negation komplexer Strukturen. Als zusätzlich hilfreiches Feature benötigen die Transformationen die Möglichkeit innere Komponenten durch Subkomponenten zu ersetzen.

Transformationen zur Vorbereitung der Codegenerierung

6 der 10 Transformationen dienen der Vorbereitung des Modells auf die Prüfung der Kontextbedingungen. Bei diesen Transformationen handelt es sich ebenfalls um normalisierende Transformationen [MVG06]. 4 der 10 Transformationen benötigen Stringmanipulation und 2 brauchen die Negation komplexer Strukturen. Ähnlich wie die Transformationen zur Vorbereitung auf die Überprüfung von Kontextbedingungen benötigen auch diese Transformationen keine Collections komplexer Strukturen. Als zusätzliches Feature benötigen die Transformationen die Möglichkeit der Abstraktion von Portrichtungen.

3.2.4 Fazit der Analyse

Es wurden 64 Transformationen untersucht. Darunter waren 54 Transformationen für Klassendiagramme und 10 Transformationen für MontiArc-Modelle. Die Analyse hat bestätigt, dass die Features Stringkonkatenation (47 von 64), Negation von Teilstrukturen (55 von 64) und die Integration mit dem Templateerweiterungsmechanismus (46 von 64) generatorübergreifend benötigt werden, und damit in die Anforderungen dieser Arbeit aufgenommen werden mussten. Auffällig ist, dass Collections von Teilstrukturen nur

von 10 der 64 Transformationen benötigt wurde. Da es dennoch von 2 der 3 Generatoren benötigt wird, wurde auch dieses Feature in die Anforderungen aufgenommen. Zusätzlich wurden für Klassendiagrammtransformationen 6 und für MontiArc-Transformationen 2 Anforderungen für komfortablere Schreibweisen identifiziert.

3.3 Anforderungen an MontiTrans

In den vorhergehenden Abschnitten wurden Szenarien vorgestellt, aus denen Anforderungen abgeleitet wurden. Außerdem wurde die Analyse bestehender Transformationen im Kontext von MontiCore vorgestellt, aus der sich weitere Anforderungen ableiten ließen. In diesem Abschnitt werden die identifizierten Anforderungen zusammengeführt und um weitere Details ergänzt. Jede Anforderung hat ein Kürzel der Form **GR**, **LR** oder **TR** plus eine Nummerierung und gegebenenfalls Subnummerierung. Hierbei steht **G** für Anforderungen an den DSTL-Generator, **L** für Anforderungen an die Transformationssprachen und **T** für Anforderungen an Transformationen.

3.3.1 Anforderungen an den DSTL-Generator

In diesem Abschnitt werden die Anforderungen an den DSTL-Generator beschrieben. Diese Anforderungen leiten sich sowohl aus den Szenarien als auch aus der in Abschnitt 3.2 beschriebenen Analyse von handgeschriebenen Modelltransformationen für Klassendiagramme und MontiArc-Modelle ab.

GR 1 Grammatikformat: Damit die Generierung für eine beliebige MontiCore-Grammatik möglich ist, müssen alle Arten von Produktionen unterstützt werden, d.h. neben Klassenproduktionen müssen auch die restlichen Produktionsarten des Grammatikformats unterstützt werden.

- **GR 1.1 Interface-Nichtterminale:** Interface-Nichtterminale sind eine Erweiterung von Alternativen. Wird innerhalb des Rumpfes einer Produktion ein Interface-Nichtterminal verwendet, dürfen an dieser Stelle alle Nichtterminale, die dieses Interface implementieren, vorkommen. Die Generierung von Transformationssprachen muss diese Form der Alternativen unterstützen, um die vollständige Unterstützung des Grammatikformats zu gewährleisten.
- **GR 1.2 Abstrakte Nichtterminale:** Abstrakte Nichtterminale sind eine weitere Form der Erweiterung von Alternativen. Verwendet eine Produktion ein solches Nichtterminal, dürfen an dieser Stelle alle Nichtterminale, die dieses erweitern, vorkommen. Die Generierung von Transformationssprachen muss diese Form der Alternativen unterstützen, um die vollständige Unterstützung des Grammatikformats zu gewährleisten.
- **GR 1.3 Externe Nichtterminale:** Externe Nichtterminale definieren innerhalb einer Sprache Erweiterungspunkte. Diese Erweiterungspunkte dürfen nicht

zu inkorrekten Transformationssprachen führen. Des Weiteren müssen die Erweiterungen bei der Generierung beachtet werden.

GR 1.4 Nichtterminalerweiterung: Das MontiCore-Grammatikformat erlaubt, dass Nichtterminale von anderen Nichtterminale erweitert werden. Eine solche Erweiterung entspricht dem Hinzufügen einer Alternative zum Rumpf der erweiterten Produktion. Verwendet eine Produktion in ihrem Rumpf ein erweitertes Nichtterminal, sind beide Alternativen möglich. Dies muss durch die Transformationssprachen behandelt werden.

GR 1.5 Redefinition von Nichtterminalen: MontiCore erlaubt es, Nichtterminale einer Supersprache in einer Subsprache zu redefinieren. Die Generierung von Transformationssprachen muss diese Form der Nichtterminaldefinition unterstützen, um die vollständige Unterstützung des Grammatikformats zu gewährleisten.

GR 2 Modulare Sprachdefinition: Das Grammatikformat von MontiCore bietet die Möglichkeit, Sprachen durch Wiederverwendung und Erweiterung von vorhandenen Sprachen oder Sprachkomponenten zu entwickeln. Die Generierung von Transformationssprachen muss daher eine modulare Sprachdefinition unterstützen.

GR 2.1 Sprachvererbung: Eine Möglichkeit zur modularen Definition von Sprachen ist Sprachvererbung. Um Transformationssprachen für nicht monolithische Sprachdefinitionen generieren zu können, müssen solche Arten der Sprachdefinition durch den Generierungsprozess unterstützt werden.

GR 2.2 Spracheinbettung: Die zweite Möglichkeit zur modularen Sprachdefinition ist Spracheinbettung. Dieses Konzept muss ebenfalls durch den Generierungsprozess berücksichtigt werden.

GR 3 Usability: Generierte Transformationssprachen sollen leicht verwendbar sein. Neben der Verwendung der konkreten Syntax beinhaltet dies:

GR 3.1 Wohlgeformtheit: Der Transformationsentwickler muss im Fall von syntaktisch korrekten, aber nicht wohlgeformten Transformationen, durch Fehlermeldungen auf die nicht wohlgeformten Teile seiner Transformation hingewiesen werden.

GR 3.2 Typinferenz: Explizite Typabgaben für Variablen erfordern ein Verständnis der zugrundeliegenden abstrakten Syntax. Da dies dem Konzept von Transformationen in konkreter Syntax entgegenwirkt, soll der Typ einer Variablen nach Möglichkeit inferiert werden.

GR 4 Mehrere Modelle: Die Praxis zeigt, dass komplexe Systeme häufig durch mehrere Modelle beschrieben werden. Es werden beispielsweise verschiedene strukturelle Aspekte durch verschiedene Klassendiagramme oder strukturelle und Verhaltensaspekte durch die Verwendung von Struktur- sowie Verhaltensmodellen dargestellt. Die Transformationen dürfen daher nicht auf die Transformation

einzelner Modelle beschränkt sein. Die Transformation mehrerer Eingabe- zu mehreren Ausgabemodellen muss möglich sein. Dies beinhaltet insbesondere:

GR 4.1 Matching in mehreren Modellen: Generierte Transformationssprachen müssen es ermöglichen, mehrere Eingabemodelle zu verarbeiten und hierbei diese Modelle zusammenzuführen, d.h. es muss möglich sein Pattern über mehrere Modelle anzugeben und Modifikationen basierend auf diesen Pattern in einem oder mehreren Modellen vorzunehmen, beispielsweise um diese Modelle zu verschmelzen.

GR 4.2 Modifikation in mehreren Modellen gleichzeitig: Wird ein Pattern über mehrere Modelle hinweg formuliert, muss es zudem möglich sein Modifikationen an allen betroffenen Modellen innerhalb der gleichen Transformation auszudrücken.

GR 4.3 Mehrere Modelle als Ausgabe: Neben dem Teilen von Modellen muss es auch möglich sein mehrere Modelle mittels einer Transformation zu erstellen und somit mehrere Ausgabemodelle zu kreieren.

GR 5 Mehrere Sprachen: In der modellgetriebenen Softwareentwicklung werden Systeme häufig mit Modellen verschiedener Sprachen modelliert, so wird die Struktur beispielsweise durch Klassen- oder Architekturdiagramme (oder eine Kombination dieser beiden) modelliert, während Verhalten durch Statecharts, Aktivitätsdiagramme oder Sequenzdiagramme beschrieben wird. Diese Modelle stehen häufig in Bezug zueinander. Ändern sich beispielsweise Klassen innerhalb eines Klassendiagramms, hat dies in der Regel Änderungen der Architekturbeschreibung oder Verhaltensbeschreibung zur Folge. Aus diesem Grund muss es möglich sein, Transformationssprachen zu erstellen, die sich auf Modelle verschiedener Sprachen beziehen.

GR 6 Modularisierung von Transformationen: Um die Wiederverwendung von Transformationsregeln zu unterstützen, müssen komplexe Transformationen modular gestaltet werden können. Dazu muss es möglich sein Transformationsregeln isoliert zu definieren und a posteriori durch eine Anwendungsstrategie zu komplexen Transformationen zu kombinieren.

GR 7 Zusammenfassen von gleichen Transformationen: Die Analyse handgeschriebener Transformationsimplementierungen hat gezeigt, dass Generalisierungsmechanismen innerhalb von Pattern wichtig sind, da die Anzahl der Transformationen ansonsten schnell anwächst, obwohl sich die Modelle nur geringfügig unterscheiden. Beispielweise kann häufig von der Richtung oder Art einer Assoziation abstrahiert werden. Darüber hinaus kann die Anzahl notwendiger Transformationen weiter reduziert werden, wenn Teile optional mit behandelt werden, falls sie vorhanden sind. Es ergeben sich folgende Anforderungen:

GR 7.1 Relaxation: Die Transformationssprachen müssen bei der Beschreibung von Pattern mehr Freiheiten bieten als die zugrundeliegende Modellie-

rungssprache. Es muss erlaubt sein Modellelemente, die im Modell verpflichtend sind innerhalb von Pattern auszulassen, falls diese für das Pattern unerheblich sind.

GR 7.2 **Optionale Elemente:** In manuellen Transformationsimplementierungen kann leicht auf weitere Teile innerhalb des transformierten Modellausschnitts reagiert werden. Die Transformationssprachen müssen ebenfalls eine Möglichkeit anbieten, solche Transformationen auszudrücken.

GR 8 **Stringkonkatenation:** Die Analyse handgeschriebener Transformationsimplementierungen hat gezeigt, dass Namen von neu eingeführten Modellelementen häufig nicht direkt aus dem Modell bzw. Pattern stammen, sondern ausgehend von diesen berechnet werden. Ein Beispiel ist die Benennung von Zugriffsmethoden für Attribute, die typischerweise aus den Namen des Attributes inklusive eines Präfixes bestehen. Generierte Transformationssprachen müssen die Möglichkeit bieten Namen basierend auf anderen Modellelementen zu berechnen.

GR 9 **Negation für beliebige Strukturen:** In [Wei12] wurde ein Operator vorgestellt, der die Definition negativer Elemente zur Formulierung von Negative Application Conditions erlaubt. In der vorgestellten Umsetzung ist dieser Operator auf Knoten ohne ausgehende Kanten, d.h. „atomare“ Modellelemente beschränkt. Die Analyse vorhandener Modellierungssprachen hat gezeigt, dass die Mehrheit der Modellelemente aus mehreren Knoten komponiert ist. Der Operator muss daher für nicht atomare Modellelemente erweitert werden.

GR 10 **Matchen beliebig vieler gleicher Strukturen:** Bereits in [Wei12] wurde ein Operator vorgestellt, der die Definition mehrerer gleicher Modellelemente erlaubt. In der in [Wei12] vorgestellten Umsetzung war dieser Operator auf Knoten ohne ausgehende Kanten, d.h. „atomare“ Modellelemente beschränkt. Die Analyse vorhandener Modellierungssprachen hat gezeigt, dass die Mehrheit der Modellelemente aus mehreren Knoten komponiert ist. Der Operator muss daher für nicht atomare Modellelemente erweitert werden.

GR 11 **Direkte Manipulation:** Neben der Möglichkeit Modellmodifikationen innerhalb der Transformationssprache auszudrücken, soll die Transformationssprache auch die Möglichkeit bieten nur den Pattern Matching Mechanismus zu nutzen und Manipulationen programmatisch auf dem Modell zu beschreiben.

GR 12 **Zuweisung von Schemavariablen:** Neben der Belegung von Schemavariablen durch das Pattern Matching muss es auch die Möglichkeit geben, Schemavariablen explizit Werte zu zuweisen. Zusätzlich solle es möglich sein, dass diese Werte berechnet werden.

GR 13 **Erweiterbarkeit:** Die generierten DSTLs sind schematisch erstellt. Damit der Sprachentwickler die Sprache für seine Nutzer weiter anpassen kann, muss es einen Erweiterungsmechanismus geben.

GR 14 Kombination von Transformationen und templatebasierter Generierung: MontiCore bietet mithilfe der Templateengine Freemarker die Möglichkeit der templatebasierten Generierung. Zusätzlich erlaubt MontiCore das Anhängen von Templates an Modellelemente. Diese Templates werden dann während der Generierung berücksichtigt und entsprechend verwendet. Die Funktionalität hierfür bietet die Klasse `GlobalExtensionManagement`. Um eine Kombination aus Transformationen und templatebasierter Generierung zu erleichtern, müssen die Transformationen die Möglichkeit haben, Templates an Modellelemente zu hängen.

3.3.2 Anforderungen Transformationssprachen

Im Folgenden werden die Anforderungen für konkrete Transformationssprachen beschrieben. Diese Anforderungen leiten sich sowohl aus den Szenarien als auch aus der in Abschnitt 3.2 beschriebenen Analyse von handgeschriebenen Modelltransformationen für Klassendiagramme bzw. MontiArc-Modelle ab.

LR 1 Klassendiagrammtransformationssprache: Um Transformationen von Klassendiagrammen in konkreter Syntax zu ermöglichen, muss es eine Klassendiagrammtransformationssprache geben. Diese muss neben den in [Wei12] vorgestellten Eigenschaften außerdem ermöglichen Folgendes auszudrücken:

- **LR 1.1 Attribute und Methoden:** Attribute und Methoden stehen in engem Bezug zu einander. Beispielsweise können Attribute durch Methoden gekapselt werden oder abgeleitete Attribute durch entsprechende Berechnungsmethoden ersetzt werden. Um die Ersetzung von Attributen durch Methoden bzw. Methoden durch Attribute zu erleichtern, muss die integrierte Notation eine direkte Ersetzung dieser Modellelemente durch einander erlauben.
- **LR 1.2 Interface, Enums und Klassen:** Interfaces, (abstrakte) Klassen und Enums werden häufig gegeneinander ausgetauscht, z.B. bei der Umwandlung von Attributen in Konstanten oder einer (abstrakten) Oberklasse in ein Interface. Die integrierte Notation muss für Klassendiagramme daher ermöglichen, diese Modellelemente direkt durch einander zu ersetzen anstatt das eine zu entfernen und das andere hinzuzufügen.
- **LR 1.3 Modifikatoren:** Die Modifikatoren `public`, `protected` und `private` werden häufig gegeneinander ausgetauscht, beispielsweise bei der Kapselung von Attributen durch Zugriffsmethoden, wobei die Attribute von `public` auf `private` bzw. `protected` gesetzt werden. Eine direkte Ersetzung dieser Sichtbarkeiten muss daher von der Klassendiagrammtransformationssprache unterstützt werden.
- **LR 1.4 Ändern des Assoziationstyps:** Es muss möglich sein den Typ einer Assoziation (Assoziation oder Komposition) zu ändern, ohne dass dafür die Assoziation zunächst entfernt und eine neue des anderen Typs hinzugefügt werden muss.

LR 1.5 Abstraktion von der Assoziationsrichtung: Es muss möglich sein Pattern über Assoziationen zu beschreiben, ohne dass die Richtung der Assoziation hierbei angegeben werden muss. Dadurch kann verhindert werden, dass Transformationen, bei denen die Richtung der Assoziation nicht von Interesse ist, jeweils einmal für jede Richtung (links nach rechts, rechts nach links, ohne Richtungsangabe und bidirektional) formuliert werden muss.

LR 1.6 Abstraktion von dem Assoziationstyp: Es muss möglich sein Pattern über Assoziationen zu beschreiben, ohne dass der Typ der Assoziation (Assoziation oder Komposition) hierbei angegeben werden muss. Dadurch kann verhindert werden, dass Transformationen, bei denen der Typ der Assoziation nicht von Interesse ist, jeweils für beide Typen formuliert werden müssen.

LR 2 MontiArc-Transformationsprache: Um Transformationen von Architekturdiagrammen in konkreter Syntax zu ermöglichen, muss eine MontiArc-Transformationssprache zur Verfügung stehen, mit deren Hilfe diese MontiArc-Transformationen ausgedrückt werden können. Diese muss neben den in [Wei12] vorgestellten Eigenschaften außerdem ermöglichen Folgendes auszudrücken:

LR 2.1 Abstraktion von der Portrichtung: Es muss möglich sein, Pattern über Ports zu beschreiben, ohne dass die Richtung des Ports hierbei angegeben werden muss. Dadurch kann verhindert werden, dass Transformationen, bei denen die Richtung des Ports nicht von Interesse ist, jeweils einmal für eingehende und einmal für ausgehende Ports formuliert werden müssen.

LR 2.2 Subkomponenten und innere Komponenten: Die Ersetzung von inneren Komponenten durch (externe) Subkomponenten soll einfach ausgedrückt werden können. Die MontiArc-Transformationssprache muss daher eine direkte Ersetzung von inneren Komponenten durch Subkomponenten und vice versa erlauben.

LR 3 Transformationssprachen für Basissprachen: Zu den von MontiCore bereitgestellten und häufig verwendeten Basissprachen *Literals*, *Types* und *Common* muss es Transformationssprachen bzw. Transformationssprachkomponenten geben. Diese dienen als Bausteine für DSTLs zu DSLs, die auf diesen Modellierungssprachen aufbauen.

LR 4 MontiArc-Klassendiagramm-Transformationssprache: Um eine Übersetzung von MontiArc-Modellen zu Klassendiagrammen bzw. Klassendiagrammen zu MontiArc-Modellen sowie die Koevolution von Klassendiagrammen und MontiArc-Modellen durch Transformationen in konkreter Syntax zu unterstützen, muss es eine Transformationssprache geben, mit der solche Transformationen ausgedrückt werden können.

3.3.3 Anforderungen Transformationen

Im Folgenden werden die Anforderungen an die Transformationsbibliotheken beschrieben. Diese Anforderungen leiten sich sowohl aus den Szenarien als auch aus der in Abschnitt 3.2 beschriebenen Analyse von handgeschriebenen Modelltransformationen für Klassendiagramme bzw. MontiArc-Modelle ab.

TR 1 Transformationsbibliothek für Klassendiagramme: Um die modellgetriebene Softwareentwicklung mittels Klassendiagrammen zu unterstützen, soll eine Bibliothek von Transformationen zur einfachen Wiederverwendung in verschiedenen Softwareprojekten bereitgestellt werden. Diese Bibliothek soll typische Refactorings für Klassendiagramme enthalten, um Modellierern automatische Refactorings ihrer Klassendiagramme zu ermöglichen.

TR 2 Transformationsbibliothek für Architekturdiagramme: Um die modellgetriebene Softwareentwicklung mittels Architekturdiagrammen zu unterstützen, soll eine Bibliothek von Transformationen zur einfachen Wiederverwendung in verschiedenen Softwareprojekten bereitgestellt werden. Die Bibliothek soll Transformationen zur Normalisierung von Architekturdiagrammen enthalten, die die syntaktische Komplexität der Modelle verringert und so beispielsweise als Vorbereitung auf einen Codegenerierungsschritt verwendet werden können.

TR 3 Schnittstellen: Durch geeignete Schnittstellen sollen die Transformationen der Bibliothek gekapselt und dem Modellierer zugänglich gemacht werden. Eine Schnittstelle repräsentiert jeweils die verschiedenen Verwendungsmöglichkeiten einer (komplexen) Transformation.

TR 4 Modularität: Transformationen einer Bibliothek sollen modular gestaltet sein, sodass auch neben der Verwendung von vordefinierten komplexen Transformationen auch eigene Konfigurationen der Transformationsregeln möglich sind.

Teil II

Domänenspezifische Transformationssprachen

Kapitel 4

Struktur und Operatoren der domänenspezifischen Transformationssprachen mit MontiTrans

Zur Beschreibung von Transformationen gibt es verschiedene Ansätze [MVG06, CH06, KC15]. Ein wesentlicher Unterschied besteht darin, ob eine Transformationssprache auf der abstrakten Syntax oder der konkreten Syntax von Modellierungssprachen basiert [BW07, RW11]. General Purpose Transformationssprachen (GPTLs) erlauben die Beschreibung beliebiger Transformationen auf Modellen beliebiger Modellierungssprachen. Beispiele für GPTLs sind ATL [JABK08], PROGRES [Sch91], Fujaba [FNTZ00], eMoflon [LAS14] oder Henshin [ABJ+10]. Um diese Sprachunabhängigkeit zu erreichen, werden Transformationen gegen die interne Repräsentation des Modells, d.h. die abstrakte Syntax der Modellierungssprache, formuliert. Diese Repräsentation ist für die Verarbeitung von Modellen – beispielsweise innerhalb von Editoren oder Generatoren – gedacht. Damit ist die abstrakte Syntax nicht für Modellierer oder Domänenexperten entwickelt und in der Regel vor ihnen verborgen. Dem gegenüber stehen domänenspezifische Transformationssprachen (DSTLs) [BW07, RW11, Sch07, GMP09, SVM+13]. Eine DSTL ist spezifisch für eine zugrundeliegende Modellierungssprache und verwendet das Vokabular der Modellierungssprache, d.h. die konkrete Syntax der Modellierungssprache, zur Beschreibung von Transformationen. Eine DSTL ist daher an eine Modellierungssprache gebunden und muss spezifisch für diese entwickelt werden, erlaubt im Gegenzug aber die Einbindung von Domänenexperten bei der Definition von Modelltransformationen. In [Wei12] wurde eine domänenspezifische Notation für Transformationen von Modellen der MontiCore Modellierungssprachen vorgestellt. Zusätzlich wurde ein Generierungsframework entwickelt, das es ermöglicht diese DSTLs inklusive Infrastruktur zur Anwendung der Transformationen automatisch aus den Grammatiken der Modellierungssprachen zu generieren. Diese Notation diente in dieser Arbeit als Grundlage und wurde im Rahmen dieser Arbeit verfeinert und weiterentwickelt. Die resultierende Syntax und Semantik sowie die Unterschiede zur in [Wei12] vorgestellten Notation werden im weiteren Verlauf dieses Kapitels genauer vorgestellt. Darüber hinaus ergänzt diese Arbeit die in [Wei12] vorgestellten Konzepte um eine detaillierte Erläuterung sowie eine Erklärung der möglichen Verwendungsformen.

Der Fokus in diesem Kapitel liegt auf der Erklärung der verschiedenen Operatoren der im Rahmen dieser Arbeit entwickelten DSTLs (vgl. Abbildung 4.1). Dieses Kapitel

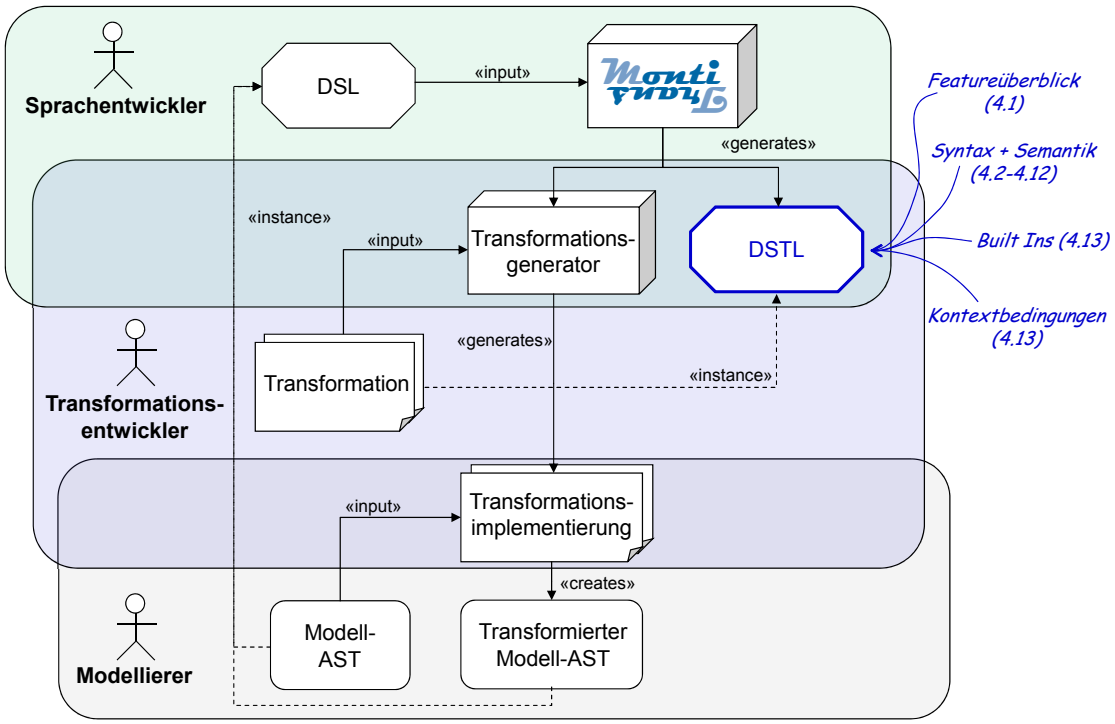


Abbildung 4.1: Übersicht über die verschiedenen Aspekte von MontiTrans mit Fokus auf den in diesem Kapitel vorgestellten Teil (Syntax und Semantik der DSTLs).

adressiert sowohl den Sprachentwickler als auch den Transformationsentwickler, da der Sprachentwickler DSTLs mit der hier vorgestellten Syntax durch MontiTrans generieren lassen kann und sie somit dem Transformationsentwickler zur Verfügung stellt. Exemplarisch wird hierzu die DSTL MATrans zur Transformation von MontiArc-Modellen verwendet, jedoch nicht detaillierter auf deren MontiArc-spezifische Syntax eingegangen. Die eigentlichen DSTLs – darunter auch MATrans – werden in Kapitel 5 vorgestellt. Guidelines zur Entwicklung von Transformationsregeln und komplexen Transformationen hingegen werden in Kapitel 9 beschrieben.

Die wichtigsten Ergebnisse dieses Kapitels sind:

- Die Beschreibung der Syntax der Operatoren und Erläuterung der Semantik der Operatoren der DSTLs
- Neue Features für die DSTLs: der Optional Operator, die Stringmanipulation, die Built-Ins, die Zuweisungen an Schemavariablen, die direkte Manipulation durch einen imperativen Anweisungsblock
- Die Vorstellung der entwickelten Kontextbedingungen zur Wohlgeformtheit von Transformationsregeln

4.1 Featureübersicht

Eine Modelltransformation besteht aus einem Pattern und (optional) einer Modifikation. Neben der für DSTLs typischen Verwendung der konkreten Syntax zur Beschreibung von Pattern bieten die DSTLs in MontiCore eine Reihe von Features und Operatoren zur Patterndefinition sowie verschiedene Möglichkeiten Modifikationen zu beschreiben. Tabelle 4.2 gibt eine Übersicht über die verschiedenen Konzepte und Operatoren, die in folgenden Abschnitten detaillierter beschrieben werden.

Tabelle 4.2: Überblick der Features der DSTLs.

Feature	Kurzbeschreibung	Teil des/der	Abschnitt
Konkrete Syntax	Wird zur Patternbeschreibung verwendet.	Pattern	4.3
Relaxation	Erlaubt es, nur relevante Teile des Modells zu beschreiben.	Pattern	4.3
Schemavariablen	Erlauben Abstraktion und Binden von Modellelementen an Variablen.	Pattern	4.4
Replacement Operator	Ermöglicht Hinzufügen, Löschen, Ersetzen und Verschieben von Modellelementen.	Modifikation	4.5
Negative Elemente	Ermöglichen Verbieten von Modellstrukturen.	Pattern	4.6
Collection Operator	Ermöglicht Finden mehrerer ähnlicher Modellstrukturen.	Pattern	4.7
Optional Operator	Ermöglicht Pattern, um optionale Strukturen zu erweitern.	Pattern	4.8
Folding Operator	Erlaubt nicht-isomorphes Matching, d.h. ein Modellelement für mehrere Patternelemente.	Pattern	4.9
Variablenzuweisung	Ermöglicht die Belegung von Schemavariablen.	Modifikation	4.10
Application Constraint	Ermöglicht Einschränkung der Anwendbarkeit.	Pattern	4.11
Direkte Manipulation	Erlaubt Modifikationen imperativ zu beschreiben (Java).	Modifikation	4.12
Finale Aktionen	Erlaubt Aktionen im Anschluss an eine Transformation zu spezifizieren.	-	4.12

4.2 Durchgängiges Beispiel

Die Operatoren der DSTLs werden an einem durchgängig verwendeten Beispiel vorgestellt. Listing 4.4 zeigt das Ausgangsmodell. Gegeben ist hier ein MontiArc-Modell, das eine Systemkomponente darstellt. Diese Systemkomponente umfasst zwei Komponenten: `Monitor` (Zeile 2-5) und `Controller` (Zeile 6-8). Die Komponente `Monitor` hat hierbei zwei Ports; einen eingehenden vom Typ `Signal` namens `trigger` (Zeile 3) und einen ausgehenden vom Typ `State` namens `state` (Zeile 4). Die Controllerkomponente hingegen hat nur einen eingehenden Port vom Typ `State` namens `state` (Zeile 7). Außerdem umfasst das System eine Subkomponente vom Typ `Monitor` namens `m`.

```
1  component System {                                          MA
2    component Monitor monitor {
3      port in Signal trigger,
4           out State state;
5    }
6    component Controller control {
7      port in State state;
8    }
9    component Monitor m;
10 }
```

Listing 4.4: Startmodell einer System-, Monitor- und Controllerkomponente.

Das Startmodell verfügt bislang über keine Konnektoren. Das bedeutet, dass die Ports nicht verbunden sind. Durch Transformationen sollen die beiden Stateports so verbunden werden, dass ein Konnektor den ausgehenden Stateport der Monitorkomponente mit dem eingehenden Stateport der Controllerkomponente verbindet. Zusätzlich ist der eingehende Triggerport Teil der Monitorkomponente, soll im resultierenden System jedoch Teil der Controllerkomponente sein. Hier muss eine Transformation dafür sorgen, dass der Port verschoben wird. Schließlich soll der Name der Controllerkomponente von `control` zu `controller` geändert werden und die überflüssige Subkomponente vom Typ `Monitor` names `m` entfernt werden. Das resultierende System in Listing 4.5 zu sehen.

4.3 Pattern in konkreter Syntax

Um das Startmodell aus Listing 4.4 in das Zielmodell zu transformieren, muss der Signalport von der Komponente `Monitor` in die Komponente `Controller` verschoben werden. Dazu muss zunächst der Teil des Modells gefunden werden, der verändert werden soll. Dazu wird ein Pattern formuliert [CH06], das diesen Teil des Modells beschreibt. Ein Pattern, um die beiden Komponenten `Monitor` und `Controller` und den Signalport im Modell zu finden, ist in Listing 4.6 zu sehen.

Dieses Pattern ist sehr ähnlich zum eigentlichen Modell, da domänenspezifische Transformationssprachen zur Beschreibung von Patterns die konkrete Syntax der zugehörigen

```
component System {                                    MA
  component Monitor monitor {
    port out State state;
  }
  component Controller controller {
    port in Signal trigger,
         in State state;
  }

  connect monitor.state -> controller.state;
}
```

Listing 4.5: Zielmodell der System-, Monitor- und Controllerkomponente.

```
component Monitor monitor {                           MATrans
  port in Signal trigger;
}
component Controller control {
}
```

Listing 4.6: Pattern zum Matchen der Monitorkomponente.

Modellierungssprache nutzen. Das heißt, sie nutzen das Vokabular, das auch zur Beschreibung der zu transformierenden Modelle verwenden wird [RW11, BW07].

Strukturell sind Patterns genauso aufgebaut wie Modelle der zugehörigen Modellierungssprache. Ein wichtiger Unterschied ist, dass der Transformationsentwickler zur Beschreibung von Pattern mehr Freiheiten hat, als es die Modellierungssprache gewährt [KMS+10, SGV13]. Dies wird auch als „Relaxation“ bezeichnet [KMS+10]. Das bedeutet, die Constraints der Sprache müssen gelockert werden, zum Beispiel muss das Auslassen von Modellelementen, die im Modell verpflichtend, jedoch im Pattern irrelevant sind, erlaubt sein.

Außerdem enthält das Pattern in Listing 4.6 die Systemkomponente – die Komponente, die die beiden Komponenten umgibt – nicht. In MontiArc entsteht diese aus dem Startsymbol der Grammatik und ist somit zwingend erforderlich. Ein Pattern sollte nicht mit dem Startsymbol der zugehörigen Modellierungssprache beginnen müssen. Zum einen ist es häufig nicht relevant, an welcher Stelle im Modell ein bestimmtes Pattern auftritt, zum anderen soll sich der Transformationsentwickler auf die Beschreibung der für die Transformation relevanten Stelle konzentrieren können. Hierzu erlauben die DSTLs – im Gengensatz beispielsweise zu Deltasprachen [HHK+13] – nur den relevanten Modellausschnitt zu beschreiben. Das bedeutet, eine Patternbeschreibung muss nicht vom Startsymbol der Modellierungssprache ausgehen. Stattdessen kann jedes Modellelement das „oberste“ Element eines Pattern sein. Dadurch wird im Vergleich zu Deltasprachen der Overhead zur Beschreibung eines Pattern deutlich reduziert, da der Weg von dem Startsymbol der Modellierungssprache bis hin zu dem zu transformierenden Modellelement durch das Pattern Matching (vgl. Abschnitt 8.3) gehandhabt wird und nicht vom

Transformationsentwickler angegeben werden muss. Die DSTLs für MontiCore erlauben innerhalb der Patternbeschreibung direkt mit dem relevanten Modellelement zu beginnen. Jedes Modellelement der Modellierungssprache kann als Ausgangspunkt für die Patternbeschreibung dienen. Für das Patternbeispiel bedeutet dies, dass zwei Komponenten unabhängig voneinander gesucht werden, die dem jeweiligen Pattern entsprechen. Eine Beziehung der Komponenten zueinander ist hierbei nicht angegeben und wird somit weder verlangt noch ausgeschlossen (vgl. Abschnitt 4.6).

Eine weitere Besonderheit, die das Pattern in Listing 4.6 verdeutlicht, ist, dass nur Elemente, die für das Pattern relevant sind, auch im Pattern angeben werden. Für das Beispiel bedeutet dies, dass insbesondere die Stateports der beiden Komponenten weggelassen wurden. Ein Pattern muss keine (irrelevanten) Details der relevanten Modellelemente spezifizieren. Modellelemente müssen nicht vollständig beschrieben werden. Das bedeutet, Eigenschaften, die das Modellelement im Eingabemodell haben kann, aber für die Transformation keine Rolle spielen, können und sollten ausgelassen werden. Wird dies nicht gemacht, muss für jede mögliche Variante des Modellelements eine eigene Transformation geschrieben werden. Die DSTLs für MontiCore bieten daher die Möglichkeit irrelevante Eigenschaften von Modellelementen auszulassen. Listing 4.6 verdeutlicht dieses Feature. Für das Pattern wurde ausschließlich die innere Komponente `Monitor` angegeben. Weder die übergeordnete Komponente `System`, noch die innerhalb der Monitorkomponente vorhandenen Ports wurden hierbei im Pattern angegeben. Diese Teile des Modells waren für die Beschreibung des Pattern irrelevant und konnten daher weggelassen werden. Es gilt, was nicht im Pattern angegeben ist, kann beliebig vorhanden sein und insbesondere wird durch das Auslassen von Modellelementen im Pattern nicht gefordert, dass an dieser Stelle keine Elemente vorkommen. Soll ein Element explizit innerhalb des Pattern verboten werden, müssen hingegen negative Elemente angegeben werden (vgl. Abschnitt 4.6).

Aus den gleichen Gründen muss erlaubt werden, dass Modellelemente, die im Modell vorhanden sein müssen, im Pattern ausgelassen werden können, beispielsweise die Richtung eines Ports. Ein Port ist zwingend entweder eingehend oder ausgehend, daher müsste der Transformationsentwickler jeweils eine Transformation für eingehende und ausgehende Ports schreiben, selbst wenn die Richtung der Ports für die Transformation irrelevant ist. Gäbe es noch weitere Alternativen, so müsste hier jeweils eine Transformation pro Alternative geschrieben werden. Ist im Beispielpattern in Listing 4.6 die Richtung des Ports nicht relevant, müsste der Transformationsentwickler in diesem Fall zwei Transformationen schreiben, um alle möglichen Richtungen des Ports abzudecken.

Das Beispiel in Listing 4.6 zeigt eine weitere Eigenschaft der Pattern der DSTLs in MontiCore: Ein Pattern kann beliebig viele Patternelemente nebeneinander beschreiben. Im Beispiel sind dies die Komponenten `Monitor` und `Controller`. Hierdurch wird keine Angabe dazu gemacht, wo im Modell sie vorkommen. Insbesondere könnte auch eine Komponente innerhalb einer anderen vorkommen. Dies ist ein weiteres für Pattern wichtiges Konzept. Ein Pattern kann – anders als in der zugehörigen Modellierungssprache – mehr als ein Element beschreiben, ohne dass diese ein gemeinsames Vaterelement haben müssen. Gibt es diese Möglichkeit nicht, so muss der Transformationsentwickler immer

ein gemeinsames Vaterelement aller im Pattern beschriebenen Elemente spezifizieren. Schlimmstenfalls muss das Pattern beim Startsymbol der Modellierungssprache ansetzen und alle Zwischenelemente bis hin zu den für das Pattern relevanten Patternelementen beschreiben. Die DSTLs in MontiCore erlauben daher beliebig viele Patternelemente, deren entsprechende Modellelemente im Modell auf gleicher oder verschiedener Ebene vorkommen, innerhalb des Pattern nebeneinander zu beschreiben. Dadurch wird keine Beziehung der Modellelemente zueinander vorgegeben.

4.4 Schemavariablen

Das Pattern in Listing 4.6 ist genau für das Modell in Listing 4.4 passend. Wird das Eingabemodell nun leicht verändert, beispielsweise wie in Listing 4.7, indem der Signalport nicht länger `trigger` sondern `trig` heißt, dann wird das Pattern aus Listing 4.6 nicht länger gefunden. Häufig wird eine Transformation nicht für ein spezielles Modell formuliert, sondern soll vielmehr auf eine Vielzahl von Modellen anwendbar sein [CMIC15, SMM+12, ISH08]. Neben den in Abschnitt 4.3 vorgestellten Möglichkeiten zur Verallgemeinerung (z.B. Auslassen von nicht relevanten Elementen) muss es daher weitere Möglichkeiten geben, ein Pattern zu verallgemeinern.

MA

```
1 component System {
2   component Monitor monitor {
3     port in Signal trig,
4          out State state;
5 }
6   component Controller control {
7     port in State state;
8   }
9 }
```

Listing 4.7: Startmodell mit System-, Monitor- und Controllerkomponente.

Beispiele für allgemeinere Transformationen sind das Einziehen von Design Pattern [FR11, Fow06, GHJV95] oder die Refaktorisierung von Modellen zur Qualitätsverbesserung [Fow99]. Eine domänenspezifische Transformationssprache muss daher neben der Möglichkeit Modellelemente in Pattern auszulassen auch die Möglichkeit haben von speziellen Eigenschaften – wie Namen – zu abstrahieren bzw. diese nur im Bezug auf andere Elemente zu beschreiben (z.B. „Komponente X hat den gleichen Namen wie Y plus das Präfix Z“). Die DSTLs bieten daher das Konzept der Schemavariablen. Der Begriff wurde bereits in [Wei12] verwendet und stammt aus [BBB+85, BEH+87]. Schemavariablen dienen mehreren Zwecken. Sie erlauben die Abstraktion von beispielsweise Namen (vgl. Abschnitt 4.4.1) oder auch anderen Modellelementen (Abschnitt 4.4.2), ermöglichen das gefundene Modellelement an eine Schemavariable zu binden, um beispielsweise einen Application Constraint über dieses Element zu formulieren (vgl. Abschnitt 4.11), es zu verschieben (vgl. Abschnitt 4.5) oder als Eingabe oder Ausgabe einer Transformation zu verwenden (vgl. Abschnitt 9.4).

4.4.1 Schemavariablen für Namen

Eine Erweiterung des Beispielpattern aus Listing 4.6 zeigt Listing 4.8. Für den Signalport wird hier als Name nicht `trigger` vorgegeben, sondern eine Schemavariable `$signal` verwendet. Dadurch kann ein Match für das Pattern sowohl im Modell in Listing 4.4 als auch im Modell in Listing 4.7 gefunden werden.

```
component Monitor monitor {
  port in Signal $signal;
}
component Controller control {
}
```
MATrans

Listing 4.8: Pattern inklusive einer Schemavariable für den Namen des Signalsports aus Listing 4.4.

Die einfachste Art der Schemavariablen ist die für Namen innerhalb von Modellen. Eine Schemavariable besteht auf einem mit einem $-Zeichen beginnenden Namen (vgl. `$signal` in Listing 4.8). Schemavariablen für Namen sind innerhalb eines Pattern überall dort erlaubt, wo im Modell ein Name stehen würde, also beispielsweise der Name eines Ports oder einer Komponente. Anstelle eines konkreten Namens kann hier eine Schemavariable im Pattern verwendet werden. Durch die Verwendung einer Schemavariable wird keine Aussage über den Namen getroffen, der an dieser Stelle im Modell auftritt.

```
component Monitor monitor {
  port in Signal $signal,
       out $type $portName;
}
component Controller control {
  port in $type $portName;
}
```
MATrans

Listing 4.9: Pattern inklusive Schemavariablen, die eine namentliche Übereinstimmung der Portnamen und -typen fordern.

Eine Erweiterung des Beispiels in Listing 4.8 zeigt Listing 4.9. Neben den bisher beschriebenen Komponenten und dem Signalport werden hier zwei weitere Ports beschrieben, wobei für deren Typ und Namen die Schemavariablen `$type` bzw. `$portName` verwendet wurde. Es fällt auf, dass die Schemavariablen für den Typ und den Namen dieser beiden Ports übereinstimmen. Dadurch wird sichergestellt, dass auch im Modell Typ und Name der Ports übereinstimmen müssen.

Textuelle Modelle bilden Baumstrukturen. Um dennoch Beziehungen zwischen nicht verbundenen Modellelemente herzustellen, können Namen als Referenz auf andere Modellelemente verwendet werden [NTVW15]. In MontiArc verbindet ein Konnektor zwei Ports. Hierbei wird der entsprechende Port jedoch nicht komplett wiederholt oder als Teil des Konnektors modelliert, stattdessen verwendet der Konnektor die Namen der Ports.

Das gleiche Prinzip wird innerhalb eines Patterns auch für Schemavariablen verwendet. Schemavariablen der DSTLs können dazu genutzt werden, Beziehungen zwischen Modellelementen, die gefunden werden sollen, auszudrücken. Es gilt, wird die gleiche Schemavariable innerhalb eines Pattern an mehreren Stellen verwendet, fordert das Pattern, dass im Modell die Namen an diesen Stellen übereinstimmen.

Dieser Mechanismus vereinfacht die Beschreibung von Pattern insofern, als dass es nicht nötig ist zum Ausdrücken von Namensübereinstimmung ein zusätzliches Konstrukt zu verwenden. Mit diesem Vorteil geht der Nachteil einher, dass an allen Stellen, an denen Schemavariablen verwendet werden, unterschiedliche Schemavariablen gewählt werden müssen. Um diesem Nachteil entgegen zu wirken, bieten die Transformationssprachen anonyme Schemavariablen.

MATrans

```
1 component Monitor monitor {
2   port in Signal $_,
3     out $type $portName;
4 }
5 component Controller control {
6   port in $type $portName;
7 }
```

Listing 4.10: Pattern mit anonymer Schemavariable für den Namen des Signalports.

Listing 4.10 zeigt eine Erweiterung des Beispiels in Listing 4.9, bei dem für den Namen des Signalports eine anonyme Schemavariable verwendet wird. Der Name dieses Ports ist für die Transformation irrelevant und wird nicht weiter benötigt. Dadurch kann eine Schemavariable verwendet werden, auf die im restlichen Pattern kein Bezug genommen werden kann. Anonyme Schemavariablen bestehen aus einem `$`-Zeichen und einem Unterstrich (`$_`). Diese Schemavariablen können an allen Stellen verwendet werden, an denen von dem Namen abstrahiert werden soll und die Schemavariable nicht weiter benötigt wird. Wird diese Form der Schemavariablen an zwei oder mehr Stellen verwendet, ist keine Beziehung zwischen den beiden Stellen gefordert, aber auch nicht verboten.

Im Gegensatz zur Forderung, dass Namen bei gleichen Schemavariablen übereinstimmen, fordern unterschiedlich gewählte Schemavariablen nicht automatisch, dass die entsprechenden Namen im Modell verschieden sein müssen. Soll ein Pattern dies explizit ausdrücken, kann hierfür ein Constraint formuliert werden, der entsprechend definiert, dass die Namen verschieden sind (vgl. Abschnitt 4.11). Des Weiteren können Schemavariablen für Namen als Platzhalter dienen, sofern sie Teil eines hinzugefügten oder veränderten Modellelements sind und somit nicht durch das Pattern Matching belegt werden. Um diesen Schemavariablen einen Wert zuzuweisen, bieten die DSTLs in MontiCore den Zuweisungsblock (vgl. Abschnitt 4.10).

Schemavariablen innerhalb des Pattern werden während des Pattern Matching mit dem zugehörigen Namen des Modellelements belegt. Dies erlaubt die Formulierung von Application Constraints über Namen (vgl. Abschnitt 4.11). Außerdem können Schemavariablen als Ein- bzw. Ausgabe einer Transformation genutzt werden (vgl. Abschnitt 9.4) und somit Ergebnisse der einen Transformation der nächsten als Eingabe dienen.

4.4.2 Abstraktion von Modellelementen durch Schemavariablen

Im Startmodell gibt es eine Subkomponente. Um vom Start- zum Zielmodelle zu kommen, soll diese entfernt werden. Dazu muss sie Teil des Pattern werden, um so anschließend durch den später erläuterten Replacement Operator entfernt werden zu können. Da es im Modell nur eine Subkomponente gibt, ist deren konkrete Struktur für ihre Entfernung nicht relevant und es kann stattdessen eine Schemavariable für die komplette Subkomponente verwendet werden. Listing 4.11 zeigt eine Erweiterung des Pattern aus Listing 4.9. Neu hinzugekommen ist die Schemavariable `$S` in Zeile 8 vom Typ `Subcomponent`, wodurch zusätzlich eine beliebige Subkomponente innerhalb des Modells gesucht wird.

```
1 component Monitor monitor {                                    MATrans
2   port in Signal $signal,
3        out $type $portName;
4 }
5 component Controller control {
6   port in $type $portName;
7 }
8 Subcomponent $S
```

Listing 4.11: Pattern für Komponenten aus Listing 4.4.

Neben der Möglichkeit Namen durch Schemavariablen zu ersetzen, bieten die DSTLs auch die Möglichkeit andere Modellelemente und ganze Teilmodelle durch Schemavariablen zu ersetzen. In MontiArc können dies die durch Nichtterminale definierten Modellelemente sein, wie beispielsweise Komponenten, Ports oder Konnektoren. Hierzu werden ebenfalls Schemavariablen verwendet, welche zusätzlich durch das Nichtterminal der Modellierungssprache, das sie darstellen, getypt sind. Im Fall von Komponenten beispielsweise `Component`. Hierbei wird die Beschreibung eines Patternelements in konkreter Syntax komplett durch die Angabe einer getypten Schemavariable ersetzt, wodurch keine Aussage über die Struktur des Elements innerhalb des Modells gemacht wird. Im Gegensatz zu der in [Wei12] vorgestellten Form werden diese zur besseren Integration in die Transformation nicht durch ein Semikolon beendet.

Schemavariablen innerhalb eines Pattern werden während des Patten Matching mit dem zugehörigen Modellelement belegt. Hierdurch können mithilfe des Ersetzungsoperators Verschiebungen realisiert (s. Abschnitt 4.5) und Constraints über das Element formuliert werden (s. Abschnitt 4.11). Außerdem kann eine solche Schemavariable auch als Ein- bzw. Ausgabe einer Transformation genutzt werden (s. Abschnitt 9.4).

4.4.3 Binden von (Teil-)Pattern an Schemavariablen

Um das Startmodell in das Zielmodell zu transformieren muss der Signalport von der Komponente `Monitor` in die Komponente `Controller` verschoben werden. Um ihn zu verschieben, wird der später erklärte Replacement Operator benutzt. Damit der Port

verschoben werden kann, statt gelöscht und neu erstellt zu werden, muss das Element an eine Schemavariable gebunden werden. Das Pattern in Listing 4.12 erweitert das Pattern aus Listing 4.11 um diese Schemavariable. In Zeile zwei wurde der eingehende Signalport in doppelte eckige Klammern eingefasst und eine Schemavariable `$P` davor ergänzt.

```
1 component Monitor monitor {                                    MATrans
2   port $P [[in Signal $_]],
3              out $type $portName;
4 }
5 component Controller control {
6   port in $type $portName;
7 }
8 Subcomponent $S
```

Listing 4.12: Pattern für Komponenten aus Listing 4.4.

Diese Verwendung von Schemavariablen ist eine Kombination aus einer Schemavariable und einem Pattern in konkreter Syntax, um die Struktur des Elements zu beschreiben. Dadurch lassen sich die Vorteile beider zuvor erläuterten Möglichkeiten kombinieren. Zum einen kann eine Struktur für das zu findende Modellelement angegeben werden, zum anderen kann eine Schemavariable – wie in Abschnitt 4.4.2 beschrieben – verwendet werden. Die Syntax hierfür sieht wie folgt aus:

```
ElementType? SchemaVar "[[" Element "]]"                    (4.1)
```

wobei der Typ der Schemavariablen optional ist. Die Verwendung von Schemavariablen für Patternelemente in konkreter Syntax funktioniert, indem das Pattern in doppelte eckige Klammern eingeschlossen wird und eine Schemavariable davor geschrieben wird.

Anders als bei der in Abschnitt 4.4.2 vorgestellten Schemavariablenverwendung und als Erweiterung zu der in [Wei12] vorgestellten Form von Schemavariablen kann hier auf eine explizite Typisierung der Schemavariablen verzichtet werden. Der Typ kann aus dem Pattern innerhalb der Klammern inferiert werden. Die explizite Typangabe hingegen kann verwendet werden, wenn beispielsweise eine Oberklasse (z.B. abstraktes Nichtterminal) verwendet werden soll.

4.5 Modifikation

Für Transformationen gibt es unterschiedliche Möglichkeiten, um Änderungen auszudrücken. Für Graphtransformationen ist es üblich eine Transformation mittels linker und rechter Regelseite anzugeben [Nag79]. Hierbei entspricht die linke Regelseite dem Pattern, zu dem ein Match im Modell gefunden werden soll, und die rechte Regelseite stellt den gleichen Modellausschnitt nach Anwendung der Transformation dar. Die Modifikation, die durch die Transformation vorgenommen wird, ergibt sich aus dem Unterschied zwischen der linken und der rechten Regelseite. Ein Nachteil dieser Art der Spezifikation ist, dass die rechte Regelseite alle unveränderten Elemente wiederholen

muss, da ansonsten von einer Löschung dieser Elemente ausgegangen wird [RW11]. Um diesen Nachteil zu vermeiden, verwenden die DSTLs in MontiCore eine integrierte Notation. Hierbei wird die linke und rechte Regelseite kombiniert und Veränderungen direkt am zu verändernden Element angegeben.

LHS

```
1  // Linke Regelseite
2  component System {
3    component Monitor monitor {
4      port in Signal trigger,
5           out State state;
6    }
7    component Controller control {
8
9       port in State state;
10   }
11
12   component Monitor m;
13
14
15 }
```

RHS

```
// Rechte Regelseite
component System {
  component Monitor monitor {

   port out State state;
 }
  component Controller controller {
    port in Signal trigger,
         in State state;
 }


  connect monitor.state ->
          controller.state;
}
```

Listing 4.13: Transformation von Start- zu Zielmodell bei separierten Regelseiten.

Bei einer Graphtransformation mit separierter linke und rechter Regelseite sieht eine Transformationsregel, die Beispieltransformation wie in Abbildung 4.13 dargestellt aus. Hierbei zeigt der linke Teil die linke Regelseite (das Pattern). Dargestellt ist hier die Systemkomponente mit ihren inneren Komponenten sowie deren Ports (vgl. Listing 4.4). Die rechte Seite zeigt die gleichen Komponenten nach der Transformation (vgl. Listing 4.5). Aus dem Unterschied zwischen der linken und rechten Seite ergibt sich eine Transformationsregel, die die Monitorkomponente `m` entfernt, den Signalport verschiebt, die Controllerkomponente umbenennt und den Konnektor hinzufügt.

Das Beispiel verdeutlicht, dass in diesem Fall alles, was sich nicht verändern soll, auf der rechten Regelseite wiederholt werden muss. Im Beispiel etwa die Komponenten. Außerdem ist es für den Transformationsentwickler bei komplexeren Transformationsregeln schnell unübersichtlich, welche Änderungen eine Transformationsregel beschreibt, da er den Unterschied selbst erkennen muss. Die DSTLs bieten hingegen einen *Replacement Operator* an, um Änderungen anzugeben. Dieser bietet die Möglichkeit Änderungen an den innerhalb des Pattern spezifizierten Elementen vorzunehmen, sowie neue Elemente hinzuzufügen, vorhandene Elemente zu entfernen oder zu verschieben. In den folgenden Abschnitten werden die verschiedenen Möglichkeiten des Replacement Operators an kleinen Patternausschnitten und schließlich in Kombination anhand des durchgängigen Beispiels vorgestellt. Die äquivalente Transformationsregel zu der Transformationsregel in Abbildung 4.13 ist in Listing 4.18 dargestellt.

4.5.1 Ersetzen von Modellelementen

Bei der Transformation des Startmodells zum Zielmodell soll der Name der Controllerkomponente von `control` auf `controller` geändert werden. Die Transformationsregel

in Listing 4.14 beschreibt diese Veränderung des Modells (Zeile 1). In doppelten eckigen Klammern wird der Replacment Operator `:-` verwendet, um `control` durch `controller` zu ersetzen. Hierbei ist `control` Teil des Pattern und muss somit im Modell gefunden werden. Ist dies der Fall wird `control` durch `controller` ersetzt.

```
1 component Controller [[control :- controller]] {                    MATrans
2 }
```

Listing 4.14: Ersetzen von Modellelementen.

Der Replacement Operator ermöglicht das Ersetzen von Modellelementen. Dazu wird der Operator an der Stelle im Pattern verwendet, an der das Modellelement ersetzt werden soll. Um ein Modellelement zu ersetzen, wird auf der linken Seite des Operators das im Modell zu findende Modellelement beschrieben. Auf der rechten Seite des Operators wird ein Modellelement beschrieben, das das auf der linken Seite ersetzt.

4.5.2 Hinzufügen von Modellelementen

Um aus dem Startmodell das Zielmodell durch Transformationen zu erhalten, muss dem Modell ein Konnektor zwischen den Stateports hinzugefügt werden. Die Transformationsregel in Listing 4.15 fügt der Systemkomponente einen Konnektor hinzu (Zeile 2). Dazu wird der Replacement Operator verwendet. In den doppelten eckigen Klammern ist die linke Seite des Operators leer gelassen, wodurch kein zu ersetzendes Element definiert wurde. Die rechte Seite beschreibt den Konnektor, dadurch wird dieser hinzugefügt. Der Konnektor ist nicht Teil des Pattern. Findet die Transformationsregel im Modell die Komponente `System`, fügt sie dieser Komponente den beschriebenen Konnektor hinzu.

```
1 component System {                                                  MATrans
2   [[ :- connect monitor.state ->  controller.state;  ]]
3 }
```

Listing 4.15: Hinzufügen von Modellelementen.

Der Replacement Operator ermöglicht auch das Hinzufügen von Modellelementen. Dazu wird der Operator an der Stelle im Pattern verwendet, an der das Modellelement hinzugefügt werden soll. Um ein Modellelement hinzuzufügen, wird die linke Seite des Operators leer gelassen und das hinzuzufügende Modellelement auf der rechten Seite des Operators beschrieben.

4.5.3 Entfernen von Modellelementen

In der Beispieltransformation von Start- zu Zielmodell soll die Subkomponente aus dem Modell entfernt werden. Die Transformationsregel in Listing 4.16 beschreibt diese Transformation. Das Pattern beschreibt die Systemkomponente. Außerdem wird der Replacement Operator dazu genutzt die Subkomponente aus dieser Komponente zu entfernen

(Zeile 2). In den doppelten eckigen Klammern ist die Subkomponente beschrieben. Damit ist die Subkomponente ebenfalls Teil des Pattern und ein Match für diese Subkomponente muss im Modell gefunden werden. Anschließend wird sie durch die rechte (leere) Seite des Operators ersetzt. Dadurch wird die Subkomponente aus dem Modell entfernt.

```
1 component System {                                          MATrans
2   [[ Subcomponent $S :- ]]
3 }
```

Listing 4.16: Entfernen von Modellelementen.

Der Replacement Operator ermöglicht auch das Entfernen von Modellelementen. Dazu wird der Operator an der Stelle im Pattern verwendet, an der das Modellelement gefunden und entfernt werden soll. Um ein Modellelement zu entfernen, wird die rechte Seite des Operators leer gelassen und das zu entfernende Modellelement auf der linken Seite des Operators beschrieben. Damit ist es Teil des Pattern und ein entsprechender Match muss im Modell gefunden werden, damit die Transformationsregel anwendbar ist. Wird ein Match gefunden, wird dieser durch die rechte Seite des Operators ersetzt. In diesem Fall ist die rechte Seite leer, wodurch das Element entfernt wird. Listing 4.16 zeigt ein Beispiel für die Verwendung des Operators zum Entfernen einer Subkomponente aus einer Komponente.

4.5.4 Verschieben von Modellelementen

In der Beispieltransformation soll der Signalport von der Komponente `Monitor` in die Komponente `Controller` verschoben werden. Die Transformationsregel in Listing 4.17 realisiert dieser Verschiebung. Dazu wird der Replacement Operator an zwei Stellen eingesetzt. Zum einen entfernt der Operator in Zeile 2 den eingehenden Signalport aus der Monitorkomponente, zum anderen fügt er diesen Port in Zeile 6 der Controllerkomponente hinzu. Damit es sich um eine Verschiebung und nicht um ein Entfernen des alten und Hinzufügen eines neuen Ports handelt, wurde der Port an die Schemavariable `$P` gebunden (vgl. Zeile 2 und 6).

```
1 component Monitor monitor {                                 MATrans
2   port[[  $P [[in Signal $_]] :- ]];
3 }
4
5 component Controller control {
6   port [[ :- $P [[in Signal $_]] ]];
7 }
```

Listing 4.17: Transformation zum Verschieben von Elementen.

Der Replacement Operator ermöglicht das Verschieben von Modellelementen, wenn diese an Schemavariablen gebunden werden. Dazu wird der Operator an der Stelle im Pattern verwendet, an der das Modellelement gefunden und entfernt werden soll, sowie

```
1  component System {                                                MATrans
2    component Monitor monitor {
3      port[[  $P [[in Signal $_]] :- ]],
4                  out $type $portName;
5    }
6    component Controller [[control :- controller]] {
7      port in $type $portName,
8           [[ :- Port $P ]];
9    }
10   [[ Subcomponent $S :- ]]
11   [[ :- connect monitor.$portName -> controller.$portName ]]
12 }
```

Listing 4.18: Transformationsregel mit Replacement Operator.

an der Stelle, an der das Element eingefügt werden soll. Hierbei wird das Element an der Stelle, an der das Element entfernt werden soll, an eine Schemavariable gebunden. Diese Schemavariable wird an der Stelle, an die das Element verschoben werden soll, ebenfalls verwendet. Dies legt fest, dass das bereits gefundene Element hier eingefügt wird. Anstelle der Wiederholung der gewünschten Syntax kann in Zeile 6 auch die Schemavariablenform ohne Angabe konkreter Syntax gewählt werden (`Port $P`).

4.5.5 Modifikation mittels Replacement Operator

In den vorhergehenden Abschnitten wurde gezeigt wie der Replacement Operator zum Ersetzen, Hinzufügen, Entfernen und Verschieben von Modellelementen verwendet werden kann. Die Beispieltransformation verschiebt den Signalport der Monitorkomponente in die Controllerkomponente, benennt die Controllerkomponente von `control` zu `controller` um, fügt einen Konnektor zwischen den Stateports hinzu und entfernt die Subkomponente aus dem Modell. Die Transformationsregel in Listing 4.18 realisiert diese Transformation. Die Transformationsregel beschreibt die Systemkomponente (Zeile 1-12). Innerhalb dieser Komponente ist eine Monitorkomponente (Zeile 2-5) beschrieben, deren Signalport an die Schemavariable `$P` gebunden und aus dieser Komponente entfernt wird (Zeile 3). Zusätzlich wird der Stateport gematcht, wobei hier für den Typ des Ports die Schemavariable `$type` und für den Namen die Schemavariable `$portName` verwendet wird (Zeile 4). In Zeile 6 bis 9 wird die Controllerkomponente beschrieben. Der Stateport wird auch hier unter Verwendung der Schemavariablen `$type` und `$portName` beschrieben. Dadurch wird die Übereinstimmung des Typs und des Namen sichergestellt. Außerdem wird der Controllerkomponente in Zeile 8 der aus der Monitorkomponente entfernte Signalport hinzugefügt. In Zeile 10 wird die Subkomponente entfernt und in Zeile 11 der Konnektor zwischen den Stateports hinzugefügt. Hierbei wird für den Konnektor auf die Schemavariable der Namen der Ports zurückgegriffen.

Die allgemeine Syntax des Operators ist die folgende:

```
"[[" Element? ":-" Element? "]]"
```
(4.2)

Der Replacement Operator besteht aus dem Zeichen `:-` und zwei optionalen Modellelementen links und rechts von diesem Zeichen. Eingeschlossen wird dies in doppelte eckige Klammern. Das Element links des `:-` Zeichens ist Teil des Pattern, wohingegen das Modellelement auf der rechten Seite des Zeichens Teil der rechten Regelseite ist.

4.6 Negative Elemente: Negative Application Condition

Bei der Transformation des Startmodell in das Zielmodel wurde davon ausgegangen, dass der Konnektor zwischen den beiden Stateports im Modell nicht existiert. Durch die Transformation in Listing 4.18 wird diese Annahme jedoch nicht überprüft. Das heißt, falls das Modell diesen Konnektor bereits enthält, würde die Transformation trotzdem angewendet. Dadurch entsteht ein invalides Modell, das den gleichen Konnektor doppelt enthält. Um eine solche Annahme innerhalb der Transformation prüfbar zu machen, bieten die DSTLs negative Elemente. Diese ermöglichen es, Modellelemente zu beschrieben, die im Modell nicht vorhanden sein dürfen, damit die Transformation anwendbar ist. Listing 4.19 zeigt eine Erweiterung des Pattern aus Listing 4.18. In Zeile 11 wird ein negatives Element verwendet, um auszudrücken, dass der einzufügende Konnektor im Modell noch nicht vorhanden sein darf.

```
1  component System {                                                    MATrans
2    component Monitor monitor {
3      port[[  $P [[in Signal $_]] :- ]],
4                  out $type $portName;
5    }
6    component Controller [[control :- controller]] {
7      port in $type $portName,
8           [[ :- Port $P ]];
9    }
10   [[ Subcomponent $S :- ]]
11   not [[ connect monitor.$portName -> controller.$portName ]]
12   [[ :- connect monitor.$portName -> controller.$portName ]]
13 }
```

Listing 4.19: Transformationsregel mit negativem Element.

Ein Pattern spezifiziert grundsätzlich nur den Teil des Modells, der für die Transformation bzw. das Finden der richtigen Stelle im Modell relevant ist. Das heißt, innerhalb eines Pattern wird jedes Element, das weder verändert wird noch dessen Anwesenheit für die Anwendbarkeit der Transformationsregel entscheidend ist, weggelassen. Das Auslassen von Elementen trifft keine Aussage darüber, ob diese im Modell vorkommen dürfen und insbesondere werden hierdurch keine Elemente verboten. Für viele Transformationen ist es relevant, bestimmte Elemente innerhalb des Pattern zu verbieten, weil die Transformation genau diese Elemente hinzufügen soll. Die DSTLs in MontiCore bieten hierfür das Konzept der Negative Application Conditions [HHT96] in Form von *negativen Elementen*. Hierdurch lassen sich Modellelemente angeben, die „negiert“ sind. Das heißt,

an der angegebenen Stelle im Modell darf es kein Element geben, das dem negativen Element entsprechen würde. Andernfalls ist die Transformationsregel nicht anwendbar, da kein Match für das Pattern gefunden wird. Die Syntax zur Spezifikation negativer Elemente ist die folgende:

```
"not" "[[" Element "]]"
```
(4.3)

Eine Transformationsregel ist nur anwendbar, wenn

1. für alle einfachen, nicht speziell markierten sowie auf der linken Seite von Ersetzungen vorkommenden Patternelemente, Modellelemente gefunden werden und
2. es keinen Match für die negativen Elemente der Transformationsregel gibt.

Negative Elemente dürfen – wie in [Wei12] – nicht verschachtelt werden. Das bedeutet, innerhalb von negativen Elementen dürfen keine negativen Elemente vorkommen (vgl. Abschnitt 4.14). Dadurch wird die Mächtigkeit des Operators zwar eingeschränkt, diese Einschränkung wird durch die Möglichkeit beliebige Constraints anzugeben aufgewogen. Als Erweiterung zu der in [Wei12] vorgestellten Version negativer Elemente kann ein Pattern mit der in dieser Arbeit entwickelten Version beliebig viele negative Elemente enthalten. Werden mehrere negative Elemente angegeben, werden diese Elemente gemeinsam verboten. Das heißt, wenn ein Teil der negativen Elemente im Modell enthalten ist, es aber mindestens ein negatives Element ohne Match gibt, kann die Transformation angewendet werden. Des Weiteren können anders als in [Wei12] nicht nur einzelne Modellelemente sondern ganze Modellteile verboten werden. In MontiArc können beispielsweise Komponenten inklusive Substrukturen wie Subkomponenten oder Ports verboten werden. Auch hier gilt, dass eine Transformation nur dann nicht anwendbar ist, wenn alle Teile des negierten Modellteils vorhanden sind.

MATrans

```
1  component $_ {
2    not[[
3      component Controller {
4        port out Integer $_;
5      }
6    ]]
7
8    not[[
9      component Monitor {
10     }
11   ]]
12 }
```

Listing 4.20: Pattern mit negativem Element.

Listing 4.20 beschreibt ein Pattern, bei dem eine Komponente gesucht wird, die keine innere Komponente `Monitor` und keine innere Komponente mit ausgehendem Port vom Typ `Integer` hat. Das Beispiel demonstriert sowohl die Möglichkeit mehrere negative Elemente anzugeben als auch die Möglichkeit negative Elemente mit Subelementen

(innere Komponente und ihr Port) anzugeben. Für das hier angegebene Pattern wäre also eine Komponente mit innerer Komponente `Monitor` ein valides Match, solange es keine Komponente Controller gibt, die einen ausgehenden Port vom Typ `Integer` hat. Genauso wäre die Transformation anwendbar, wenn es eine Komponente mit zwei Subkomponenten `Monitor` und `Controller` gibt, wobei die Controller Komponente keinen ausgehenden Port vom Typ `Integer` hat.

4.7 Collection Operator: Mengen von Elementen

Die Transformation von Startmodell zum Zielmodell geht davon aus, dass es genau eine zu entfernende Subkomponente gibt. Ändert sich das Modell auf das die Transformation angewendet werden soll, sodass es nun zwei oder mehr Subkomponenten gibt, ist das Zielmodell nicht länger das Ergebnis der Transformation. Sollen alle Subkomponenten der Systemkomponente entfernt werden, könnten entsprechend viele Subkomponenten durch jeweils einen Replacement Operator entfernt werden. Die Transformation hängt dadurch stark von der Anzahl der Subkomponenten ab. Um dies zu vermeiden und alle Subkomponenten zu finden (und zu entfernen), bieten die DSTLs einen eigenen Operator den *Collection Operator*.

Listing 4.21 zeigt eine Erweiterung der Transformationsregel aus Listing 4.19. In Zeile 10 wurde der Collection Operator auf die Subkomponente angewendet. Dadurch wird hier nicht mehr eine einzelne Subkomponente beschrieben, sondern eine Menge von Subkomponenten. Weiterhin wird hier der Replacement Operator verwendet, um diese Menge von Subkomponenten zu entfernen.

```
1  component System {                                                    MATrans
2    component Monitor monitor {
3     port[[  $P [[in Signal $_]] :- ]],
4                 out $type $portName;
5    }
6    component Controller [[control :- controller]] {
7      port in $type $portName,
8           [[ :- Port $P ]];
9    }
10   [[ list [[Subcomponent $S]] :- ]]
11   not [[ connect monitor.$portName -> controller.$portName ]]
12   [[ :- connect monitor.$portName -> controller.$portName ]]
13 }
```

Listing 4.21: Pattern für Komponenten aus Listing 4.4.

Die bisher vorgestellten Operatoren ermöglichen es Pattern mit Hilfe der konkreten Syntax zu beschreiben, Schemavariablen zu verwenden und Elemente innerhalb des Modells zu verbieten. In der Praxis zeigt sich, dass diese Simplizität der (Graph-)Transformationsansätze zwar Grundlage für deren Erfolg ist, der Mangel an High-Level-Konstrukten jedoch die Benutzerfreundlichkeit für Transformationsentwickler ein-

schränkt [GKMP13]. Mengen ähnlicher Modellteile (Subgraphen) zu finden ist ein wiederkehrendes Problem der Transformationsentwicklung. PROGRES bietet die Möglichkeit *set nodes* anzugeben [SWZ95] während in Fujaba *multi objects* angegeben werden können [GZ06]. In beiden Fällen werden jedoch lediglich einzelne Knoten im Graphen als mengenwertig markiert. Im Gegensatz hierzu erlaubt der hier vorgestellte Operator Mengen von Teilgraphen und nicht nur einzelnen Knoten anzugeben, da dies in der Regel zu restriktiv ist. Andere Ansätze hingegen bieten Collection Operatoren, die es explizit erlauben, innerhalb eines Pattern eine Menge von Teilgraphen zu beschreiben [GKMP13, FT08, MH08, BNN+07, Ren06, HJvE06, dLETE04]. Anders als bei diesen Ansätzen liegt der Fokus für den in diesem Abschnitt vorgestellten Operator darauf, sowohl die Möglichkeit der Spezifikation von Mengen von Teilgraphen zu bieten als auch diese Spezifikation mittels der konkreten Syntax der Modellierungssprache anzubieten, um Domänenexperten den Umgang mit diesem Operator möglichst zu erleichtern. Die DSTLs bieten daher einen *Collection Operator* der folgenden Form an:

```
"list" "[[" Element "]]"
```
(4.4)

Eingeleitet durch das Keyword `list` und eingefasst in doppelte eckige Klammen kann die konkrete Syntax der Modellierungssprache verwendet werden, um wie in Abschnitt 4.3 beschrieben einen Modellausschnitt zu beschreiben.

Der Collection Operator funktioniert hierbei so, dass mindestens ein Element für das innerhalb der Klammern angegebene Pattern gefunden werden muss, damit die Transformation anwendbar ist. Eine Transformationsregel ist daher nur anwendbar, wenn

1. für alle einfachen, nicht speziell markierten sowie auf der linken Seite von Ersetzungen vorkommenden Patternelemente, Modellelemente gefunden werden,
2. es keinen Match für die negativen Elemente der Transformationsregel gibt und
3. für alle Collection Operatoren mindestens ein Match gefunden wird.

Dadurch wird durch den Collection Operator ein Pattern grundsätzlich vergrößert, jedoch niemals verkleinert. Das bedeutet, dass an den Stellen, an denen der Operator verwendet wird, im Gegensatz zum einfachen Pattern jetzt ein oder mehrere Elemente gefunden werden. Ein Match für ein Pattern mit Collection Operator wird nur erfolgreich gefunden, wenn auch ein Match für das Pattern ohne den Operator (jedoch inklusive des Teils innerhalb der Klammern des Operators) erfolgreich gefunden würde. Um ein Match für das Pattern auch zu finden, wenn Teile des Pattern fehlen, d.h. das Finden eines Matches für Teile des Pattern – wie beispielsweise das Pattern innerhalb des Collection Operators – kann hierfür der Optional Operator verwendet werden (s. Abschnitt 4.8). Verwandte Konzepte sind Rekursion [GdL07], Iteration oder der Sternoperator [LLP07] für hierarchische oder rekursive Strukturen. Das Finden von Mengen von ähnlichen Modellteilen können beide Ansätze jedoch nicht lösen.

Ein weiteres Beispiel für den Collection Operator ist in Listing 4.22 zusehen. Hier wurde der Collection Operator neben der Systemkomponente angegebenen (Zeile 3). Für

```
1 component System {                                                MATrans
2 }
3 list [[SubComponent $Subs]]
```

Listing 4.22: Collection Operator für alle Subkomponenten.

die Transformationsregel bedeutet dies, dass der Collection Operator kein umgebendes Patternelement hat. In Listing 4.21 war das umgebende Element die Systemkomponente. Ohne ein umgebendes Element werden die Subkomponenten im gesamten Modell gesucht. Es werden insbesondere auch Subkomponenten innerhalb der inneren Komponenten (in Listing 4.4 wären dies `Monitor` und `Controller`) gesucht. Umfasst das gesamte Modell mindestens eine Subkomponente und eine Komponente `System`, ist die Transformationsregel anwendbar.

```
1 componente System {                                               MATrans
2   component Monitor {}
3   component Controller {}
4   list [[component $_ {}]]
5   list [[SubComponent $Subs]]
6 }
```

Listing 4.23: Collection Operator für alle inneren Komponenten.

Listing 4.23 zeigt ein Beispiel für zwei Collection Operatoren in einem Pattern. Das Pattern besteht aus einer Systemkomponente, die zwei innere Komponenten (`Monitor` und `Controller`) hat. Zusätzlich ist ein Collection Operator für innere Komponenten (Zeile 4) und ein Collection Operator für Subkomponenten (Zeile 5) angegeben. Damit die Transformation anwendbar ist, muss das Eingabemodell also mindestens die Systemkomponente, deren innere Komponenten `Monitor` und `Controller` sowie eine weitere innere Komponente und eine Subkomponente innerhalb der Systemkomponente umfassen. Damit ist diese Transformationsregel nicht auf das Modell in Listing 4.4 anwendbar, da es keine dritte innere Komponente innerhalb der Systemkomponente gibt. Diese wäre notwendig, da für alle Patternelemente – ohne die Verwendung des Folding Operators (vgl. Abschnitt 4.9) – verschiedene Modellelemente gefunden werden müssen. Da nicht mindestens ein Match für das Pattern innerhalb des Collection Operators für innere Komponenten gefunden werden kann, ist die Transformationsregel nicht anwendbar.

Mit dem in [Wei12] vorgestellten Listenoperator ist es bereits möglich, ähnlich wie in PROGRES, einzelne Modellelemente als mengenwertig zu markieren. Anders als in PROGRES ist es jedoch nicht möglich, innerhalb des Pattern zusätzlich Strukturen anzugeben, die sich unterhalb der mengenwertigen Elementen befinden. Beispielsweise konnten für eine mit dem Listenoperator markierte Komponente keine Ports der Komponente oder innere Komponenten angegeben werden. Als Erweiterung zu der in [Wei12] vorgestellten Version des Collection bzw. Listenoperators erlaubt die in dieser Arbeit vorgestellte, weiterentwickelte Version des Operators die Angabe von Teilgraphen.

4.8 Optional Operator: Pattern mit Optionalen Elementen

Durch die Erweiterung der Beispieltransformation um den Collection Operator in Abschnitt 4.7 ist es möglich, die Transformation auf ein Modell anzuwenden, das mehrere Subkomponenten enthält. Enthält die Systemkomponente des Modells, auf das die Transformation angewendet werden soll, keine Subkomponente wäre die Transformation nicht anwendbar. Mit den bisher vorgestellten Möglichkeiten zur Beschreibung von Pattern können Modellelemente beschrieben werden, die im Modell vorhanden sein müssen, oder – im Fall von negativen Elementen – im Modell verboten sind, damit die Transformation angewendet werden kann. In manchen Fällen sind diese Forderungen jedoch zu streng.

MATrans

```
1  component System {
2    component Monitor monitor {
3      port[[  $P [[in Signal $_]] :- ]],
4                  out $type $portName;
5    }
6    component Controller [[control :- controller]] {
7      port in $type $portName,
8           [[ :- Port $P ]];
9    }
10   not [[Subcomponent $S]]
11   not [[ connect monitor.$portName -> controller.$portName ]]
12   [[ :- connect monitor.$portName -> controller.$portName ]]
13 }
```

Listing 4.24: Pattern für Komponenten aus Listing 4.4.

Um beide Fälle – mit und ohne Subkomponente – abzudecken, müsste mit den bisher vorgestellten Operatoren zusätzlich die Transformation in Listing 4.24 angewendet werden. Diese unterscheidet sich von der in Listing 4.21 nur in Zeile 10. Hier wird explizit verboten, dass es eine Subkomponente gibt. Damit deckt diese Transformation den Fall ab, dass es keine Subkomponente gibt. Um diesen Overhead zu vermeiden und Modellelemente zu finden, falls sie vorhanden sind, bieten die DSTLs den Optional Operator.

Listing 4.25 zeigt eine Erweiterung der Transformationsregel aus Listing 4.21. Hier wurde der optional Operator auf die Menge der Subkomponenten angewendet, wodurch es optional wird, eine Menge von Subkomponenten zu finden. Falls Subkomponenten vorhanden sind, wird durch den verwendeten Collection Operator weiterhin jede Subkomponente gefunden. Falls keine Subkomponente vorhanden ist, schlägt die Anwendung der Transformation nicht fehl, da der optional Operator angibt, dass ein Match für diesen Teil des Pattern optional zu finden ist.

Dieses Beispiel verdeutlicht, dass eine strikte Einordnung in Elemente, die vorhanden sein müssen, und Elemente, die nicht vorhanden sein dürfen, zu einer Vielzahl sehr ähnlicher Transformationen führen kann. Soll eine Transformation geschrieben werden bei der n Elemente optional vorhanden sein können, führt dies mit den in [Wei12] entwickelten Operatoren im schlimmsten Fall zu 2^n Transformationen, die ein Transformationsentwickler für diesen Fall schreiben müsste. Um Transformationsentwicklern die Beschrei-

```
component System {                                                    MATrans
  component Monitor monitor {
    port[[  $P [[in Signal $_]] :- ]],
                out $type $portName;
  }
  component Controller [[control :- controller]] {
    port in $type $portName,
          [[ :- Port $P ]];
  }
  [[ opt [[ list [[Subcomponent $S]] ]] :- ]]
  not [[ connect monitor.$portName -> controller.$portName ]]
  [[ :- connect monitor.$portName -> controller.$portName ]]
}
```

Listing 4.25: Pattern für Komponenten aus Listing 4.4.

bung von Transformationen zu erleichtern, bieten die DSTLs für MontiCore daher einen expliziten *Optional Operator*, der es erlaubt, Patternelemente als optional zu markieren. Dieser Operator hat die folgende Syntax:

```
"opt" "[[" Element "]]"                                               (4.5)
```

Im Gegensatz zu dem Collection Operator für den möglichst viele aber mindestens ein Modellelement gefunden werden muss, wird für ein Patternelement, das durch den Optional Operator markiert ist, versucht, maximal ein Modellelement zu finden. Die Transformation ist auch dann anwendbar, wenn dieses nicht gefunden wird. Das heißt, an den Stellen, an denen der Optional Operator verwendet wird, werden kein oder maximal ein Modellelement gefunden. Daraus er gibt sich, dass für das Pattern in Listing 4.26 sowohl eine Komponente mit Subkomponente `ControlSystem` als auch eine ohne diese Subkomponente gültige Matches sind.

```
component $system {                                                   MATrans
  opt [[ component ControlSystem; ]]
}
```

Listing 4.26: Pattern einer Komponente mit optionaler Subkomponente.

Der Optional Operator ist hierbei auch für ganze Modellteile wie beispielsweise Komponenten inklusive Ports anwendbar (vgl. Listing 4.27, Zeile 2-6). Hierbei gilt, dass der Modellteil zwar insgesamt optional ist, damit er erfolgreich im Modell gefunden wird, müssen die spezifizierten Strukturen innerhalb des Operators in der beschriebenen Form gefunden werden. Das heißt, der Teil innerhalb des Operators kann als ein Pattern betrachtet werden, das an der durch den Operator spezifizierten Stelle gefunden werden soll. Die Patternelemente innerhalb des Operators sind für dieses Teilpattern verpflichtend, während das gesamte Teilpattern optional ist. Ein erfolgreicher Fund für den Patternausschnitt in Zeile 2-6 von Listing 4.27 muss daher eine innere Monitorkomponente sein. Diese Komponente muss hierbei über einen Port vom Typ `Integer` verfügen.

```
1  component $system {                                    MATrans
2    opt [[
3      component Monitor {
4        port out Integer $_;
5      }
6    ]]
7    opt [[
8      component ControlSystem{
9        opt [[ component Monitor;]]
10     }
11   ]]
12 }
```

Listing 4.27: Mehrfache Verwendung und Verschachtelung des Optional Operators.

Der Optional Operator darf auch geschachtelt auftreten. Das heißt, ein optionaler Teil des Pattern kann wiederum optionale Teile haben. Wie zuvor erläutert kann der Teil innerhalb des Operators als eigenes Pattern betrachtet werden, innerhalb dessen nicht optionale Teile verpflichtend sind. Hierbei ist es wiederum erlaubt, Teile als optional zu markieren. Die Zeilen 7 bis 11 in Listing 4.27 zeigen ein Beispiel hierfür. Hier wird eine Subkomponente `ControlSystem` beschrieben. Diese kann außerdem optional eine innere Komponente `Monitor` haben.

Des Weiteren darf der Operator innerhalb einer Transformation beliebig oft „nebeneinander“ vorkommen, sodass an verschiedenen Stellen im Pattern Teile optional sind. Ist dies der Fall, wird versucht für die optionalen Teilpattern jeweils ein Match zu finden. Für jedes einzelne optionale Teilpattern kann hierbei ein Match gefunden oder nicht gefunden werden. Im Beispiel in Listing 4.27 ist der Optional Operator an drei Stellen verwendet worden: Für die inneren Komponenten `Monitor` und `ControlSystem` sowie innerhalb der optionalen inneren Komponente für die Subkomponente `Monitor`. Für jeden optionalen Teil wird separat versucht diesen im Modell zu finden. Die Transformation kann daher auch dann angewendet werden, wenn nur eine der inneren Komponenten gefunden wird oder wenn die innere Komponente `ControlSystem` existiert, sie jedoch keine Subkomponente `Monitor` hat.

Darüber hinaus kann der Optional Operator auch mit den anderen Operatoren der DSTLs kombiniert werden. Der Optional Operator darf nicht innerhalb des Operators zur Negation von Modellteilen verwendet werden. Man könnte dies zwar durch den Pattern Matching Algorithmus unterstützen, da diese Kombination jedoch keinen Mehrwert für die Spezifikation verbotener Modellteile bietet, sich jedoch negativ auf die Laufzeit des Algorithmus auswirkt, wird diese Kombination durch die DSTLs nicht unterstützt. Die entsprechende Kontextbedingung wird in Abschnitt 4.14 detaillierter erläutert. Im Gegensatz dazu ist es wichtig, dass auch innerhalb von optionalen Modellteilen Negation möglich ist. In diesem Fall wird – ähnlich wie bei nicht optionalen Pattern – ein Match für den optionale Teil nur dann gefunden, wenn ein Match für alle nicht negierten Teile des Pattern im Modell gefunden wird und es keinen möglichen Match für die negativen

Teile des Pattern gibt. Das Beispiel in Listing 4.28 verdeutlicht die Möglichkeit von negativen Teilen innerhalb des Optional Operators. Hier wird eine Komponente gesucht. Außerdem wird optional auch deren innere Komponente `ControlSystem` gesucht. Diese innere Komponente darf jedoch keine Subkomponente `Monitor` haben.

```
1 component $system {                                          MATrans
2   opt [[
3     component ControlSystem{
4       not [[ component Monitor;]]
5     }
6   ]]
7 }
```

Listing 4.28: Pattern mit optionaler Komponente, die ein negatives Element hat.

Im Folgenden wird die Kombination aus dem Optional und dem Replacement Operator erläutert. Ähnlich wie negative Elemente dürfen optionale Elemente nicht auf der rechten Seite des Replacement Operators auftreten. Optionale Elemente auf der rechten Seite einer Ersetzung würden zu einem nichtdeterministischen Hinzufügen von Modellelementen führen, was die Benutzerfreundlichkeit für Transformationsentwickler aufgrund schlechterer Nachvollziehbarkeit der Auswirkungen einer Transformation verringern würde. Daher erlauben die DSTLs keine optional hinzugefügten Modellelemente. Die entsprechende Kontextbedingung wird in Abschnitt 4.14 genauer erläutert. Auf der linken Seite des Replacement Operators hingegen, dürfen optionale Elemente vorkommen. Das Beispiel in Listing 4.29 zeigt eine Transformationsregel, bei der eine optional vorhandene Subkomponente `ControlSystem` durch eine `Monitor` Subkomponente ersetzt wird. Hierbei wird der Monitor auch dann der äußeren Komponente hinzugefügt, wenn keine Subkomponente `ControlSystem` gefunden wird.

```
1 component $system {                                          MATrans
2   [[ opt [[component ControlSystem;]] :- component Monitor; ]]
3 }
```

Listing 4.29: Ersetzung von optionalen Elementen.

Neben der Ersetzung von optionalen Modellteilen können auch Ersetzungen innerhalb von optionalen Modellteilen beschrieben werden. In Zeile 1-5 in Listing 4.30 wird eine optionale Komponente beschrieben, die eine innere Komponente hat, die an die Schemavariable `$C` gebunden wird. Wird diese Komponente gefunden, wird die innere Komponente aus der äußeren entfernt. Zusätzlich wird in Zeile 7-11 eine weitere optionale Komponente beschrieben. Dieser Komponente wird die innere Komponente, die aus der anderen Komponente entfernt wird, hinzugefügt, sofern sie gefunden wurde. An Schemavariablen gebundene Patternelemente können auch außerhalb ihres eigenen Optional Operators verwendet werden. Wird ein Match für das optionale, an eine Schemavariable gebundene Modellelement gefunden, werden die von diesem Fund abhängigen Aktionen

durchgeführt. Wird kein Match für das Patternelement gefunden, wird die abhängige Aktion nicht durchgeführt. Im vorangegangenen Beispiel betrifft dies die zu verschiebende, innere Komponente. Zusätzlich kann auch innerhalb der Java-Anteile auf diese Schemavariable zugegriffen werden. Hier ist ein Java `Optional` verwendet, um die Optionalität zu repräsentieren. Entsprechend kann über die Optional-API (`isPresent()` und `get()`) abgefragt werden, ob ein Match gefunden wurde, und – falls ein Match gefunden wurde – dieser abgerufen werden. Für den Application Constraint (vgl. Abschnitt 4.11) wird die Behandlung der optionalen Patternelemente durch die Transformation übernommen, sodass der Application Constraint auswertbar bleibt, falls ein optionales Patternelement nicht gefunden wird. Hierbei werden Aussagen über nicht gefundene Elemente als erfüllt betrachtet, wenn das optionale Patternelement innerhalb einer Konjunktion auftritt, und als falsch angenommen, wenn es innerhalb einer Disjunktion vorkommt. Details der technischen Realisierung können der betreuten Vorarbeit [Wil17] entnommen werden.

MATrans

```
1  opt [[
2    component $one {
3      [[ Component $C :- ]]
4    }
5  ]]
6
7  opt [[
8    component $two {
9      [[ :- Component $C ]]
10   }
11 ]]
```

Listing 4.30: Ersetzungen in optionalen Teilen des Pattern.

4.9 Folding Operator: Nichtisomorphes Matching

Die DSTLs in MontiCore realisieren einen Graphtransformationsansatz [CH06]. Wie für diese Ansätze üblich, werden alle Elemente des Pattern auf verschiedene Elemente im Modell abgebildet. Das heißt beispielsweise, dass für zwei Komponenten in einem Pattern auch zwei Komponenten im Modell gefunden werden müssen. Diese strikte Einschränkung ist jedoch nicht in allen Fällen wünschenswert und wirkt sich zum Teil negativ auf die Benutzerfreundlichkeit des Ansatzes aus.

Listing 4.31 beschreibt ein Pattern, in dem eine Komponente mit zwei inneren Komponenten und einem Konnektor gefunden werden soll. Die inneren Komponenten haben kompatible Ports. Das bedeutet, der Typ und der Name der Ports stimmen überein, wobei der Port der ersten Komponente (Zeile 2-6) ausgehend und der Port der zweiten Komponente (Zeile 8-12) eingehend ist. Der Konnektor verbindet die Ports der beiden inneren Komponenten.

```
component $outer {                                          MATrans
  component $_ $f {
    port out $type $out;
  }

  component $_ $t {
    port in $type $in;
  }

  connect $f.$out -> $t.$in;
}
```

Listing 4.31: Pattern: Isomorphes Matching.

```
component System {                                          MA
  component Monitor m {
    port in  State incoming;
    port out State outgoing;
  }
  connect m.outgoing -> m.incoming;
}
```

Listing 4.32: MontiArc Modell mit Komponente mit Feedback-Konnektor.

Das Modell in Listing 4.32 hat eine äußere und eine innere Komponente. Die innere Komponente hat zwei Ports von Typ `State`, wobei der eine eingehend ist und `incoming` heißt und der andere ausgehen ist und `outgoing` heißt. In diesem Modell kann kein Match für das Pattern aus Listing 4.31 gefunden werden. Es gibt zwar eine Komponente deren innere Komponente die Eigenschaften beider inneren Komponenten aus dem Pattern erfüllt, da es jedoch nur eine innere Komponente gibt und für die Patternelemente unterschiedliche Modellelemente gefunden werden müssen, kann die Komponente `Monitor` nicht als Match für beide inneren Komponenten des Pattern gefunden werden. Ein Feedback-Konnektor ist damit kein gültiger Match für den hier gesuchten Konnektor inklusive der Start- und Zielkomponente. Um sowohl Feedback-Konnektoren als auch Konnektor zwischen zwei verschiedenen Komponenten als Match zu erlauben, bieten die DSTLs einen *Folding Operator*. Dieser erlaubt zu definieren, für welche Patternelemente beim Pattern Matching die gleichen Modellelemente als Match gefunden werden dürfen. Der Folding Operator hat hierzu die folgende Syntax:

```
"folding" "{"
   SchemaVarTuple+                                          (4.6)
"}"
```

Der Folding Operator besteht also aus dem Schlüsselwort `folding` und einer Liste von beliebig langen Tupeln aus Schemavariablen. Diese Tupel geben jeweils an, welche

Patternelemente auf die gleichen Modellelemente abgebildet werden dürfen. Listing 4.33 zeigt ein Beispiel für die Verwendung des Folding Operators. Das Pattern beschreibt hierbei die gleiche Struktur wie das Pattern in Listing 4.31 mit dem Unterschied, dass die beiden inneren Komponenten an die Schemavariablen `$FROM` bzw `$TO` gebunden wurden. Zusätzlich wurde der Folding Operator verwendet, um anzugeben, dass die beiden inneren Komponenten `$TO` und `$FROM` auf die gleiche innere Komponente im Modell abgebildet werden können. Im Gegensatz zu dem Pattern aus Listing 4.31 kann für das Pattern in Listing 4.33 in dem Modell in Listing 4.32 ein Match gefunden werden, indem sowohl für die Komponente `$FROM` als auch für die Komponente `$TO` die innere Komponente `Monitor` gefunden wird.

MATrans

```
1  component $outer {
2    $FROM [[ component $_ $f {
3               port out $type $port;
4             } ]]
5
6    $TO [[ component $_ $t {
7             port in $type $port;
8           } ]]
9
10   connect $f.$port -> $t.$port;
11 }
12
13 folding {
14   ($FROM, $TO)
15 }
```

Listing 4.33: Pattern: Nichtisomorphes Matching.

4.10 Zuweisung von Schemavariablen

In der Beispieltransformation wird der Name `control` durch den Namen `controller` ersetzt. Der Name bekommt nur das zusätzliche Suffix `ler`. Die Transformation kann daher auch hier auf Schemavariablen setzen und per Berechnung aus `control` `controller` erstellen[1]. Die Transformationsregel in Listing 4.34 ist eine Erweiterung der Transformationsregel in Listing 4.25. In Zeile 6 wurden die Namen `control` und `controller` durch die Schemavariabelen `$oldName` und `$newName` ersetzt. Der Wert für die nur auf der rechte Seite des Replacement Operators vorkommende Schemavariable `$newName` wird im `assign`-Block (Zeile 15-17) berechnet und der Schemavariable zugewiesen.

Die Werte von Schemavariablen innerhalb von Pattern (inklusive der linken Seite von Ersetzungen) ergeben sich aus den entsprechenden Werten der Modellelemente, die für die Patternelemente gefunden werden. Dies kann beispielsweise der Namen einer

[1] Diese Berechnung dient Demonstrationszwecken.

```
component System {                                                    MATrans
  component Monitor monitor {
    port[[  $P [[in Signal $_]] :- ]],
                out $type $portName;
  }
  component Controller [[$oldName :- $newName]] {
    port in $type $portName,
         [[ :- Port $P ]];
  }
  [[ opt [[ list [[Subcomponent $S]] ]] :- ]]
  not [[ connect monitor.$portName -> controller.$portName ]]
  [[ :- connect monitor.$portName -> controller.$portName ]]
}

assign {
  $newName = $oldName +"ler"
}
```

Listing 4.34: Pattern: Zuweisung von Variablen.

Komponente oder eine Komponente selbst sein. Schemavariablen, die auf der rechten Seite einer Ersetzung vorkommen, werden nicht auf diese Weise mit Werten belegt. Die DSTLs in MontiCore bieten hierfür zwei Möglichkeiten. Zum einen können die Werte der Transformation als Parameter übergeben werden (vgl. Abschnitt 9.4), zum anderen bieten die DSTLs einen hierfür vorgesehenen Zuweisungsblock, der die Zuweisung von Werten innerhalb der Transformation erlaubt. Die Syntax des Zuweisungsblocks ist:

```
"assign" "{"
   (SchemaVar "=" Expression ";")+                                    (4.7)
"}"
```

Der Zuweisungsblock besteht aus dem Schlüsselwort `assign` und einer Menge von Zuweisungen innerhalb von geschweiften Klammern. Eine Zuweisung besteht hierbei immer aus einer Schemavariablen auf der linken Seite, einem Gleichheitszeichen `=` und einer Berechnung des Schemavariablenwerts auf der rechten Seite. Eine Zuweisung wird durch ein Semikolon abgeschlossen. Auf diese Weise berechnete Werte können auch im Application Constraint verwendet werden (vgl. Abschnitt 4.11). Im Vergleich zu [Wei12] ist durch die Einführung eines Zuweisungsblocks die Verwendung von Schemavariablen auf der rechten Regelseite möglich. Eine weitere in [Wei12] vorhandene Limitierung, dass Werte – beispielsweise für Namen von Modellelementen – nur direkt aus anderen Modellelementen übernommen aber nicht angepasst oder vollständig berechnet werden können, wird ebenfalls durch den Zuweisungsblock aufgehoben.

4.11 Application Constraint

In der Beispieltransformation wurden das Pattern für die Stateports komplett durch Schemavariablen beschrieben. Durch die Verwendung der Schemavariabeln ist sichergestellt, dass die Namen und Typen der Ports übereinstimmen. Dass die Ports den Typ `State` haben, wird nicht mehr sichergestellt. Um dies zusätzlich sicherzustellen, kann über die verwendeten Schemavariablen ein Bedingung formuliert werden – der Application Constraint –, die erfüllt sein muss, damit die Transformationsregel anwendbar ist. Dies wird auch als (positive) application condition bezeichnet [EH86, HP05]. Die Transformationsregel in Listing 4.35 ist eine Erweiterung der Transformationsregel aus Listing 4.34. Die Transformationsregel fordert zusätzlich, dass der Name der Stateports (Schemavariable `$portname`) `state` ist. Dies wird durch die Formulierung eines entsprechenden Constraints mit Hilfe des Bedingungsblocks in Zeile 15-17 sichergestellt.

```
1  component System {                                                    MATrans
2    component Monitor monitor {
3      port[[  $P [[in Signal $_]] :- ]],
4                  out $type $portName;
5    }
6    component Controller [[$oldName :- $newName]] {
7      port in $type $portName,
8           [[ :- Port $P ]];
9    }
10   [[ opt [[ list [[Subcomponent $S]] ]] :- ]]
11   not [[ connect monitor.$portName -> controller.$portName ]]
12   [[ :- connect monitor.$portName -> controller.$portName ]]
13 }
14
15 where {
16   $portName.equals("state")
17 }
18
19 assign {
20   $newName = $oldName +"ler"
21 }
```

Listing 4.35: Transformationsregel deren Anwendbarkeit über einen Application Constraint eingeschränkt ist.

Wie gezeigt, erlauben die DSTLs die Angabe von Application Constraints, die die Anwendbarkeit von Transformationsregeln weiter einschränken. Dazu bieten die DSTLs einen Bedingungsblock mit folgender Syntax an:

```
"where" "{"
   BooleanExpression                                                    (4.8)
"}"
```

Der Bedingungsblock besteht aus dem Schlüsselwort `where` und einem in geschweiften Klammern eingefassten booleschen Ausdruck. Umfasst eine Transformationsregel einen Application Constraint, dann muss dieser erfüllt sein, damit die Transformationsregel angewendet werden kann. Das bedeutet, eine Transformationsregel ist anwendbar, wenn

1. für alle einfachen, nicht speziell markierten sowie auf der linken Seite von Ersetzungen vorkommenden Patternelemente, Modellelemente gefunden werden,
2. es keinen Match für die negativen Elemente der Transformationsregel gibt,
3. für alle Collection Operatoren mindestens ein Match gefunden wird und
4. der Application Constraint erfüllt ist.

Der boolesche Ausdruck eines Bedingungsblocks wird durch eine Java-Expression [Ull12] der durch MontiCore zur Verfügung gestellten *JavaDSL* [MSN17] realisiert. Innerhalb des Constraints sind alle, nicht anonymen, im Pattern verwendeten Schemavariablen verfügbar sowie die Methoden der abstrakten Syntax der Modellelemente, wie zum Beispiel `getName()` für Komponenten. Insbesondere sind anders als in [Wei12] alle Schemavariablen im Application Constraint verfügbar. Der Application Constraint ist ein Feature, das nicht allein auf der konkreten Syntax arbeitet. Damit benötigt der Transformationsentwickler zur Nutzung dieses Features grundlegende Java-Kenntnisse und in vielen Fällen eine grundlegende Kenntnis der abstrakten Syntax.

4.12 Direkte Manipulation und abschließende Aktionen

Bisher wurde erklärt, dass die DSTLs einen Replacement Operator anbieten, der es erlaubt Veränderungen innerhalb des zum Pattern passenden Modellteils zu beschreiben. Als Erweiterung von [Wei12] bieten die DSTLs – neben der Möglichkeit Veränderungen mittels Replacement Operator anzugeben – auch eine Möglichkeit das Modell bzw. die durch die Transformation gefundenen Modellteile durch direkte Manipulation zu verändern [CH06, MVG06, Bie10, KC15]. Des Weiteren können nach Ausführung einer Transformation bzw. am Ende der Ausführung noch abschließende Aktionen ausgeführt werden. Zum Beispiel können Reports erstellt oder Templates an Modellelemente entsprechend der Templateerweiterungsmechanismen angehängt werden (vgl. Abschnitt 4.13.4). Um dies zu ermöglichen bieten die DSTLs einen Anweisungsblock mit folgender Syntax:

```
"do" "{"
   JavaCode                                                    (4.9)
"}"
```

Am Ende einer Transformationsregel kann somit ein Anweisungsblock eingefügt werden, der durch das Schlüsselwort `do` eingeleitet wird und in geschwungenen Klammern Java-Statements enthält. Dieser Java-Code wird als Abschluss der Transformationsregel ausgeführt. Innerhalb dieses Java-Codes hat der Transformationsentwickler Zugriff

auf die Modellelemente, die innerhalb des Pattern an Schemavariablen gebunden wurden. Außerdem kann der Transformationsentwickler die in Abschnitt 4.13 beschriebenen Built-Ins verwenden. Dadurch wird es beispielsweise möglich nur den Pattern Matching Teil der DSTLs zu nutzen und die gesamte Veränderung des Modells manuell, imperativ in Java zu beschreiben. Der Transformationsentwickler ist jedoch nicht gezwungen, sich zwischen deklarativer und imperativer Modifikation zu entscheiden. Die DSTLs erlauben auch eine Kombination aus beiden Möglichkeiten, wodurch der Entwickler flexibel entscheiden kann, welche Teile er intuitiver deklarativ beschreibt und welche er imperativ umsetzt. Im Folgenden werden diese Möglichkeiten an Beispielen erklärt.

```
MATrans
component System {
  component Monitor monitor {
    port[[  $P [[in Signal $_]] :- ]],
                out $type $portName;
  }
  component Controller [[$oldName :- $newName]] {
    port in $type $portName,
         [[ :- Port $P ]];
  }
  [[ opt [[ list [[Subcomponent $S]] ]] :- ]]
  not [[ connect monitor.$portName -> controller.$portName ]]
  [[ :- connect monitor.$portName -> controller.$portName ]]
}

where {
  $portName.equals("state")
}

assign {
  $newName = $oldName +"ler";
}

do {
  $P.setName("trig");
}
```

Listing 4.36: Imperative beschriebene Ersetzung des Namens eines Ports mittels Anweisungsblock.

Direkte Manipulation

Der Anweisungsblock erlaubt dem Transformationsentwickler die direkte, imperativ beschriebene Manipulation des Modells. Dazu kann das Pattern mit Hilfe der zuvor beschriebenen Operatoren formuliert werden. Zusätzlich muss der Transformationsentwickler die Modellelemente, auf die er im Anweisungsblock Zugriff braucht, an Schemavariablen binden. Dadurch können sie direkt im Anweisungsblock verwendet werden.

Zusätzlich können die Methoden der abstrakten Syntax sowie die in Abschnitt 4.13 beschriebenen Built-Ins verwendet werden. Listing 4.36 zeigt ein Beispiel für eine imperativ beschriebene Modifikation des Beispielmodells aus Listing 4.4.

Die Transformationsregel ist eine Erweiterung der Transformationsregel in Listing 4.35. Die Transformationsregel wurde um einen Anweisungsblock in Zeile 23 bis 25 ergänzt. Innerhalb dieses Blocks wird durch direkte Modellmanipulation der Name des Signalport (gebunden an die Schemavariable `$P`) auf `trig` gesetzt. Dazu wurde die von der AST-Klasse des Ports zu Verfügung gestellte Methode `setName()` verwendet.

```
1  component System {                                                    MATrans
2    component Monitor monitor {
3      port[[  $P [[in Signal $_]] :- ]],
4        out $type $portName;
5    }
6    component Controller [[$oldName :- $newName]] {
7      port in $type $portName,
8      [[ :- Port $P ]];
9    }
10   [[ opt [[ list [[Subcomponent $S]] ]] :- ]]
11   not [[ connect monitor.$portName -> controller.$portName ]]
12   [[ :- connect monitor.$portName -> controller.$portName ]]
13 }
14
15 where {
16   $portName.equals("state")
17 }
18
19 assign {
20   $newName = $oldName +"ler";
21 }
22
23 do {
24   reportChange($P, "name", $P.getName(), "trig");
25   $P.setName("trig");
26 }
```

Listing 4.37: Pattern: Reporting als abschließende Aktion.

Abschließende Aktionen

Neben der zuvor erläuterten imperativen Beschreibung von Modifikationen durch eine Transformationsregel kann der Anweisungsblock auch für weitere Aktionen, wie beispielsweise Reporting, das Anhängen von Templates oder Analysen des Modells oder Teilen des Modells, genutzt werden. Hierfür können sowohl die an Schemavariablen gebundenen Modellelemente als auch die in Abschnitt 4.13 beschriebenen Built-Ins genutzt werden. Darüber hinaus ist der Transformationsentwickler frei in der Entscheidung, wel-

che Aktionen er am Ende der Transformationsregel ausführen lassen möchte. Listing 4.37 zeigt ein Beispiel für eine Transformationsregel, die den Anweisungsblock zum Reporting verwendet und hierbei Gebrauch von Built-Ins macht.

Ähnlich wie die Transformationsregel in Listing 4.36 ersetzt die Transformationsregel in Listing 4.37 den Namen des Ports. Zusätzlich verwendet die Transformationsregel jedoch das Built-In `reportChange`, um diese Änderung als Report auszugeben.

4.13 Built-Ins

Um die Benutzerfreundlichkeit für Transformationsentwickler zu erhöhen, bieten die DSTLs für häufig benötigte Funktionen Hilfsmethoden in Form von Built-Ins an. Ein Beispiel hierfür ist die Stringmanipulation. Built-Ins sind ein im Vergleich zu [Wei12] neues Feature. Diese können innerhalb der Java-Anteile einer Transformationsregel genutzt werden. Die DSTLs stellen solche Built-Ins für Stringmanipulation, die Handhabung von Listen, die Ausgabe von Reports sowie die Handhabung von (angehängten) Templates bereit. Im Folgenden werden die verschiedenen Built-Ins detailliert.

4.13.1 Stringmanipulation

Häufige Anwendungsfälle bei der Verwendung von Zeichenketten. Ein Beispiel hierfür ist das Konkatenieren sowie die Beeinflussung der Groß- oder Kleinschreibung von Strings bei der Berechnung von Namen von Modellelementen. Hierzu bieten die DSTLs verschiedene Built-Ins, die innerhalb der Java-Anteile einer Transformation, also dem Zuweisungs-, dem Bedingungs- und dem Anweisungsblock, verwendet werden können.

`String uncapitalize(String)` nimmt einen String als Parameter und gibt den gleichen String mit kleinem Anfangsbuchstaben zurück.

`String capitalize(String)` nimmt einen String als Parameter und gibt den gleichen String mit großem Anfangsbuchstaben zurück.

`String qualifiedName(Iterable<String>)` erwartet ein Iterable von Strings als Parameter und gibt einen qualifizierten Namen zurück. Das heißt, aus der Liste `["a", "b", "c"]` wird `"a.b.c"`.

Joiner Der Joiner basiert auf dem in Guava [Bej13] vorhandenen Joiner und bietet die Möglichkeit Iterables von Strings zu einem String zusammenzusetzen. Hierzu gibt es folgende Methoden:

`Joiner on(Char)` erwartet das Zeichen, das als Trennzeichen zwischen den Strings verwendet werden soll und liefert den entsprechenden Joiner zurück.

`String join(Iterable<String>)` erwartet ein Iterable von Strings, die zusammengesetzt werden sollen, und setzt einen String aus den Teilstrings zusammen. Das über on() übergebene Zeichen separiert hierbei die verschiedenen Teilstrings.

Listing 4.38 zeigt ein Beispiel für die Verwendung des Joiners:

```java
List<String> toBeJoined = newArrayList("a", "b");
String joined = Joiner.on(":").join(toBeJoined); //a:b
```

Listing 4.38: Beispiel für die Verwendung des Joiners.

Splitter Der Splitter basiert auf dem in Guava [Bej13] vorhandenen Splitter und kann einen String an bestimmten Zeichen splitten und so eine Liste bzw. ein Iterable von Strings erzeugen.

`Splitter on(Char)` gibt das Zeichen an, an dem der String gesplittet wird soll und liefert den entsprechenden Splitter zurück.

`Iterable<String> split(String)` erwartet einen String als Eingabe und liefert ein Iterable von Strings zurück.

Listing 4.39 zeigt ein Beispiel für die Verwendung des Splitters:

```java
String toBeSplit = "a:b";
Iterable<String> split = Splitter.on(":")
                              .split(toBeSplit); // ["a", "b"]
```

Listing 4.39: Beispiel für die Verwendung des Splitters.

4.13.2 Listenbehandlung

Zum Erstellen von Listen, sowohl leeren als auch solchen basierend auf vorhandenen Listen oder Modellelementen, bieten die DSTLs einige Built-Ins an, um dem Transformationsentwickler die Handhabung zu erleichtern.

`List<E> newArrayList(E...)` ermöglicht es Listen aus einer beliebigen Anzahl Elementen des gleichen Typs zu erstellen. Ein Beispiel hierfür ist in Listing 4.38 gezeigt. Hier wird eine Liste `List<String>` aus den Elementen `"a"` und `"b"` erstellt. Die Methode liefert eine Liste der übergebenen Elemente zurück.

`List<E> newArrayList(Iterable<? extends E>)` ermöglicht die Erstellung einer neuen Liste vom Typ `ArrayList` anhand eines übergebenen `Iterable`. Das Iterable kann zum Beispiel eine Liste, ein Array oder ein Set sein. Die Methode liefert die erstellte Liste zurück.

`List<E> newArrayList()` ermöglicht das Erstellen einer leeren Liste vom Typ `ArrayList`. Die Methode liefert die erstellte Liste zurück.

`List<E> newLinkedList()` ermöglicht das Erstellen einer leeren Liste vom Typ `LinkedList`. Die Methode liefert die erstellte Liste zurück.

`List<E> newLinkedList(Iterable<? extends E)` ermöglicht die Erstellung einer neuen Liste vom Typ `LinkedList` anhand eines übergebenen `Iterable`. Das Iterable kann zum Beispiel eine Liste, ein Array oder ein Set sein. Die Methode liefert die neue Liste zurück.

4.13.3 Reporting and Logging

Reporting und Logging bieten die Möglichkeit das Verhalten einer Transformation nachzuvollziehen. Während der deklarative Anteil einer Transformation automatisch entsprechende Reportevents erzeugt, muss dies für den imperativen Teil, also innerhalb des Anweisungsblocks, der Transformationsentwickler selbst übernehmen. Um dem Transformationsentwickler dies zu erleichtern, bieten die DSTLs hierfür spezielle Built-Ins an. Reporting zielt hierbei mehr darauf ab, den Ablauf einer Transformation nachzuvollziehen und realisiert damit eine Art Tracing [CH06] der Transformation. Logging hingegen ermöglicht es dem Transformationsentwickler Fehler, Warnungen und Hinweise zu melden bzw. zu protokollieren, fokussiert dabei aber weniger den Transformationsablauf.

`reportMatch (ASTNode)` erwartet den gefundenen AST-Knoten als Parameter und schreibt erzeugt eine Nachricht über das gefundene Modellelement für einen Transformationsreport.

`reportDeletion(ASTNode)` erwartet den zu löschenden AST-Knoten als Parameter und erzeugt eine Nachricht für das zu löschende Modellelement für einen Transformationsreport.

`reportCreation(ASTNode)` erwartet den erstellten AST-Knoten als Parameter und erzeugt eine Nachricht über das erzeugte Modellelement für einen Transformationsreport.

`reportChange(ASTNode, String attr, String old, String new)` erstellt eine Nachricht über eine Änderung von Attributwerten für einen Report. Erwartet den betroffenen AST Knoten, den Namen des Attributes, den alten Wert und den neuen Attributwert.

`reportChange(ASTNode, String attr, ASTNode old, ASTNode new)` erstellt eine Nachricht über eine Änderung von Kompositionen für einen Report. Erwartet den betroffenen AST-Knoten (Kompositum), den Namen der Komposition, die alte Komponente und die neue Komponente.

`error(String)` sollte verwendet werden, wenn – beispielsweise im Anweisungsblock – ein kritischer Fehler auftritt, der die Ausführung der Transformation verhindert. Die Methode erwartet einen String mit einer Nachricht, die den Fehler beschreibt, schreibt diese in den Log und beendet die Ausführung der Transformation.

`warn(String)` sollte verwendet werden, wenn – beispielsweise im Anweisungsblock – ein Fehler auftritt, der die Ausführung der Transformation beeinträchtigen kann. Die Methode erwartet einen String mit einer Nachricht, die den Fehler beschreibt und schreibt diese in den Log. Die Transformation wird weiter ausgeführt.

`info(String msg, String logger)` sollte verwendet werden, um – beispielsweise während der Ausführung des Anweisungsblocks – eine Information zu protokollieren. Die Methode erwartet einen String mit einer Nachricht und die Angabe des Namens des zu verwendenden Loggers.

`debug(String msg, String logger)` sollte verwendet werden, um – beispielsweise während der Ausführung des Anweisungsblocks – eine Debuginformation zu protokollieren. Die Methode erwartet einen String mit einer Nachricht und die Angabe des Namens des zu verwendenden Loggers.

4.13.4 Templateerweiterung und Hook Points

MontiCore bietet für die templatebasierte Codegenerierung einen Templateerweiterungsmechanismus an, der es erlaubt Templates an spezifische Modellelemente zu hängen, sodass diese bei der Codegenerierung verwendet werden können. Technisch ist dieses Features durch die Klasse `GlobalExtensionManagement` des MontiCore Runtime Environment realisiert, deren Instanz durch die Generatoren und die Templates verwendet wird. Das Anhängen von Templates an Modellelemente passiert vor der eigentlichen Generierung. Werden zusätzlich Transformation verwendet, die neue Modellelemente erzeugen, so muss es möglich sein auch diese mit Templates zu versehen. Dazu bieten die DSTLs die Möglichkeit einer Transformation eine Instanz der Klasse `GlobalExtensionManagement` zu übergeben und innerhalb der Java-Anteile der Transformation auf diese Instanz zu zugreifen. Dadurch kann der Transformationsentwickler für erstellte Modellelemente spezifische Templates registrieren, indem er die vordefinierte Schemavariable `glex` innerhalb der Java-Anteile der Transformation verwendet. Listing 4.40 zeigt ein Beispiel für die Verwendung von `glex` innerhalb einer Transformationsregel.

MATrans

```
1 component System {
2   [[ :- $P [[ port Integer state; ]] ]]
3 }
4
5 do {
6   glex.replaceTemplate("ma.DefaultPort", $P, "ma.WrappedPort");
7 }
```

Listing 4.40: Verwendung von `glex` innerhalb einer Transformationsregel zum Austausch eines Templates.

Die Transformationsregel in Listing 4.40 sucht nach einer Komponente `System`. Wird diese gefunden, fügt die Transformationsregel dieser Komponente einen neuen Port `state` hinzu. Der Generator verwendet zur Generierung von Code für Ports das Template `ma.DefaultPort`, welches durch die Transformation speziell für den hier eingefügten Port durch das Template `ma.WrappedPort` ersetzt wird. Die Anwendung von Transformationen, die `glex` verwenden, wird in Kapitel 9 detailliert. Die Funktionsweise des Templateerweiterungsmechanismus ist in [Rot17] erläutert.

4.14 Kontextbedingungen

Bisher wurden die verschiedenen Operatoren und Konzepte der DSTLs vorgestellt. Hierbei wurden Einschränkungen in deren Kombinationsmöglichkeiten weitestgehend außer

Acht gelassen. In diesem Abschnitt werden nun die geltenden Bedingungen erläutert. Die Grammatiken der DSTLs definieren die parsbaren Transformationen und erlauben durch ihre Orientierung an der Struktur der Modellierungssprache, dass Modellelemente nur in valider Konstellation beschrieben werden können (z.B. keine Komponentendefinitionen als Portnamen). Kontextbedingungen – zum Teil auch statische Semantik genannt – werden nach dem Parsen eines Modells geprüft und schränken die möglichen Transformationen weiter ein, sodass nur wohlgeformte Modelle erlaubt sind [Völ11]. Damit stellen sie sicher, dass eine Transformation verarbeitbar ist und eine ausführbare Java-Implementierung für diese erzeugt werden kann. Im Folgenden werden die für die DSTLs geltenden Kontextbedingungen, die beispielsweise das Einfügen negativer Modellelemente verhindern, beschrieben.

4.14.1 Wohlgeformtheit bezüglich Schemavariablen

Schemavariablen werden während des Pattern Matchings mit Modellelementen belegt. Um eine korrekte Belegung der Schemavariablen zu ermöglichen, gelten eine Reihe von Kontextbedingungen, die im Folgenden erläutert werden.

Schemavariablen für Modellelemente unterschiedlichen Typs sind verschieden.

Die Zuordnung von Modellelementen zu Schemavariablen muss eindeutig sein. Um dies zu ermöglichen, darf eine Schemavariable nur für ein Modellelement verwendet werden. Innerhalb einer Transformation darf die gleiche Schemavariable daher nicht für Elemente unterschiedlichen Typs verwendet werden. Eine Schemavariable darf im Pattern, der rechten Regelseite bzw. dem Zuweisungsblock sowie im Application Constraint verwendet werden und bezieht sich in allen Fällen auf des gleiche Modellelement. Die folgende Kontextbedingung überprüft dies:

Bedingung:	Die gleiche Schemavariable darf nicht für verschiedene Modellelemente unterschiedlichen Typs verwendet werden.
Fehlercode:	0xF0C01
Schweregrad:	Error
Hinweis:	Umbenennung bzw. Verwendung eines anderen Namen für die Schemavariable führt in diesem Fall zu einer validen Transformation.

MATrans

```
1 $A [[
2   component Monitor;
3 ]]
4 $A [[ // ⊗ 0xF0C01
5   connect c -> m;
6 ]]
```

Listing 4.41: Transformation die die Kontextbedingung 0xF0C01 verletzt.

Listing 4.41 zeigt ein Beispiel, das diese Kontextbedingung verletzt. Die hier angegebene Transformationsregel verwendet die Schemavariable `$A` zum einen für eine Subkomponente (Zeile 1-3) und zum anderen für einen Konnektor (Zeile 4-6). Korrekterweise müssen hier stattdessen zwei verschiedene Schemavariablen verwendet werden.

Schemavariablen für verschiedene Modellelemente gleichen Typs müssen verschieden sein.

Die Zuordnung von Modellelementen zu Schemavariablen muss eindeutig sein. Um dies zu ermöglichen, darf eine Schemavariable nur für ein Modellelement verwendet werden. Innerhalb einer Transformation darf die gleiche Schemavariable daher nicht für verschiedene Modellelemente verwendet werden, selbst wenn deren Typ übereinstimmt. Eine Schemavariable darf im Pattern, der rechten Regelseite bzw. dem Zuweisungsblock sowie im Application Constraint verwendet werden und bezieht sich in allen Fällen auf des gleiche Modellelement. Eine Ausnahme dieser Regel bilden Schemavariablen für Namen. Wird hier die gleiche Schemavariable verwendet, fordert dies die Gleichheit der Namen statt der Identität der Namen. Die folgende Kontextbedingung überprüft dies:

Bedingung:	Die gleiche Schemavariable darf nicht für verschiedene Modellelemente des gleichen Typs verwendet werden.
Fehlercode:	0xF0C02
Schweregrad:	Error
Hinweis:	Ausnahme von dieser Regel sind Namen von Modellelementen. Hier wird nicht die Identität sondern die Gleichheit gefordert.

```
1 $A [[                                                    MATrans
2   component Monitor;
3 ]]
4 $A [[ // ⊗ 0xF0C02
5   component Controller;
6 ]]
```

Listing 4.42: Transformation die die Kontextbedingung 0xF0C02 verletzt.

Listing 4.42 zeigt ein Beispiel, das diese Kontextbedingung verletzt. Die hier angegebene Transformationsregel verwendet die Schemavariable `$A` für zwei Subkomponenten, die sich in ihrem Namen unterscheiden. Korrekterweise müssen hier stattdessen zwei verschiedene Schemavariablen verwendet werden.

Schemavariablen für Namen eingefügter Modellelemente müssen belegt sein.

Eingefügte Modellelemente müssen valide sein. Werden Schemavariablen für Namen verwendet, so müssen diese innerhalb des Modells oder über eine explizite Zuweisung gegeben sein. Ist dies nicht der Fall besteht darüber hinaus die Möglichkeit den Wert für

diese Schemavariable als Parameter beim Anwenden der Transformation zu übergeben. Wird ein Wert nicht innerhalb der Transformation gesetzt, kann dies ein Fehler oder eine bewusste Entscheidung sein. Die folgende Kontextbedingung informiert den Transformationsentwickler darüber, dass der Wert als Parameter übergeben werden muss.

Bedingung:	Schemavariablen für Namen auf rechten Regelseiten müssen entweder im Pattern vorkommen, innerhalb des Zuweisungsblocks zugewiesen oder durch den Benutzer der Transformation übergeben werden.
Fehlercode:	0xF0C03
Schweregrad:	Information
Hinweis:	Eine Behebung dieser Meldung ist nicht notwendig, da es sich nur um eine Information für den Transformationsentwickler handelt.

```
1 [[                                                    MATrans
2   connect c -> m;
3 :-
4   component $monitor ; // 0xF0C03
5 ]]
```

Listing 4.43: Transformation, die die Kontextbedingung 0xF0C03 verletzt.

Listing 4.43 zeigt ein Beispiel, das diese Kontextbedingung verletzt. Die hier angegebene Transformationsregel fügt eine Subkomponente ein, für deren Name die Schemavariable `$monitor` verwendet wurde. Diese Schemavariable wird innerhalb der Transformation mit keinem Wert belegt, wodurch der Wert als Parameter an die Transformation übergeben werden muss. Eine Korrektur ist nicht notwendig.

Schemavariablen für eingefügte Modellelemente müssen belegt sein.

Eingefügte Modellelemente müssen valide sein. Werden Schemavariablen ohne die Angabe der Struktur des Elements verwendet, so müssen diese innerhalb des Modells oder über eine explizite Zuweisung gegeben sein. Ist dies nicht der Fall besteht darüber hinaus die Möglichkeit das Modellelement für diese Schemavariable als Parameter beim Anwenden der Transformation zu übergeben. Wird das Modellelement nicht innerhalb der Transformation gesetzt, kann dies einen Fehler oder eine bewusste Entscheidung darstellen. Die folgende Kontextbedingung weist den Transformationsentwickler darauf hin, dass das Modellelement als Parameter übergeben werden muss.

Bedingung:	Schemavariablen auf rechten Regelseiten müssen entweder im Pattern vorkommen, innerhalb des Zuweisungsblocks zugewiesen werden oder durch den Benutzer der Transformation übergeben werden.
Fehlercode:	0xF0C04
Schweregrad:	Information

```
[[                                                          MATrans
  connect c -> m;
:-
  Connector $C // ⓘ 0xF0C04
]]
```

Listing 4.44: Transformation, die die Kontextbedingung 0xF0C04 verletzt.

Listing 4.44 zeigt ein Beispiel, das diese Kontextbedingung verletzt. Die hier angegebene Transformationsregel fügt einen Konnektor ein, der an die Schemavariable `$C` gebunden ist, ohne dass diese im Pattern der Transformationsregel vorkommt. Dadurch muss der Konnektor als Parameter an die Transformation übergeben werden. Eine Korrektur ist nicht notwendig.

Schemavariablen im Application Constraint müssen existieren.

Der Application Constraint erlaubt die Formulierung einer Bedingung unter Verwendung der innerhalb der Transformationsregel definierten Schemavariablen. Damit die Transformationsregel anwendbar ist, muss dieser erfüllt sein. Eine Aussage über nicht belegte Schemavariablen ist nicht erlaubt und wird daher über eine Kontextbedingung verhindert. Das heißt, alle im Application Constraint verwendeten Schemavariablen müssen innerhalb der Transformationsregel definiert sein. Die folgende Kontextbedingung überprüft dies:

Bedingung:	Schemavariablen, die innerhalb des Application Constraints verwendet werden, müssen innerhalb der Transformationsregel existieren.
Fehlercode:	0xF0C05
Schweregrad:	Error

Listing 4.45 zeigt ein Beispiel, das diese Kontextbedingung verletzt. Die hier angegebene Transformationsregel sucht nach einer Komponente `Monitor`. Der Application Constraint fordert, dass der Wert der Schemavariable `$in` mit einem `M` beginnt. Diese Schemavariable ist nicht innerhalb der Transformationsregel definiert, wodurch die Transformationsregel invalide wird. Dieser Fehler kann korrigiert werden, indem die Schemavariable entweder innerhalb der Transformationsregel eingeführt wird oder der Constraint reformuliert wird.

```
component Monitor;                                          MATrans

where {
  $in.startsWith("M") // ✖ 0xF0C05
}
```

Listing 4.45: Transformation, die die Kontextbedingung 0xF0C05 verletzt.

Wertzuweisungen sind nur an existierende, nicht durch das Pattern Matching belegte Schemavariablen möglich.

Innerhalb der Zuweisungsblocks können Schemavariablen, die nicht durch das Pattern Matching belegt werden, Werte zugewiesen werden. Eine Zuweisung ist nur erlaubt, wenn der Wert nicht durch das Pattern Matching für die Transformationsregel bereits gesetzt wird. Das heißt, nur Schemavariablen, die ausschließlich auf der rechten Seite von Ersetzungen vorkommen, dürfen zugewiesen werden. Für die Übergabe von Werten für Schemavariablen innerhalb der Pattern werden die Parameter verwendet (vgl. Abschnitt 9.4). Die folgende Kontextbedingung überprüft dies:

Bedingung:	Schemavariablen, die innerhalb des Zuweisungsblock zugewiesen werden, müssen auf der rechten Regelseite von Ersetzungen existieren.
Fehlercode:	0xF0C06
Schweregrad:	Error

```
1 connect c -> [[$in :- foo]];                                    MATrans
2
3 assign {
4   $in = "msgIn"; // ⊗ 0xF0C06
5 }
```

Listing 4.46: Transformation, die die Kontextbedingung 0xF0C06 verletzt.

Listing 4.46 zeigt ein Beispiel, das diese Kontextbedingung verletzt. Die hier angegebene Transformationsregel sucht nach einem Konnektor und ändert dessen Ziel. Im Zuweisungsblock wir nun versucht, die Schemavariable `$in` zu setzen, die jedoch Teil des Pattern ist.

Keine anonymen Schemavariablen auf der rechten Regelseite.

Schemavariabeln auf der rechten Seite von Ersetzungen müssen mit Werten belegt werden, damit ein gültiges Modellelement dem Modell hinzugefügt werden kann. Anonyme Schemavariablen dienen lediglich als Platzhalter. Sie können weder als Parameter dienen noch können sie durch Zuweisungen mit Werten belegt werden. Auf der rechten Regelseite einer Ersetzung dürfen daher keine anonymen Schemavariablen vorkommen. Die folgende Kontextbedingung überprüft dies:

Bedingung:	Schemavariablen, die nur innerhalb der rechten Regelseite von Ersetzungen vorkommen, dürfen nicht anonym sein.
Fehlercode:	0xF0C15
Schweregrad:	Error

```
1  connect c -> [[$in :- $_]]; // ⊗ 0xF0C15                    MATrans
```

Listing 4.47: Transformation, die die Kontextbedingung 0xF0C15 verletzt.

Listing 4.47 zeigt ein Beispiel, das diese Kontextbedingung verletzt. Die hier angegebene Transformationsregel sucht nach einem Konnektor und ändert dessen Ziel, wobei für das neue Ziel eine anonyme Schemavariable verwendet wird. Dadurch is unklar, auf welches Ziel der Konnektor zeigen soll. Diese Transformationsregel ist nicht erlaubt.

4.14.2 Wohlgeformtheit bezüglich der Kombination von Operatoren

Einige Kombinationen aus Operatoren sind innerhalb der Transformationsregeln nicht erlaubt. Im Folgenden werden die Kontextbedingungen erläutert, die sicherstellen, dass nur erlaubte Kombinationen der Operatoren innerhalb der Transformationsregeln verwendet werden.

Keine negative Elemente einfügen.

Negative Elemente sind Elemente, die im Modell an der spezifizierten Stelle nicht vorkommen dürfen. Das heißt, sie schränken das Pattern einer Transformationsregel weiter ein. Modifikationen erlauben es dem Modell neue Modellelemente hinzuzufügen. Es können keine Elemente eingefügt werden, die nicht vorkommen dürfen. Daher sind negative Elemente auf der rechten Seite des Replacement Operators nicht erlaubt. Für Modifikationen, die Modellelemente verschieben, müssen die negativen Elemente innerhalb des Pattern, das das zu verschiebende Element beschreibt, spezifiziert werden. Die folgende Kontextbedingung überprüft dies:

Bedingung: Negative Elemente dürfen nicht erstellt bzw. hinzugefügt werden.

Fehlercode: 0xF0C07

Schweregrad: Error

```
1  component Controller {                                       MATrans
2    [[ :-
3      component Monitor {
4        not [[component $_;]] // ⊗ 0xF0C07
5      }
6    ]]
7  }
```

Listing 4.48: Transformation, die die Kontextbedingung 0xF0C07 verletzt.

Listing 4.48 zeigt ein Beispiel, das diese Kontextbedingung verletzt. Die hier angegebene Transformationsregel sucht eine Komponente `Controller` und fügt dieser eine innere Komponente `Monitor` hinzu. Ein negatives Element soll hier verbieten, dass die Komponente `Monitor` eine Subkomponente hat. Diese Spezifikation ist nicht erlaubt. Da die `Monitor` Komponente hinzugefügt wird und damit keine Subkomponente hat,

wenn keine angegeben ist, kann diese Transformationsregel korrigiert werden, indem das negative Element aus der Transformationsregel entfernt wird.

Keine Schachtelung von negativen Elementen.

Zur leichteren Verständlichkeit der Transformationsregeln erlauben die DSTLs keine Schachtelung von negativen Elementen, d.h. es dürfen keine negativen Elemente innerhalb von negativen Elementen spezifiziert werden. Die hiermit verbundene geringere Ausdrucksmächtigkeit gleicht der Application Constraint aus, der es erlaubt Constraints auch über negative Elemente zu formulieren. Dies überprüft folgende Kontextbedingung:

Bedingung:	Negative Elemente dürfen nicht verschachtelt werden.
Fehlercode:	0xF0C08
Schweregrad:	Error
Hinweis:	Der Application Constraint ermöglicht es negative Elemente weiter einzuschränken.

```
1 component Controller {                                          MATrans
2   not [[ component Monitor {
3      not [[ component $_; ]] // ⊗ 0xF0C08
4   } ]]
5 }
```

Listing 4.49: Transformation, die die Kontextbedingung 0xF0C08 verletzt.

Listing 4.49 zeigt ein Beispiel, das diese Kontextbedingung verletzt. Die hier angegebene Transformationsregel sucht eine Komponente `Controller` und fordert, dass diese keine innere Komponente `Monitor` hat. Ein negatives Element soll hier verbieten, dass die Komponente `Monitor` eine Subkomponente hat. Diese Spezifikation ist nicht erlaubt, da die `Monitor` Komponente ein negatives Element ist. Innerhalb dieses Elements darf kein weiteres negatives Element spezifiziert werden. Stattdessen muss der Application Constraint die Subkomponente innerhalb der Monitorkomponente verbieten.

Keine Modifikation negativer Elemente.

Negative Elemente sind Elemente, die im Modell an der spezifizierten Stelle nicht vorkommen dürfen. Sie schränken das Pattern einer Transformationsregel weiter ein. Eine Modifikation nicht vorhandener Elemente ist nicht möglich. Daher verbieten die DSTLs die Verwendung des keine Replacement Operators innerhalb von negativen Elementen. Die folgende Kontextbedingung überprüft dies:

Bedingung:	Negative Elemente können nicht verändert werden.
Fehlercode:	0xF0C09
Schweregrad:	Error

```
component Controller {                                          MATrans
  not[[
    component Monitor {
      [[ :- component Foo;]] // ✖ 0xF0C09
     }
  ]]
}
```

Listing 4.50: Transformation, die die Kontextbedingung 0xF0C09 verletzt.

Listing 4.50 zeigt ein Beispiel, das diese Kontextbedingung verletzt. Die hier angegebene Transformationsregel sucht eine Komponente `Controller` und fordert, dass diese keine innere Komponente `Monitor` hat. Innerhalb der Monitorkomponente soll eine Subkomponente `Foo` eingefügt werden. Diese Spezifikation ist nicht erlaubt, da die `Monitor` Komponente ein negatives Element ist und nicht verändert werden kann.

Keine optionalen Elemente einfügen.

Optionale Elemente sind Elemente, die im Modell an der spezifizierten Stelle vorkommen können aber nicht müssen. Modifikationen erlauben es, dem Modell neue Modellelemente hinzuzufügen. Beim Einfügen von optionalen Elementen wäre es nicht eindeutig, ob ein Element hinzugefügt würde oder nicht. Daher ist der Optional Operator auf der rechten Seite des Replacement Operators nicht erlaubt. Für Modifikationen, die Modellelemente verschieben, müssen die optionalen Elemente innerhalb des Pattern, das das zu verschiebende Element beschreibt, spezifiziert werden. Die folgende Kontextbedingung überprüft dies:

Bedingung:	Optionale Elemente dürfen nicht erstellt bzw. hinzugefügt werden.
Fehlercode:	0xF0C10
Schweregrad:	Error
Hinweis:	Als Korrektur reicht in der Regel die Entfernung des optionalen Elements aus der Transformationsregel.

```
component Controller {                                          MATrans
  [[ :-
    component Monitor {
      opt [[component $_;]] // ✖ 0xF0C10
    }
  ]]
}
```

Listing 4.51: Transformation, die die Kontextbedingung 0xF0C10 verletzt.

Listing 4.51 zeigt ein Beispiel, das diese Kontextbedingung verletzt. Die hier angegebene Transformationsregel sucht eine Komponente `Controller` und fügt dieser eine

innere Komponente `Monitor` hinzu. Der Optional Operator wird hier verwendet, um auszudrücken, dass die Komponente `Monitor` optional eine Subkomponente hat. Diese Spezifikation ist nicht erlaubt.

Keine optionalen Elemente innerhalb von negativen Elementen.

Optionale Elemente sind Elemente, die im Modell an der spezifizierten Stelle vorkommen können aber nicht müssen. Negative Elemente verbieten Modellelemente. Optionale Elemente innerhalb von negativen Elementen bieten keinen Mehrwert, da sie nicht vorkommen müssen. Das heißt, ein negatives Element wird bereits dann gefunden, wenn das darin enthaltene optionale nicht gefunden wird. Da sich solche Spezifikationen zusätzlich negativ auf die Laufzeit der Transformation auswirken können, erlauben die DSTLs solche Spezifikationen nicht. Die folgende Kontextbedingung überprüft dies:

Bedingung:	Der Optional Operator darf nicht innerhalb von negativen Elementen vorkommen.
Fehlercode:	0xF0C11
Schweregrad:	Error

```
1 component Controller {                                          MATrans
2   not[[
3     component Monitor {
4       opt [[component $_;]] // ✖ 0xF0C11
5     }
6   ]]
7 }
```

Listing 4.52: Transformation, die die Kontextbedingung 0xF0C11 verletzt.

Listing 4.52 zeigt ein Beispiel, das diese Kontextbedingung verletzt. Die hier angegebene Transformationsregel sucht eine Komponente `Controller`, die keine innere Komponente `Monitor` hat. Der Optional Operator wird hier verwendet, um auszudrücken, dass die Komponente `Monitor` optional eine Subkomponente hat. Diese Spezifikation ist nicht erlaubt.

Keine leeren Ersetzungen

Beim Replacement Operator sind sowohl die linke als auch die rechte Seite optional. Dadurch können leere Ersetzungen, d.h. Ersetzungen bei denen sowohl die linke als auch die rechte Regelseite leer sind, formuliert werden. Eine solche Ersetzung ist zwar kein Fehler, sie kann aber auf einen solchen hindeuten. Zum Beispiel kann der Transformationsentwickler vergessen haben, ein Element hinzuzufügen oder zu entfernen. Der Transformationsentwickler muss daher hierauf hingewiesen werden. Die folgende Kontextbedingung prüft dies:

Bedingung:	Ersetzungen sollten nicht leer sein.
Fehlercode:	0xF0C12
Schweregrad:	Warnung
Hinweis:	Eine leere Ersetzung deutet auf ein versehentlich weggelassenes, zu entfernendes oder hinzuzufügendes Modellelement hin.

```
1 component Controller {                                    MATrans
2   [[ :- ]] // ⚠ 0xF0C12
3 }
```

Listing 4.53: Transformation, die die Kontextbedingung 0xF0C12 verletzt.

Listing 4.53 zeigt ein Beispiel, das diese Kontextbedingung verletzt. Die hier angegebene Transformationsregel sucht eine Komponente `Controller` innerhalb dieser Komponente findet eine leere Ersetzung statt. Die Kontextbedingung weist den Transformationsentwickler auf diese Ersetzung hin.

4.14.3 Konventionen

Konventionen können ebenfalls über Kontextbedingungen geprüft werden. Kontextbedingungen für Konventionen produzieren Warnungen, um den Transformationsentwickler drüber zu informieren, dass er entgegen der Konventionen arbeitet.

Schemavariablen für Namen werden klein geschrieben.

Eine Konvention, die aus [Wei12] adaptiert wurde, ist, dass Schemavariablen für Namen klein geschrieben werden sollten. Die folgende Kontextbedingung prüft dies:

Bedingung:	Schemavariablen, die für Namen verwendet werden, sollten klein geschrieben werden.
Fehlercode:	0xF0C13
Schweregrad:	Information

```
1 component $MONITOR { // ⓘ 0xF0C13                         MATrans
2 }
```

Listing 4.54: Transformation, die die Kontextbedingung 0xF0C13 verletzt.

Listing 4.54 zeigt ein Beispiel, das diese Kontextbedingung verletzt. Die hier angegebene Transformationsregel sucht eine Komponente und verwendet für deren Namen die Schemavariable `$MONITOR`. Die Kontextbedingung weist den Transformationsentwickler auf die Konvention hin.

Schemavariablen für Modellelemente beginnen mit Großbuchstaben.

Eine Konvention, die aus [Wei12] adaptiert wurde ist, das Schemavariablen für Modellelemente groß geschrieben werden sollten. Die folgende Kontextbedingung prüft dies:

Bedingung:	Schemavariablen, die für Modellelemente verwendet werden, sollten groß geschrieben werden.
Fehlercode:	0xF0C14
Schweregrad:	Information

```
1 $c [[ // 0xF0C14                                          MATrans
2   component $_ {
3   }
4 ]]
```

Listing 4.55: Transformation, die die Kontextbedingung 0xF0C14 verletzt.

Listing 4.55 zeigt ein Beispiel, das diese Kontextbedingung verletzt. Die hier angegebene Transformationsregel sucht eine Komponente und bindet diese an die Schemavariable `$c`. Die Kontextbedingung weist den Transformationsentwickler auf die Konvention hin.

4.14.4 Application Constraint, Zuweisungen und Direkte Manipulation

Der Application Constraint, der Zuweisungsblock sowie der Anweisungsblock basieren auf Java-Expressions. Neben den hier bereits vorgestellten Kontextbedingungen, die den Constraint und die Zuweisungen betreffen, gelten für diese Teile der Transformation die für Java gültigen Kontextbedingungen [Ora17].

4.15 Diskussion und verwandte Arbeiten

Die am stärksten mit dieser Arbeit verwandte Arbeit ist die als Grundlage dienende Arbeit von Ingo Weisemöller [Wei12]. Die grundlegenden Konzepte dieser Arbeit, wie beispielsweise die Formulierung von Pattern in konkreter Syntax, die Art der Formulierung von Ersetzungen, Constraints und die Grundfunktion der negativen Elemente sowie des Listenoperators wurden beibehalten. Darauf aufbauend wurde der Operator zum Negieren von Elementen erweitert und der Listenoperator zum Collection Operator weiterentwickelt (vgl. GR 9 und GR 10). Statt einzelne Patternelemente zu negieren bzw. den Collection Operator auf einzelne Patternelemente anzuwenden, können diese Operatoren auf Teilpattern angewendet werden. Ein Beispiel für ein Teilpattern ist eine Klasse inklusive der enthaltenen Attribute. Des Weiteren wurde der Optional Operator als Feature hinzugefügt (vgl. GR 7.2). Dieser ermöglicht es Patternelemente als optional zu markieren, wodurch sich Transformationen zusammenfassen lassen (vgl. GR 7). Ein Zuweisungsblock erlaubt Schemavariablen auf der rechten Regelseite von Ersetzungen mit (möglicherweise berechneten) Werten zu belegen (vgl. GR 12 und GR 8), wodurch beispielsweise die Berechnung von Namen für hinzugefügte Methoden möglich

wird. Der hinzugefügte Anweisungsblock ermöglicht dem Transformationsentwickler sowohl zwischen deklarativer und imperativer Beschreibung von Modellveränderungen zu wählen als auch Analysen oder Reporting in die Transformationen zu integrieren (vgl. GR 11). Zusätzlich wurde Typinferenz für Schemavariablen ermöglicht (vgl. GR 3 und GR 3.2). Für die auf Java basierenden Anteile der Transformationsregeln wurden außerdem eine Vielzahl an Built-Ins geschaffen und diesen alle in einer Transformationsregel verwendeten Schemavariablen zugänglich gemacht (vgl. GR 12). Um die Entwicklung wohlgeformter Transformationsregeln zu ermöglichen, wurden zudem eine Reihe von Kontextbedingungen entwickelt, die die Wohlgeformtheit sicherstellen (vgl. GR 3 und GR 3.1). Schließlich wurde eine Integration mit der Klasse `GlobalExtensionManagement` geschaffen, die die Nutzung des Templateerweiterungsmechanismus, wie er in [Rot17] entwickelt wurde, ermöglicht (vgl. GR 14).

Die Designentscheidung keine Schachtelung von negativen Elementen zu unterstützen schränkt die Möglichkeiten zur Patterndefinition scheinbar ein. Durch den Application Constraint wird diese Einschränkung jedoch relativiert. Der Application Constraint basiert auf Java und erlaubt die Verwendung von Hilfsklassen. Diese können verwendet werden, um weitere Eigenschaften der negativen Elemente zu überprüfen. Die Application Constraints können somit auch das Vorkommen bestimmter Elemente innerhalb der negativen Elemente einschränken.

Die DSTL kombinieren die konkrete Syntax der Modellierungssprache mit zusätzlichen Transformationsoperatoren wie dem Replacement Operator in doppelten eckigen Klammern. Abhängig von der konkreten Syntax der Modellierungssprache kann die Syntax der Operatoren mit der Syntax der Modellierungssprache in Konflikt stehen. In diesem Fall können die Spracherweiterungsmechanismen von MontiCore genutzt werden, um eine Subsprache zu definieren, in der die konkrete Operatorensyntax redefiniert wird.

In [Wei12] wurde für MontiCore – ähnlich wie beispielsweise in PROGRES [Sch91], eMoflon [LAS14] und Henshin [ABJ+10, SBG+17], GReAT [BNvBK06], AGG [Tae04] und viele weitere [KC15] – ein Graphtransformationsansatz [CH06] gewählt. Dieser spiegelt sich beispielsweise in der Notwendigkeit eine Folding Operators wider, hat jedoch den Vorteil, dass der Ansatz auch für nicht baumförmige Modelle (z.B. nach dem Auflösen von Referenzen zu tatsächlichen Verbindungen innerhalb der Modelle) verwendet werden kann. In dieser Arbeit wurde der Ansatz zu einem hybriden Ansatz erweitert, der den Graphtransformationsansatz um imperative Features erweitert [KC15]. Einen hybriden Ansatz verfolgen auch MOLA [KBC05], VMTS [LLMC05], Fujaba [FNTZ00] und GrGen.NET [JBK10]. Anders als diese Ansätze basiert der in dieser Arbeit verwendete Ansatz auf der konkreten statt abstrakten Syntax der Modellierungssprachen.

Die meisten Ansätze zur Transformation in konkreter Syntax sind für graphische Modellierungssprachen konzipiert [ASS16, SVM+13, Grø09, GMP09, GMPO09b, Sch07, RKR+06]. Auf Grund der textuellen Natur der Modellierungssprachen, für die der hier vorgestellte Ansatz konzipiert ist, wurde zur Transformationsbeschreibung eine textuelle Syntax gewählt. Die Ansätze zur Verwendung textueller konkreter Syntax, auch als „Concrete Object Syntax“ bezeichnet, stammen aus dem Bereich Programmtransformation und nutzen statt Graphersetzung einen Termersetzungsansatz [Vis02, vdBHKO02,

KvdSV09, HKV12]. Diese Ansätze eignen sich besonders gut für Baumstrukturen, wie sie in Programmiersprachen üblich sind, wohingegen sich Graphtransformationsansätze auch für Graphstrukturen eignen. MontiTrans unterstützt mit dem umgesetzten Graphtransformationsansatz auch das Matching von Pattern in unzusammenhängenden Graphen bzw. mehreren Modellen.

Ein weiterer ähnlicher Bereich ist die Inferenz von Modelltransformation aus Beispielen [SGW15, KLR+12, LWK10, BLS+09, SWG09, NMSS09]. Hierbei wird die Modelltransformation aus Beispielmodellen vor und nach der Transformation abgeleitet. Dadurch muss der Nutzer sich zunächst nicht mit der abstrakten Syntax beschäftigen. Die hierdurch entstehende Transformation muss in der Regel jedoch noch weiter verfeinert bzw. generalisiert werden, um die nötige Abstraktion in die Transformation zu integrieren bzw. deren Anwendbarkeit zu beschränken. Diese Anpassung erfolgt auf Basis der abstrakten Syntax.

Kapitel 5

Domänenspezifische Transformationssprachen für Klassendiagramme und MontiArc-Modelle

In Kapitel 4 wurden die verschiedenen Operatoren der DSTLs anhand einer exemplarischen, mit MATrans formulierten Transformationsregel vorgestellt. Neben der Erweiterung der Struktur und der Operatoren der DSTLs, wurden in dieser Arbeit DSTLs für MontiCore-basierte Modellierungssprachen wie beispielsweise CD4Analysis und MontiArc entwickelt. Mit Hilfe dieser Sprachen wird dem Transformationsentwickler, der diese Sprache verwendet, eine intuitive Möglichkeit der Beschreibung von Modelltransformationen bereitgestellt. Darüber hinaus wurden wiederverwendbare Modelltransformationen entwickelt und in Bibliotheken zusammengefasst. Das Ziel von Kapitel 4 war die Erläuterung der verschiedenen Operatoren und der allgemeinen Struktur von Transformationen und Transformationsregeln. Im Gegensatz dazu werden in diesem Kapitel die verschiedenen, entwickelten DSTLs und die durch sie bereitgestellte Syntax erklärt. Dazu wird jeweils an Beispielen erläutert, wie sich aus der Modellierungssprache die Syntax der jeweiligen Transformationssprache ergibt. Abbildung 5.1 zeigt erneut die Übersicht der DSTL- und Transformationsentwicklung mit MontiTrans. Hervorgehoben ist der in diesem Kapitel betrachtete, mittlere Teil. Dieses Kapitel adressiert den Transformationsentwickler. In Kapitel 6 werden Regeln und die Systematik zur Erstellung neuer DSTLs nach dem hier vorgestellten Schema erläutert. Wiederverwendbare Transformationen für die hier vorgestellten DSTLs werden in Kapitel 7 vorgestellt.

Die wichtigsten Ergebnisse dieses Kapitels sind:

- Die DSTL CDTrans zur Transformation von CD4Analysis- und CD4Code-Modellen.
- Die DSTL MATrans zur Transformation von MontiArc-Modellen.
- Die Erläuterung der Herkunft der Syntax von CDTrans und MATrans.
- Die Beschreibung der DSTL MACDTrans als Kombination der DSTLs CDTrans und MATrans.
- Die Erklärung der Entwicklung exogener DSTLs mit MontiTrans.

Dieses Kapitel stellt somit sowohl ein eigenständiges Forschungsergebnis als auch die Demonstration der Anwendbarkeit der Ergebnisse aus Kapitel 4 und Kapitel 6 dar.

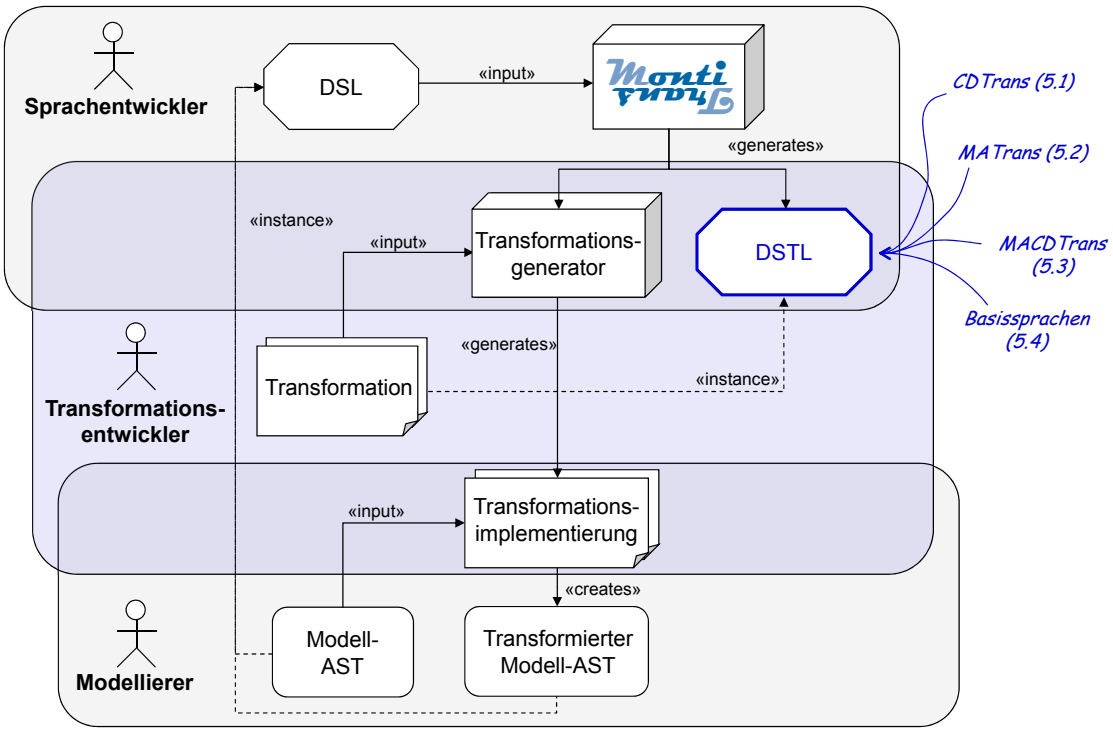


Abbildung 5.1: Übersicht über die verschiedenen Aspekte von MontiTrans mit Fokus auf den in diesem Kapitel vorgestellten Teil (Die DSTLs CDTrans, MATrans und MACDTrans zur Transformation von Klassendiagrammen und MontiArc-Modellen).

5.1 CDTrans: Eine DSTL für Klassendiagramme

CDTrans ist eine DSTL für Modelle der Sprachen *CD4Analysis* und *CD4Code*. Bei diesen Klassendiagrammsprachen handelt es sich um reduzierte Formen der Klassendiagrammsprache der UML/P [Rum16, Sch12]. Ziel von CDTrans ist es, dem Modellierer eine komfortable Möglichkeit zu bieten, Transformationen auf Modellen der DSLs CD4Analysis und CD4Code zu beschreiben. Die beiden Sprachen – CD4Analysis und CD4Code – sind durch die gleiche Grammatik (vgl. Anhang C.1) definiert. Die beiden Sprachen unterscheiden sich darin, dass für CD4Analysis stärkere Einschränkungen durch Kontextbedingungen gemacht werden, sodass diese im Vergleich zu CD4Code keine Methoden, Konstruktoren sowie keine Sichtbarkeiten (`private`, `public`, `protected`) der Attribute erlaubt. Im Folgenden wird in Abschnitt 5.1.1 CD4Code und die daraus resultierende DSTL beschrieben. Anschließend werden in Abschnitt 5.1.2 CD4Code-spezifische Erweiterungen der abgeleiteten gegenüber der allgemeinen Struktur von DSTLs – wie sie in Abschnitt 5.1.1 und Kapitel 4 erläutert wurde – vorgestellt.

5.1.1 CD4Code und CDTrans

In diesem Abschnitt wird jeweils an einer viergeteilten Abbildung die Herkunft der einzelnen Aspekte der klassendiagrammspezifischen Syntax entsprechend der systematischen Ableitung (vgl. Kapitel 6) erläutert (vgl. Abbildung 5.2 - 5.8). Dazu ist jeweils oben links der relevante Ausschnitt des Klassendiagramms `SocNet` dargestellt. Unten links ist der CD4Code-Grammatikausschnitt abgebildet, der die Syntax des Klassendiagramms definiert. Oben rechts zeigt die Abbildung die Transformationsregel und unten rechts den dazu passenden Ausschnitt der Grammatik der CDTrans DSTL. Die Grammatik mit allen Nichtterminalen der DSTL ist im Anhang zu finden (vgl. Anhang C.2). Als Beispieltransformation wird die Pull Up Attribute Transformation, die auch Teil der Transformationsbibliothek ist (vgl. Abschnitt 7.1.1), verwendet. Die Transformation findet gleiche Attribute in Klassen mit gemeinsamer Superklasse. Anschließend verlagert die Transformation die gemeinsamen Attribute in die Superklasse und entfernt sie aus den Subklassen. Zur Demonstration wird die Transformationsregel schrittweise aufgebaut. Dazu muss zunächst ein Pattern entwickelt werden, das den entsprechenden Teil des Modells bestehend aus der Super- und den Subklassen sowie dem zu verschiebenden Attribut beschreibt. Dieses Pattern wird zunächst sehr modellspezifisch formuliert (vgl. Abbildung 5.2 und 5.3) und dann schrittweise generalisiert. Es werden Schemavariablen (vgl. Abbildung 5.4), ein Application Constraint (vgl. Abbildung 5.5), negative Elemente (vgl. Abbildung 5.6) und der Collection Operator (vgl. Abbildung 5.7) verwendet, um das Pattern allgemein anwendbar zu gestalten. Schließlich wird noch die Modifikation hinzugefügt (vgl. Abbildung 5.8).

Das Beispieleingabemodell `SocNet` zeigt eine abstrakte Klasse `Profile` sowie deren Subklassen `Person` und `Group`. Beide Subklassen haben das Attribut `String address`. Die zur Demonstration schrittweise aufgebaute Transformation entfernt das gemeinsame Attribut aus den beiden Subklassen `Person` und `Group` und fügt es der Superklasse `Profile` hinzu.

Abbildung 5.2 zeigt den ersten Schritt zur Definition des Patterns. Um das Pattern zu formulieren, wurde der relevante Ausschnitt aus dem Modell in das Pattern übernommen. Im Klassendiagramm sind dies die Klassen `Profile`, `Person` und `Group` sowie die beiden Attribute. Die erlaubte Syntax für Klassen im Klassendiagramm ergibt sich aus der Produktion `CDClass` der Klassendiagrammgrammatik (unten links). Das Pattern sieht für diesen Modellteil zunächst fast identisch aus. Wie in Abschnitt 4.3 beschrieben, muss ein Pattern nur die Elemente und die Eigenschaften dieser Elemente beschreiben, die für die Transformation relevant sind. Der Modifier `abstract` und die „Hülle“, also `classdiagram SocNet { ... }`, wurden weggelassen, da

1. die angestrebte Transformationsregel sowohl für eine abstrakte als auch nicht abstrakte Superklasse Profil anwendbar und
2. nicht auf das Klassendiagramm `SocNet` beschränkt sein soll.

Um für das Pattern die gleiche Syntax wie im Modell verwenden zu können, muss die konkrete Syntax der DSL CD4Code in die DSTLs CDTrans übernommen werden.

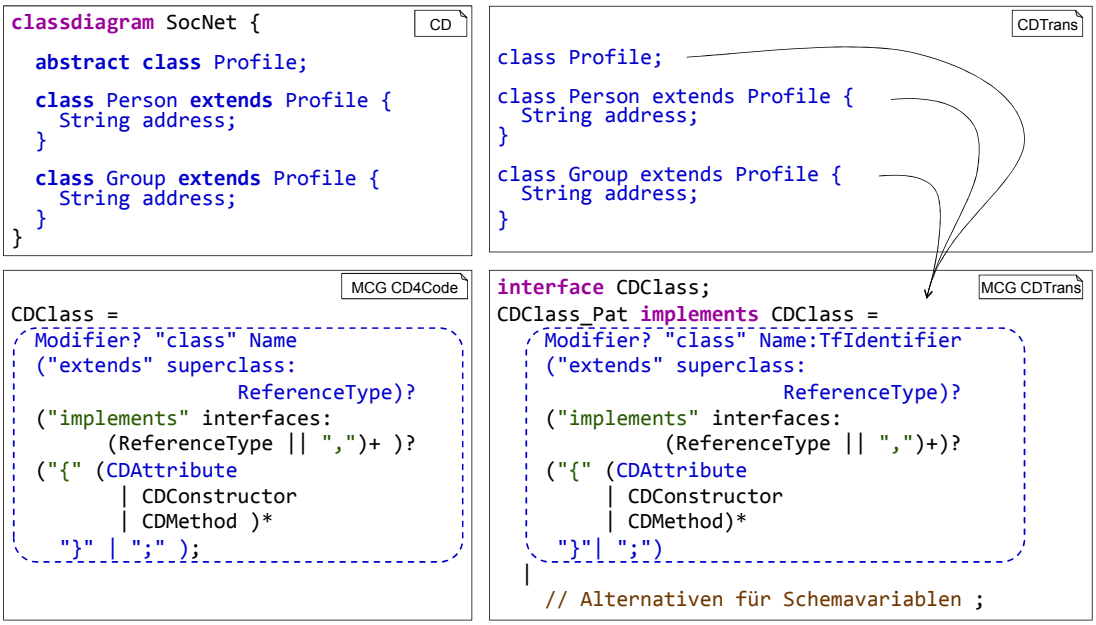


Abbildung 5.2: Matchen der Klassen in konkreter Syntax (Fokus: Klassen).

Der Rumpf der Produktion `CDClass` wurde daher in den Rumpf der Produktion `CDClass_Pat` übernommen. Hier formt es eine der insgesamt drei Alternativen. Die anderen (nicht gezeigten) Alternativen erlauben die Verwendung von Schemavariablen, die später erläutert wird. Die gestrichelten Kästen in den unteren beiden Listings markieren die übernommene Syntax. `CDClass_Pat` ermöglicht die Beschreibung von Pattern für Klassen. Ein im Modell gültiges Elemente ist daher immer auch ein gültiger Teil des Patterns. Für jedes Klassendiagrammelement, wie zum Beispiel Klassen, Attribute und Assoziationen, umfasst die CDTrans-Grammatik ein `_Pat`-Nichtterminal, das die Syntax des entsprechenden Nichtterminals der CD4Code-Grammatik bereitstellt. Die allgemeine Ableitung der `_Pat`-Nichtterminale findet sich in Kapitel 6.

Damit nicht nur die Syntax der einzelnen Patternelemente sich an der Syntax der entsprechenden Modellelemente orientiert, sondern auch die Struktur des Pattern an der Struktur der Modelle, muss auch die Struktur der Sprache übernommen werden. Dadurch kann zum einen der Ausschnitt aus dem Modell übernommen werden und zum anderen ist Nutzern von CD4Code diese Struktur vertraut. Genau wie in Klassendiagrammen können daher Klassen auch im Pattern Attribute enthalten. In der Klassendiagrammgrammatik erlaubt dies die Produktion `CDClass` durch das verwendete Nichtterminal `CDAttribute`. In der CDTrans-Grammatik bildet sich diese Beziehung ähnlich ab (vgl. Abbildung 5.3). `CDClass_Pat` verweist auf das Nichtterminal `CDAttribute`. In der CDTrans Grammatik gibt es zu jedem Nichtterminal der CD4Code-Grammatik ein entsprechendes Interface-Nichtterminal, das den gleichen Namen hat wie das Nichtterminal der CD4Code-Grammatik. Die `_Pat`-Nichtterminale implementieren das jeweilige Interface-Nichtterminal (vgl. Abbildung 5.3). Außerdem implementieren die Nichttermi-

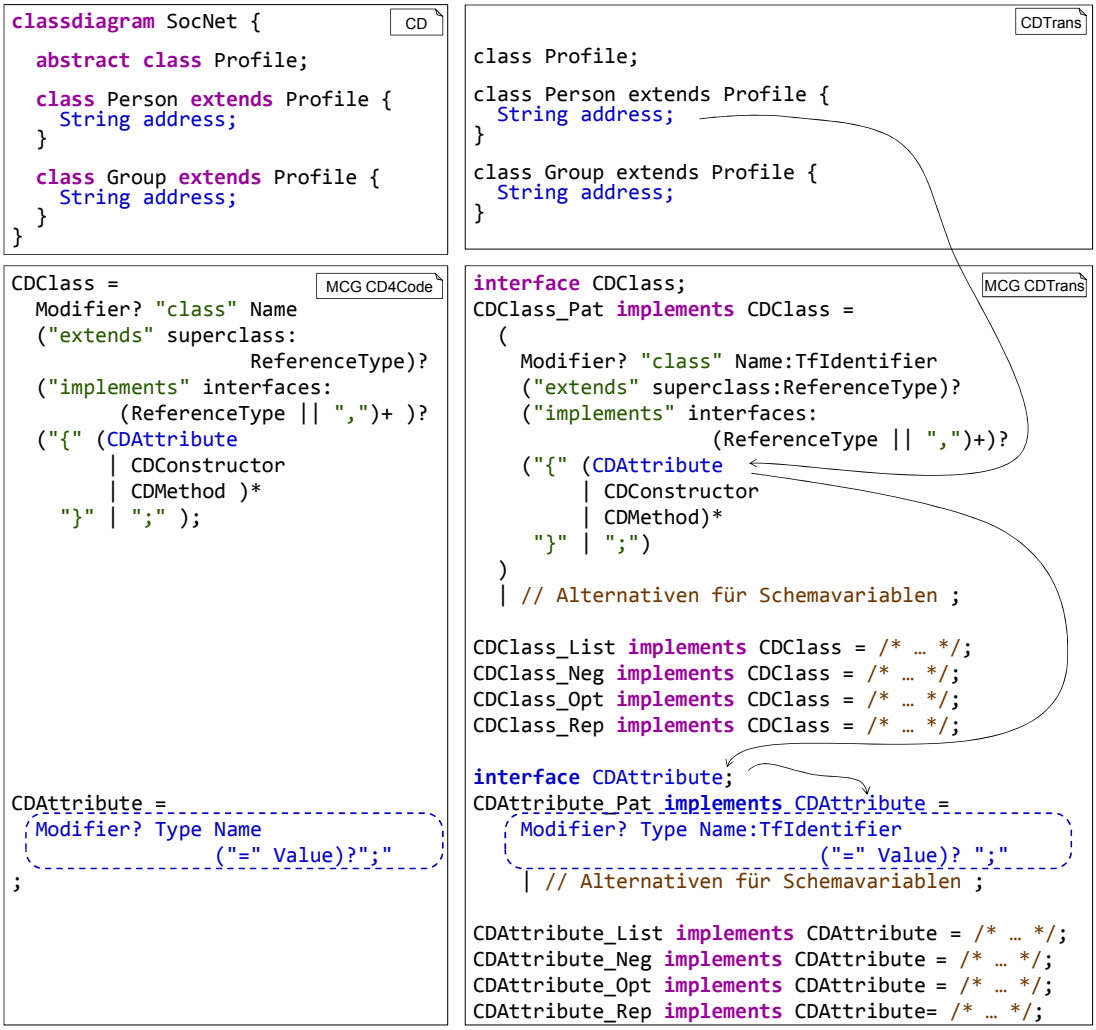


Abbildung 5.3: Matchen der Klassen in konkreter Syntax (Fokus: Attribute).

nale für die verschiedenen Operatoren (vgl. Kapitel 4) dieses Interface-Nichtterminal: Collection (Suffix `_List`), Optional, (Suffix `_Opt`), Negative Elemente (Suffix `_Neg`), und der Replacement Operator (Suffix `_Rep`)).

Wie in Abschnitt 4.4 beschrieben, können an bestimmten Stellen im Pattern Schemavariablen verwendet werden, um beispielsweise vom Eingabemodell zu abstrahieren und die Transformation dadurch zu generalisieren. Hierbei gibt es zwei Arten von Schemavariablen: solche für Namen und solche für ganze Modellelemente. In Abbildung 5.4 werden für die Namen der Klassen die Schemavariablen `$_` und `$parent` sowie für die kompletten Attribute die Schemavariable `$A1` verwendet. An allen Stellen, an denen im Modell ein Name vorkommt, kann die einfache Form der Schemavariablen bestehend aus einem `$`-Zeichen und einem Bezeichner für die Schemavariable – im Fall

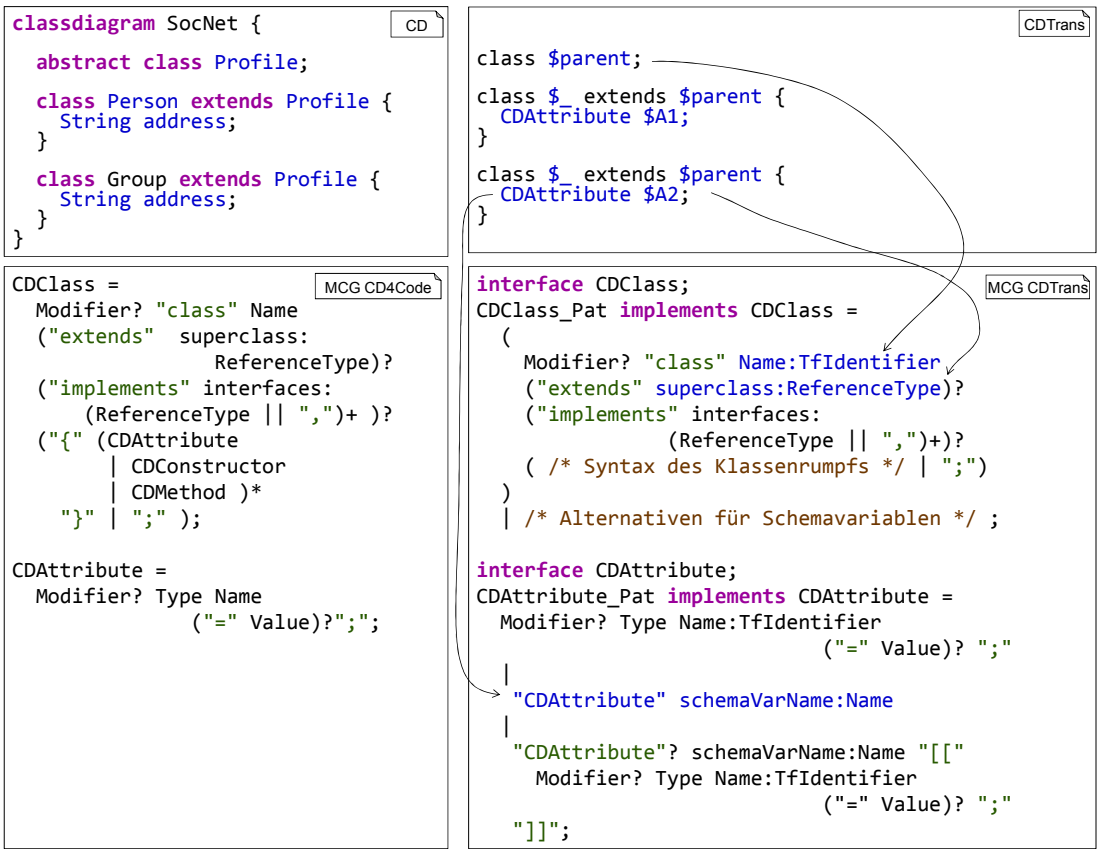


Abbildung 5.4: Verwendung von Schemavariablen für Attribute und Klassennamen.

einer anonymen Schemavariable ein Unterstrich `_` – verwendet werden. Beispiele hierfür sind der Name des Klassendiagramms, einer Klasse oder von Attributen. In der CDTrans-Grammatik spiegelt sich dies dadurch wider, dass in der aus den Produktionen der CD4Code-Grammatik übernommenen Syntax alle Vorkommen des Nichtterminals `Name` durch das Nichtterminal `TFIdentifier` ersetzt wurden. Dieses erlaubt weiterhin einfache Namen und zusätzlich auch Schemavariablen und Ersetzungen von Namen. Im Beispiel in Abbildung 5.4 wird dies für den Namen der Klasse verwendet. CDTrans erlaubt anstelle eines Modellelements Schemavariablen für dieses zu verwenden. Alternativ zur Angabe eines Patternelements wie einer Klasse oder einem Attributes, kann im Pattern eine Schemavariable angegeben werden. Dies bildet die zweite Alternative im Rumpf der `CDAttribute_Pat` Produktion. In Abbildung 5.4 wird diese für die Attribute verwendet.

Die dritte – in der Beispieltransformation nicht verwendete – Alternative des Rumpfs erlaubt die in Abschnitt 4.4.3 vorgestellte Kombination aus Schemavariable und Modellsyntax. Die Alternative erlaubt es beispielsweise `$A2 [[ String address; ]]` aufzuschreiben. Auch hierfür wurde der Rumpf der Produktion der CD4Code-Grammatik übernommen und Vorkommen des Name Nichtterminals entsprechend ersetzt.

Abbildung 5.5: Application Constraint, der die Übereinstimmung der Attribute prüft.

Durch die Generalisierung in Abbildung 5.4 wurde von der konkreten Struktur der Attribute abstrahiert und somit die Transformationsregel allgemeingültiger formuliert. Ein Application Constraint bietet die Möglichkeit über die verwendeten Schemavariablen Aussagen zu formulieren (vgl. Abschnitt 4.11) und damit die Anwendbarkeit der Transformationsregel einzuschränken [EH86, HHT96, BH02, HP05]. Der Application Constraint basiert auf Java und erlaubt die Verwendung von Methoden der abstrakten Syntax der Modellelemente. In der Transformationsregel in Abbildung 5.5 wurde das Pattern um einen Application Constraint ergänzt, der ausdrückt, dass die Attribute, die an die Schemavariablen `$A1` und `$A2` gebunden sind, gleich sind. `deepEquals` ist eine Methode, die jedes Modellelement bietet und die den Vergleich zweier Modelldemente erlaubt. Der Application Constraint verwendet keine klassendiagrammspezifische konkrete Syntax und wird daher von der Basisgrammatik `TFCommons` zur Verfügung gestellt (vgl. Abschnitt 6.3). Eingebunden in die CDTrans-Grammatik wird der Application Constraint durch das Startsymbol `TFRule` der CDTrans-Grammatik. Dieses erlaubt zunächst das Pattern in konkreter Syntax zu beschreiben und anschließend optional einen Application Constraint zu ergänzen (Nichtterminal `TFWhere`). Die Produktion `TFWhere` verwendet das aus der JavaDSL stammende Nichtterminal `Expression` für den Constraint. Analog werden das Folding (vgl. Abschnitt 4.9), der Anweisungsblock (vgl. Abschnitt 4.12) und der Zuweisungsblock (vgl. Abschnitt 4.10) von der Basisgrammatik zur Verfügung gestellt und durch das Startsymbol der CDTrans Grammatik eingebunden.

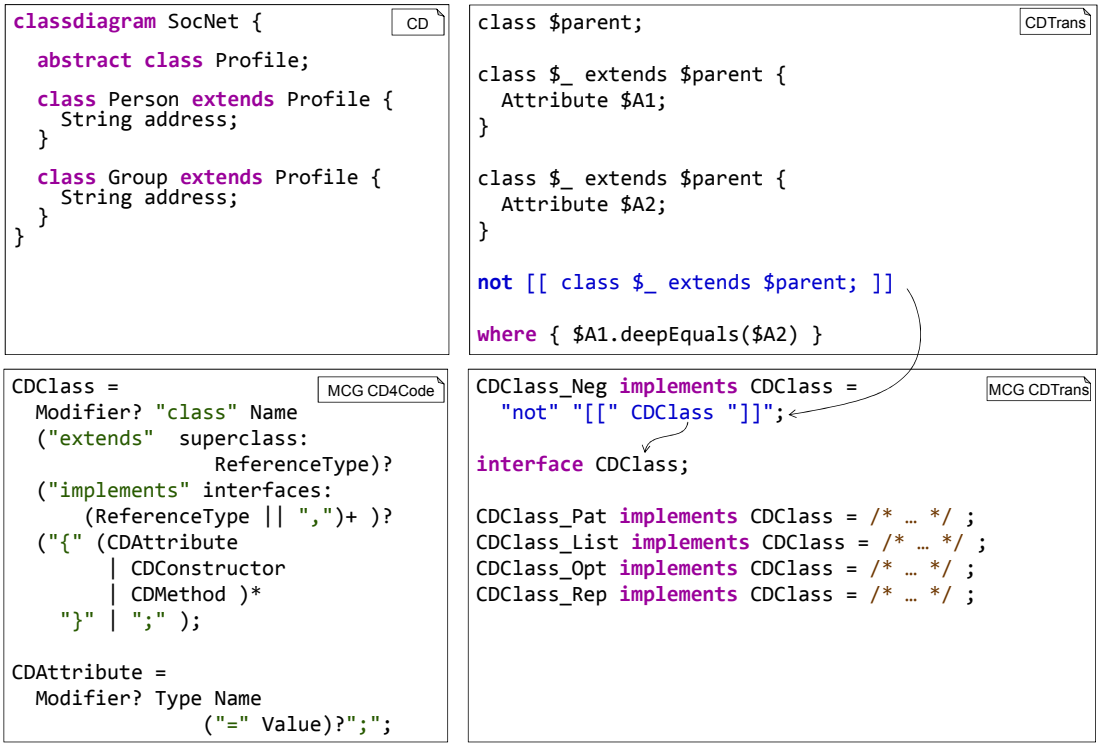


Abbildung 5.6: Verwendung eines negativen Elements, um weitere Subklassen ohne passendes Attribut zu verbieten.

Um das Pattern weiter zu generalisieren, darf die betrachtete Pull up Attribute Transformationsregel nur dann anwendbar sein, wenn es keine andere Subklasse gibt, die das Attribute nicht enthält. Um auszudrücken, dass bestimmte Elemente nicht vorkommen dürfen, bietet CDTrans negative Elemente (vgl. Abschnitt 4.6), diese entsprechen Negative Application Conditions [HHT96]. Das Pattern wurde um ein negatives Element ergänzt, das weitere Subklassen im Modell verbietet (vgl. Abbildung 5.6, oben rechts). Die Syntax hierzu ist `not [[ ]]`, wobei zwischen den doppelten eckigen Klammern ein Modellelement in konkreter Syntax beschrieben wird. Im Beispiel in Abbildung 5.6 wurde dies für eine Klasse gemacht. Dies ermöglicht das Nichtterminal `CDClass_Neg`. `CDClass_Neg` definiert hierbei die Syntax `not [[ ]]` und verweist auf das Nichtterminal `CDClass` innerhalb von CDTrans. In gleicherweise kann jedes Modellelement negiert werden, also beispielsweise auch Attribute, Methoden oder Interfaces. Dies ist möglich, da für jedes Nichtterminal der Modellierungssprache ein `_Neg`-Nichtterminal existiert, das auf das zugehörige Interface-Nichtterminal verweist. In ähnlicher Weise ist der Optional Operator (vgl. Abschnitt 4.8) realisiert. Der Optional Operator definiert die Syntax `opt [[ ]]` und erlaubt zwischen den doppelten eckigen Klammern ebenfalls ein Modellelement in Modellsyntax. Die Nichtterminale dieses Operators haben das Suffix `_Opt` und verweisen auf das zugehörige Interface-Nichtterminal.

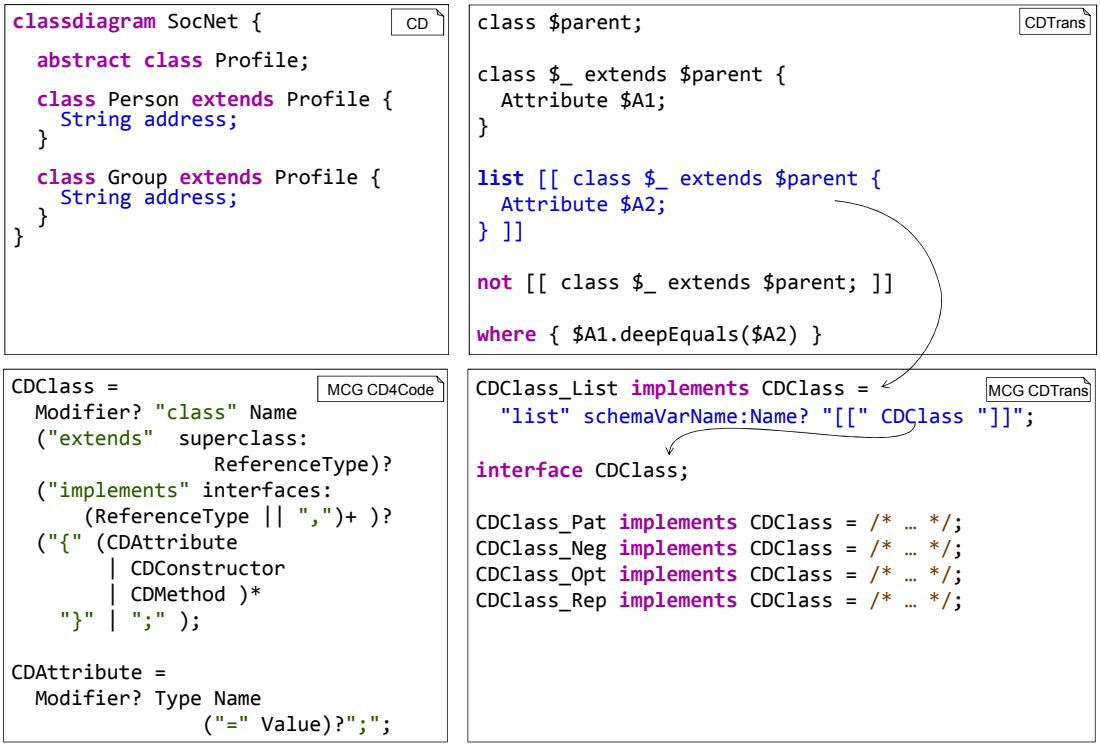


Abbildung 5.7: Erweiterung des Patterns auf eine beliebige Anzahl von Subklassen mittels Collection Operator.

Das bisher gezeigte Pattern funktioniert für den Fall, dass genau zwei Klassen mit gleichem Attribut gefunden werden sollen. Die Pull Up Transformation soll jedoch für eine beliebige Anzahl von Subklassen mit gleichem Attribut funktionieren. Um das Pattern weiter zu generalisieren, muss das Pattern dahingehend angepasst werden, dass es für eine beliebige Anzahl von Klassen mit gleichem Attribut funktioniert. CDTrans bietet hierfür den Collection Operator (vgl. Abschnitt 4.7). Das Pattern wurde in Abbildung 5.7 erweitert und verwendet für die zweite Klasse den Collection Operator. Dieser hat die Syntax `list [[ ]]`, wobei auch hier zwischen den doppelten eckigen Klammern ein Modellelement mithilfe der Modellsyntax beschrieben wird. Im Beispiel wurde dies für eine Klasse gemacht. Dies ermöglicht das Nichtterminal `CDClass_List`. `CDClass_List` definiert hierbei die Syntax `list [[ ]]` und verweist auf das Nichtterminal `CDClass` innerhalb von CDTrans. Zusätzlich kann diese Collection auch an eine Schemavariable gebunden werden. In gleicherweise kann für jedes Modellelement der Collection Operator verwendet werden, also beispielsweise auch Attribute, Methoden oder Interfaces. Dies ist möglich, da für jedes Nichtterminal der Modellierungssprache ein `_List`-Nichtterminal existiert, das auf das zugehörige Interface-Nichtterminal verweist.

Bisher wurde lediglich das Pattern der Pull up Attribute Transformationsregel beschrieben. Die Pull Up Attribute Transformation soll die gefundenen Attribute jedoch

Abbildung 5.8: Verschieben der Attribute.

in die Superklasse verlagern. Um das Modell zu verändern, muss die Transformationsregel noch um eine Modifikation ergänzt werden. Dazu bietet CDTrans den Replacement Operator (vgl. Abschnitt 4.5). Ziel der Transformation ist die Entfernung der Attribute aus den Subklassen und Hinzufügen des Attributs in der Superklasse. Die Transformationsregel wurde an drei Stellen um einen Replacement Operator ergänzt. In den beiden Subklassen (die "normale" und die innerhalb des Collection Operators) wird der Replacement Operator verwendet, um das Attribut zu entfernen (vgl. Abbildung 5.8). In der Superklasse hingegen wird der Replacement Operator benutzt, um das Attribut der klasse hinzuzufügen. Die Syntax des Operators ist `[[ :- ]]`, wobei sowohl vor als auch nach dem Operator `:-` ein Modellelement in Modellsyntax beschrieben werden kann. Im Beispiel wurde der Replacement Operator jeweils für Attribute verwendet. Dies ermöglicht das Nichtterminal `CDAttribute_Rep`. `CDAttribute_Rep` definiert hierbei die Syntax `[[ :- ]]` und verweist vor und nach dem `:-` auf das `CDClass` Interface-Nichtterminal innerhalb von CDTrans. In gleicherweise kann für jedes Modellelement der Replacement Operator verwendet werden, also beispielsweise auch Klassen, Methoden oder Interfaces. Dies ist möglich, da für jedes Nichtterminal der Modellierungssprache ein `_Rep`-Nichtterminal existiert, das auf das zugehörige Interface-Nichtterminal verweist.

5.1.2 CD4Code-spezifische Erweiterungen

Im vorherigen Abschnitt wurde die Syntax der DSTL CDTrans, die sich aus der Ableitung der DSTL aus der Modellierungssprache CD4Code ergibt, erläutert. Hier wurde ebenfalls verdeutlicht, wie sich diese Syntax aus der Syntax der CD4Code Modellierungssprache ableitet. Im Folgenden werden die CD4Code-spezifischen, handgeschriebenen Erweiterungen vorgestellt. Diese Erweiterungen erlauben es Klassendiagrammtransformationen noch spezieller an die CD4Code-Sprache angepasst zu formulieren. Die Erweiterungen basieren auf den in Abschnitt 3.2 ermittelten Anforderungen. Diese Erweiterungen sind aufgrund der Struktur der CD4Code-Grammatik sinnvoll und notwendig. Für die Erweiterungen wurde die zweite in Abschnitt 9.2 vorgestellte Methodik verwendet. Die generierte Grammatik wurde als Vorlage verwendet, um eine eigene DSTL-Grammatik zu erstellen. Hierbei wurde die Nichtterminalstruktur übernommen und erweitert. Die Vor- und Nachteile dieser Erweiterungsform werden in Abschnitt 9.2 diskutiert. Durch die Verwendung von Visitoren für die Übersetzung der Transformationsregeln nach Java und weil lediglich neue Interface-Nichtterminale, die Nichtterminale in Beziehung setzen, eingeführt wurden, ist für die im Folgenden beschriebenen Grammatikanpassungen keine Anpassung der Traversierung durch die Visitoren notwendig. Die Traversierung der neuen Nichtterminal übernimmt hierbei die Implementierung der `traverse`-Methoden in den generierten Visitor-Interfaces [HMSNRW16]. Die Grammatikanpassungen werden in den folgenden Abschnitten jeweils kurz erklärt und das Verfahren hierfür in Abschnitt 5.5 diskutiert.

Attribute durch Methoden ersetzen

Methoden und Attribute kommen innerhalb der Sprache CD4Code innerhalb von Klassen und Interfaces vor. Um ein Attribut durch eine Methode oder umgekehrt zu ersetzen, muss das Attribut entfernt und die Methode hinzugefügt werden. Dies ist der Fall, da die Sprache CD4Code einzelne Nichtterminale für diese beiden Modellelemente verwendet und die automatische Ableitung dafür sorgt, dass Attribute nur durch Attribute und Methoden nur durch Methoden ersetzbar sind. Ein Beispiel für diese Ersetzung sind abgeleitete Attribute, die – beispielsweise durch den Data Explorer Generator – durch jeweils eine Zugriffsmethode ersetzt werden. Hierzu sind entsprechend der systematischen Herleitung der Sprache zwei Replacement Operatoren notwendig (vgl. Abbildung 5.9, links oben). Um diese Ersetzung komfortabler zu ermöglichen, wurde die DSTL CDTrans dahingehend erweitert, dass stattdessen die Ersetzung mit nur einem Replacement Operator formuliert werden kann (vgl. Abbildung 5.9, oben rechts).

Zur Umsetzung wurde die Grammatik CDTrans erweitert. Zudem wurde ein Interface `CDAttributeOrMethod` eingeführt, dass Methoden und Attribute zusammenfasst (vgl. Abbildung 5.9, unten). Die Ersetzung für Attribute und Methoden wurde abgeändert, so dass auf der rechten Seite der Ersetzung Attribute oder Methoden erlaubt sind. Dies erfüllt Anforderung LR 1.1.

```
CDTrans
class Person {
  [[ / String fullName; :- ]]
  [[ :- String getFullName(); ]]
}
```

```
CDTrans
class Person {
  [[ / String fullName; :- String getFullName(); ]]
}
```

```
MCG CDTrans
CDAttribute_Replacement implements CDAttribute =
        "[[" lhs:CDAttribute? ":-" rhs:CDAttributeOrMethod? "]]";

CDMethod_Replacement implements CDMethod =
        ("[[" lhs:CDMethod? ":-" rhs:CDAttributeOrMethod? "]]");

interface CDAttributeOrMethod;

interface CDAttribute;
interface CDMethod;
CDAttribute_Pat implements CDAttributeOrMethod, CDAttribute = /* … */;
CDMethod_Pat implements CDAttributeOrMethod, CDMethod = /* … */;
```

Abbildung 5.9: Ersetzung eines abgeleiteten Attributs durch eine Zugriffsmethode.

Interfaces, Klassen und Enums durch einander ersetzen

Ähnlich wie Attribute und Methoden kommen Klassen, Interfaces und Enums innerhalb des Klassendiagramms an der gleichen Stelle vor, nämlich innerhalb der Klassendiagrammdefinition. Auch hier gilt, dass Ersetzungen dieser verschiedenen Modellelemente – entsprechend der systematischen Herleitung – durch einzelne Replacement Operatoren realisiert werden müssen (vgl. Abbildung 5.10, links). Dies ist der Fall, da diese Modellelemente in CD4Code durch verschiedene Nichtterminale definiert sind. Um auch hierfür eine komfortablere Schreibweise zu ermöglichen (vgl. Abbildung 5.10, rechts), wurde auch hier die Grammatik erweitert. Die Erweiterung erfolgte analog zur Erweiterung für Attribute und Methoden und erfüllt Anforderung LR 1.2.

```
CDTrans
[[ class Person; :- ]]
[[ :- interface Person; ]]
```

```
CDTrans
[[ class Person; :- interface Person;]]
```

Abbildung 5.10: Ersetzung einer Klasse durch ein Interface.

Direkte Ersetzung von Sichtbarkeiten

Die Sichtbarkeiten `public`, `private` und `protected` werden häufig durch einander ersetzt. Ein Beispiel hierfür ist die Kapselung von Attributen durch Zugriffsmethoden. Hierzu sind entsprechend der systematischen Herleitung der DSTL zwei Replacement Operatoren notwendig (vgl. Abbildung 5.11, links oben). Ein Replacement Operator drückt aus, dass die alte Sichtbarkeit entfernt wird und der andere, dass die neue Sichtbarkeit hinzugefügt wird. Dies wäre anders, wenn CD4Code für die Sichtbarkeiten ein Interface-Nichtterminal und für die einzelnen Sichtbarkeiten entsprechende Nichttermi-

nale, die das Interface-Nichtterminal implementieren, verwenden würde. In diesem Fall würde bereits die systematische Ableitung dazu führen, dass die Sichtbarkeiten mit nur einem Replacement Operator ausgetauscht werden können. Um diese Ersetzung komfortabler zu ermöglichen, wurde die DSTL CDTrans so erweitert, dass die Ersetzung mit einem Replacement Operator formulierbar ist (vgl. Abbildung 5.11, oben rechts).

CDTrans

```
class Person {
  [[ public :- ]] [[ :- private ]]
   String fullName;
}
```

CDTrans

```
class Person {
  [[ public :- private ]] String fullName;
}
```

MCG CDTrans

```
Public_Rep implements Protected =
        "[[" lhs:Public_Pat? ":-" rhs:PublicProtectedPrivate? "]]";

Protected_Rep implements Protected =
        "[[" lhs:Protected_Pat? ":-" rhs:PublicProtectedPrivate? "]]";

Private_Rep implements Protected =
        "[[" lhs:Private_Pat? ":-" rhs:PublicProtectedPrivate? "]]";

interface PublicProtectedPrivate;
Public_Pat implements PublicProtectedPrivate, Public = /* … */;
Private_Pat implements PublicProtectedPrivate, Private = /* … */;
Protected_Pat implements PublicProtectedPrivate, Protected = /* … */;
```

Abbildung 5.11: Ersetzung von Sichtbarkeiten.

Zur Umsetzung wurde die Grammatik erweitert. Es wurde ein weiteres Interface `PublicProtectedPrivate` eingeführt, dass die Sichtbarkeiten `public`, `protected` und `private` zusammenfasst (vgl. Abbildung 5.11, unten). Die Ersetzung für `public`, `protected` und `private` wurde so abgeändert, dass auf der rechten Seite der Ersetzung nun ebenfalls `public`, `protected` und `private` erlaubt sind. Diese Erweiterung erfüllt Anforderung LR 1.3.

Assoziation durch Komposition ersetzen

In CD4Code gibt es zwei Arten von Assoziationen: „einfache“ Assoziationen und Kompositionen. Um den Typ einer Assoziation zu ändern muss entsprechend der systematischen Herleitung die Assoziation entfernt und eine identische Assoziation des anderen Typs hinzugefügt werden (vgl. Abbildung 5.12). Hierbei kann es leicht passieren, dass Eigenschaften der ursprünglichen Assoziation vergessen werden anzugeben, wodurch diese Eigenschaften bei den neuen Assoziation fehlen würden. In Abbildung 5.12 wurden beispielsweise keine Kardinalitäten angegeben, wodurch ggf. vorhandene Kardinalitäten der „einfachen“ Assoziation nicht in die Komposition übernommen werden. Um diese Ersetzung weniger fehleranfällig zu ermöglichen, wurde die DSTL CDTrans dahingehend erweitert, dass stattdessen mit nur einem Replacement Operator, der den Typ der Assoziation ändert, die Ersetzung formuliert werden kann (vgl. Abbildung 5.12, oben rechts). Die Erweiterung erfolgte analog zur Erweiterung für die Änderung von Sichtbarkeiten und erfüllt die Anforderung LR 1.4.

```
[[ association Person -> Profile; :- ]]
[[ :- composition Person -> Profile; ]]
```
CDTrans

```
[[ association :- composition ]]
  Person -> Profile;
```
CDTrans

Abbildung 5.12: Ändern des Assoziationstyps.

```
classdiagramm System {
  association A -> B ;
}
```
CDTrans

```
classdiagramm System {
  association A <- B ;
}
```
CDTrans

```
classdiagramm System {
  association A $_ B ;
}
```
CDTrans

Abbildung 5.13: Abstraktion von der Assoziationsrichtung.

Assoziationsrichtung und -typ abstrahieren

Wie zuvor erläutert gibt es in CD4Code zwei Arten von Assoziationen. Beide Arten haben je vier mögliche Richtungsangaben. In einigen Fällen ist die Richtung einer Assoziation oder deren Typ jedoch nicht von Interesse. Dies ist beispielsweise der Fall, wenn alle Assoziationen gefunden werden sollen, die mit einer bestimmten Klasse verbunden sind. Mittels der hergeleiteten DSTL CDTrans müssten in diesem Fall jeweils Transformationen für die verschiedenen Richtungen (vgl. Abbildung 5.13, links) bzw. Assoziationstypen (vgl. Abbildung 5.14, links) geschrieben werden. Auch dies ist durch die Struktur der Sprache CD4Code bedingt. Gäbe es in CD4Code ein Nichtterminal für die Assoziationsrichtung und ein weiteres für den Assoziationstyp, würde bereits die automatische Ableitung dazu führen, dass es Schemavariablen zur Abstraktion gibt. Um diese Transformationen dennoch zusammenfassen zu können, wurde die Grammatik der CDTrans DSTL erweitert, so dass von der Richtung und dem Typ mittels anonymer Schemavariable abstrahiert werden kann (vgl. Abbildung 5.13 und 5.14, jeweils rechts).

```
classdiagramm System {
  association A -> B ;
}
```
CDTrans

```
classdiagramm System {
  composition A -> B ;
}
```
CDTrans

```
classdiagramm System {
  $_ A -> B ;
}
```
CDTrans

Abbildung 5.14: Abstraktion von dem Assoziationstyp.

Umgesetzt wurde dies, indem die Pattern-Produktion für Assoziationen erweitert wurde. Hierbei wurde an zwei Stellen zusätzlich die Möglichkeit geschaffen, eine Schemavariable anzugeben (vgl. Abbildung 5.15). Zum einen als Alternative zur Typangabe der Assoziation (`dc_Type`) und zum anderen als Alternative zur Richtungsangabe der As-

soziation (vgl. Abbildung 5.15). Dies erfüllt die Anforderungen LR 1.5 und LR 1.6.

```
CDAssociation_Pat implements CDAssociation =                                MCG CDTrans
  Stereotype?
  (Association | Composition | dC_Type:TfIdentifier)
  (Derived)?
   Name:TfIdentifier?
  leftModifier:Modifier?
  leftCardinality:Cardinality?
  leftReferenceName:QualifiedName
  ("[" leftQualifier:CDQualifier "]")?
  ("(" leftRole:TfIdentifier ")")?
  (LeftToRight | RightToLeft |  Bidirectional | Unspecified  |  dC_Dir:TfIdentifier)
  ("(" rightRole:TfIdentifier ")")?
  ("[" rightQualifier:CDQualifier "]")?
  rightReferenceName:QualifiedName
  rightCardinality:Cardinality?
  rightModifier:Modifier? ";"
| /* Alternativen mit Schemavariablen */ ;
```

Abbildung 5.15: Grammatikerweiterung zur Abstraktion von der Assoziationsrichtung und dem -typ.

5.2 MATrans: Eine DSTL für MontiArc-Modelle

MATrans ist eine DSTL für Modelle der Sprache *MontiArc* [HRR12, Hab16]. Ziel von MATrans ist es, dem Modellierer eine komfortable Möglichkeit zu bieten, Transformationen auf MontiArc Modellen zu beschreiben. Im Folgenden wird in Abschnitt 5.2.1 MontiArc und die daraus resultierende DSTL beschrieben. Anschließend werden in Abschnitt 5.2.2 MontiArc-spezifische Erweiterungen gegenüber der allgemeinen Struktur von DSTLs – wie sie in Abschnitt 5.2.1 und Kapitel 4 beschrieben wurde – vorgestellt.

5.2.1 MontiArc und MATrans

In Kapitel 4 wurden die Operatoren bereits anhand einer MontiArc Transformation vorgestellt. Hierbei wurde jedoch der Fokus auf die Erläuterung der Operatoren gelegt. In diesem Abschnitt wird die Herkunft der MontiArc-spezifischen Syntax der DSTL MATrans entsprechend der systematischen Ableitung (vgl. Kapitel 6) erläutert. Wie in Abschnitt 5.1.1 wird die Syntax über eine viergeteilte Abbildung erläutert. Oben links befindet sich jeweils das Startmodell, das auch in Kapitel 4 verwendet wurde. Oben rechts zeigt die Abbildung die Transformationsregel, die schrittweise während der Erläuterung ausgebaut wird. Hierbei wird zunächst sehr nah am Modell ein Pattern definiert und schrittweise zu einer allgemeiner formulierten Transformationsregel entwickelt. Im unteren Bereich der Abbildungen ist jeweils links ein Ausschnitt aus der MontiArc-Grammatik und rechts ein Ausschnitt aus der MATrans-Grammatik dargestellt, der zu der aktuell erläuterten Syntax passt. Die Grammatik mit allen Nichtterminalen der DSTL MATrans ist im Anhang zu finden (vgl. Anhang C.3).

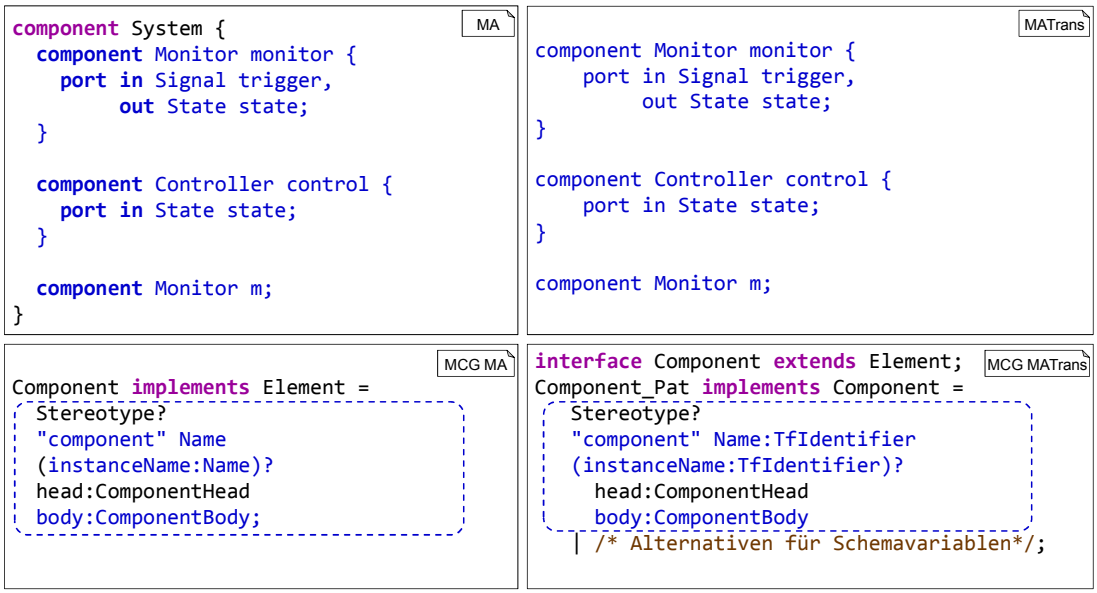


Abbildung 5.16: Modellspezifisches Pattern für die Beispieltransformation (Fokus: Komponente).

In Kapitel 4 wurde das Startmodell bereits vorgestellt. Dieses ist in Abbildung 5.16 erneut zusehen. Das Modell besteht aus einer Systemkomponente mit zwei inneren Komponenten `Monitor` und `Controller` sowie einer Subkomponente vom Typ `Monitor`. Die Monitorkomponente hat zwei Ports: einen eingehenden Signalport und einen ausgehenden Stateport. Die Controllerkomponente hat einen eingehenden Stateport. Ziel der Transformationsregel ist es den Signalport aus der Monitor- in die Controllerkomponente zu verschieben, die Stateports mittels Konnektor zu verbinden, `control` in `controller` umzubenennen und die Subkomponente zu entfernen (vgl. Kapitel 4).

Dazu muss zunächst ein Pattern definiert werden, dass den entsprechenden Teil des Modells bestehend aus den zwei inneren Komponenten, deren Ports und der Subkomponente beschreibt. Außerdem muss die Modifikation beschrieben werden (vgl. Abbildung 5.19), also die Entfernung der Subkomponenten, das Umbenennen der Komponente, das Hinzufügen des Konnektors und die Verschiebung des Ports. Das Pattern kann zunächst sehr modellspezifisch formuliert (vgl. Abbildung 5.16 und 5.17) und anschließend verallgemeinert werden (vgl. Abbildung 5.18, 5.20-5.22), um es auf verschiedenen Eingabemodelle anwenden zu können. Beispiele für Verallgemeinerungen sind das Finden aller vorhandenen Subkomponenten und kompatibler Ports statt explizit Stateports.

Abbildung 5.16 zeigt den ersten Schritt zur Definition des Patterns. Hierzu wurde der relevante Ausschnitt aus dem Modell in das Pattern übernommen. Im MontiArc-Modell sind dies die inneren Komponenten `Monitor` und `Controller` sowie die Subkomponente vom Typ `Monitor`. Die erlaubte Syntax für Komponenten im MontiArc-Modell ergibt sich aus der Produktion `Component` der MontiArc-Grammatik (unten links). Das Pattern sieht für diesen Modellteil zunächst fast identisch aus. Ein Pattern muss die für die Transformation relevanten die Elemente und deren relevante Eigenschaften beschrei-

ben. Die „Hülle“, also `component System { ... }`, wurde weggelassen, da diese zu diesem Zeitpunkt nicht relevant für das Pattern ist.

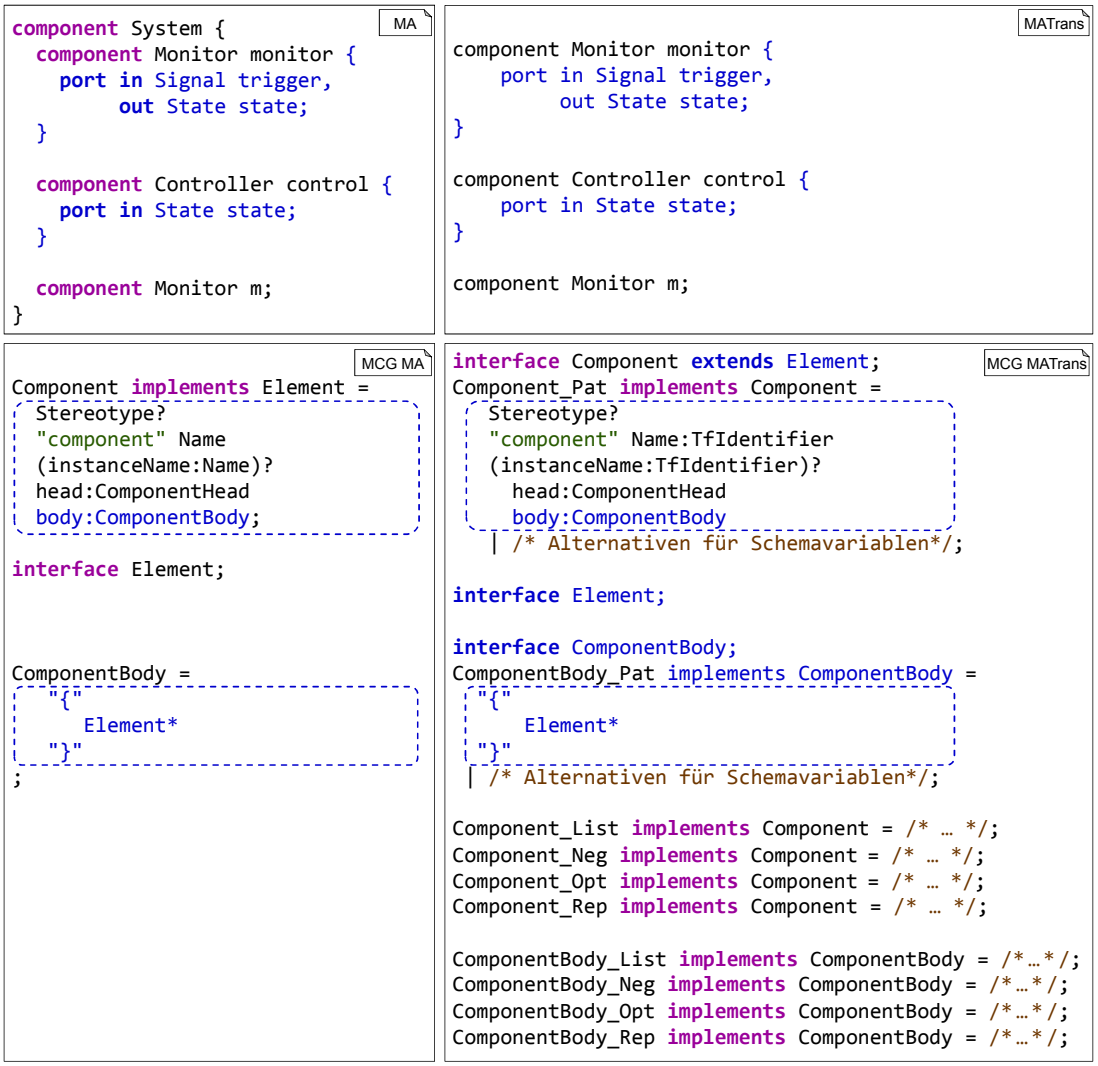


Abbildung 5.17: Modellspezifisches Pattern für die Beispieltransformation (Fokus: Komponentenrumpf).

Auch für die DSTL MATrans wurde bei der Entwicklung der DSTL-Grammatik die Syntax der Modellierungssprache – hier MontiArc – übernommen. Der Rumpf der Produktion `Component` wurde in den Rumpf der Produktion `Component_Pat` übernommen. Hier formt es eine der insgesamt drei Alternativen. Die anderen (nicht gezeigten) Alternativen erlauben die Verwendung von Schemavariablen, die später in diesem Abschnitt erläutert wird. Die übernommene Syntax ist in den unteren beiden Listings in Abbildung 5.16 erneut durch gestrichelte Kästen markiert. `Component_Pat` ermöglicht die Beschreibung von Pattern für Komponenten. Ein im Modell gültiges Elemente – beispielsweise eine Komponente – ist daher immer auch ein gültiger Teil des Patterns.

Für jedes MontiArc-Element, zum Beispiel Komponenten, Subkomponenten und Ports, umfasst die MATrans-Grammatik ein `_Pat`-Nichtterminal, das die Syntax des entsprechenden Nichtterminals der MontiArc Grammatik bereitstellt. Die allgemeine Ableitung der `_Pat`-Nichtterminale findet sich in Kapitel 6.

Auch für MATrans wurde die Struktur der zugehörigen Modellierungssprache MontiArc übernommen. Wie in MontiArc-Modellen können Komponenten auch im Pattern innere Komponenten und Subkomponenten sowie Ports enthalten. In der MontiArc-Grammatik erlaubt dies die `Component`-Produktion durch das verwendete Interface-Nichtterminal `Element`, das von den verschiedenen Nichtterminalen der Modellelemente implementiert wird. In der MATrans-Grammatik bildet sich diese Beziehung ähnlich ab (vgl. Abbildung 5.17). `Component_Pat` verwendet das Nichtterminal `ComponentBody`, das im Rumpf wiederum das Nichtterminal `Element` verwendet. In der MATrans Grammatik gibt es zu jedem Nichtterminal der MontiArc-Grammatik ein entsprechendes Interface-Nichtterminal, das den gleichen Namen wie das Nichtterminal der MontiArc-Grammatik hat. Dies gilt auch für das Interface Nichtterminal `Element` der MontiArc-Grammatik. Die `implements`-Beziehung bildet sich über die Interface-Nichtterminale ab. Hier erweitert das Interface-Nichtterminal `Component` der MATrans-Grammatik das Interface-Nichtterminal `Element` der MATrans-Grammatik. Die `_Pat`-Nichtterminale implementieren die korrespondierenden Interface-Nichtterminal (vgl. Abbildung 5.17). Das heißt beispielsweise implementiert `Component_Pat` das Interface-Nichtterminal `Component`. Außerdem implementieren die Nichtterminale für die verschiedenen Operatoren (vgl. Kapitel 4) das Interface-Nichtterminal für das korrespondierende Nichtterminal: Collection (Suffix `_List`), Optional, (Suffix `_Opt`), Negative Elemente (Suffix `_Neg`), und der Replacement Operator (Suffix `_Rep`).

In MATrans können wie in Abschnitt 4.4 beschrieben an bestimmten Stellen im Pattern Schemavariablen verwendet werden, um beispielsweise vom Eingabemodell zu abstrahieren und die Transformation dadurch zu generalisieren. Hierbei gibt es zwei Arten von Schemavariablen, solche für Namen und solche für ganze Modellelemente. In Abbildung 5.18 werden Schemavariablen für Namen an fünf Stellen verwendet: Für die Portnamen und Typenbezeichner der Stateports (`$portName`, `$type`) sowie den Namen des Signalports (`$_`). Schemavariablen für ganze Modellelemente werden für den Signalport (`$P`) sowie die Subkomponente verwendet (`$S`). Die einfache Form der Schemavariablen besteht aus einem `$`-Zeichen und einem Bezeichner für die Schemavariable bzw. einem Unterstrich im Falle einer anonymen Variable. Diese Form kann an allen Stellen, an denen im Modell ein Name vorkommt, verwendet werden. Beispiele hierfür sind der Name einer Komponente, einer Subkomponente oder von Ports. In der MATrans-Grammatik spiegelt sich dies dadurch wider, dass in der aus den Produktionen der MontiArc-Grammatik übernommenen Syntax alle Vorkommen des Nichtterminals `Name` durch das Nichtterminal `TFIdentifier` ersetzt wurden. Dieses erlaubt weiterhin einfache Namen aber auch Schemavariablen und Ersetzungen von Namen. Im Beispiel in Abbildung 5.18 sieht man dies für den Namen des Ports. MATrans erlaubt außerdem anstelle eines Modellelements Schemavariablen für selbiges zu verwenden. Alternativ zur Angabe eines Modellelements, wie zum Beispiel einer Komponente oder eines Ports, kann

MA

```
component System {
  component Monitor monitor {
    port in Signal trigger,
         out State state;
  }

  component Controller control {
    port in State state;
  }

  component Monitor m;
}
```

MATrans

```
component Monitor monitor {
  port $P [[in Signal $_]],
  out $type $portName;
}

component Controller control {
  port in $type $portName;
}

SubComponent $S
```

MCG MA

```
Component implements Element =
  Stereotype?
  "component" Name
  (instanceName:Name)?
  head:ComponentHead
  body:ComponentBody;

ComponentBody =
   "{"
      Element*
    "}"
;

Port =
  Stereotype?
  (incoming:["in"] | outgoing:["out"])
  Type Name?;


SubComponent implements Element =
  Stereotype?
  "component"
  type:ReferenceType
  ("(" (arguments:Expression || ",")* ")")?
  (instances:SubComponentInstance
    ("," instances:SubComponentInstance)*)?
  ";";
```

MCG MATrans

```
interface Component extends Element;
Component_Pat implements Component =
   Stereotype?
   "component" Name:TfIdentifier
   (instanceName:TfIdentifier)?
     head:ComponentHead
     body:ComponentBody
   | /* Alternativen für Schemavariablen*/;

Port_Pat implements Port =
  /* Alternativen für Modellsyntax
       bzw. Schemavariablen*/
  |
    schemaVarName:Name "[["
      Stereotype?
      (Incoming | Outgoing)
      Type Name:TfIdentifier?
    "]]";

SubComponent_Pat implements SubComponent =
  /* Alternative für Modellsyntax */
  |
    "SubComponent" schemaVarName:Name
  |
  /* Schemavariable + Modellsyntax */;
```

Abbildung 5.18: Verallgemeinerung des Patterns durch Schemavariablen.

im Pattern eine Schemavariable angegeben werden. Dies bildet jeweils die zweite Alternative im Rumpf der `_Pat`-Nichtterminale. Im Beispiel ist dies für die Subkomponente und die zugehörige `SubComponent_Pat` Produktion gezeigt.

Die dritte Alternative des Rumpfs erlaubt die in Abschnitt 4.4.3 vorgestellte Kombination aus Schemavariable und Modellsyntax. Im Beispiel wird dies für den Signalport verwendet. Auch hierfür wurde der Rumpf der Produktion der MontiArc-Grammatik übernommen und Vorkommen des Nichtterminals `Name` entsprechend ersetzt.

Bisher wurde das Pattern schrittweise aufgebaut. Um das Modell zu verändern, muss die Transformationsregel auch hier um eine Modifikation ergänzt werden. Dazu bietet auch MATrans den Replacement Operator (vgl. Abschnitt 4.5). Ziel der Transformati-

MA

```
component System {
  component Monitor monitor {
    port in Signal trigger,
         out State state;
  }

  component Controller control {
    port in State state;

  }

  component Monitor m;
}
```

MATrans

```
component System {
  component Monitor monitor {
    port [[  $P [[in Signal $_]] :- ]],
              out $type $portName;
  }

 component Controller [[control :- controller]] {
    port in $type $portName,
     [[ :- Port $P ]];
  }

  [[ SubComponent $S :- ]]
  [[ :- connect monitor.$portName ->
                              controller.$portName ]]
}
```

MCG MA

```
Component
      implements Element =
 /* … */;

ComponentBody = /* … */;

Port = /* … */;

SubComponent implements Element =
 /* … */;

Connector implements Element =
 /* … */;
```

MCG MATrans

```
Port_Rep implements Port =
  "[[" lhs:Port? ":-" rhs:Port?  "]]";

SubComponent_Rep implements SubComponent =
  "[[" lhs:SubComponent? ":-" rhs:SubComponent? "]]";

Connector_Rep implements Connector =
  "[[" lhs:Connector? ":-" rhs:Connector? "]]";

interface Port;
interface SubComponent extends Element;
interface Connector extends Element;
```

Abbildung 5.19: Modifikation des Modells.

on ist die Entfernung der Subkomponente, Hinzufügen eines Konnektors zwischen den Stateports, Verschieben des Signalports in die Controllerkomponente und Umbenennen der Controllerkomponenten von `control` in `controller`. Dazu wurde die Transformationsregel um fünf Replacement Operatoren ergänzt (vgl. Abbildung 5.19). Die Syntax des Operators ist `[[ :- ]]`, wobei sowohl vor als auch nach dem Operator `:-` ein Modellelement beschrieben werden kann. In der Monitorkomponente wird der Replacement Operator verwendet, um den Signalport zu entfernen. Der hier entfernte und an die Schemavariable `$P` gebundene Port wird durch den Replacement Operator innerhalb der Controllerkomponente eingefügt. Der Port wird dadurch verschoben. Ermöglicht wird dies durch die Produktion `Port_Rep`. `Port_Rep` definiert hierbei die Syntax `[[ :- ]]` und verweist vor und nach dem `:-` auf das `Port` Interface-Nichtterminal innerhalb von MATrans. Außerdem wurde der Replacement Operator zur Umbenennung der Controllerkomponenten verwendet. Dies ermöglicht die Produktion `TFIdentifier` der Basisgrammatik `TFCommons`. Schließlich wurde der Operator noch zum Entfernen der Subkomponente und Hinzufügen des Konnektors verwendet. Dies ermöglichen die Produktionen `Subcomponent_Rep` und `Connector_Rep`. In gleicherweise kann für jedes Modellelement der Replacement Operator verwendet werden. Dies ist möglich, da für jedes Nichtterminal der Modellierungssprache ein `_Rep`-Nichtterminal existiert, das auf das zugehörige Interface-Nichtterminal verweist.

MA

```
component System {
  component Monitor monitor {
    port in Signal trigger,
         out State state;
  }

  component Controller control {
    port in State state;

  }

  component Monitor m;
}
```

MATrans

```
component System {
  component Monitor monitor {
    port [[  $P [[in Signal $_]] :- ]],
              out $type $portName;
  }

 component Controller [[control :- controller]] {
    port in $type $portName,
     [[ :- Port $P ]];
  }

  [[ SubComponent $S :- ]]
  not [[connect monitor.$portName ->
                                controller.$portName ]]
  [[ :- connect monitor.$portName ->
                                controller.$portName ]]
}
```

MCG MA

```
Component
      implements Element =
 /* … */;

ComponentBody = /* … */;

Port = /* … */;

SubComponent implements Element =
 /* … */;

Connector implements Element =
 /* … */;
```

MCG MATrans

```
interface Connector implements Element;

Connector_Neg implements Connector =
  "not" "[[" Connector "]]";

Connector_Pat implements Connector = /* … */;
```

Abbildung 5.20: Einschränkung der Anwendbarkeit der Transformationsregel auf den Fall, dass noch kein Konnektor existiert.

Um das Pattern weiter zu generalisieren, darf die betrachtete Transformationsregel nur dann anwendbar sein, wenn es den Konnektor noch nicht gibt. Um auszudrücken, dass bestimmte Elemente nicht vorkommen dürfen, bietet MATrans – genauso wie CDTrans – negative Elemente (vgl. Abschnitt 4.6). Das Pattern wurde um ein negatives Element ergänzt, das den Konnektor im Modell verbietet (vgl. Abbildung 5.20, oben rechts). Die Syntax hierzu ist `not [[ ]]`, wobei zwischen den doppelten eckigen Klammern ein Modellelement in konkreter Syntax beschrieben wird. Im Beispiel wurde dies für einen Konnektor gemacht. Dies ermöglicht das Nichtterminal `Connector_Neg`. `Connector_Neg` definiert hierbei die Syntax `not [[ ]]` und verweist auf das Nichtterminal `Connector` innerhalb von MATrans. In gleicherweise kann jedes Modellelement negiert werden, also beispielsweise auch Komponenten, Ports oder Subkomponenten. Dies ist möglich, da für jedes Nichtterminal der Modellierungssprache ein `_Neg`-Nichtterminal existiert, das auf das zugehörige Interface-Nichtterminal verweist. In ähnlicher Weise ist der Optional Operator (vgl. Abschnitt 4.8) realisiert. Der Optional Operator definiert die Syntax `opt [[ ]]` und erlaubt zwischen den doppelten eckigen Klammern ein Modellelement. Die Nichtterminale dieses Operators haben das Suffix `_Opt` und verweisen auf das zugehörige Interface-Nichtterminal.

MA

```
component System {
  component Monitor monitor {
    port in Signal trigger,
         out State state;
  }

  component Controller control {
    port in State state;

  }

  component Monitor m;
}
```

MATrans

```
component System {
  component Monitor monitor {
    port [[  $P [[in Signal $_]] :- ]],
              out $type $portName;
  }

 component Controller [[control :- controller]] {
    port in $type $portName,
     [[ :- Port $P ]];
  }

  [[ list [[ SubComponent $S ]] :- ]]
  not [[connect monitor.$portName ->
                                controller.$portName ]]
  [[ :- connect monitor.$portName ->
                                controller.$portName ]]
}
```

MCG MA

```
Component
      implements Element =
 /* … */;

ComponentBody = /* … */;

Port = /* … */;

SubComponent implements Element =
 /* … */;

Connector implements Element =
 /* … */;
```

MCG MATrans

```
interface SubComponent implements Element;

SubComponent_List implements SubComponent =
  "list" schemaVarName:Name? "[[" SubComponent "]]";

SubComponent_Pat implements SubComponent = /* … */;
```

Abbildung 5.21: Entfernen aller Subkomponenten mithilfe des Collection Operators.

Die bisher gezeigte Transformationsregel funktioniert für den Fall, dass genau eine Subkomponente entfernt werden soll. Um die Transformationsregel weiter zu generalisieren, also alle vorhandenen Subkomponenten zu entfernen, muss das Pattern dahingehend angepasst werden, dass es für eine beliebige Anzahl von Subkomponenten funktioniert. Wie auch CDTrans bietet MATrans hierfür den Collection Operator (vgl. Abschnitt 4.7). Die Transformationsregel wurde in Abbildung 5.21 erweitert und verwendet für die Subkomponente den Collection Operator. Dieser hat die Syntax `list [[ ]]`, wobei wiederum zwischen den doppelten eckigen Klammern ein Modellelement beschrieben wird. Im Beispiel wurde dies für eine Subkomponente gemacht. Dies ermöglicht das Nichtterminal `SubComponent_List`. `SubComponent_List` definiert hierbei die Syntax `list [[ ]]` und verweist auf das Nichtterminal `SubComponent` innerhalb von MATrans. Zusätzlich kann diese Collection auch an eine Schemavariable gebunden werden. In gleicherweise kann für jedes Modellelement der Collection Operator verwendet werden, also beispielsweise auch für Komponenten, Ports oder Subkomponenten. Dies ist möglich, da für jedes Nichtterminal der Modellierungssprache ein `_List`-Nichtterminal existiert, das auf das zugehörige Interface-Nichtterminal verweist.

Durch die Generalisierung in Abbildung 5.18 wurde von den tatsächlichen Namen der Stateports abstrahiert und somit die Transformationsregel allgemeingültiger formuliert.

MA

```
component System {
  component Monitor monitor {
    port in Signal trigger,
         out State state;
  }

  component Controller control {
    port in State state;

  }

  component Monitor m;
}
```

MATrans

```
component System {
  component Monitor monitor {
    port[[  $P [[in Signal $_]] :- ]],
                out $type $portName;
  }

component Controller [[$oldName :- $newName]] {
    port in $type $portName,
         [[ :- Port $P ]];
  }

 [[ list [[SubComponent $S]] ]] :- ]]
  not [[ connect monitor.$portName ->
                                controller.$portName ]]
  [[ :- connect monitor.$portName ->
                                controller.$portName ]]
}

where {
  $portName.equals("state")
}

assign {
  $newName = $oldName +"ler"
}
```

MCG MA

```
Component
      implements Element =
 /* … */;

ComponentBody = /* … */;

Port = /* … */;

SubComponent implements Element =
 /* … */;

Connector implements Element =
 /* … */;
```

MCG MATrans

```
TFRule =
  (Component | ComponentBody | Port
    | SubComponent | Connector | /*…*/ )*
  TFFolding? TFAssignments? TFWhere? TFDo? ;

// Geerbt von Supergrammatik TFCommons
TFWhere =
  "where" "{" constraint:Expression "}";

TFAssignments = "assign" "{" Assign* "}";

Assign = variable:Name "=" value:Expression ";" ;
```

Abbildung 5.22: Einschränkung der Anwendbarkeit durch einen Application Constraint und Zuweisungen von Schemavariablen.

Dadurch ist nicht länger sichergestellt, dass es sich um Ports mit dem Namen `state` handelt. Ein Application Constraint bietet die Möglichkeit, über die verwendeten Schemavariablen Aussagen zu formulieren (vgl. Abschnitt 4.11) und damit die Anwendbarkeit der Transformationsregel einzuschränken [EH86, HHT96, BH02, HP05]. Der Application Constraint basiert auf Java und erlaubt die Verwendung der Methoden der abstrakten Syntax. In der Transformationsregel in Abbildung 5.22 wurde die Transformationsregel um einen Application Constraint ergänzt, der ausdrückt, dass der Name der Ports `state` entsprechen muss. Diese Forderung kann auch innerhalb der Transformationsregel ausgedrückt werden, indem die Schemavariable durch `state` ersetzt wird. Aus Demonstrationszwecken wurde stattdessen der Application Constraint verwendet. Der Application Constraint erlaubt beliebige Aussagen, die einen booleschen Wert ergeben. Der App-

lication Constraint verwendet keine MontiArc-spezifische Syntax und wird daher von der Basisgrammatik `TFCommons` zur Verfügung gestellt. Eingebunden in die MATrans Grammatik wird er durch das Startsymbol `TFRule` der MATrans Grammatik. Dieses erlaubt zunächst das Pattern in konkreter Syntax zu beschreiben und anschließend optional einen Application Constraint zu ergänzen (Nichtterminal `TFWhere`). Die Produktion `TFWhere` verwendet das aus der JavaDSL stammende Nichtterminal `Expression` für den Constraint. Ebenfalls zu Demonstrationszwecken wurde die Transformationsregel so abgeändert, dass sie nicht direkt `control` durch `controller` ersetzt, sondern den neuen Namen aus dem alten Namen mithilfe des Zuweisungsblocks berechnet. Ähnlich wie der Application Constraint verwendet auch der Zuweisungsblock keine MATrans-spezifische Syntax und wird daher von der Basisgrammatik `TFCommons` zur Verfügung gestellt (Nichtterminal `TFAssignments`) und durch das Nichtterminal `TFRule` eingebunden (vgl. Abbildung 5.22). Analog werden das Folding (vgl. Abschnitt 4.9) und der Anweisungsblock (vgl. Abschnitt 4.12) von der Basisgrammatik zur Verfügung gestellt und durch das Startsymbol der MATrans-Grammatik eingebunden.

5.2.2 MontiArc-spezifische Elemente

Im vorherigen Abschnitt wurde die Syntax der Sprache MATrans erläutert, die sich aus der Ableitung der DSTL aus der MontiArc-Grammatik ergibt. Hier wurde insbesondere verdeutlicht, wie sich diese Syntax aus der Syntax der MontiArc-Grammatik ergibt. Im Folgenden werden die MontiArc-spezifischen Erweiterungen vorgestellt. Diese Erweiterungen erlauben es MontiArc-Modelltransformationen noch spezieller an die MontiArc-DSL angepasst zu formulieren. Die Erweiterungen basieren auf den in Abschnitt 3.2 ermittelten Anforderungen. Diese Erweiterungen sind aufgrund der Struktur der MontiArc-Grammatik notwendig. In den folgenden Abschnitten wird dies jeweils kurz erklärt und in Abschnitt 5.5 diskutiert.

Abstraktion der Portrichtung

Analog zu Assoziationen in Klassendiagrammen gibt es zwei Arten von Ports in MontiArc-Modellen: eingehende und ausgehende Ports. Die beiden Arten werden durch die Schlüsselworte `in` bzw. `out` unterschieden. In einigen Fällen ist die Richtung eines Ports jedoch nicht von Interesse, beispielsweise, wenn Ports eines bestimmten Typs gefunden werden sollen. Mittels der hergeleiteten DSTL MATrans müssten in diesem Fall jeweils eine Transformation für jede Portrichtung (vgl. Abbildung 5.23, links oben) geschrieben werden. Dies liegt an der Struktur der MontiArc Grammatik. Würde die Grammatik dahingehend verändert, dass die Portrichtung über ein eigenes Nichtterminal definiert ist, dann würde durch die Ableitung automatisch die Möglichkeit der Abstraktion durch Schemavariablen entstehen. Um diese Transformationen dennoch zusammenfassen zu können, wurde die Grammatik der DSTL erweitert, so dass von der Richtung (vgl. Abbildung 5.23, rechts oben) mittels anonymer Schemavariable abstrahiert werden kann.

Umgesetzt wurde dies analog zur Abstraktion von Assoziationsrichtungen bzw. -typen in CDTrans. Die Pattern-Produktion für Ports wurde erweitert. Hierbei wurde die Mög-

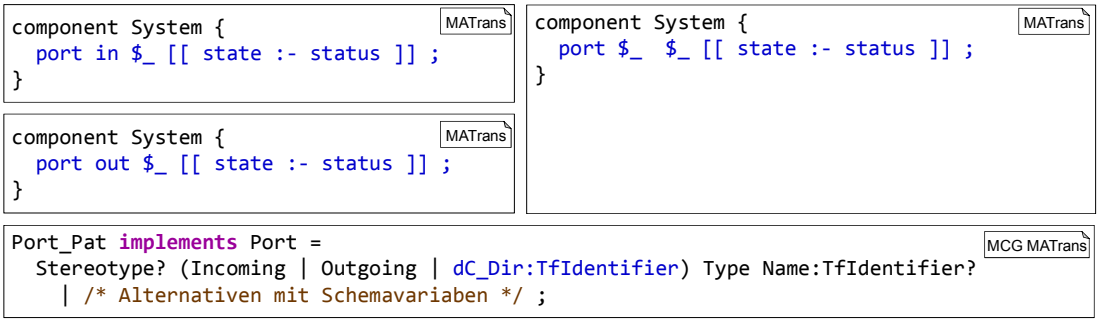


Abbildung 5.23: Abstraktion von der Portrichtung.

lichkeit geschaffen, eine Schemavariable als Alternative zur Portrichtung anzugeben (vgl. Abbildung 5.23). Diese Erweiterung erfüllt Anforderung LR 2.1.

Innere Komponenten durch Subkomponenten ersetzen

Eine weitere Anforderung war die direkte Ersetzung von inneren durch Subkomponenten und Subkomponenten durch innere Komponenten (vgl. Anforderung LR 2.2). In MontiArc kommen innere Komponenten und Subkomponenten innerhalb von Komponenten vor. Um eine innere Komponente durch eine Subkomponente oder umgekehrt zu ersetzen, muss die innere Komponente entfernt und die Subkomponente hinzugefügt werden. Dies kommt beispielsweise vor, wenn eine innere Komponente nicht mehr länger nur von einer sie umgebenden Komponente, sondern von verschiedenen Komponenten benötigt wird. In diesem Fall muss die innere Komponente entfernt werden und stattdessen wird eine Subkomponente benötigt. Dies kann durch zwei Replacement Operatoren ausgedrückt werden (vgl. Abbildung 5.24, links oben). Eine komfortablere Schreibweise ist es, dies über einen Replacement Operator auszudrücken (vgl. Abbildung 5.24, rechts oben). Eine spezielle Erweiterung der MATrans-Grammatik ist durch die weiterentwickelte Generierung und Ableitung von DSTLs nicht notwendig. Die beiden Nichtterminale `SubComponent` und `Component`, die Subkomponenten bzw. innere Komponenten ermöglichen, implementieren beide das Interface-Nichtterminal `Element` der MontiArc-Grammatik. Die weiterentwickelte Generierung berücksichtigt diese Beziehung und bildet sie auf Ebene der Interface-Nichtterminale ab. In MATrans erweitern daher die beiden Interface-Nichtterminale `SubComponent` und `Component` das Interface-Nichtterminal `Element` (vgl. Abbildung 5.24, unten). Zusätzlich wird für das Nichtterminal `Element` ein Replacement Operator generiert, der eine Ersetzung von inneren Komponenten durch Subkomponenten und umgekehrt ermöglicht. Dadurch ist Anforderung LR 2.2 erfüllt.

5.3 MACDTrans: Eine DSTL für Klassendiagramme und MontiArc-Modelle

Klassendiagramme und MontiArc-Modelle bilden zwei verschiedene Möglichkeiten die Struktur eines Systems zu beschreiben. In [Hab16] wurde außerdem die Möglichkeit

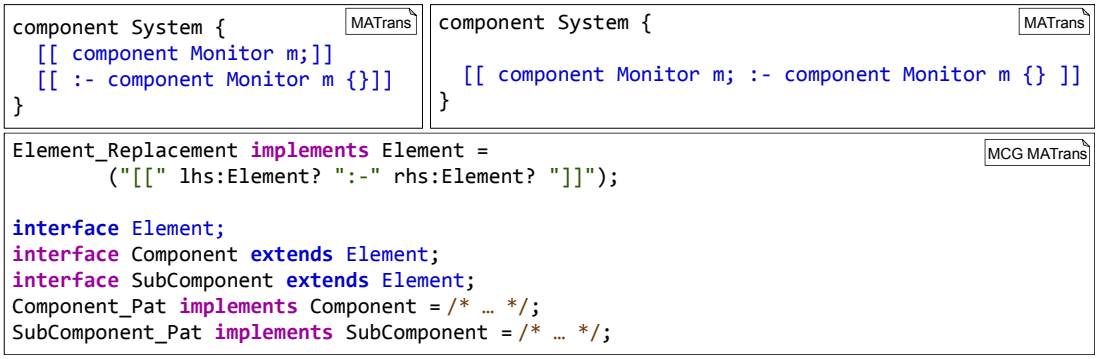


Abbildung 5.24: Ersetzen einer Subkomponente durch eine innere Komponente.

geschaffen die in MontiArc verwendeten Datentypen mittels Klassendiagrammen zu definieren. Hinzu kommt, dass – ähnlich wie es für Grammatiken in MontiCore gemacht wird – zur Codegenerierung aus einer Systembeschreibung mittels MontiArc-Modellen eine Übersetzung in Klassendiagramm möglich ist, aus der schließlich der Code generiert werden kann. Eine DSTL, die es erlaubt sowohl Klassendiagramme als auch MontiArc-Modelle zu transformieren ist daher notwendig. Im Rahmen dieser Arbeit wurde basierend auf den beiden zuvor vorgestellten DSTLs zur Transformation von Klassendiagrammen bzw. MontiArc-Modellen die DSTL *MACDTrans* entwickelt.

MACDTrans ist eine DSTL, die sowohl endogene Transformationen [CH06, MVG06] auf Klassendiagrammen als auch endogene Transformationen auf MontiArc-Modellen erlaubt. Zusätzlich ermöglicht MACDTrans exogene Transformationen [CH06, MVG06] zwischen Klassendiagrammen und MontiArc-Modellen sowie eine Koevolution beider Modellarten. MACDTrans kombiniert hierfür die Syntax der DSTLs CDTrans und MATrans. Im Folgenden wird die Verwendung und Syntax von MACDTrans anhand von Beispielen erläutert. Dazu sind jeweils im oberen Bereich der Abbildung links die Eingabemodelle und rechts die Ausgabemodelle zu sehen (gekennzeichnet durch die Tags `CD` und `MA`). Der untere Bereich der Abbildungen zeigt jeweils die angewendete Transformationsregel in MACDTrans (gekennzeichnet durch das Tag `MACDTrans`). Es werden drei verschiedenen Use Cases für die Sprache erläutert: Erstellung eines Klassendiagramms anhand einer MontiArc-Modells, die Erstellung eines MontiArc-Modells anhand eines Klassendiagramms und die Koevolution von Klassendiagramm und MontiArc-Modell.

5.3.1 Von MontiArc zu Klassendiagrammen

Analog zum MontiCore-Generator kann auch ein MontiArc-Generator so realisiert werden, dass ein Modell zunächst in ein Klassendiagramm übersetzt und anschließend aus diesem Code generiert wird. Die DSTL MACDTrans unterstützt diesen Use Case, da sie dem Nutzer die Möglichkeit bietet ein Pattern für ein MontiArc-Modell zu formulieren und basierend auf den gefundenen Modellelementen wie Komponenten oder Ports ein Klassendiagramm zu erzeugen. Abbildung 5.25 zeigt ein Beispiel hierfür. Das Ausgangsmodell basiert auf dem durchgängigen Beispiel aus Kapitel 4. Das Modell wurde auf die

Systemkomponente, deren innere Monitorkomponente sowie deren Signalport reduziert. Ziel der Transformation ist die Erstellung eines Klassendiagramms für das System, wobei die Systemkomponente durch das Klassendiagramm und die inneren Komponenten durch Klassen abgebildet werden sollen. Die Transformation dient hierbei Demonstrationszwecken und stellt keine vollständige Abbildung von MontiArc-Modellen zu Klassendiagrammen dar. Das Ergebnis der Transformationsregel ist oben rechts in Abbildung 5.25 zu sehen. Das Eingabemodell bleibt erhalten und ist deswegen auf der rechten Seite erneut zu sehen. Zusätzlich ist das erstellte Klassendiagramm auf der rechten Seite vorhanden.

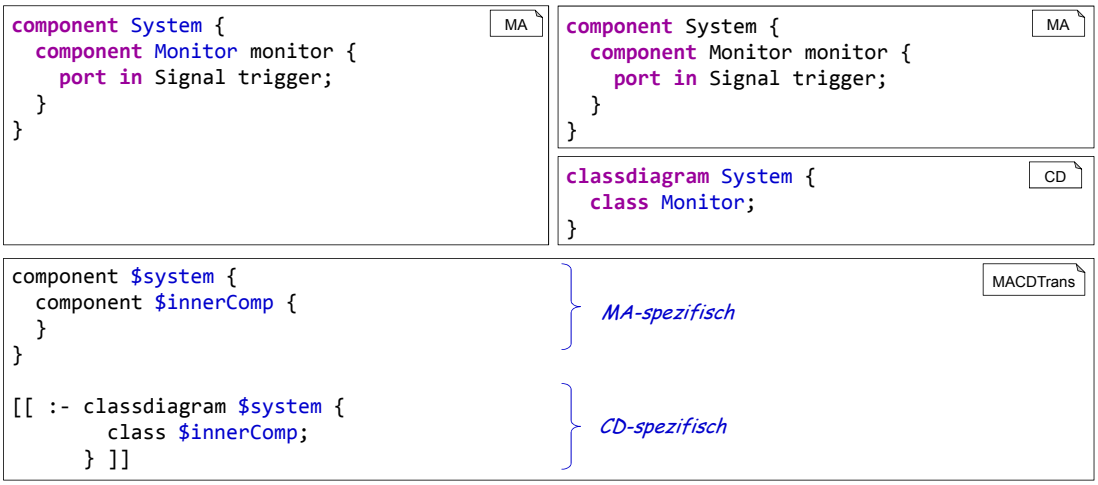


Abbildung 5.25: Grammatikstruktur der MACDTrans DSTL.

Im unteren Teil von Abbildung 5.25 ist die hierzu passende Transformationsregel abgebildet. Die ersten vier Zeilen der Transformationsregel verwenden die MontiArc-spezifische Syntax und beschreiben eine Komponente und deren innere Komponente, wobei für die Namen dieser Komponenten die Schemavariablen `$system` und `$innerComp` verwendet wurden. Im unteren Bereich wird die klassendiagrammspezifische Syntax verwendet. Hier wird das Klassendiagramm erzeugt, wobei der Name des Diagramms den Namen der äußeren Komponente (gebunden an die Variable `$system`) erhält. Innerhalb dieses Klassendiagramms wird außerdem eine Klasse erstellt. Diese hat den Namen der inneren Komponente. In einem nachgelagerten Schritt können für mögliche weitere innere Komponenten Klassen innerhalb des Klassendiagramms und Attribute und Klassen für die Ports der Komponenten erstellt werden. Eine mögliche Abbildung von MontiArc nach Java ist in [Hab16] beschrieben. Aufgrund der Ähnlichkeit von Java und Klassendiagrammen lässt sich diese Abbildung auf Klassendiagramme übertragen. MACDTrans bietet die Grundlage zur Definition entsprechender Transformationen.

5.3.2 Von Klassendiagrammen zu MontiArc

Neben der Möglichkeit Klassendiagramme aus MontiArc-Modellen zu erstellen, ermöglicht MACDTrans auch die Gegenrichtung, also MontiArc-Modelle für Klassendiagramme zu erstellen. Die DSTL MACDTrans unterstützt diesen Use Case, da sie dem Nutzer

auch die Möglichkeit bietet, ein Pattern für ein Klassendiagramm zu formulieren und basierend auf den gefundenen Klassendiagrammelementen ein MontiArc-Modell zu erzeugen. Abbildung 5.26 zeigt ein Beispiel hierfür. Das Ausgangsmodell ist in diesem Fall ein Klassendiagramm `System` mit einer Klasse `Monitor`. Ziel ist es zu dem Klassendiagramm eine äußere Systemkomponente zu erzeugen und zu der Klasse eine innere Komponente Monitor innerhalb der Systemkomponente. Das Ergebnis der Transformationsregel ist oben rechts in Abbildung 5.26 zu sehen. Die Transformation dient Demonstrationszwecken und ist keine vollständige Abbildung von Klassendiagrammen zu MontiArc-Modellen. Das Eingabemodell bleibt erhalten, ist es auf der rechten Seite erneut zu sehen. Zusätzlich ist das erstellte MontiArc-Modell auf der rechten Seite vorhanden.

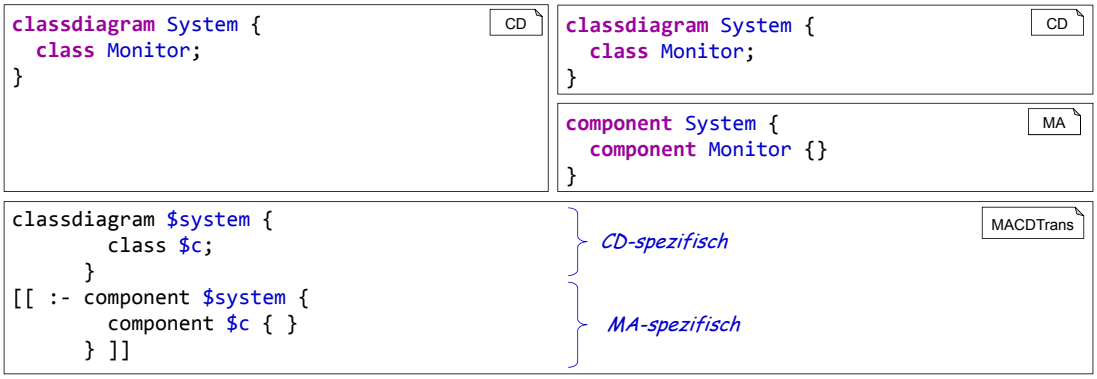


Abbildung 5.26: Grammatikstruktur der MACDTrans DSTL.

Im unteren Teil von Abbildung 5.26 ist die hierzu passende Transformationsregel abgebildet. Die ersten drei Zeilen der Transformationsregel verwenden die klassendiagrammspezifische Syntax und beschreiben ein Klassendiagramm und eine darin enthaltene Klasse. Für die Namen des Klassendiagramms und der Klasse wurden die Schemavariablen `$system` und `$c` verwendet. Im unteren Teil des Listings wird die MontiArc-spezifische Syntax verwendet. Hier wird das MontiArc-Modell erzeugt, wobei der Name der äußeren Komponente der des Klassendiagramms und der Name der inneren Komponente dem der Klasse entspricht. Dies wird durch die verwendeten Schemavariablen erreicht. In einem nachgelagerten Schritt können beispielsweise weitere innere Komponenten für weitere Klassen innerhalb des Klassendiagramms erstellt werden. Das Matching und die parallele Transformation von MontiArc-Modellen und Klassendiagrammen ist ebenfalls mittels MACDTrans möglich und wird im folgenden Use Case zur Koevolution von MontiArc-Modellen und Klassendiagrammen gezeigt.

5.3.3 Koevolution von MontiArc-Modellen und Klassendiagrammen

Neben der bisher gesehenen Erzeugung eines Klassendiagramms auf Basis eines MontiArc-Modells und umgekehrt, ermöglicht MACDTrans auch das Matching und die Modifikation in beiden Arten von Modellen gleichzeitig. Abbildung 5.27 zeigt ein Beispiel hierfür. Das Ausgangsmodell ist angelehnt an das durchgängige Beispiel aus Kapitel 4

und zeigt die Monitorkomponente sowie deren eingehenden Signalport. Außerdem sind die Typen der Ports durch ein nebenliegendes Klassendiagramm `DataTypes` definiert [Hab16]. Ziel der Transformation ist eine Änderung eines Klassennamens innerhalb des Klassendiagramms von `Signal` zu `State` sowie die parallele Änderung der Ports, die diesen Typen verwenden. Die Ausgangssituation zeigen die beiden Modelle im oberen linken Teil der Abbildung 5.27. Die Zielsituation zeigen die entsprechenden Modelle im oberen rechten Bereich der gleichen Abbildung.

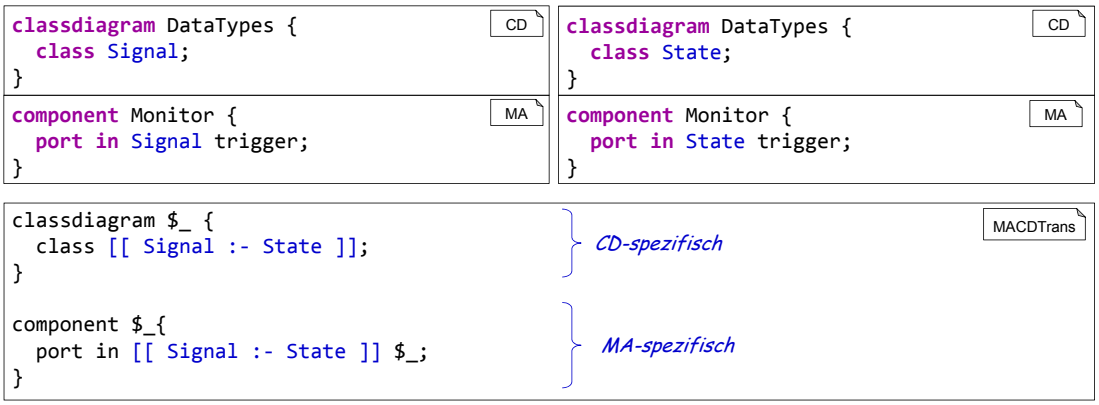


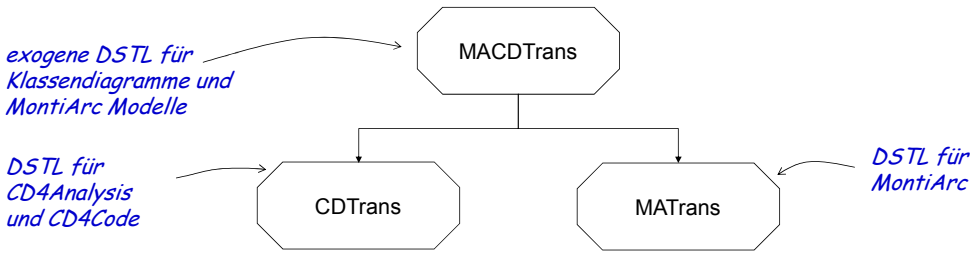
Abbildung 5.27: Grammatikstruktur der MACDTrans DSTL.

Im unteren Teil von Abbildung 5.27 ist die hierzu passende Transformationsregel abgebildet. Die ersten drei Zeilen der Transformationsregel verwenden die Klassendiagramm-spezifische Syntax und beschreiben die Klasse `Signal` im Klassendiagramm. Im unteren Teil des Listings wird die MontiArc-spezifische Syntax verwendet. Hier wird die Komponente mit ihrem Port von Typ `Signal` beschrieben. Sowohl für die Namensänderung der Klasse als auch für die Änderung des Typs des Ports wird der Replacement Operator verwendet und ersetzt hier Signal durch `State`. Im Folgenden wird die Realisierung von MACDTrans detailliert.

5.3.4 Realisierung von MACDTrans

MACDTrans wurde basierend auf den beiden DSTLs MATrans und CDTrans entwickelt. Dazu wurde der Sprachvererbungsmechanismus [LNPR+13, HLMSN+15b, HLMSN+15a] von MontiCore genutzt (vgl. Abschnitt 2.1.4). Abbildung 5.28 verdeutlicht die Beziehung der Grammatiken der beteiligten Sprachen. Die Grammatik der MACDTrans DSTL erweitert die Grammatiken der MATrans und der CDTrans DSTL. Damit erbt MACDTrans die Produktionen beider Sprachen und damit verbunden die Syntax zur Beschreibung von Pattern und Modifikationen in der Syntax für Klassendiagramme und MontiArc-Modelle. Um innerhalb einer Transformationsregel beide Syntaxen verwenden zu können, verknüpft die `TFRule` Produktion der MACDTrans-Grammatik die Nichtterminale der MATrans und der CDTrans Grammatik in einer Alternative und erlaubt beliebig viele dieser Nichtterminale innerhalb der Transformationsregel. Dadurch ergeben sich die in den vorangegangenen Abschnitten erläuterten Möglichkeiten der Transfor-

mation von Klassendiagrammen und MontiArc-Modellen. Zusätzlich gewährleistet MACDTrans, dass innerhalb einer Transformationsregel nur Transformationen formulierbar sind, die keine hybriden Modelle, das heißt Modelle, die zum Teil zur Klassendiagrammgrammatik und zum Teil zur MontiArc-Grammatik konform sind, erzeugt werden. Als eine Erweiterung des Prinzips wäre vorstellbar eine Modellierungssprache zu erstellen, die solche hybriden Modelle erlaubt und anschließend eine DSTL für diese zu erzeugen.




```
grammar MCCDTR extends MontiArcTR, CD4AnalysisTR {
  TFRule = (CDClass | CDAttribute | /*…*/ | Component | Port | /*…*/ )*
           TFFolding? TFAssignments? TFWhere? TFDo? ;
}
```

MCG

Abbildung 5.28: Grammatikstruktur der MACDTrans DSTL.

Um die systematische Herleitung (vgl. Kapitel 6) oder MontiTrans zur Generierung von DSTLs für eine Kombination an Modellierungssprachen zu nutzen, muss zunächst eine gemeinsame Subsprache der beteiligten Modellierungssprachen gebildet werden. Anschließend kann entsprechend der Ableitungsregeln oder mittels Generator die DSTL automatisiert erzeugt werden.

5.4 Aufbau von CD-, MA-, MACDTrans und der Basissprachen

MontiCore unterstützt eine modulare, kompositionale Entwicklung von Modellierungssprachen durch die Wiederverwendung bereits entwickelter Grammatiken mittels verschiedener Kompositionsmechanismen wie Sprachvererbung, -einbettung und -aggregation. Um die Entwicklung neuer DSLs weiter zu erleichtern, bietet MontiCore außerdem eine Reihe wiederverwendbarer Basisgrammatiken [Sch12]:

Literals definiert Nichtterminale für Literale in Form von Zahlen und Strings,

Types definiert Nichtterminale für verschiedene Typen wie Referenztypen oder primitive Typen und

Common definiert Nichtterminale für Stereotypen, Modifier und Kardinalitäten.

Für jede dieser Sprachen wurde eine entsprechende DSTL abgeleitet, d.h. für Literals LiteralsTrans, für Types TypesTrans und für Common CommonTrans. Die Sprachen CD4Code und CD4Analysis und MontiArc sind modular definiert. MontiArc verwendet

außerdem eine eigene Common Sprache, welche die Common aus MontiCore erweitert. Auch für diese DSL wurde eine entsprechende DSTLs, genannt MACommonTrans, entwickelt. Die Modularität spiegelt sich im Aufbau der DSTLs wider (vgl. Abbildung 5.29).

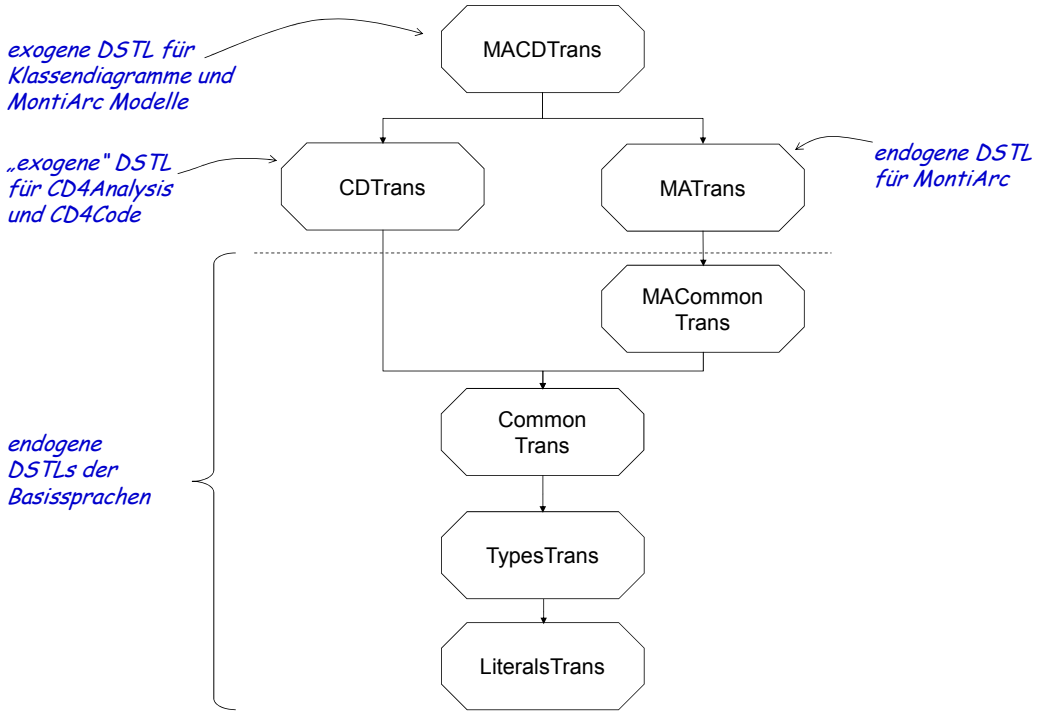


Abbildung 5.29: Zusammenhang der Basis-DSTLs und CD-, MA- und MACDTrans.

Sowohl MATrans als auch der CDTrans basieren auf den DSTLs für die Basissprachen. Abbildung 5.29 verdeutlicht die Sprachvererbungsstruktur. Die Abbildung zeigt die DSTLs MACDTrans, MATrans, CDTrans, MACommonTrans, CommonTrans, TypesTrans und LiteralsTrans. Hierbei bildet LiteralsTrans das Fundament, auf welchem die TypesTrans aufsetzt, genauso wie Types auf Literals basiert. CommonTrans basiert auf der TypesTrans und bildet die Grundlage für CDTrans und MACommonTrans. MATrans basiert auf MACommonTrans. Bei den Basis-DSTLs Literals-, Types-, Common- und MACommonTrans handelt es sich um endogene DSTLs die Transformationsregeln innerhalb der gleichen Sprache also beispielsweise von Literals nach Literals erlauben. Die DSTLs sind nicht zur eigenständigen Nutzung vorgesehen, sondern dienen als Bausteine für DSTLs, deren zugehörige DSL auf den Basisgrammatiken von MontiCore bzw. MontiArc basieren. MATrans ist eine endogene DSTL für MontiArc-Modelle. Die DSTL erlaubt es m MontiArc-Modelle zu n MontiArc-Modellen zu transformieren. CDTrans ist ein DSTL für Modelle der Grammatik, die die Sprachen CD4Code und CD4Analysis definiert. Da CD4Analysis eine Teilmenge der Modelle von CD4Code erlaubt, ist CDTrans, die sowohl für CD4Code als auch für CD4Analysis Modelle verwendet werden kann, im Grunde ein Sonderfall für eine exogene DSTL, da die beteiligten DSLs in einer Subsprachenbeziehung stehen. MACDTrans ist eine exogene DSTL für Klassendiagram-

me und MontiArc-Modelle. Wie in Abschnitt 5.3 beschrieben, erlaubt die DSTL sowohl Transformationen nur auf Klassendiagrammen, nur auf MontiArc-Modellen sowie die Übersetzung von Klassendiagrammen nach MontiArc bzw. von MontiArc zu Klassendiagrammen und eine Koevolution beider Modelltypen.

5.5 Diskussion und verwandte Arbeiten

In diesem Kapitel wurden die DSTLs CDTrans und MATrans für die auf MontiCore basierenden Sprachen CD4Code, CD4Analysis und MontiArc vorgestellt (vgl. LR 1 und LR 2). Es wurde erläutert wie sich die Syntax der DSTLs aus der Syntax der jeweiligen, zugehörigen Modellierungssprache ergibt. Zusätzlich wurde der Aufbau der DSTLs basierend auf DSTLs für die Basissprachen von MontiCore erläutert (vgl. LR 3). Außerdem wurden Erweiterungen der generierten DSTLs CDTrans und MATrans vorgestellt, die die Benutzbarkeit von CDTrans und MATrans verbessern (vgl. LR 1.1-LR 1.4). Hierbei wurden für CDTrans-Transformationen abkürzende Schreibweisen für die Ersetzung von Attributen durch Methoden (vgl. Punkt LR 1.1) sowie Interfaces durch Klassen (vgl. LR 1.2) und umgekehrt entwickelt. Außerdem wurde die Veränderung von Sichtbarkeiten beispielsweise für Attribute und Methoden vereinfacht (vgl. LR 1.3). Des Weiteren wurden für Assoziationen Möglichkeiten zur Abstraktion geschaffen (vgl. LR 1.4-LR 1.6), sodass Transformationen, für die die Assoziationsrichtung oder der -typ irrelevant sind, zusammengefasst werden können (vgl. GR 7.1). MATrans wurde erweitert, um zu ermöglichen, dass Transformationen, für die die Richtung eines Ports irrelevant ist (vgl. LR 2.1), zusammengefasst werden können (vgl. GR 7.1). Diese Erweiterungen waren aufgrund der Struktur der Grammatiken von CD4Code bzw. MontiArc notwendig. Alternativ zu diesen Erweiterungen könnte man auch die Grammatiken der DSLs verändern, sodass diese Anpassungen nicht notwendig sind, da entsprechende Operatoren bereits durch die Ableitung oder Generierung von DSTLs geschaffen werden. Für eine direkte Ersetzung von inneren Komponenten durch Subkomponenten (vgl. LR 2.2) musste MATrans hingegen nicht erweitert werden, da hier eine Zusammenfassung durch ein Interface-Nichtterminal vorliegt, welche von der weiterentwickelten Generierung von DSTLs berücksichtigt wird (vgl. Abschnitt 6.2).

Darüber hinaus wurde die exogene Transformationssprache MACDTrans als Kombination aus CDTrans und MATrans vorgestellt (vgl. LR 4). Dazu wurde zunächst eine Grammatik erstellt, die die Grammatiken der Sprachen CD4Code und MontiArc erweitert und für diese die DSTL MACDTrans mithilfe von MontiTrans entwickelt. Analog können weitere exogene DSTLs entwickelt werden. Bei der Bildung einer Grammatik als Kombination mehrere Grammatiken durch Sprachvererbung kann es zu Namenskonflikten zwischen den verwendeten Nichtterminalen kommen. Diese werden von MontiCore behandelt, indem in diesem Fall nur eine der Definitionen dieses Nichtterminals verwendet wird und die anderen verworfen werden. Welche Definition verwendet wird, hängt von der Reihenfolge der kommaseparierten Liste von erweiterten Grammatiken ab. MontiCore verwendet die erste gefundenen Definition, das heißt die Definition in der am weitesten vorne stehenden Grammatik. Ist dies nicht das gewünschte Verhalten, muss

der Sprachentwickler den Konflikt durch eine Anpassung der Grammatiken auflösen.

MACDTrans ermöglicht sowohl die Übersetzung zwischen Klassendiagrammen und MontiArc-Modellen als auch ein Koevolution beider Modellarten (vgl. GR 4 und GR 4.1-GR 4.3). Die vorgestellten DSTLs wurden mithilfe von MontiTrans entwickelt. Dies zeigt, dass MontiTrans für die Entwicklung von DSTLs für typische mit MontiCore entwickelte Modellierungssprachen geeignet ist. Mit MACDTrans wurde außerdem gezeigt, wie MontiTrans für die Entwicklung exogener DSTLs bzw. zur Koevolution geeigneter DSTLs für zwei oder mehr verschiedene Modellierungssprachen verwendet werden kann (vgl. GR 5). Die möglichen Verwendungsformen von MACDTrans, das heißt Übersetzung von Klassendiagrammen zu MontiArc-Modellen bzw. MontiArc-Modellen zu Klassendiagrammen und die Koevolution beider Modelltypen wurden jeweils anhand einer exemplarischen Transformation erläutert. Hierbei wurde jedoch kein vollständige Abbildung von MontiArc-Modellen auf Klassendiagramme oder umgekehrt gezeigt. Dieses kann jedoch mithilfe von MACDTrans beispielsweise basierend auf der Abbildung von MontiArc-Modellen nach Java [Hab16] entwickelt werden.

Gegenüber Ansätzen, die auf der abstrakten Syntax der Modelle aufsetzen, wie beispielsweise GPTLs, haben diese DSTLs den Vorteil, dass sie eine Einbindung von Domännexperten bzw. Nutzern der korrespondierenden Modellierungssprachen ermöglichen. Beispiele für GPTLs sind ATL [JK06]) und C-SAW [ZLG05] oder UML-basierte Tools wie UMLX [UML17b] und UML-RSDS [UML17a]. Es existieren bereits DSTLs für verschiedene Bereiche [Sch07, Grø09, GMPO09a], unter anderem UML. Beispielweise existieren DSTLs für einige Sprachen der UML [Sch07, Grø09, GMPO09a] und insbesondere für Aktivitätsdiagramme [GMPO09a, Grø09], Sequenzdiagramme [Grø09, WJE+09] und Statecharts [Grø09, WJE+09] und Klassendiagramme [WJE+09]. Im Gegensatz zu den hier präsentierten DSTLs adressieren diese DSTLs jedoch graphische Modelle.

Neben MATrans existieren für ADLs bereits verschiedene Transformationssprachen [Gru05, BLMDL08, BGS14]. Diese entwickelten entweder eine eigene neue Art Transformationen zu beschreiben, basieren auf der abstrakten Syntax der ADL oder sind sehr eingeschränkt in ihrer Funktionalität [BLMDL08], indem sie beispielsweise das Entfernen von Modellelementen verbieten. Im Gegensatz dazu fokussiert MATrans wie auch die anderen vorgestellten DSTLs die Verwendung der konkreten Syntax von MontiArc zur besseren Einbindung von Domänenexperten, ohne dass hierbei die Funktionalität der Transformationssprache eingeschränkt werden muss.

Die hier vorgestellte DSTL-Sammlung hat die Grundlage für eine Familie von DSTLs geschaffen, innerhalb der jede DSTL zugeschnitten auf die entsprechende Modellierungssprache ist, für die sie erstellt wurde. Dies passt zu der in [Cua12] vorgestellten Idee der Transformationssprachenfamilie, in der jede Transformationssprache einem speziellen Zweck dient und für diesen besonders geeignet ist.

Kapitel 6

Ableitung und Generierung domänenspezifischer Transformationssprachen

Bisher wurden in Kapitel 4 die Operatoren der DSTLs und in Kapitel 5 konkrete DSTLs unter anderem CDTrans zur Transformation von Klassendiagrammen und MATrans für MontiArc-Modellen vorgestellt. In diesem Kapitel wird nun auf die automatische Generierung von DSTLs durch eine Menge von Ableitungsregeln eingegangen, die genutzt wurden, um die in Kapitel 5 vorgestellten DSTLs zu entwickeln. CDTrans und MATrans wurden schematisch erläutert. Diese Erklärung orientierte sich bereits an den hier vorgestellten Ableitungsregeln. Zusätzlich wird die Architektur der Transformationsengine und der generierten DSTLs sowie deren technische Realisierung erläutert.

Die in Kapitel 4 vorgestellte Struktur der DSTLs und deren Operatoren lassen sich in zwei Kategorien einteilen: modellierungssprachenunabhängige und -spezifische. Die unabhängigen Teile sind für die verschiedenen DSTLs gleich und können über eine wiederverwendbare Basissprache zur Verfügung gestellt werden. Die spezifischen Anteile hingegen unterscheiden sich von DSTL zu DSTL.

Die wichtigsten Ergebnisse dieses Kapitels sind:

- Die Regeln zur Ableitung einer DSTL aus einer DSL.
- Die Demonstration der Ableitung einer DSTL aus einer DSL.
- Die Erläuterung des Aufbaus von DSTL für modulare DSL Definitionen.
- Die Beschreibung der Generierung von DSTLs mittels MontiTrans.

Abbildung 6.1 zeigt einen Überblick über die in diesem Kapitel beschriebenen Aspekte von MontiTrans. Im Folgenden wird zunächst die Struktur von DSTLs für nicht monolithische Sprachdefinitionen erläutert. Anschließend werden die Ableitungsregeln für neue DSTLs in Abschnitt 6.2 vorgestellt und am Beispiel einer `Automatons` DSL demonstriert. Als nächstes wird in Abschnitt 6.3 die gemeinsame Basisgrammatik der DSTLs vorgestellt. Schließlich wird die Generierung neuer DSTLs mit MontiTrans in Abschnitt 6.4 beschrieben.

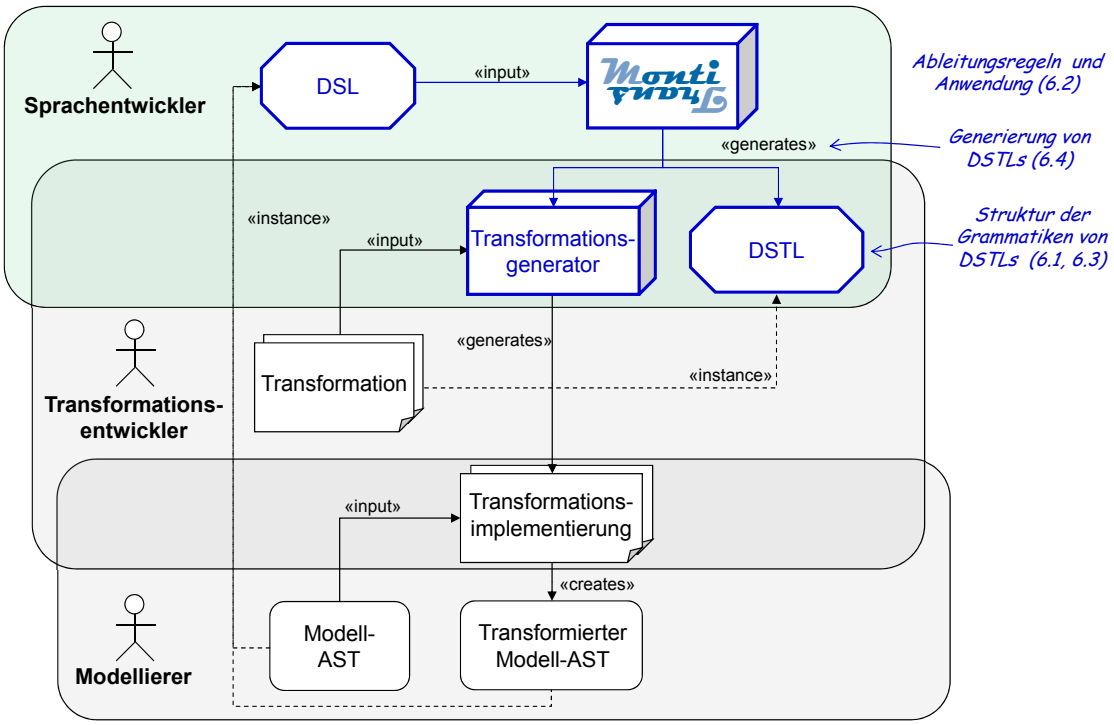


Abbildung 6.1: Übersicht über die verschiedenen Aspekte von MontiTrans mit Fokus auf den in diesem Kapitel vorgestellten Teil (Ableitung und Generierung von DSTLs).

6.1 Struktur von DSTLs für modular definierte Modellierungssprachen

Die Struktur einer DSTL für eine Modellierungssprache ist komplett systematisch, wobei die Bestandteile einer DSTL in zwei Arten unterteilen lassen: sprachunabhängige und sprachspezifische. Der sprachunabhängige Teil einer DSTL ist für alle Modellierungssprachen gleich und kann daher über eine gemeinsame Basissprache aller DSTLs zur Verfügung gestellt werden. Der sprachspezifische Anteil hingegen hängt von der zugrundeliegenden Modellierungssprache ab und muss für jede Modellierungssprache erstellt werden. Die Struktur des sprachspezifischen Teiles einer DSTL ist komplett systematisch und kann daher aus Basis von Ableitungsregeln systematisch und automatisiert aus der Grammatik der Modellierungssprache abgeleitet werden.

Abbildung 6.2 verdeutlicht die Beziehungen zwischen Modellierungs- und Transformationssprachen sowie zwischen den sprachspezifischen und sprachunabhängigen Anteilen der DSTLs. Auf der linken Seite der Abbildung sind drei Modellierungssprachen dargestellt: `Automaton`, `ActionAutomaton` als erweiterte Variante von Automaten und `Website`, eine Sprache zur Beschreibung von Websites (Abbildung 6.2, links). `Action-Automaton` basiert auf `Automaton`, was sich auf Grammatikebene durch die Verwen-

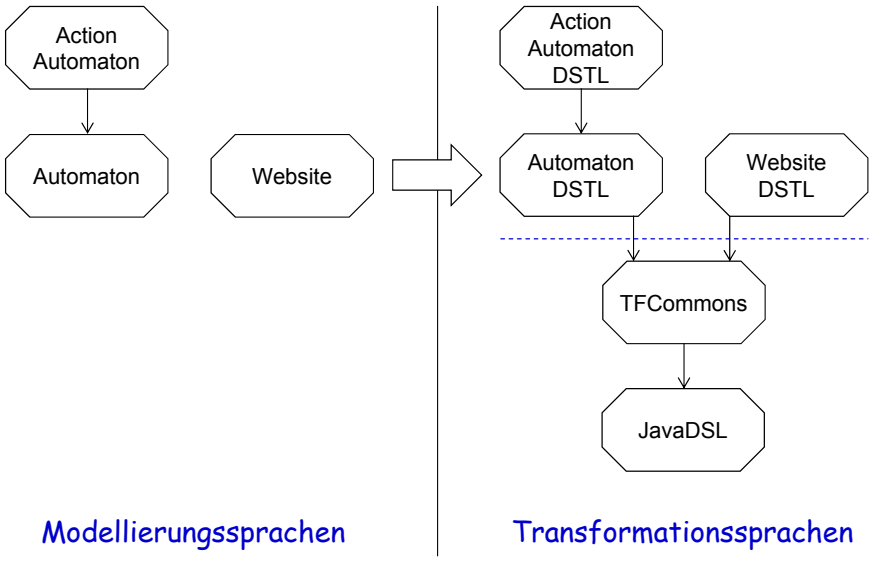


Abbildung 6.2: Hierarchie der DSTLs.

dung von Sprachvererbung widerspiegelt. Zu jeder Modellierungssprache gibt es eine entsprechend zugehörige DSTL. Die zugehörigen DSTLs sowie die Basissprache sind auf der rechten Seite dargestellt: `AutomatonDSTL`, `ActionAutomatonDSTL` und `WebsiteDSTL` sind die DSTLs und `TFCommons`, die wiederum auf der JavaDSL aufbaut, sind die Basissprachen für die DSTLs. Hierbei stehen die DSTLs der Sprachen in der gleichen Beziehung zueinander wie die Modellierungssprachen, d.h. genauso wie `ActionAutomaton` von `Automaton` erbt, erbt `ActionAutomatonDSTL` von `AutomatonDSTL`.

6.2 Ableitungsregeln

Die Grammatik einer DSTL wird systematisch aus der Grammatik der gegebenen Modellierungssprache abgeleitet. Die Grammatik einer DSTL erbt modellierungssprachenunabhängige Anteile von der Basissprache `TFCommons` (vgl. Abschnitt 6.3), die restlichen modellierungssprachenspezifischen Nichtterminale werden durch die im Folgenden erläuterten Ableitungsregeln erstellt. Die Struktur der resultierenden DSTL ist hierbei von der Nichtterminalstruktur der Modellierungssprache abhängig. Relevant für die DSTL sind hierbei sowohl Nichtterminale als auch Schlüsselworte, die semantisch relevant sind, d.h. die im AST gespeichert werden, beispielsweise `abstract` oder `public` bei Klassen. Solche Schlüsselworte werden im Folgenden nur noch als relevante Schlüsselworte bezeichnet. Für die Nichtterminale und relevanten Schlüsselworte der Modellierungssprache erstellen die Ableitungsregeln Nichtterminale für folgende Möglichkeiten in der DSTL:

- Verwendung der konkreten Syntax der Modellierungssprache für Pattern (Nichtterminale mit Suffix `_Pat`),

- Verwendung von Schemavariablen für Elemente und Namen (Nichtterminale mit Suffix `_Pat`),
- einen Replacement Operator (Nichtterminale mit Suffix `_Rep`),
- Beschreibung negativer Elemente (Nichtterminale mit Suffix `_Neg`),
- einen Collection Operator (Nichtterminale mit Suffix `_List`) und
- einen Optional Operator (Nichtterminale mit Suffix `_Opt`).

In [HRW15] werden ebenfalls Ableitungsregeln zur Ableitung von DSTLs vorgestellt. Dies umfasst Regeln für Pattern inklusive Schemavariablen, den Replacement Operator sowie negative Elemente und das Startsymbol der DSTLs. In dieser Arbeit werden Erweiterungen dieser Regeln vorgestellt. Diese Erweiterungen umfassen eine Regel für das Grammatikgerüst inklusive Sprachhierarchien sowie zusätzliche Regeln für den Collection sowie den Optional Operator. Zur Demonstration wird die `Automatons` Grammatik in Abbildung 6.3 (oben) verwendet. Die Grammatik ermöglicht die Definition hierarchischer Automaten. Die Grammatik umfasst drei Nichtterminale: `Automaton`, `State` und `Transition`. `Automaton` ist die Startsymbol der Grammatik und erlaubt das „Gerüst“ des Automaten. Ein Automat hat einen Namen und kann beliebige States und Transitionen haben. Ein State kann als `initial`, `final` oder beides markiert werden, hat einen Namen und kann Substates haben. Transitionen verbinden zwei States miteinander und haben eine Quell- und eine Zielangabe (`from` bzw. `to`).

6.2.1 Regel 1: Grammatikgrundgerüst

Die erste Regel erstellt das „Grammatikgerüst“ der DSTLs, innerhalb dessen in den nächsten Regeln Produktionen für die einzelnen Operatoren erstellt werden. Die Regel ist abhängig davon, ob es sich um eine monolithische oder modulare Sprachdefinition handelt:

Regel 1 *Für die Sprache L erstelle eine neue Grammatik für die Sprache TL. Falls L monolithisch ist, lasse TL von $TFCommons$ erben:*

```
grammar TL extends TFCommons {
}
```

sonst für die Supersprachen $SL_1 \dots SL_n$ von L erstelle die DSTLs $TSL_1 \dots TSL_n$. Erstelle erstelle ein Grammatikgerüst der folgenden Form:

```
grammar TL extends TSL1, ..., TSLn {
}
```

Hierbei leitet sich der Name für TL aus dem Namen von L ab. Für TL wird der Name von L plus das Suffix `TR` (für Transformationsregel) verwendet.

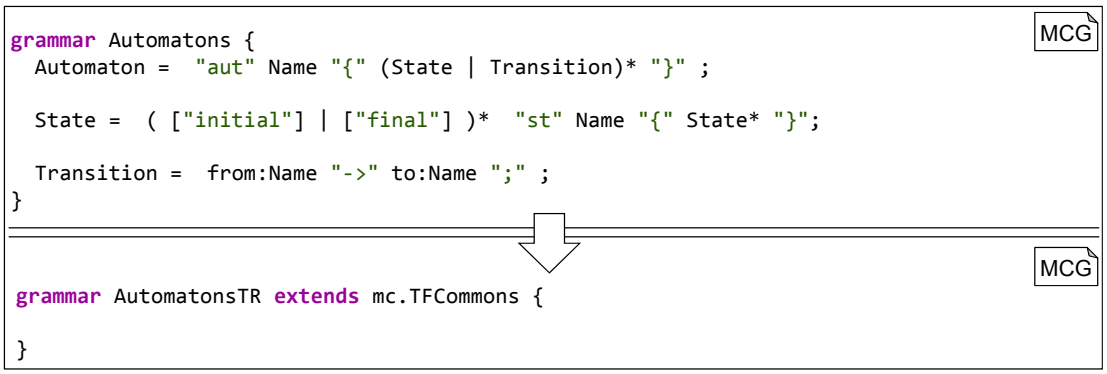


Abbildung 6.3: Regel 1 der DSTL-Ableitung.

Der erste Ableitungsschritt für die `Automatons` Grammatik ist in Abbildung 6.3 dargestellt. Die Grammatik mit allen abgeleiteten Nichtterminalen findet sich in Anhang C.4. Die erste Regel der Ableitung von DSTLs leitet das Grammatikgerüst der DSTL ab. Hierbei wird eine neue Grammatik erstellt. Als Name wird hier der Name der alten Grammatik plus das Suffix `TR` verwendet. Entsprechend Regel 1 erweitert die erstellte Sprache die Basissprache `TFCommons`, falls die Ausgangssprache keine Supergrammatik hat. Die resultierende DSTL hat den Namen `AutomatonsTR`. Da die Beispielgrammatik `Automatons` keine Supersprache hat, erweitert `AutomatonsTR` die Basissprache `TFCommons`.

6.2.2 Regel 2: Interface-Nichtterminale und Externe Nichtterminale

Initial wird für jedes Nichtterminal und jedes relevante Schlüsselwort ein Interface-Nichtterminal in der DSTL-Grammatik erstellt. Dieses Nichtterminal wird genutzt, um die Nichtterminale für die verschiedenen Operatoren, die in den nachfolgenden Regeln erstellt werden, als Alternativen bzw. Implementierungen dieses Nichtterminals zu bündeln. Im Folgenden meint L die Modellierungssprache, TL die zu erstellende DSTL, N, K und I sind Nichtterminale und k ein relevantes Schlüsselwort. Diese Regel teilt sich in die folgenden fünf Teilregeln auf:

Regel 2a *Für jedes Interface-Nichtterminal $N \in L$ erstelle ein Interface-Nichtterminal N in der DSTL TL. Falls $N \in L$ ein Interface-Nichtterminal $I \in L$ erweitert, erweitert $N \in TL$ das Interface-Nichtterminal $I \in TL$.*

Regel 2b *Für jedes abstrakte Nichtterminal $N \in L$ erstelle ein Interface-Nichtterminal N in der DSTL TL. Falls $N \in L$ ein Interface-Nichtterminal $I \in L$ implementiert, erweitert $N \in TL$ das Interface-Nichtterminal $I \in TL$. Falls $N \in L$ ein abstraktes Nichtterminal $Y \in L$ erweitert, erweitert $N \in TL$ das Interface-Nichtterminal $Y \in TL$.*

Regel 2c *Für jedes normale Nichtterminal $N \in L$, falls N kein Nichtterminal einer Supersprache SL von L überschreibt, erstelle ein Interface-Nichtterminal N in der DSTL*

TL. Falls $N \in L$ ein Interface-Nichtterminal $I \in L$ implementiert, erweitert $N \in TL$ das Interface-Nichtterminal $I \in TL$. Falls $N \in L$ ein abstraktes oder normales Nichtterminal $Y \in L$ erweitert, erweitert $N \in TL$ das Interface-Nichtterminal $Y \in TL$.

Regel 2d *Für jedes externe Nichtterminal $N \in L$ erstelle ein externes Nichtterminal N in der DSTL TL.*

Die Vererbungsbeziehungen der Nichtterminale werden innerhalb der DSTLs auf Ebene der in dieser Regel erstellten Interface-Nichtterminale abgebildet. Als Erweiterung der in [Wei12] vorgestellten Generierung von DSTLs wird Vererbung auf Nichtterminalebene damit unterstützt. Das Überschreiben von Nichtterminalen wird unterstützt, in dem auch innerhalb der DSTLs die Nichtterminale einander überschreiben, wobei kein überschreibendes Interface-Nichtterminal erzeugt wird, da Interface-Nichtterminale in MontiCore-Grammatiken nicht überschrieben werden dürfen. Schließlich wird Spracheinbettung unterstützt, indem DSTLs externe Nichtterminale haben können. An diesen Stellen wird dann die DSTLs der eingebetteten Sprache in die DSTLs eingebettet. Abschnitt 6.1 gibt eine detaillierte Erläuterung zu DSTLs für modular definierten DSLs.

Regel 2e *Für jedes relevante Schlüsselwort $k \in L$ erstelle ein Interface-Nichtterminal K in der DSTL TL.*

Der Name dieses Interface-Nichtterminale leitet sich auf die gleiche Weise aus dem Schlüsselwort ab, wie MontiCore den Namen der AST Attribute für Schlüsselworte berechnet, wobei jedoch der erste Buchstabe großgeschrieben wird. Das heißt, aus dem Schlüsselwort `abstract` wird `Abstract`, und `+` wird zu `Plus`.[1]

Der Ableitungsschritt der Interface-Nichtterminale für die `Automatons` Grammatik ist in Abbildung 6.4 dargestellt. Die zweite Regel leitet für alle Nichtterminale sowie jedes relevante Schlüsselwort ein Interface-Nichtterminal innerhalb der abgeleiteten DSTL ab. Diese Interface-Nichtterminale bilden die Basis für die in den nachfolgenden Regeln abgeleiteten Nichtterminale. In der `Automatons`-Grammatik gibt es drei Nichtterminale: `Automaton`, `State` und `Transition`. Daraus entstehen in der abgeleiteten `AutomatonsTR`, die namentlich gleichen Interface-Nichtterminale. Die `Automatons`-Grammatik umfasst außerdem zwei relevante, im AST gespeicherte Schlüsselworte, `initial` und `final`. In der Grammatik ist dies anhand der eckigen Klammern erkennbar. Für jedes dieser relevanten Schlüsselworte wird ebenfalls ein Interface-Nichtterminal in der DSTL abgeleitet. Die Namen dieser Nichtterminale leiten sich aus den Namen der Schlüsselworte ab.

6.2.3 Regel 3: Pattern und Schemavariablen

Um die konkrete Syntax der Modellierungssprache zur Beschreibung von Pattern nutzen zu können, muss diese in die DSTL übertragen werden. Außerdem muss sie dahingehend

[1] Um Namenskollisionen zu vermeiden, kann es notwendig sein alle Interface-Nichtterminale mit einem Präfix wie beispielsweise `ITF` zu versehen bzw. die Interface-Nichtterminale für Schlüsselworte mit einem Suffix wie beispielsweise `_Constant`. Das heißt, aus dem Nichtterminal `Component` wird in der DSTL `ITFComponent` und aus `abstract` wird `Abstract_Constant`.

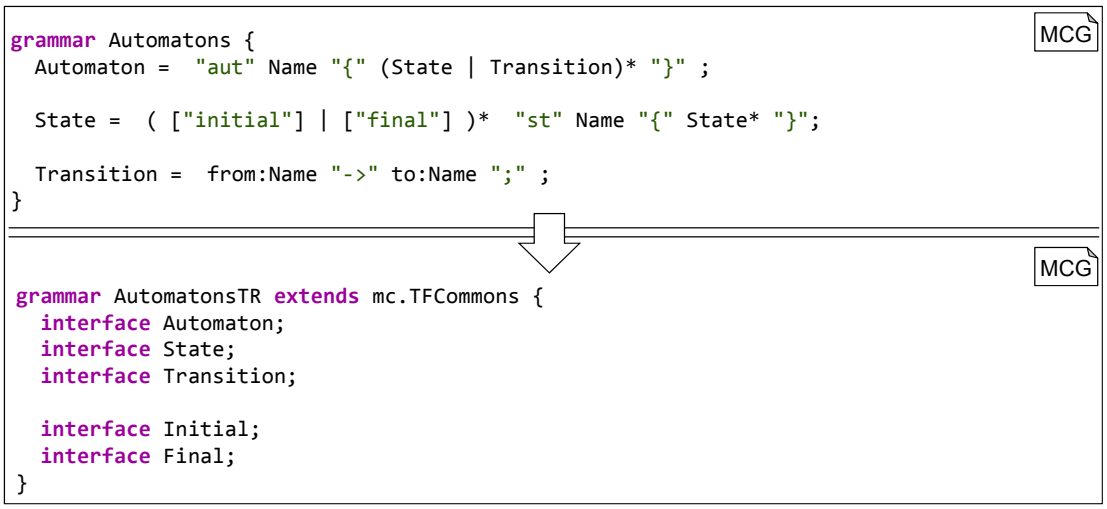


Abbildung 6.4: Regel 2 der DSTL-Ableitung.

angepasst werden, dass die Verwendung von Schemavariablen für das Modellelement sowie für Namen möglich ist. Die folgenden drei Teilregeln beschreiben die Ableitung der Nichtterminale, die diese Beschreibung ermöglichen:

Regel 3a *Für jedes normale Nichtterminal $N \in L$, erstelle ein Nichtterminal $N\_Pat \in TL$ der folgenden Form:*

```
N_Pat implements N =  SyntaxOfN
                    | "N" SchemaVar
                    | "N"? SchemaVar "[[" SyntaxOfN "]]";
```

wobei $SyntaxOfN$ für die angepasste Kopie des Rumpfes der Produktion für $N \in L$ steht. Innerhalb dieser Kopie werden

- *alle Vorkommen von relevanten Schlüsselworten $k \in L$ durch die entsprechenden Interface-Nichtterminale $K \in TL$ ersetzt und*
- *alle Vorkommen des Nichtterminals `Name` durch das Nichtterminal `TfIdentifier`, definiert in der Basisgrammatik `TFCommons`, ersetzt.*

Die Vorkommen von Nichtterminalen im Rumpf von N_Pat zeigen in der DSTL daher nicht auf die Nichtterminale, die die konkrete Syntax erlaubten, sondern auf die in Regel 2 erstellten Interface-Nichtterminale. Dadurch werden an diesen Stellen alle Implementierungen dieser Interface-Nichtterminale erlaubt. Das heißt, es wird sowohl die Verwendung der konkreten Syntax inklusive Schemavariablen ermöglicht als auch die Verwendung der in den folgenden Ableitungsregeln erstellten Operatoren.[2]

[2] Wurden in Regel 1 Suffixe oder Präfixe für die Namen der Nichtterminale erstellt, so müssen die Nichtterminale im Rumpf N entsprechend angepasst werden, um auf die Interface-Nichtterminale aus Regel 1 zu verweisen.

Regel 3b *Für jedes abstrakte und jedes Interface-Nichtterminal $N \in L$, erstelle ein Nichtterminal $N\_Pat \in TL$ der folgenden Form:*

```
N_Pat implements N =
        "N" SchemaVar | "N"? SchemaVar "[[" N "]]";
```

Interface und abstrakte Nichtterminale definieren selbst keine konkrete Syntax. Die konkrete Syntax dieser Nichtterminale ist durch ihre Implementierungen bzw. Erweiterungen durch normale Nichtterminale definiert. Um dem Transformationsentwickler jedoch die Möglichkeit zu geben es offen zu lassen, welche Implementierung an der Stelle im Modell erwartet wird, erstellt die Regel 3b Schemavariablen vom Typ des Interface- bzw. abstrakten Nichtterminals. Dies ist beispielsweise für das Interface `Type` der von MontiCore bereitgestellten Types-Grammatik nützlich, welches von verschiedenen Nichtterminale implementiert wird. Ist es für die Transformation irrelevant, welche Implementierung von `Type` im Modell beispielsweise als Teil eines Attributs vorkommt, kann hierfür die Schemavariable für `Type` verwenden.

Regel 3c *Für jedes relevante Schlüsselwort $k \in L$, erstelle ein Nichtterminal $K\_Pat \in TL$ der folgenden Form:*

```
K_Pat implements K = k;
```

Die konkrete Syntax eines Schlüsselworts ist das Schlüsselwort selbst. Es wird hier keine Schemavariable benötigt und somit gibt es im Rumpf keine Alternativen.

```
State =  ( ["initial"] | ["final"] )*  "st" Name "{" State* "}";
```

MCG …

MCG …

```
grammar AutomatonsTR extends mc.TFCommons {

  State_Pat implements State  =
   (
     (Initial | Final)* "st" name:TfIdentifier "{" State* "}"
   ) |
     "State" schemaVarName:Name
   | (
     "State"? schemaVarName:Name "[["(
       (Initial | Final)* "st" name:TfIdentifier "{" State* "}")
     "]]");

  Initial_Pat implements Initial =
     "initial";
}
```

Abbildung 6.5: Regel 3 der DSTL-Ableitung.

Abbildung 6.5 zeigt die Ableitung für das `State`-Nichtterminal der `Automatons`-DSL sowie das relevante Schlüsselwort `initial`. Die dritte Ableitungsregel produziert

die Nichtterminale, die die Syntax der Modellierungssprache in der DSTL bereitstellen. Für jedes normale Nichtterminal der Modellierungssprache sowie für jedes relevante Schlüsselwort werden Nichtterminale mit dem Suffix `_Pat` erzeugt. Diese Nichtterminale implementieren das zugehörige Interface-Nichtterminal das durch Regel 2 erstellt wurde. Dadurch bilden sie die erste erstellte Implementierung dieses Interface-Nichtterminals. Für das `State`-Nichtterminal wird ein `State_Pat` Nichtterminal erstellt. Das `State_Pat`-Nichtterminal implementiert das Interface-Nichtterminal `State` der DSTL. Für den Rumpf der abgeleiteten `State_Pat` Produktion leitet diese Regel drei Alternativen ab. Die erste Alternative ist eine angepasste Kopie des Rumpfes der `State`-Produktion. Die Kopie wurde dabei an zwei Stellen angepasst. Zum einen wurden die relevanten Schlüsselworte durch Nichtterminale ersetzt. Hierbei wurde `["initial"]` durch das Nichtterminal `Initial` und `["final"]` durch `Final` ersetzt. An diesen Stellen wird statt des Schlüsselworts das Interface-Nichtterminal verwendet, wodurch an diesen Stellen alle Implementierungen des Interface-Nichtterminals erlaubt werden. Die Implementierungen werden in den Regeln 3-7 erzeugt. Zum anderen wurden alle Vorkommen des Nichtterminals `Name` durch `TfIdentifier` ersetzt. `TfIdentifier` stammt aus der Basisgrammatik `TFCommons` und ermöglicht wie zuvor die Angabe eines Namens. Zusätzlich erlaubt `TfIdentifier` Ersetzungen von Namen sowie die Verwendung von Schemavariablen. Die sonstigen verwendeten Nichtterminale werden nicht ersetzt. Im Rumpf auftretende Nichtterminale zeigen bereits auf die erstellten Interface-Nichtterminale, da für diese der gleiche Name wie der der Nichtterminale in der Modellierungssprache verwendet wurde. Für das `State`-Nichtterminal bedeutet dies, dass die Verwendung des `State`-Nichtterminals im Rumpf der Produktion während der Ableitung nicht verändert wurde.

Die zweite Alternative des Rumpfs der `State_Pat`-Produktion ermöglicht die Abstraktion durch getypte Schemavariablen wie in Abschnitt 4.4.2 beschrieben. Dazu wurde der Name des Nichtterminals – State – als Terminal und damit als Typangabe der Schemavariablen verwendet.

Die dritte Alternative erlaubt die Kombination aus Schemavariable und konkreter Syntax. Hierbei ist das Terminal `State` und damit die Typangabe optional, da diese aus der folgenden konkreten Syntax abgeleitet werden kann. Innerhalb der eckigen Klammern – den Terminalen, die die Trennzeichen `[[` und `]]` ermöglichen – findet sich ebenfalls eine Kopie des Rumpfes der `State`-Produktion. Die gleichen Anpassungen wie bei der Kopie für die erste Alternative wurden vorgenommen: `Name` wurde durch `TfIdentifier`, `["initial"]` durch `Initial` und `["final"]` durch `Final` ersetzt.

Für `["initial"]` wurde das Nichtterminal `Initial_Pat` erstellt. `Initial_Pat` implementiert das Interface-Nichtterminal `Initial`. Für den Rumpf wurde die konkrete Syntax des Schlüsselworts, also `"initial"` übernommen.

6.2.4 Regel 4: Replacement Operator

Der Replacement Operator erlaubt die Spezifikation von Modifikationen am Modell. Um eine Modifikation jedes Modellelements zu erlauben, wird für jedes Nichtterminal und jedes relevante Schlüsselwort aus L ein Replacement Operator erstellt. Die Modifikation

von Namen wird bereits durch das Nichtterminal `TfIdentifier` der Basisgrammatik `TFCommons` ermöglicht. Die Ersetzung von Modellelementen und Schlüsselworten unterscheidet sich, daher teilt sich diese Regel in die folgenden zwei Teilregeln:

Regel 4a *Für jedes normale, abstrakte und Interface-Nichtterminal $N \in L$, erstelle ein Nichtterminal $N\_Rep \in TL$ der folgenden Form:*

```
N_Rep implements N = [[ lhs:N? :- rhs:N? ]];
```

Durch die *N_Rep*-Nichtterminale wird es für Modellelemente möglich diese durch einander zu ersetzen, hinzuzufügen oder aus dem Modell zu entfernen.

Regel 4b *Für jedes relevante Schlüsselwort $k \in L$, erstelle ein Nichtterminal $K\_Rep \in TL$ der folgenden Form:*

```
K_Rep implements K =
        [[ k :- ]] | [[ :- k ]];
```

Schlüsselworte werden zu Nichtterminalen geliftet und jeweils einzeln behandelt. Da die Ersetzung eines Schlüsselwortes durch sich selbst nicht benötigt wird, erlaubt der Operator für Schlüsselworte lediglich das Hinzufügen oder Entfernen des jeweiligen Schlüsselworts. Es ist zu beachten, dass eine solch systematische Behandlung von Schlüsselworten nicht in jedem Fall intuitiv ist. Beispielsweise ist das Entfernen von `private` und das Hinzufügen von `public` weniger intuitiv für den Transformationsentwickler als die direkte Ersetzung von `public` durch `private`. Daher kann in einigen Fällen die Benutzbarkeit der systematisch erstellten DSTL durch manuelle Erweiterung optimiert werden.

```
State =  ( ["initial"] | ["final"] )*  "st" Name "{" State* "}";
```

MCG
...

MCG
...

```
grammar AutomatonsTR extends mc.TFCommons {

  State_Rep implements State =
    "[[" lhs:State? ":-" rhs:State? "]]";

  Initial_Rep implements Initial =
    "[[" lhs:Initial_Pat ":-" "]]"
    |
    "[[" ":-" rhs:Initial_Pat "]]";
}
```

Abbildung 6.6: Regel 4 der DSTL-Ableitung.

Abbildung 6.6 zeigt diesen Ableitungsschritt für das `State`-Nichtterminal sowie für das relevante Schlüsselwort `initial`. Die vierte Regel erstellt Nichtterminale für den Replacement Operator. Diese werden sowohl für Nichtterminale als auch für relevante

Schlüsselworte erstellt. Für das `State`-Nichtterminal wird das `State_Rep`-Nichtterminal erstellt. `State_Rep` implementiert das Interface-Nichtterminal `State`. Der Rumpf der `State_Rep`-Produktion erlaubt die konkrete Syntax des Operators (`[[ :- ]]`) sowie rechts und links des `:-` jeweils optional ein `State`-Nichtterminal. Die Nichtterminale `Automaton_Rep` für `Automaton` und `Transition_Rep` für Transition werden analog abgeleitet, implementieren die Interface-Nichtterminale `Automaton` bzw. `Transition` und verwenden im Rumpf das Nichtterminal `Automaton` bzw. `Transition`. Für `initial` wird das Nichtterminal `Initial_Rep` erzeugt. `Initial_Rep` implementiert das Interface-Nichtterminal `Initial`. Der Rumpf erlaubt die konkrete Syntax des Operators (`[[ :- ]]`) sowie rechts oder links des `:-` jeweils das Schlüsselwort `initial`, da `intial` durch sich selbst ersetzen keinen Mehrwert schafft. Das Nichtterminal `Finale_Rep` wird analog erzeugt.

6.2.5 Regel 5: Negative Elemente

Des Weiteren werden Nichtterminale für jedes Nichtterminal und jedes relevante Schlüsselwort erstellt, die die Spezifikation negativer Elemente erlauben. Die folgenden Teilregeln beschreiben die Ableitung und den Aufbau dieser Nichtterminale:

Regel 5a *Für jedes normale, abstrakte und Interface-Nichtterminal $N \in L$, erstelle ein Nichtterminal N_Neg der folgenden Form:*

```
N_Neg implements N = not [[ N ]];
```

Regel 5b *Für jedes relevante Schlüsselwort $k \in L$, erstelle ein Nichtterminal $K\_Neg \in TL$ der folgenden Form:*

```
K_Neg implements K = not [[ k ]];
```

Nach Anwendung dieser Regel existiert in der DSTL die Möglichkeit die verschiedenen Modellelemente zu verbieten. Außerdem kann auch das Auftreten relevanter Schlüsselworte verboten werden.

```
State =  ( ["initial"] | ["final"] )*  "st" Name "{" State* "}";
```

MCG …

```
grammar AutomatonsTR extends mc.TFCommons {

  State_Neg implements Automaton =
    "not" "[[" State "]]";

  Initial_Neg implements Initial =
    "not" "[[" "initial" "]]";
}
```

MCG …

Abbildung 6.7: Regel 5 der DSTL-Ableitung.

Abbildung 6.7 zeigt diesen Ableitungsschritt für das `State`-Nichtterminal sowie für das relevante Schlüsselwort `initial`. Die fünfte Regel leitet Nichtterminale für negative Elemente ab. Diese werden sowohl für Nichtterminale als auch für relevante Schlüsselworte erstellt. Für das `State`-Nichtterminal wird das `State_Neg`-Nichtterminal erstellt. `State_Neg` implementiert das Interface-Nichtterminal `State`. Der Rumpf der `State_Neg`-Produktion erlaubt die konkrete Syntax des Operators (`not [[ ]]`) sowie ein `State`-Nichtterminal zwischen den doppelten eckigen Klammern. Die Nichtterminale `Automaton_Neg` für `Automaton` und `Transition_Neg` für Transition werden analog abgeleitet, implementieren die Interface-Nichtterminale `Automaton` bzw. `Transition` und verwenden im Rumpf das Nichtterminal `Automaton` bzw. `Transition`. Für `initial` wird das Nichtterminal `Initial_Neg` erzeugt. `Initial_Neg` implementiert das Interface-Nichtterminal `Initial`. Der Rumpf erlaubt die konkrete Syntax des Operators (`not [[ initial ]]`). Das Nichtterminal `Final_Neg` für das Schlüsselwort `final` wird analog abgeleitet, implementiert das Interface-Nichtterminal `Final` und verwenden im Rumpf das Schlüsselwort `final`.

6.2.6 Regel 6: Collection Operator

Diese Ableitungsregel erstellt die notwendigen Nichtterminale zur Verwendung des Collection Operators. Die Nichtterminale werden wie folgt erstellt:

Regel 6 *Für jedes normale, abstrakte und Interface-Nichtterminal $N \in L$, erstelle ein Nichtterminal $N\_List \in TL$ der folgenden Form:*

```
N_List implements N = list SchemaVar? [[ N ]];
```

Die Erstellung eines Collection Operators für Schlüsselworte ist nicht notwendig. Jedes Schlüsselwort wird auf Nichtterminalebene geliftet und somit separat behandelt. Dadurch würde eine Collection dieser Schlüsselworte bestenfalls ein Modell beschreiben, dass an der beschriebenen Position mehrfach das gleiche Schlüsselwort hat. Diese Information wird durch MontiCore jedoch nicht im AST hinterlegt (vgl. Abschnitt 2.1.2), wodurch ein Collection Operator hier keinen Mehrwert hat, sondern das Verständnis für den Transformationsentwickler erschwert.

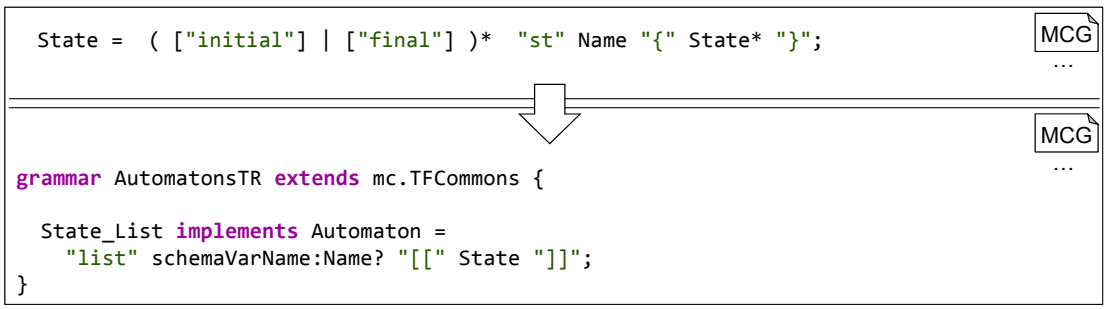


Abbildung 6.8: Regel 6 der DSTL-Ableitung.

Abbildung 6.8 zeigt diesen Ableitungsschritt für das `State`-Nichtterminal. Die sechste Regel leitet Nichtterminale für den Collection Operator ab. Diese werden nur für Nichtterminale und nicht für Schlüsselworte erstellt. Für das `State`-Nichtterminal wird das `State_List`-Nichtterminal erstellt. `State_List` implementiert das Interface-Nichtterminal `State`. Der Rumpf der `State_List` Produktion erlaubt die konkrete Syntax des Operators (`list [[ ]]`) sowie ein `State`-Nichtterminal zwischen den doppelten eckigen Klammern. Die Nichtterminale `Automaton_List` für `Automaton` und `Transition_List` für `Transition` werden analog abgeleitet, implementieren die Interface-Nichtterminale `Automaton` bzw. `Transition` und verwenden im Rumpf das Nichtterminal `Automaton` bzw. `Transition`.

6.2.7 Regel 7: Optional Operator

Die letzte Regel zur Erstellung von Nichtterminalen für Operatoren erstellt die Nichtterminale für den Optional Operator. Auch diese Regel gliedert sich in zwei Teilregeln: eine für Nichtterminale und eine für relevante Schlüsselworte.

Regel 7a *Für jedes normale, abstrakte und Interface-Nichtterminal $N \in L$, erstelle ein Nichtterminal $N\_Opt \in TL$ der folgenden Form:*

```
N_Opt implements N = opt [[ N ]];
```

Regel 7b *Für jedes relevante Schlüsselwort $k \in L$, erstelle ein Nichtterminal $K\_Opt \in TL$ der folgenden Form:*

```
K_Opt implements K = opt [[ k ]];
```

Nach Anwendung dieser Regel existiert in der DSTL die Möglichkeit die verschiedenen Patternelemente als optional zu markieren. Außerdem kann auch das Auftreten relevanter Schlüsselworte optional sein.

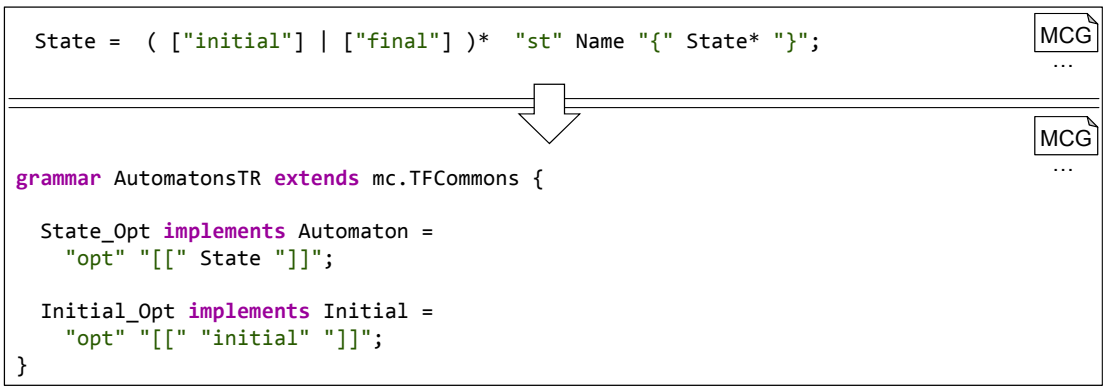


Abbildung 6.9: Regel 7 der DSTL-Ableitung.

Abbildung 6.9 zeigt diesen Ableitungsschritt für das `State`-Nichtterminal sowie für das relevante Schlüsselwort `initial`. Die siebte Regel leitet Nichtterminale für den Optional Operator ab. Diese werden sowohl für Nichtterminale als auch für relevante Schlüsselworte erstellt. Für das `State`-Nichtterminal wird das `State_Opt`-Nichtterminal erstellt. `State_Opt` implementiert das Interface-Nichtterminal `State`. Der Rumpf der `State_Opt`-Produktion erlaubt die konkrete Syntax des Operators (`opt [[ ]]`) sowie ein `State`-Nichtterminal zwischen den doppelten eckigen Klammern. Die Nichtterminale `Automaton_Opt` für `Automaton` und `Transition_Opt` für `Transition` werden analog abgeleitet, implementieren die Interface-Nichtterminale `Automaton` bzw. `Transition` und verwenden im Rumpf das Nichtterminal `Automaton` bzw. `Transition`. Für `initial` wird das Nichtterminal `Initial_Opt` erzeugt. `Initial_Opt` implementiert das Interface-Nichtterminal `Initial`. Der Rumpf erlaubt die konkrete Syntax des Operators (`opt [[ initial ]]`). Das Nichtterminal `Final_Opt` für das Schlüsselwort `final` wird analog abgeleitet, implementiert das Interface-Nichtterminal `Final` und verwenden im Rumpf das Schlüsselwort `final`.

6.2.8 Regel 8: Startsymbol der DSTLs

Als finaler Schritt der Ableitung muss das Startsymbol der DSTL erstellt werden. Dazu werden die in den vorherigen Regeln erstellten Nichtterminale durch die folgende Regel kombiniert und um die modellierungssprachenunabhängigen Anteile, d.h. beispielsweise den Application Constraint, den Zuweisungsblock und den Anweisungsblock, ergänzt:

Regel 8 *Erstelle ein Nichtterminal* `TFRule` *der folgenden Form:*

```
TFRule = (AlternativeOfNTs)* Folding? Where? Assign? Do?;
```

wobei $AlternativeOfNTs$ eine Alternative aller erstellten Interface-Nichtterminale ist, die für Nichtterminale von L erstellt wurden.

Diese Regel liftet alle Modellelemente auf die oberste Ebene einer Transformationsregel. Dadurch müssen Transformationsregeln zur Beschreibung von Pattern nicht beim Startsymbol der Modellierungssprache ansetzen und erlauben eine Fokussierung auf den für die Transformationsregel relevanten Modellausschnitt. Außerdem erlaubt die Regel auch die Angabe mehrerer Modellelemente nebeneinander, ohne dass diese ein gemeinsames Elternelement haben müssen. Schließlich ist durch diese Regel gewährleistet, dass eine Transformationsregel ein Pattern über mehrere Modelle spezifizieren kann.

Für die `Automatons`-Grammatik ist dieser Ableitungsschritt in Abbildung 6.10 dargestellt. Die achte Ableitungsregel produziert das Startsymbol der `AutomatonsTR`-Grammatik. Das Startsymbol erlaubt beliebig viele Modellelemente durch Patternelemente zu beschreiben. Insbesondere ist es möglich beliebige, durch Nichtterminale definierte Modellelemente im Pattern nebeneinander zu beschreiben. Dazu leitet die Regel eine Alternative aller für Nichtterminale der Ausgangssprache erstellten Interface-Nichtterminale ab und erlaubt beliebig viele dieser Nichtterminale nebeneinander. Anschließend folgen optional der Folding Operator, die Zuweisungen, der Application Cons-

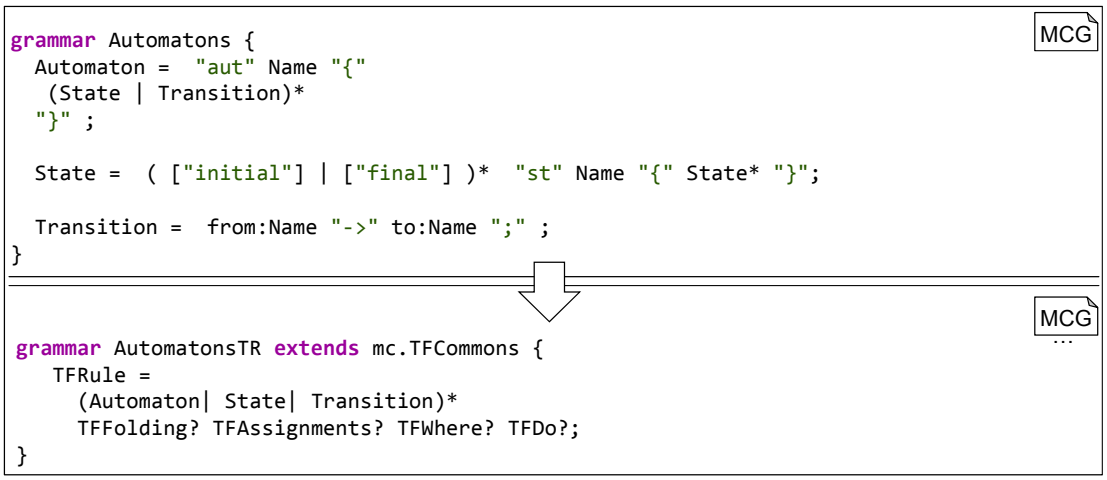


Abbildung 6.10: Regel 8 der DSTL-Ableitung.

traint sowie der Anweisungsblock. Für die `Automatons`-Grammatik besteht die Alternative aus den Interface-Nichtterminalen `Automaton`, `State`, und `Transition`. Die restlichen Nichtterminale werden von der Basissprache `TFCommons` bereitgestellt.

6.3 Basissprache TFCommons

Sprachunabhängige Konzepte, d.h. Konzepte, die für jede Transformationssprache in gleicher Form verwendet werden können, werden durch die Basissprache *TFCommons* bereitgestellt. Zu den sprachunabhängigen Konzepten der Transformationssprache gehören: der `where`-Block zur Formulierung von Constraints, der `assign`-Block zur Spezifikation von Schemavariablenzuweisungen, das Folding zur Definition von nicht-isomorphen Matches und der `do`-Block zur direkten Manipulation der Eingabemodelle sowie weiterer Aktionen wie Reporting oder Analysen.

Listing 6.11 zeigt einen Auszug aus der Basissprache *TFCommons*, der die Nichtterminale für die Modellierungssprachen-unabhängigen Features zeigt. Die *TFCommons* basiert auf der *JavaDSL* und verwendet hiervon die Nichtterminale `BlockStatement` und `Expression`. Die abgeleiteten DSTLs basieren auf dieser Basissprache und verwenden deren Nichtterminale.

6.4 Generierung einer neuen DSTL mit MontiTrans

In den Abschnitten zuvor wurden Ableitungsregeln zur Ableitung von DSTLs aus (MontiCore) Grammatiken beschrieben. Die beschriebene Methodik zur Ableitung neuer DSTLs ist komplett systematisch und damit automatisierbar. In MontiTrans wurde sie daher als Generator umgesetzt.

Abbildung 6.1 zeigt erneut die bereits in Abbildung 1.1 vorgestellte dreischichtige Darstellung der Transformationssprachen-, Transformationsentwicklung sowie Transforma-

```
grammar TFCommons extends JavaDSL {                                    MCG
  TfIdentifier =
    ident:Name | ("[[" ident:Name? ":-" newIdent:Name? "]]");

  Folding = "folding" "{" FoldingSet* "}";

  Assignments = "assign" "{" Assign* "}";

  Assign = variable:Name "=" value:Expression ";";

  Where = "where" "{" constraint:Expression "}";

  Do = "do" BlockStatement;
}
```

Listing 6.11: Die Basissprache `TFCommons`.

tionsanwendung mit Fokus auf der Generierung und Ableitung von DSTLs. Der MontiTrans DSTL-Generator befindet sich hierbei in der obersten Schicht der Transformationssprachenentwicklung und adressiert den Sprachentwickler (vgl. Abbildung 6.1). Aus Nutzersicht bekommt der Generator die Grammatik der Modellierungssprache, für die er die DSTL generieren möchte, und produziert die DSTL plus einen Generator, der Modelle der Sprache – Transformationsregeln – einliest und die entsprechende Java-Implementierung der Transformationsregeln produziert (vgl. Abbildung 6.1, oberer Bereich). Abbildung 6.12 zeigt die Details der DSTL-Generierung mit MontiTrans. Die Generierung und die beteiligten Komponenten werden im Folgenden detailliert.

MontiTrans stellt den MontiTrans DSTL-Generator zur Verfügung. Dieser wird durch das Groovy-Skript *dstlgen* [KKL+15] gesteuert (in Abbildung 6.12 mit «rte» gekennzeichnet). Das Groovy-Skript benötigt zur Ausführung eine Base Class. Die Base Class wird ebenfalls durch MontiTrans bereitgestellt. Als weitere Komponenten stellt MontiTrans außerdem einen Generator (`Java Generator` in Abbildung 6.12) zur Generierung von Java-Implementierungen für Transformationsregeln beschrieben als ODRule (Erklärung der Notation in Abschnitt 8.2) und die Basisgrammatik der DSTLs `TFCommons` zur Verfügung. Der MontiTrans DSTL-Generator bekommt eine (oder mehrere) Grammatik(en) im MontiCore-Format als Input. Basierend auf dieser Grammatik erzeugt der Generator die Grammatik der DSTL, Implementierungen der in Abschnitt 4.14 beschriebenen Kontextbedingungen, einen Generator der Transformationsregeln in DSTL-Syntax in Transformationsregeln in ODRules-Syntax übersetzt (`ODRule Generator` in Abbildung 6.12). Zusätzlich generiert MontiTrans die Komposition der beiden Generatoren `ODRule Generator` und `Java Generator` (`Transformation Generator` in Abbildung 6.12). Schließlich erzeugt der Generator eine Base Class für das Groovy-Skript, das den kompositionalen Generator steuert und eine Main-Klasse für das Command Line Tool. Die Verwendung des Generators wird in Kapitel 9 beschrieben.

Der DSTL-Generatorablauf wird durch das Groovy-Skript *dstlgen* gesteuert. Abbildung 6.13 zeigt diesen Ablauf während Abbildung 6.14 die wichtigsten Klassen des Gene-

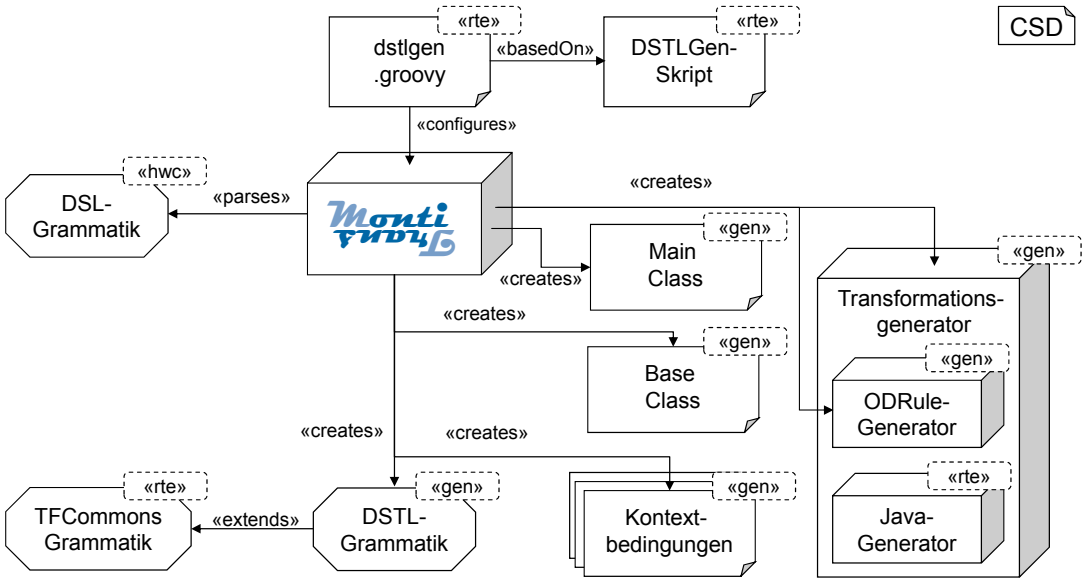


Abbildung 6.12: Generierung einer DSTL mit MontiTrans (Detailsicht).

rators darstellt. Der Generator liest die Grammatik der Modellierungssprache M ein, für die die DSTL generiert wird. Dann erstellt der Generator die Symboltabelle der Sprache. Falls M Supergrammatiken erweitert, dann werden die Symboltabellen dieser Sprachen geladen. Danach wird die Grammatik der DSTL generiert. Die Generierung der Grammatik folgt den zuvor in Abschnitt 6.2 beschriebenen Ableitungsregeln. Der Generator entspricht einer exogenen Modelltransformation [CH06, MVG06] mit anschließendem Pretty Printing des resultierenden Modells, also der DSTL-Grammatik. Die Modelltransformation erfolgt visitorbasiert durch den `DSL2DSTLVisitor` (vgl. Abbildung 6.14). Das heißt, mittels des Visitors wird die Struktur der Grammatik traversiert und abhängig von dem aktuellen Grammatikelement (Grammatikdefinition (`ASTMCGrammar`), Nichtterminale (`ASTClassProd, ASTInterfaceProd` und `ASTAbstractProd`), Konstanten für Schlüsselworte (`ASTConstantGroup`)) den Ableitungsregeln entsprechende Grammatikelemente der DSTL-Grammatik erzeugt.

Anschließend werden die Implementierungen der in Abschnitt 4.14 vorgestellten Kontextbedingungen generiert (vgl. Abbildung 6.13). Die Generierung der Implementierungen der Kontextbedingungen erfolgt templatebasiert mithilfe der von MontiCore bereitgestellten `GeneratorEngine` (vgl. Abbildung 6.14). In Kapitel 4 wurden die Kontextbedingungen allgemeingültig beschrieben, beispielsweise keine negativen Elemente auf der rechten Seite von Ersetzungen. Für jede einzelne DSTL werden diese dann spezifisch umgesetzt. Die einzelnen Operatoren werden über spezifische Nichtterminale in der DSTL abgebildet. Die von MontiCore generierte Kontextbedingungeninfrastruktur sieht vor, dass Kontextbedingungen für die einzelnen Nichtterminale einer Sprache formuliert werden. Für die DSTLs bedeutet dies, dass eine Kontextbedingung über Ersetzungen für jedes `_Rep`-Nichtterminal formuliert werden muss. Aus diesem Grund werden die Implementierungen der Kontextbedingungen zusammen mit der DSTL generiert.

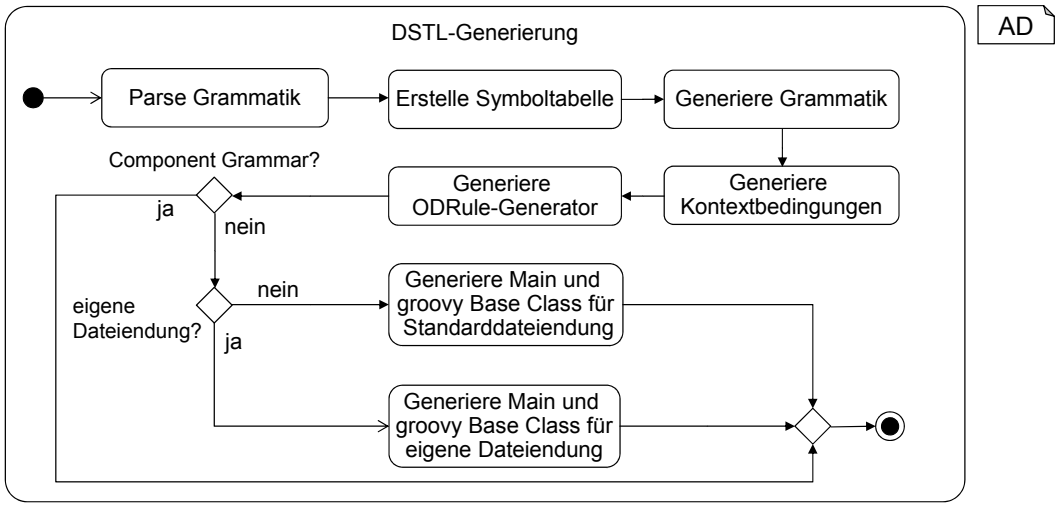


Abbildung 6.13: Ablauf der Generierung einer DSTL mit MontiTrans.

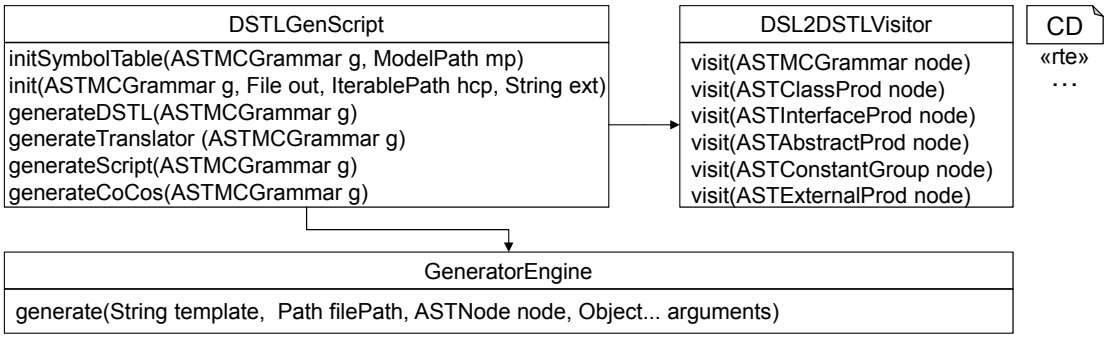


Abbildung 6.14: Struktur des MontiTrans DSTL Generators.

Abbildung 6.15 zeigt exemplarisch den Zusammenhang zwischen Nichtterminalen der DSTL und den generierten Kontextbedingungen. Auf der linken Seite ist das Nichtterminal `CDAttribute_Rep` zu sehen. Auf der rechten Seite ist die zugehörige Implementierung der Kontextbedingung, die den Optional Operator auf der rechten Regelseite des Operators verbietet, zu sehen. Generiert wird die Klasse `NoOptWithinRHSCoCo_CDAttribute`, die das Interface `CD4AnalysisTRASTCDAttribute_RepCoCo` implementiert. Das Interface wird von MontiCore für das Nichtterminal `CDattribute_Rep` der DSTL-Grammatik generiert. Die generierte Klasse implementiert die durch das Interface vorgegebene check()-Methode. Ist die rechte Regelseite im AST präsent (`rhsIsPresent()`), das heißt im Modell wurde auf der rechten Seite des Replacement Operators für Attribute ein Attribut angegeben, wird diese mittels eines Visitors [GHJV95] traversiert (accept(optChecker)). Findet der Visitor (optChecker) innerhalb der rechten Regelseite einen Optional Operator (_Opt ASTKlasseninstanz) wurde die Kontextbedingung verletzt und es wird die entsprechende Fehlermeldung ausgegeben.

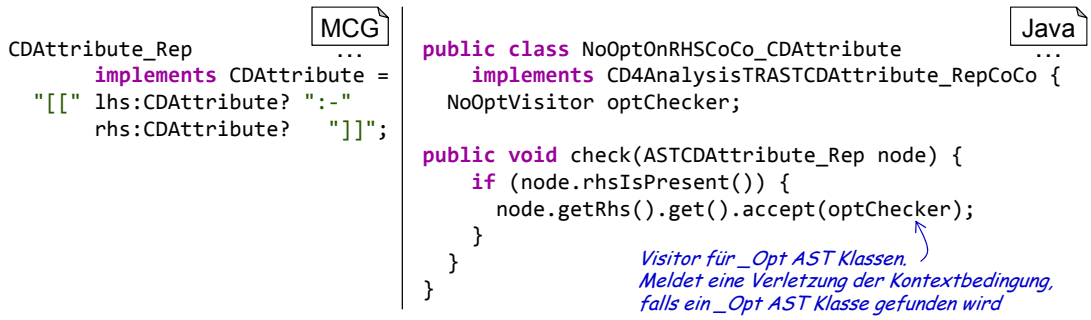


Abbildung 6.15: Mapping von `CDAttribute_Rep` Nichtterminal zur Kontextbedingung, die den Optional Operator in der RHS verbietet.

Die Implementierungen der anderen Kontextbedingungen werden nach dem gleichen Schema generiert und daher hier nicht weiter detailliert.

In nächsten Schritt wird der ODRules Generator erstellt (vgl. Abbildung 6.13). Die Generierung erfolgt – wie die Generierung der Implementierungen von Kontextbedingungen – templatebasiert mithilfe der `GeneratorEngine` (vgl. Abbildung 6.14). Ähnlich wie die Übersetzung der DSL-Grammatik zur DSTL-Grammatik erfolgt auch die Übersetzung der Transformationsregeln (Modelle der DSTL) als M2M-Transformation. Auch diese M2M-Transformation wird durch einen Visitor realisiert. Dieser Visitor traversiert den AST der Transformationsregel und erstellt dabei den AST der entsprechenden Transformationsregel in ODRules-Notation. Die ODRules-Notation sowie das Mapping zwischen DSTL-Notation und ODRules-Notation werden in Kapitel 8 erläutert.

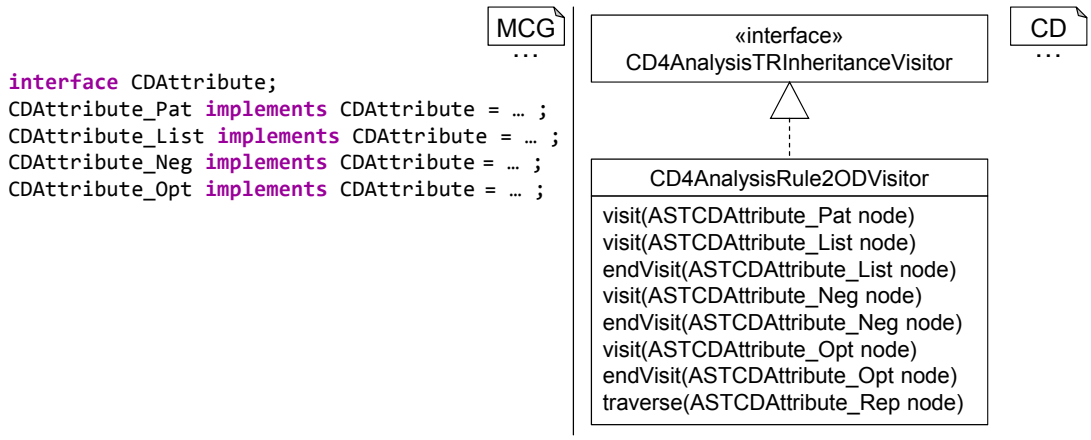


Abbildung 6.16: Mapping von `CDAttribute` Nichtterminalen zu Visitor, der die Übersetzung von CDTrans Modell zu ODRules Modell realisiert.

Abbildung 6.16 zeigt exemplarisch den Zusammenhang zwischen den Nichtterminalen der CDTrans-Grammatik, die für das `CDAttribute`-Nichtterminal der CD4Code-DSL erstellt werden, und dem generierten Visitor zur Übersetzung von CDTrans-Modellen

zu ODRules-Modellen. Links sind hierzu die Nichtterminale `CDAttribute`, `CDAttribute_Pat`, `CDAttribute_List`, `CDAttribute_Neg`, `CDAttribute_Opt` und `CDAttribute_Rep` abgebildet. Rechts in Form eines Klassendiagramms der `CD4AnalysisRule2ODVisitor` abgebildet. Der generierte `CD4AnalysisRule2ODVisitor` implementiert das von MontiCore zu der CDTrans DSTL generierte Interface `CD4AnalysisTRInheritanceVisitor`. Die Generierung der Visitorinfrastruktur ist in [HMSNRW16] erläutert. Das Interface stellt die Standardimplementierungen der `visit`-, `traverse` bzw. `endVisit`-Methoden zur Verfügung. Der generierte Visitor überschreibt daher lediglich die Methoden, bei denen die Standardimplementierung nicht ausreichend ist. Für das hier betrachtete Beispiel `CDAttribute` implementiert der Visitor die `visit`-Methoden für `CDAttribute_Pat`, `CDAttribute_List`, `CDAttribute_Neg` und `CDAttribute_Opt`. Außerdem implementiert der Visitor die `endVisit`-Methoden für `CDAttribute_List`, `CDAttribute_Neg` und `CDAttribute_Opt` sowie die `traverse`-Methode für `CDAttribute_Rep`. Insgesamt werden alle `visit`-Methoden für `_Pat`, `_List`, `_Neg`- und `_Opt`-Nichtterminale, alle `endVisit`-Methoden für `_List`, `_Neg` und `_Opt`-Nichtterminale und alle `traverse`-Methoden für `_Rep`-Nichtterminale implementiert.

Zur Generierung der Base Class für das Groovy-Skript zur Generierung von Java-Implementierungen der Transformationsregeln wird zunächst geprüft, ob es sich um eine Grammatikkomponente handelt (vgl. Abbildung 6.13). Ist dies der Fall, wird die Generierung einer Base Class für die DSTL übersprungen. Da es in diesem Fall keine Modelle von M gibt, dient die generierte DSTL nur als Baustein für eine DSTL für eine Subsprache von M. Falls es sich nicht um eine Grammatikkomponente handelt, wird überprüft, ob dem Aufruf eine Dateiendung für die Modelle der DSTL mitgegeben wurde. Ist dies der Fall wird eine Base Class für diese Endung generiert. Andernfalls wird eine Base Class für die Standarddateiendung *mtr* erzeugt. Der Aufbau und die Steuerung des Transformation Generators sowie der Ablauf der Generierung von Implementierungen zu modellierten Transformationsregeln werden in Kapitel 8 detailliert. Genau wie der Generator zur Generierung von DSTLs wird der Generator zur Generierung von Transformationsimplementierungen durch ein Groovy-Skript basierend auf einer Base Class in Java gesteuert. Das Groovy-Skript ist dabei für jede DSTL gleich, die zugrundeliegende Base Class hingegen ist DSTL-spezifisch und wird daher zusammen mit der DSTL generiert. Die Generierung erfolgt templatebasiert.

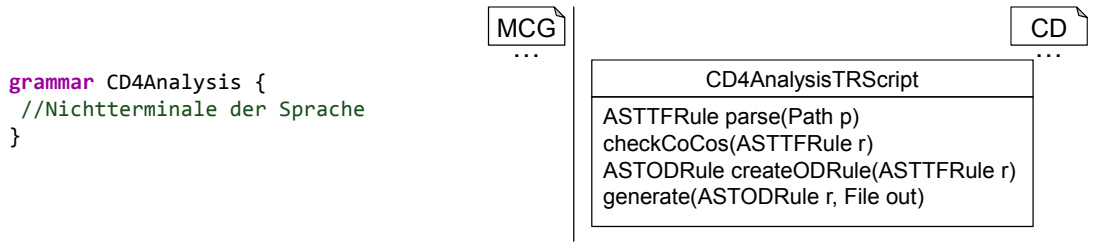


Abbildung 6.17: Mapping von CD4Analysis-Grammatik zur generierten Base Class.

Abbildung 6.17 zeigt exemplarisch für die CD4Analysis-Grammatik das Mapping von DSL zur generierten Groovy Base Class. Links ist die Grammatik und rechts die resultierende Base Class in Form eines Klassendiagramms abgebildet. Für die CD4Analysis-Grammatik wird die Base Class `CD4AnalysisTRScript` generiert. Diese Klasse bieten die Signaturen:

ASTTFRule parse(Path p) zum Einlesen von Modellen der DSTL CDTrans. Die Methode erwartet als Eingabe einen Pfad, der die Grammatik enthält, und liefert den AST des Modells zurück.

checkCoCos(ASTTFRule r) zur Überprüfung der Wohlgeformtheit anhand der generierten Kontextbedingungen. Die Methode erwartet das zu prüfende Modell in Form eines ASTs als Eingabe.

ASTODRule createODRule(ASTTFRule r) zur Erstellung der ODRule anhand einer übergebenen Transformationsregel in Form eines ASTs.

generate(ASTODRule r, File out) zur Generierung einer Java-Implementierung zu einer Transformationsregel in ODRules-Notation. Die Methode erwartet die Transformationsregel als AST sowie das Zielverzeichnis für die Generierung als Input und schreibt das Generat in das angegebene Zielverzeichnis.

Für den Fall, dass eine eigene Dateiendung für Modelle der DSTL angegeben wurde, spiegelt sich dies in der generierten `parse`-Methode wieder. Anstelle von Dateien mit der Standardendung `mtr` werden hier Modelle der angegebenen Endung verarbeitet. Die Verschaltung dieser Methoden zur Generatorsteuerung geschieht durch das von MontiTrans bereitgestellte `dstl2java` Groovy-Skript. Die Details zum Ablauf der Generierung, zur Struktur des Generators sowie des Generats werden in Kapitel 8 beschrieben.

Die Generierung der Kontextbedingungen, des ODRules Generators sowie der Base Class für das Groovy-Skript erfolgt templatebasiert. Dazu wird jeweils die von MontiCore bereitgestellte `GeneratorEngine` und entsprechende Freemarker Templates für die jeweiligen Komponenten verwendet. Zur Adaption durch den Nutzer gibt es verschiedene Möglichkeiten handgeschriebenen Code mit dem generierten zu kombinieren [GHK+15b]. Für die DSTL-Infrastruktur wird der auch in MontiCore verwendete *TOP-Mechanismus* – eine Variante des Generation Gap Pattern [VS13, Fow10, Vli98] – verwendet (vgl. Abschnitt 9.2).

6.5 Diskussion und verwandte Arbeiten

In diesem Kapitel wurden Ableitungsregeln zur systematischen Ableitung von DSTLs aus (MontiCore) Grammatiken vorgestellt (vgl. GR 1). Es wurde erklärt wie DSTLs zu modular definierten DSLs strukturiert sind und entwickelt werden (vgl. GR 2). Darüber hinaus wurde die neue Architektur des Generators vorgestellt. Dieser setzt im Vergleich zur in [Wei12] vorgestellten Umsetzung nicht auf Workflows sondern auf eine Steuerung

per Groovy-Skript. Zusätzlich erlaubt der neue Generator Erweiterungen bzw. Anpassungen des generierten Codes durch Einbindung von handgeschriebenem Code mittels des MontiCore TOP-Mechanismus (vgl. GR 13). Neben dem Generator wurde auch für das Generat bzw. den ODRule Generator eine neue, auf Groovy basierende Architektur entwickelt und umgesetzt. Außerdem wurden die generierten DSTLs um ebenfalls generierte Kontextbedingungen ergänzt. Die Ableitung berücksichtigt hierbei sowohl die in dieser Arbeit (weiter-)entwickelten Operatoren als auch die in [Wei12] nicht unterstützen Nichtterminaltypen (vgl. GR 1) wie Interface-Nichtterminale (vgl. GR 1.1), abstrakte und externe Nichtterminale (vgl. GR 1.2 und GR 1.3) sowie die Erweiterung und Redefinition von Nichtterminalen (vgl. GR 1.4 und GR 1.5). Zusätzlich wird auch eine modulare Definition von DSLs (vgl. GR 2) durch Sprachvererbung (vgl. GR 2.1) oder -einbettung GR 2.2) unterstützt. Zudem wurde in dieser Arbeit auch der MontiTrans Generator dahingehend erweitert, dass diese Operatoren und Konzepte bei der Generierung berücksichtigt und entsprechend unterstützt werden. Schließlich wurde die Ableitung mithilfe der Ableitungsregeln an der Beispielsprache `Automaton` demonstriert.

Die in diesem Kapitel beschriebene Ableitung von DSTLs ist nicht auf MontiCore-Grammatiken beschränkt, sondern lässt sich auf beliebige Sprachen übertragen, die durch Grammatiken definiert sind. Hierbei finden die Ableitungsregeln für Interface-Nichtterminale, abstrakte und externe Nichtterminale keine Anwendung. Des Weiteren wird die zweite Regel zur Ableitung von Interface-Nichtterminalen dahingehend angepasst, dass einfache Nichtterminale abgeleitet werden, deren Rumpf eine Alternative der Nichtterminale für Pattern, den Collection Operator, den Optional Operator und die negativen Elemente umfasst. Wie bereits in Kapitel 4 diskutiert ist die Struktur der abgeleiteten DSTLs von der Nichtterminalstruktur der DSL abhängig. Falls die abgeleitete DSTLs nicht die angestrebte Granularität der Operatoren aufweist, kann dies entweder durch eine Anpassung der DSTL (vgl. Abschnitt 5.1.2 und 5.2.2) an die Bedürfnisse der Nutzer oder durch eine Restrukturierung der Grammatik der DSL gelöst werden. Darüber hinaus kann es abhängig von der Syntax der DSL zu einer Kollision der Syntax der Operatoren mit der Syntax der DSL kommen. In diesem Fall können die Spracherweiterungsmechanismen von MontiCore genutzt werden, um die Syntax der Operatoren zu redefinieren oder alternativ die Ableitungsregeln angepasst werden, sodass eine andere konkrete Syntax für die Operatoren abgeleitet wird.

Zur Erstellung neuer domänenspezifischer Transformationssprachen gibt es unterschiedliche Ansätze. Neben der hier vorgestellten Idee der automatischen Ableitung gibt es auch Ansätze zur semi-automatischen [BW07, KMS+10, Grø09, GMP09] und vollständig manuellen Konstruktion aus Bausteinen für die DSTL [SVL13, SGV13, SCGdL14].

Der semi-automatische Ansatz in [BW07] und [KMS+10] generiert basierend auf einem Metamodell einer DSL eine Patternsprache. Diese Patternsprache erlaubt es dem Transformationsentwickler schließlich die LHS und RHS einer Transformation anzugeben. Im Vergleich zu der hier vorgestellten automatischen Generierung muss der Sprachentwickler jedoch eine eigene konkrete Syntax mit der erzeugten abstrakten Syntax verknüpfen, um eine DSTL zu erhalten. Ein ähnlicher Ansatz wird in [Grø09, GMP09] vorgestellt. Hier wird basierend auf einer graphischen Modellierungssprache eine graphische DSTL

generiert. Auch in diesem Ansatz muss jedoch der Sprachentwickler die Verknüpfung zwischen abstrakter DSTL-Syntax und konkreter Syntax der Modellierungssprache vornehmen. In [GMPO09a] wird dies für UML 2 Aktivitätsdiagramme gemacht.

Ein Ansatz zur manuellen Konstruktion von DSTLs wird in [SCGdL14] vorgestellt. Hierbei wird eine DSTL aus Bausteinen für Transformationssprachen zusammengesetzt. Im Gegensatz zu der hier vorgestellten Ableitung fehlt eine klare Vorgehensweise bei der Erstellung neuer DSTLs. Ein ähnliches Konzept verfolgt T-Core [SVL13]. T-Core stellt ebenfalls Bausteine zur Erstellung neuer Transformationssprachen bereit. Auch hier ist jedoch der eigentliche Erstellungsprozess manuell und muss durch den Sprachentwickler durchgeführt werden. Ein Tool, das hierauf aufbaut ist AToMPM [EHR+13, CSE16].

AToMPM [EHR+13, CSE16] ist ein Tool, das ähnlich wie MontiTrans eine automatische Erstellung von DSTLs ermöglicht. AToMPM ist eine Cloud-basierte Language Workbench [CSE16] und erlaubt die Definition von DSLs über Metamodelle. Für diese Metamodelle können verschiedene graphische konkrete Syntaxen definiert werden. Außerdem erlaubt AToMPM die automatische Erzeugung von Transformationssprachen auf Basis von einem oder mehreren Metamodellen. Hierfür basiert AToMPM auf der Sammlung von Transformationsoperatoren von T-Core. Im Gegensatz zu MontiTrans fokussiert AToMPM metamodellbasierte statt grammatikbasierte Sprachdefinitionen und setzt auf eine graphische Modellierung und Definition von Modelltransformationen. Ein weiterer automatisierter Ansatz wird in [RKR+06] skizziert. Die Autoren beschreiben die Idee eines Generatorframework für domänenspezifische Transformationen. Ausgehend von einer EBNF-Grammatik sollen Java-basierte Execution Engines generiert werden. Die Autoren erklären jedoch nicht wie dies realisiert werden soll und es ist bislang keine Implementierung bekannt.

Ebenfalls auf die Ableitung neuer Sprachen aus der Grammatik einer gegebenen Modellierungssprache setzen die Ansätze zur Ableitung von Taggingsprachen [GLRR15, Loo17] und Deltasprachen [HHK+13, HHK+15] so wie Spezifikationen von Modelleditieroperationen [RKK14, KTRK16]. Taggingsprachen ergänzen hierbei eine Sprache – ähnlich wie Stereotypen – um die Möglichkeit Informationen an Modellelemente anzuhängen. Im Gegensatz zu Stereotypen haben sie jedoch den Vorteil, dass ein Taggingmodell neben dem eigentlichen Modell liegt und somit das Modell auch bei vielen zusätzlichen Informationen nicht unübersichtlich wird. Während die Aufgabe einer Taggingsprache größtenteils durch eine Transformationssprache gelöst werden kann, ist eine Taggingsprache umgekehrt kein Ersatz für eine DSTL. Deltasprachen sind Transformationssprachen recht ähnlich. Genauso wie Transformationssprachen erlauben sie Modellveränderungen auszudrücken. Im Gegensatz zu Transformationen adressieren Deltamodelle jedoch typischerweise konkrete Modelle und bieten daher nicht die Möglichkeit der Generalisierung. Zusätzlich setzen sie nicht auf Pattern Matching, wodurch ein Deltamodell im Gegensatz zu einer Transformation wissen muss, wo innerhalb des Modells sich das zu verändernde Element befindet. Eine weitere Gemeinsamkeit der Ableitung ist, dass in beiden Fällen eine Orientierung an der Modellsyntax stattfindet. Die Art der Ableitung unterscheidet sich jedoch von der hier vorgestellten. Während der hier vorstellte Ansatz komplett eigenständige DSTL-Grammatiken erzeugt, werden für Taggingsprachen und

Deltasprachen die Grammatiken der DSLs durch Sprachvererbung wiederverwendet. Da die DSTLs jedoch zur Beschreibung von Pattern Abstraktionen von der eigentlichen Modellsyntax erlauben (vgl. Abschnitt 4.3), ist diese Möglichkeit für die Ableitung von DSTLs nicht geeignet. Der Ansatz zur Ableitung von Modelleditieroperationen leitet aus dem Metamodell einer Sprache eine Menge von konsistenzerhaltenden Editieroperationen ab. Die Unterschiede zu diesem Ansatz sind zum einen, dass der Ansatz eine Ableitung aus Metamodellen statt Grammatiken fokussiert und zum anderen das Ergebnis der Ableitung. Anders als die abgeleiteten Editieroperationen dienen die Operatoren der DSTLs der Spezifikation von Transformationsregeln, die Modelle verändern, und dienen nicht der Abbildung von Versionsunterschieden bei der Versionierung von Modellen. Die abgeleiteten Editieroperationen haben zum Ziel nach jeder angewendeten Operation ein konsistentes Modell zu erzeugen. Das bedeutet, dass alle Multiplizitäten und Konsistenzregeln nach Anwendung der Operation wieder erfüllt sind. Dies unterschiedet sich von den DSTLs, die mit dem hier vorgestellten Ansatz abgeleitet werden. Diese erlauben auch die Spezifikation von Transformationsregeln, die nicht wohlgeformte Modelle erzeugen. Die Spezifikation und Kombination von Transformationsregeln, die schließlich wieder wohlgeformte Modell erzeugen, obliegt dem Transformationsentwickler. Ein weiterer Unterschied besteht darin, dass die DSTLs die Syntax der Modellierungssprache zur Spezifikation von Modelltransformation verwenden. Die Ableitung der Editieroperationen hingegen berücksichtigt ausschließlich die abstrakte Syntax.

Schließlich kann auch die Ableitung von Transformationen aus Beispielmodellen vor und nach Anwendung der Transformation als verwandt zur Erstellung von DSTLs betrachtet werden [SGW15, KLR+12, LWK10, BLS+09, SWG09, NMSS09], wobei hier streng genommen nicht die DSTL, sondern die konkreten Transformationen abgeleitet werden. Wie in Kapitel 4 diskutiert, basiert die eigentliche Transformationsbeschreibung auf der abstrakten Syntax und die nötige Generalität einer Transformation muss durch den Transformationsentwickler nachträglich in die Transformation integriert werden.

Teil III

Domänenspezifische Transformationen

Kapitel 7

Wiederverwendbare Transformationen

In den vorangegangenen Kapiteln wurden die Operatoren von DSTLs im Allgemeinen (vgl. Kapitel 4) und die Herleitung bzw. Generierung neuer DSTLs mithilfe von MontiTrans (vgl. Kapitel 6) beschrieben. In Kapitel 5 wurden außerdem verschiedene DSTLs vorgestellt, darunter sowohl (Baustein-)DSTLs zur Entwicklung neuer DSTLs als auch DSTLs für Klassendiagramme der CD4Code und CD4Analysis DSLs und MontiArc-Modelle. In diesem Kapitel werden darauf aufbauend wiederverwendbare Transformationen vorgestellt, die die Entwicklung neuer Transformationen und Generatoren erleichtern (vgl. Abbildung 7.1). Die Verwendung sowie die Entwicklung neuer Transformationen wird in Kapitel 9 beschrieben.

Bei der Verwendung von Modellen bzw. der Entwicklung von Transformationsregeln treten häufig ähnliche, wiederkehrende Problemstellungen auf. Dies ist bereits aus der Softwareentwicklung bekannt und wird hier in der Regel über Bibliotheken wiederverwendbarer Funktionalität oder Frameworks gelöst [MBK91, FK05]. Dieser Gedanke wurde aufgegriffen und daher ist – neben der Entwicklung von DSTLs – die Entwicklung und Bereitstellung von wiederverwendbaren Transformationen ein Ziel dieser Arbeit (vgl. Abschnitt 3.3). Dazu wurden die in diesem Kapitel beschriebenen Transformationsbibliotheken entwickelt, die dem Transformationsentwickler die Entwicklung von Transformationen erleichtern und dem Modellierer deren Verwendung ohne vorherige Entwicklung ermöglichen. Dies erlaubt dem Transformationsentwickler die Wiederverwendung dieser Transformationen sowohl in der durch die Bibliothek bereitgestellten Form als auch als Teil neu zu entwickelnder Transformationen. Dieses Konzept orientiert sich an dem für Programmiersprachen typischen Konzept der Bibliotheken [MBK91]. Außerdem erlaubt die Bibliothek dem Modellierer die dort bereitgestellten Transformation zu nutzen.

Die wichtigsten Ergebnisse dieses Kapitels sind:

- Die Bibliothek von wiederverwendbaren Transformationen zum Refactoring von Klassendiagrammen.
- Die Bibliothek von wiederverwendbaren Transformationen zur Normalisierung MontiArc-Modellen und zur besseren Interoperabilität mit ROS.
- Grundlage zur Entwicklung weiterer Transformationen als Kombination vorhandener Transformationen und Realisierungen der Bibliothekstransformationen als Vorlage zur Entwicklung eigener Transformationen.

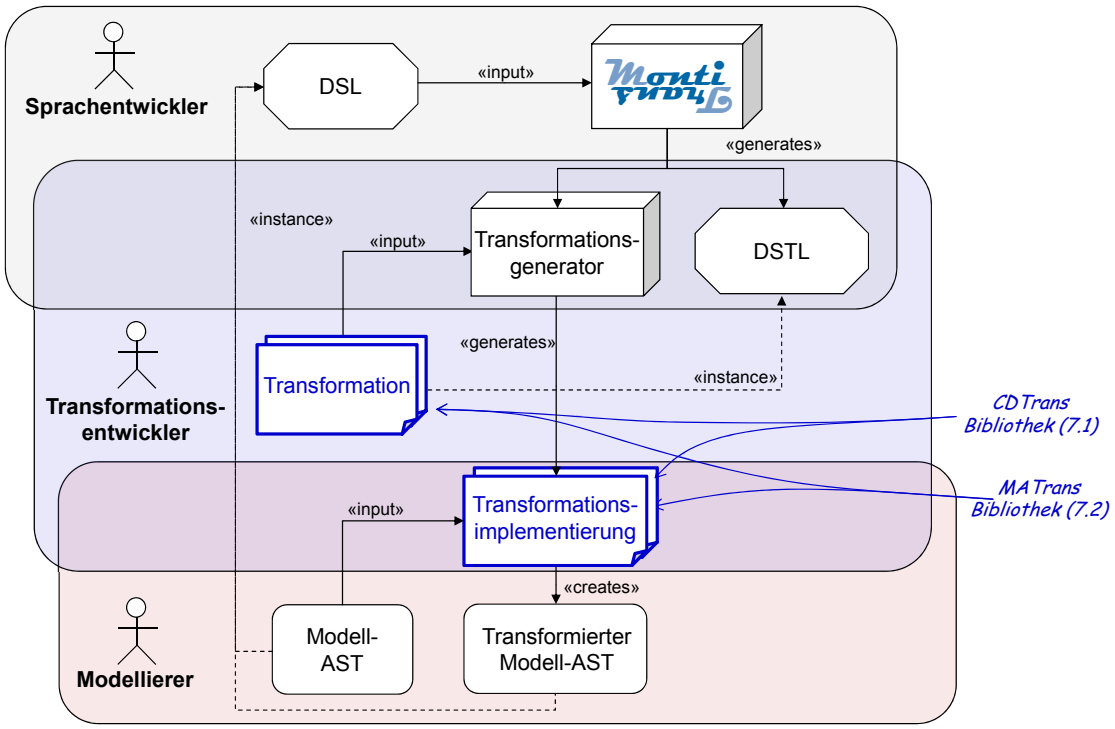


Abbildung 7.1: Übersicht über die verschiedenen Aspekte von MontiTrans mit Fokus auf den in diesem Kapitel vorgestellten Teil (Wiederverwendbare Transformationen mit CDTrans und MATrans).

7.1 Transformationsbibliothek für Klassendiagramme

In diesem Abschnitt wird die Bibliothek von Klassendiagrammtransformationen, die mittels der DSTL CDTrans entwickelt wurde, vorgestellt. Sie stellt dem Modellierer zur Anwendung sowie dem Transformationsentwickler zur Wiederverwendung Transformationen zum Refactoring bzw. Bearbeiten von Klassendiagrammen zur Verfügung. Die Entwicklung basiert zum Teil auf der betreuten Vorarbeit [Eck16].

Refactoring wird in der Softwareentwicklung bereits lange Zeit zur Verbesserung des Quellcodes genutzt [Opd92, Bro98, Fow99, PR01, SPLJ01, Rum17]. Der Begriff Refactoring wurde dabei insbesondere durch Martin Fowler geprägt [Fow99].

Fowler beschreibt Refactoring wie folgt:

> „Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.“

Das heißt, Ziel des Refactorings ist eine Verbesserung der inneren Struktur, ohne dass dies Auswirkungen auf das nach außen sichtbare Verhalten hat. Diese Definition ist auf die Objekt-Orientierte Programmierung (OOP) ausgelegt. Dennoch wird Refactoring

nicht nur auf Quellcode von Software angewendet, sondern auch auf Modelle wie beispielsweise die der UML [SPLJ01, OMG15]. Darüber hinaus können – auf Grund der Ähnlichkeiten von Klassendiagrammen und OOP – viele der für OOP entwickelten Refactorings auf Klassendiagramme übertragen werden, wie beispielsweise die Kapselung von Feldern [Fow99].

Im Folgenden wird für jedes Refactoring immer zunächst eine allgemeine Erklärung, gefolgt von einem Anwendungsbeispiel, gegeben. Anschließend werden die Ausgangssituation für die Anwendung, die Motivation für das Refactoring sowie das Ziel und die Realisierung beschrieben. Außerdem wird für jedes Refactoring der vom Nutzer erwartete Input, vorhandene Varianten und Synonyme in der Literatur erläutert sowie – falls vorhanden – das inverse Refactoring erläutert.

7.1.1 Pull Up Attributes

Pull Up Attributes ist eines der am häufigsten in der Literatur vorkommenden Refactorings [RBJ97, Fow99, Rob99, Chr05, ALC08, WR09]. Ziel dieses Refactoring ist die Reduktion von Codeduplikaten durch die Verlagerung gemeinsamer Attribute (auch Felder genannt) mehrerer Klassen in deren gemeinsame, existente Superklasse. Abbildung 7.2 zeigt ein Beispiel für die Anwendung dieses Refactorings (von links nach rechts).

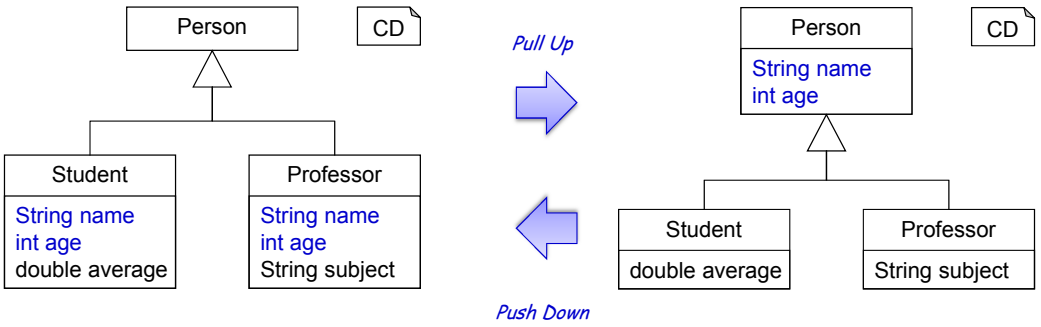


Abbildung 7.2: Verschieben von Attributen entlang von Vererbungsbeziehungen. Von links nach rechts: Anwendung des Refactorings Pull Up Attribute. Von rechts nach links: Anwendung des Refactorings Push Down Attribute.

Das Klassendiagramm in Abbildung 7.2 umfasst drei Klassen: `Person`, `Student` und `Professor`. Die Klassen `Student` und `Professor` sind Subklassen der Klasse `Person` und haben zwei gemeinsame Attribute: `String name` und `int age`. Nach Anwendung des Refactorings sind diese gemeinsamen Attribute nicht länger in den Klassen `Student` und `Professor`, sondern in der Klasse `Person` vorhanden.

Situation Alle Subklassen einer Klasse haben das gleiche Attribut.

Motivation Reduzierung der Codeduplikation durch mehrfach vorhandene gleiche Attribute. Ermöglichen der Verwendung der Superklasse anstelle der Subklasse.

Ziel/Lösung Verlagerung der gemeinsamen Attribute aus den Sub- in die Superklasse.

Realisierung Dieses Refactoring matcht alle Subklassen einer Superklasse, die das selbe Attribute haben. Hierbei ist die Transformation nur anwendbar, wenn es keine weitere Subklasse der Superklasse gibt, welche das gemeinsame Attribute nicht hat. Anschließend entfernt die Transformation das Attribut aus allen Subklassen und fügt es der Superklasse hinzu.

```
1  classdiagram $_ {                                          CDTrans
2    class $super {
3      [[ :- CDAttribute $A1 ]]
4    }
5    class $_ extends $super{
6      [[  CDAttribute $A1 :- ]]
7    }
8    list [[  class $_ extends $super {
9               [[  CDAttribute $A2  :- ]]
10            }
11       ]]
12   not [[ class $_ extends $super; ]]
13 }
14 where {  $A1.deepEquals($A2) }
```

Listing 7.3: Transformatrionsregel des Refactorings Pull Up Attribute.

Listing 7.3 zeigt einen Auszug der Transformationsregel, die diese Transformation realisiert. In Zeile 2-4 wird die Superklasse beschrieben, der durch die Transformation das gemeinsame Attribut hinzugefügt wird. Zeile 5-10 und 14 beschreiben alle Subklassen mit gleichem Attribut. Zeile 12 verbietet weitere Subklassen.

Input Dem Refactoring muss nur das Modell mitgegeben werden.

Varianten -

Synonyme aus der Literatur

- Pull Up Attribute(s) [ZLG05, MB07, KRLP+14, Are14, BEK+07]
- Pull Up Feature [Chr05, HVW11, SWS+13, Bur14]
- Pull Up Field [Fow99, ALC08, MC12, SFP17]
- Pull Up Item [SWS+13]
- Pull Up (Meta)Property [Wac07, CDEP08, Ste15, Are14]
- Pull Up Instance Variable [WR09, RBJ97, Rob99]
- Pull Up Variable From Subclass(es) [RBJ97]
- Move (optional/required) Attribute to Super Class [HBJ08]

Inverses Refactoring Push Down Attribute (vgl. Abschnitt 7.1.2)

7.1.2 Push Down Attribute

Push Down Attribute ist das inverse Refactoring zu Pull Up Attribute und ebenfalls ein häufig in der Literatur vorkommendes Refactoring [RBJ97, Fow99, Rob99, Chr05, ALC08, ZLG05]. Ausgangssituation dieses Refactorings sind Attribute einer Superklasse,

die nur von einigen der Subklassen verwendet werden, oder in der Superklasse nicht länger sichtbar sein sollen. Ziel dieses Refactoring ist daher die Verlagerung dieser Attribute in die entsprechenden Subklassen. Abbildung 7.2 zeigt ein Beispiel für die Anwendung dieses Refactoringst (von rechts nach links).

Das Klassendiagramm in Abbildung 7.2 umfasst drei Klassen: `Person`, `Student` und `Professor`. Die Klassen `Student` und `Professor` sind Subklassen der Klasse `Person`. `Person` hat die Attribute `String name` und `int age`, die in die Subklassen verlagert werden sollen. Nach Anwendung des Refactorings sind diese Attribute nicht länger in der Klasse `Person`, sondern in beiden Subklassen `Student` und `Professor` vorhanden.

Situation Mindestens ein Attribut wird nicht mehr von allen Subklassen einer Klasse benötigt oder soll in der Superklasse verborgen werden.

Motivation Kapselung der Daten und Information Hiding.

Ziel/Lösung Verlagerung der Attribute, die nur von einigen Subklassen verwendet werden, in diese Subklassen.

Realisierung Für jedes angegebene Attribut matcht dieses Refactoring alle gegebenen Subklassen der gegebenen Superklasse. Anschließend fügt die Transformation den Subklassen das Attribut hinzu und entfernt es schließlich aus der Superklasse.

CDTrans

```
1  classdiagram $_ {
2    class $super {
3       $attribute [[ Type $_ $name; ]]
4    }
5
6    class $className extends $super{
7      not [[ Type $_ $name; ]]
8        [[ :- CDAttribute $attribute ]]
9    }
10 }
```

Listing 7.4: Auszug aus der Transformation Push Down Attributes.

Listing 7.4 zeigt einen Auszug der Transformationsregel, die diese Transformation realisiert. In Zeile 2-4 wird die Superklasse beschrieben, aus der durch die Transformation die Attribute entfernt werden. Zeile 6-9 beschreibt eine Subklasse, der das Attribut hinzugefügt wird. Wurde das Attribut in den Subklassen hinzugefügt, wird in einer nachgelagerten Transformationsregel das Attribute aus der Superklasse entfernt.

Input Das Refactoring erwartet als Input die Namen der Superklasse, der Subklassen und Attribute sowie das Modell.

Varianten

Push Down aller Attribute: Hierbei reicht die Angabe der Super- und Subklassen sowie das Modell.

Push Down aller Attribute in alle Subklassen: Hierbei reicht die Angabe der Superklasse sowie das Modell.

Push Down bestimmter Attribute in alle Subklassen: Hierbei reicht die Angabe der Superklasse, der Attribute sowie das Modell.

Synonyme aus der Literatur

- Push Down Attribute(s) [ZLG05, MB07]
- Push Down Feature [Chr05]
- Push Down Field [Fow99, MC12]
- Push (Meta) Property [Wac07, CDEP08]
- Push Down Instance Variable [Rob99]
- Push Down Variable into Subclass(es) [RBJ97]

Inverses Refactoring Pull Up Attribute (vgl. Abschnitt 7.1.1)

7.1.3 Pull Up Method

Pull Up Method ist ein Refactoring mit dem Ziel der Reduktion von Codeduplikaten durch die Verlagerung gemeinsamer Methoden mehrerer Klassen in deren gemeinsame Superklasse [Fow99]. Abbildung 7.5 zeigt ein Beispiel für die Anwendung dieses Refactorings (von links nach rechts).

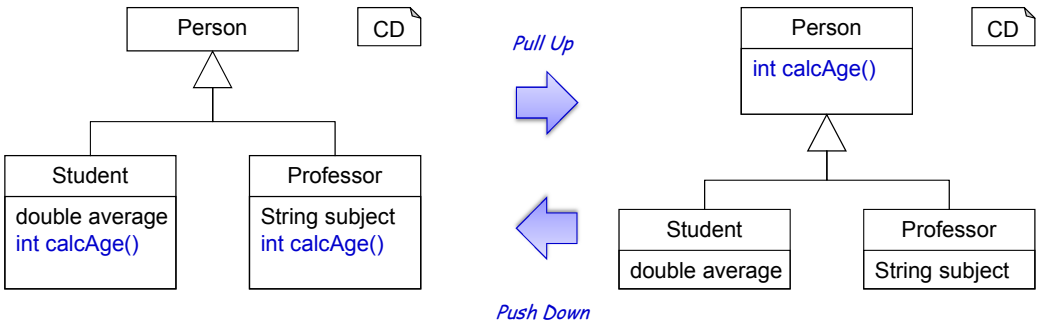


Abbildung 7.5: Verschieben von Methoden entlang von Vererbungsbeziehungen. Von links nach rechts: Anwendung des Refactorings Pull Up Method. Von rechts nach links: Anwendung des Refactorings Push down Method.

Das Klassendiagramm in Abbildung 7.5 umfasst drei Klassen: `Person`, `Student` und `Professor`. Die Klassen `Student` und `Professor` sind Subklassen der Klasse `Person` und haben die gemeinsame Methode `int calcAge()`. Nach Anwendung des Refactorings ist diese gemeinsame Methode nicht länger in den Klassen `Student` und `Professor`, sondern in der Klasse `Person` vorhanden.

Situation Alle Subklassen einer Klasse haben die gleiche Methode.

Motivation Reduzierung der Codeduplikation durch mehrfach vorhandene gleiche Methoden. Ermöglichen der Verwendung der Superklasse anstelle der Subklasse.

Ziel/Lösung Verlagerung der gemeinsamen Methoden aus den Sub- in die Superklasse.

Realisierung Dieses Refactoring matcht alle Subklassen einer Superklasse, die eine Methode mit gleicher Signatur haben. Hierbei ist die Transformation nur anwendbar, wenn es keine weitere Subklasse der Superklasse gibt, welche die gemeinsame Methode nicht hat. Anschließend entfernt die Transformation die Methode aus allen Subklassen und fügt sie der Superklasse hinzu.

CDTrans

```
1  classdiagram $_ {
2    class $super {
3      [[ :- CDMethod $M1 ]]
4    }
5    class $_ extends $super {
6      [[ CDMethod $M1  :- ]]
7    }
8    list [[ class $_ extends $super{
9              [[ CDMethod $M2 :- ]]
10           }
11   ]]
12
13   not [[ class $_ extends $super; ]]
14 }
15
16   where{ $M2.deepEquals($M1) }
```

Listing 7.6: Auszug aus der Transformation Pull Up Methods.

Listing 7.6 zeigt einen Auszug der Transformation, die dieses Refactoring realisiert. In Zeile 2-4 wird die Superklasse beschrieben, der durch die Transformation die gemeinsame Methode hinzugefügt wird. Zeile 5-11 und 16 beschreiben alle Subklassen mit gleicher Methode. Zeile 13 verbietet weitere Subklassen.

Input Dem Refactoring muss nur das Modell übergeben werden.

Varianten -

Synonyme aus der Literatur

- Pull(ed) Up Method [Fow99, VGSMD03, SVGJ05, DJ06, MMBJ09, WR09, Kos09, MC12]
- Pull Up Operation [MB07, Are14]
- Pull Up Method from Subclass(es) [RBJ97]

Inverses Refactoring Push Down Method (vgl. Abschnitt 7.1.4)

7.1.4 Push Down Method

Push Down Method ist das inverse Refactoring zu *Pull Up Method*. Ausgangssituation dieses Refactorings sind Methoden einer Superklasse, die nur von einigen Subklassen verwendet werden, oder in der Superklasse nicht länger sichtbar sein sollen [Fow99]. Ziel dieses Refactoring ist daher die Verlagerung dieser Methoden in die entsprechenden Subklassen. Abbildung 7.5 zeigt ein Beispiel für die Anwendung dieses Refactorings (von rechts nach links).

Das Klassendiagramm in Abbildung 7.5 umfasst drei Klassen: `Person`, `Student` und `Professor`. Die Klassen `Student` und `Professor` sind Subklassen der Klasse `Person`. `Person` hat die Methode `int calcAge()`, die in die Subklassen verlagert werden soll. Nach Anwendung des Refactorings ist diese Methode nicht länger in der Klasse `Person`, sondern in beiden Subklassen `Student` und `Professor` vorhanden.

Situation Eine Methode wird nicht mehr von allen Subklassen einer Klasse benötigt bzw. soll in der Superklasse verborgen werden.

Motivation Datenkapselung und Information Hiding.

Ziel/Lösung Verlagerung der Methode, die nur von einzelnen Subklassen verwendet werden, in diese Subklassen.

Realisierung Für jede angegebene Methode matcht dieses Refactoring alle gegebenen Subklassen der gegebenen Superklasse. Anschließend entfernt die Transformation die Methode aus der Superklasse und fügt sie den Subklassen hinzu.

CDTrans

```
1  classdiagram $_ {
2    class $super {
3      $M [[ ReturnType $_ $name (); ]]
4    }
5
6    class $className extends $super {
7      not [[ ReturnType $_ $name (); ]]
8        [[ :- CDMethod $M ]]
9    }
10 }
```

Listing 7.7: Auszug aus der Transformation Push Down Methods.

Listing 7.7 zeigt einen Auszug der Transformation, die dieses Refactoring realisiert. In Zeile 2-4 wird die Superklasse beschrieben, aus der durch die Transformation die Methoden entfernt werden. Zeile 6-9 beschreibt eine Subklasse, der die Methode hinzugefügt wird. Wurde die Methode in den Subklassen hinzugefügt, wird in einer nachgelagerten Transformationsregel die Methode aus der Superklasse entfernt.

Input Das Refactoring erwartet als Input die Namen der Superklasse, der Subklassen und Methoden sowie das Modell.

Varianten

Push Down aller Methoden: Hierbei reicht die Angabe der Super- und Subklassen sowie das Modell.

Push Down aller Methoden in alle Subklassen: Hierbei reicht die Angabe der Superklasse sowie das Modell.

Push Down bestimmter Methoden in alle Subklassen: Hierbei reicht die Angabe der Superklasse, der Methoden sowie das Modell.

Synonyme aus der Literatur

- Push(ed) Down Method [Fow99, MC12, DJ06, WR09, Kos09]

- Push Down Method into Subclass(es) [RBJ97]
- Push Down Operation [MB07, Are14]

Inverses Refactoring Pull Up Method (vgl. Abschnitt 7.1.3)

7.1.5 Extract Super Class

Extract Super Class ist ein Refactoring, bei dem für mehrere Klassen eine gemeinsame Superklasse erzeugt wird. Ausgangssituation dieses Refactorings sind Klassen, die gemeinsame Attribute und keine gemeinsame Superklasse haben [Fow99]. Dies unterscheidet dieses Refactoring von dem *Pull Up Attributes* Refactoring. Ziel dieses Refactoring ist die Reduktion von Codeduplikaten durch die Verlagerung gemeinsamer Attribute mehrerer Klassen in eine gemeinsame Superklasse. Abbildung 7.8 zeigt ein Beispiel für die Anwendung dieses Refactorings (von links nach rechts).

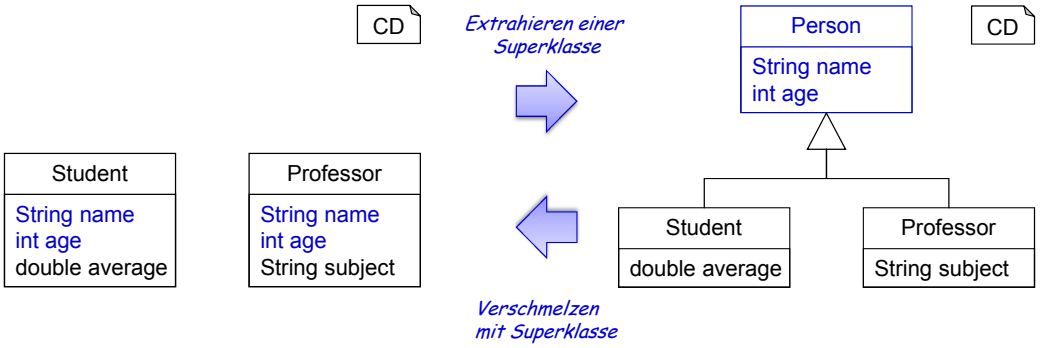


Abbildung 7.8: Extrahieren von Superklassen und Verschmelzen von Hierarchien. Von links nach rechts: Anwendung des Refactorings *Extraxt Super Class*. Von rechts nach links: Refactoring *Collapse Hierarchy*.

Das linke Klassendiagramm in Abbildung 7.8 umfasst zwei Klassen: `Student` und `Professor`. Die Klassen `Student` und `Professor` haben die gemeinsamen Attribute `String name` und `int age`. Nach Anwendung des Refactorings existiert eine gemeinsame Superklasse `Person`, die die gemeinsamen Attribute hat. Aus den Subklassen `Student` und `Professor` wurden diese Attribute entfernt.

Situation Mehrere Klassen haben gleiche Attribute und/oder Methoden. Anders als bei Pull Up Attributes/Methods existiert keine gemeinsame Superklasse.

Motivation Reduzierung der Codeduplikation durch mehrfach vorhandene gleiche Attribute. Ermöglichen der Verwendung der Superklasse anstelle der Subklasse.

Ziel/Lösung Erzeugen einer gemeinsamen Superklasse und Verlagerung der gemeinsamen Attribute und/oder Methoden aus den Subklassen in die Superklasse.

Realisierung Das Refactoring matcht eine Menge von Klassen mit gleichen Attributen und/oder Methoden. Für diese wird eine Superklasse erstellt und die gefundenen Klassen erben von der neu erstellten Superklasse. Der Name der Superklasse kann wahlweise der Transformation übergeben werden, indem der Wert der Schemavariable für

den Namen als Parameter gesetzt wird (vgl. Abschnitt 9.4), oder automatisch durch die Transformation berechnet werden. Im letzten Fall werden die Namen der Subklassen zusammengefügt und bilden den Namen der Superklasse. Zum Beispiel würde aus `Student` und `Professor` in diesem Fall `StudentProfessor`[1]. Anschließend werden alle gemeinsamen Attribute und/oder Methoden in die Superklasse verlagert.

CDTrans

```
1  classdiagram $_{
2    [[ :- class $superclass {
3      CDAttribute $A1
4    }]]
5
6    class $sub extends [[ :- $superclass ]] {
7      [[ CDAttribute $A1  :- ]]
8      }
9
10   list [[ class $_ extends [[ :- $superclass ]] {
11     [[ CDAttribute $A2  :- ]]
12   }]]
13 }
```

Listing 7.9: Auszug aus der Transformation Extract Super Class.

Listing 7.9 zeigt einen Auszug der Transformation, die dieses Refactoring für den Fall, dass der Name durch den Nutzer bestimmt wird, realisiert. In Zeile 2-4 wird die Superklasse beschrieben, die durch die Transformation erstellt wird. In den Zeilen 6-12 sind die Klassen beschrieben, die das gleiche Attribut haben. Wie bei Pull Up Attribute (vgl. Abschnitt 7.1.1) wird der Collection Operator verwendet, um alle Klassen mit gleichem Attribut zu beschreiben. Wird das Pattern gefunden, entfernt die Transformationsregel die Attribute aus den Klassen (Zeile 7 und 11), erzeugt die neue Superklasse und legt diese als Superklasse der gefundenen Klassen fest (Zeile 6 und 10). Nachgelagerte Transformationsregeln verschieben die Methoden.

Input Das Refactoring benötigt nur das Modell.

Varianten Alle Varianten existieren auch mit expliziter Angabe des Namens der zu erstellenden Superklasse.
Nur für Attribute: Das Refactoring benötigt das Modell.

Nur für Methoden: Das Refactoring benötigt das Modell.

Synonyme aus der Literatur

- Extract Superclass [Fow99, Chr05, WR09, ZLG05, Are14]
- Create Superclass [Are14]
- Introduce Inheritance [FHFB10]

Inverses Refactoring Collapse Hierarchy (vgl. Abschnitt 7.1.6)

[1] Hierzu wurde eine Hilfsklasse geschrieben, die im Anweisungsblock verwendet wird. Diese erhält die Superklasse und Subklassen, berechnet den neuen Namen als Konkatenation der Namen der gefundenen Subklassen und setzt diesen Namen für die Superklasse ein.

7.1.6 Collapse Hierarchy

Collapse Hierarchy ist ein Refactoring, bei dem eine Superklasse mit ihren Subklassen verschmolzen wird [Fow99]. Ziel ist die Reduktion der Klassenhierarchien, beispielsweise weil sich Super- und Subklassen nicht sonderlich unterscheiden oder die Vererbungsstrukturen zu komplex sind.

Abbildung 7.8 zeigt ein Beispiel für die Anwendung des *Collapse Hierarchy* Refactorings (von rechts nach links). Das Klassendiagramm in Abbildung 7.8 umfasst drei Klassen: `Student`, `Professor` und deren Superklasse `Person`. `Person` hat die Attribute `String name` und `int age`. Nach Anwendung des Refactorings existiert die Superklasse `Person` nicht mehr und die Subklassen `Student` und `Professor` haben jeweils die Attribute `String name` und `int age`.

Situation Die Klassenhierarchie hat unerwünscht viele Ebenen, Super- und Subklassen weisen nur wenige Unterschiede auf.

Motivation Reduktion der Modellgröße.

Ziel/Lösung Verlagerung der Attribute und/oder Methoden der Superklasse in die Subklassen. Entfernen der Superklasse.

Realisierung Das Refactoring matcht eine Superklasse sowie deren Subklassen. Anschließend verschiebt es alle Attribute/Methoden der Superklasse in die Subklassen und entfernt die Superklasse.

CDTrans

```
1  class $super {
2     [[CDAttribute $A1 :-]]
3  }
4
5  list [[class $_ extends $super {
6    [[ :- CDAttribute $A1]]
7    not [[ CDAttribute $A2 ]]
8  } ]]
9
10 where {$A2.deepEquals($A1)}
```

Listing 7.10: Auszug aus der Transformation Collapse Hierarchy.

Listing 7.10 zeigt einen Auszug der Transformation, die dieses Refactoring realisiert. In Zeile 1-3 wird die Superklasse inklusive eines Attributs gematcht. In den Zeilen 5-8 werden alle Subklassen gematcht. Als Modifikation wird das Attribut aus der Superklasse entfernt (Zeile 2) und den Subklassen hinzugefügt (Zeile 6). Das negative Element in Zeile 7 und der Application Constraint in Zeile 10 verhindern, dass Subklassen gefunden werden, die das Attribut bereits enthalten. Nachgelagerten Transformationsregeln verschieben auch die Methoden verschoben und entfernen die Superklasse.

Input Das Refactoring benötigt das Modell sowie den Name der Superklasse.

Varianten -

Synonyme aus der Literatur

- Collapsing Hierarchy [Kos09, Fow99, MC12, SFP17]

Inverses Refactoring Extract Super Class (vgl. Abschnitt 7.1.5)

7.1.7 Extract Intermediate Class

Extract Intermediate Class ist ein Refactoring, das als Spezialfall des *Extract Super Class* Refactorings betrachtet werden kann. Wie bei *Extract Super Class* wird hier eine gemeinsame Superklasse für Klassen mit gleichem Attribut erstellt und die gemeinsamen Attribute in diese Superklasse verlagert. Der Unterschied liegt darin, dass in diesem Fall bereits eine Superklasse existiert, die jedoch weitere Subklassen hat, die das gemeinsame Attribute nicht besitzen. Dadurch wird eine Zwischenklasse statt einer einfachen Superklasse erzeugt. Abbildung 7.11 zeigt eine Anwendung dieses Refactorings.

Das Ausgangsklassendiagramm hat vier Klassen: `Person`, `Student`, `Assistant` und `Professor`. `Student`, `Assistant` und `Professor` sind Subklassen von `Person`, wobei `Assistant` und `Professor` das gemeinsame Attribute `int salary` haben. Nach Anwendung des Refactorings existiert eine weitere Subklasse von `Person`, genannt `Employee`. Diese hat das gemeinsame Attribute und die beiden Klassen `Assistant` und `Professor` erweitern statt `Person` die neu eingeführte Klasse `Employee`.

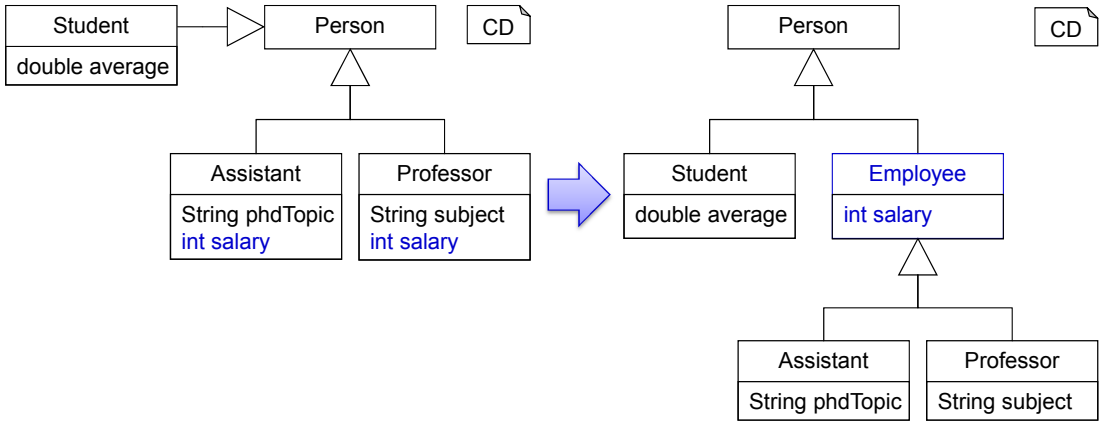


Abbildung 7.11: Anwendung des Refactorings *Extract Intermediate Class*.

Situation Eine Klasse hat mehrere Subklassen, die gleiche Attribute haben. Es gibt zusätzlich Subklassen, die kein entsprechendes Attribut haben.

Motivation Datenkapselung, Reduktion von Codeduplikaten und Verwendung einer Superklasse statt konkreter Subklassen.

Ziel/Lösung Einführung einer Klasse für die gemeinsamen Attribute als Zwischenklasse in der Klassenhierarchie.

Realisierung Das Refactoring matcht eine Menge von Klassen mit gleichen Attributen und/oder Methoden sowie deren vorhandene Superklasse. Für die gefundenen Klassen wird eine eigene Superklasse erstellt, die von der bisherigen Superklasse erbt.

Ähnlich wie bei Extract Super Class (vgl. Abschnitt 7.1.5) gibt es auch hier sowohl die Möglichkeit einen expliziten Namen für die erzeugte Klasse zu übergeben, indem der Name als Parameter an die Transformation übergeben wird (vgl. Abschnitt 9.4), oder diesen von der Transformation erzeugen zu lassen. Die gefundenen Klassen erben von der neu erstellten Superklasse[2]. Anschließend werden alle gemeinsamen Attribute und/oder Methoden in die neue Superklasse verlagert.

CDTrans

```
1  class $subclass1 extends [[ $super :- $newSuper ]] {
2    [[ CDAttribute $A1  :- ]]
3  }
4
5  list [[ class $_ extends [[ $super :- $newSuper ]] {
6          [[ CDAttribute $A2 :- ]]
7        }]]
8
9  [[ :- class $newSuper extends $super {
10    CDAttribute $A1
11 } ]]
12
13 where{ $A2.deepEquals($A1) }
```

Listing 7.12: Auszug aus der Transformation Collapse Hierarchy.

Listing 7.12 zeigt einen Auszug der Transformation, die dieses Refactoring realisiert. In Zeile 1-3 wird eine Subklasse inklusive Attribut und Superklasse gematcht. In Zeile 5-7 werden alle weiteren Subklassen der Superklasse gematcht, die das gleiche Attribut haben (Zeile 13). Die Transformationsregel fügt in Zeile 9-11 die neue Zwischenklasse als Subklasse der ursprünglichen Superklasse inklusive des gemeinsamen Attributs ein, ändert die Superklasse der Subklassen entsprechend und entfernt das Attribut der Subklassen. In nachgelagerten Transformationsregeln werden weitere gemeinsame Attribute und Methoden in die Zwischenklasse verlagert.

Input Dem Refactoring muss nur das Modell übergeben werden.

Varianten Alle Varianten existieren auch mit Angabe des Namens der Superklasse.
Nur für Attribute: Dem Refactoring muss nur das Modell übergeben werden.

Nur für Methoden: Dem Refactoring muss nur das Modell übergeben werden.

Synonyme aus der Literatur

- AddClass [RBJ97]

Inverses Refactoring -

7.1.8 Extract Class

Extract Class ist ein Refactoring, bei dem eine Klasse in zwei Klassen geteilt wird. Hierbei werden zusammengehörige Attribute und Methoden in eine speziell dafür erstellte Klasse ausgelagert. Ziel ist hierbei die Trennung von Verantwortlichkeiten.

[2]Hierzu wurde die gleiche Hilfsklasse wie für *Extract Super Class* genutzt.

Abbildung 7.13 zeigt ein Beispiel für die Anwendung der *Extract Class* Refactorings (von rechts nach links). Die Klasse `Student` beinhaltet hierbei die Attribute `int zipcode` und `String city`, die zusammen die Adresse des Studenten repräsentieren. Durch das Refactoring werden diese beiden Attribute in eine neu erstellte Klasse `Address` ausgelagert. Die neu erstellte Klasse wird über eine zu-1-Assoziation mit der Klasse `Student` verbunden.

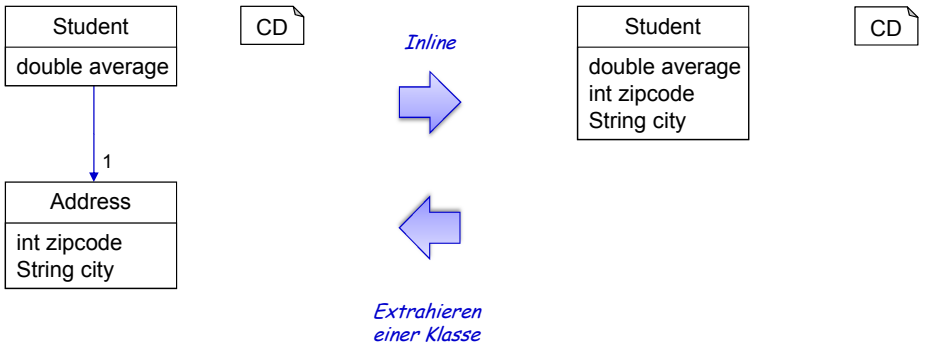


Abbildung 7.13: Trennen und Verschmelzen von (assoziierten) Klassen: Von links nach rechts: Anwendung des Refactorings *Inline Class*. Von rechts nach links Anwendung des inversen Refactorings *Extract Class*.

Situation Eine Klasse erfüllt mehrere Aufgaben, wodurch sich Gruppen von Attributen und Methoden ergeben, die zusammen gehören und eine der Aufgaben repräsentieren.

Motivation Trennung der Verantwortlichkeiten, bessere Verständlichkeit der Zuständigkeit der Klassen, Möglichkeit zur Wiederverwendung der extrahierten Klasse.

Ziel/Lösung Extrahieren einer Klasse für eine Menge von zusammengehörigen Attributen und Methoden.

Realisierung Das Refactoring matcht eine Klasse und eine Menge von deren Attributen/Methoden. Es wird eine Klasse erstellt, deren Name durch den User angegeben ist. Die Attribute/Methoden werden aus der alten Klasse entfernt und der neuen hinzugefügt. Die neue Klasse wird über eine zu-1-Assoziation mit der alten Klasse verbunden.

CDTrans

```
1 classdiagram $_ {
2   not [[ class $className; ]]
3   [[:- class $className;]]
4 }
```

Listing 7.14: Auszug aus der Transformation Extract Class.

Die Transformation ist aus Teilregeln zusammengesetzt, die die Klasse suchen, eine neue Klasse sowie neue Assoziation erstellen und die Attribute/Methoden verschieben. Listing 7.14 zeigt die Transformationsregel, die es erlaubt eine Klasse einem Klassendiagramm hinzuzufügen, falls es noch keine Klassen mit diesem Namen gibt.

Input Das Refactoring benötigt den Namen der beiden Klassen sowie Listen von Attribut- und Methodennamen und das Modell.

Varianten -

Synonyme aus der Literatur

- Create Associated Class [Are14]
- Extract Class [Fow99]

Inverses Refactoring Inline Class (vgl. Abschnitt 7.1.9)

7.1.9 Inline Class

Inline Class [Kos09, Are14] ist das inverse Refactoring zu *Extraxt Class*. Hierbei wird eine Klasse, die über eine zu-1-Assoziation mit einer anderen Klasse verbunden ist, in diese integriert, weil deren Verantwortlichkeiten gering sind. Hierbei werden die Attribute und Methoden der assoziierten Klasse in die andere Klasse verschoben. Abbildung 7.13 zeigt ein Beispiel für die Anwendung dieses Refactorings (von links nach rechts).

Die Klasse `Student` ist über ein zu-1-Assoziation mit der Klasse `Address` (im folgenden als assoziierte Klasse bezeichnet) verbunden. Da die Klasse nur 2 Attribute aufweist, soll sie in die Klasse `Student` integriert werden. Nach Anwendung des Refactoring befinden sich die Attribute `int zipcode` und `String city` in der Klasse `Student`. Die Klasse `Address` und die Assoziation wurden entfernt.

Situation Zwei Klassen sind über eine zu-1-Assoziation verbunden, wobei die assoziierte Klasse nur wenige Attribute/Methoden und nur eine geringe Verantwortlichkeit hat.

Motivation Reduktion der Modellgröße.

Ziel/Lösung Verschmelzen der zwei Klassen zu einer einzigen neuen Klasse.

Realisierung Das Refactoring matcht zwei Klassen, die über eine zu-1-Assoziation verbunden sind. Die Attribute und Methoden der assoziierten Klasse werden in die andere Klasse verschoben. Die nun leere Klasse wird entfernt.

CDTrans

```
1 class $from {
2   [[ CDAttribute $A :- ]]
3 }
4
5 class $to {
6   [[ :- CDAttribute $A ]]
7 }
8
9 association $to -> $from [1];
```

Listing 7.15: Auszug aus der Transformation Inline Class.

Die Transformation ist aus Teilregeln zusammengesetzt. Listing 7.15 zeigt die zum Verschieben der Attribute verwendete Transformationsregel. Hierbei wird das Attribut aus der einen Klasse entfernt (Zeile 1-3) und in der anderen hinzugefügt (Zeile 5-7). Zeile 9 stellt sicher, dass die Klassen über eine zu-1-Assoziation verbunden sind.

Input Das Refactoring benötigt die Namen der beiden Klassen sowie das Modell.

Varianten -

Synonyme aus der Literatur -

Inverses Refactoring Extract Class (vgl. Abschnitt 7.1.8)

7.1.10 Replace Delegation By Inheritance

Replace Delegation By Inheritance ist ein Refactoring, bei dem die Delegation von einer Klasse an eine andere durch eine Vererbungsbeziehung der beiden Klassen ersetzt wird. Dieses Refactoring ist vor allem dann einzusetzen, wenn die delegierende Klasse alle Attribute bzw. Methoden des Delegats benutzt und damit viele simple Delegationen für das gesamte Interface des Delegats durchführt. Abbildung 7.16 zeigt ein Beispiel für die Anwendung dieses Refactorings (von rechts nach links).

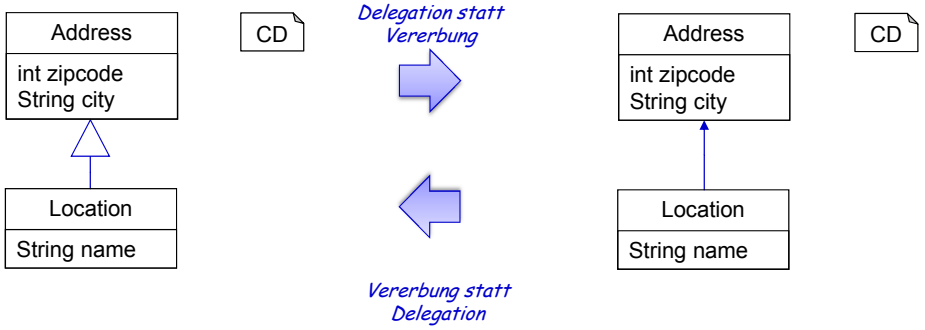


Abbildung 7.16: Vererbung durch Delegation ersetzen und vice versa: Von rechts nach links: Anwendung des Refactorings *Replace Delegation By Inheritance*. Von links nach rechts: Anwendung des inversen Refactorings *Replace Inheritance By Delegation*.

Die Klasse `Location` delegiert hierbei an die Klasse `Address`. Nach Anwendung des Refactorings existiert die Delegation (Assoziation) zwischen den beiden Klassen nicht mehr, stattdessen ist `Location` eine Subklasse der Klasse `Address`.

Situation Eine Klasse delegiert an eine andere Klasse, wobei an das komplette Interface des Delegats delegiert wird.

Motivation Reduktion der vielen simplen Delegationen.

Ziel/Lösung Ersetzen der Delegationsbeziehung durch eine Vererbungsbeziehung.

Realisierung Listing 7.17 zeigt einen Auszug aus der Transformationsregel für dieses Refactoring. Die delegierende Klasse (vgl. Zeile 2) sowie die Assoziation zwischen den beiden Klassen wird gematcht (vgl. Zeile 4). Anschließend wird die Assoziation gelöscht und das Delegat zur Oberklasse der ursprünglich delegierenden Klasse. Die gezeigte Transformationsregel würde auch dann die Superklasse setzen, wenn es bereits eine Superklasse gab. Um dies zu verhindern kann vor den Replacement Operator `not [[ $_ ]]` ergänzt werden.

```
1 classdiagram $_{                                                    CDTrans
2   class $sub extends [[  :- $super]];
3
4   [[  association $sub -> $super; :-]]
5 }
```

Listing 7.17: Auszug aus der Transformation Replace Delegation By Inheritance.

Input Das Refactoring benötigt die Namen der beiden Klassen sowie das Modell.

Varianten -

Synonyme aus der Literatur

- Replace delegation by inheritance [Fow99, Kos09, MC12, Are14, SFP17]

Inverses Refactoring Replace Inheritance By Delegation (vgl. Abschnitt 7.1.11)

7.1.11 Replace Inheritance By Delegation

Replace Inheritance By Delegation ist das inverse Refactoring zu *Replace Delegation by Inheritance*. Hierbei wird die Vererbungsbeziehung von zwei Klasse durch eine Delegation (Assoziation) von der Subklasse zur Superklasse ersetzt. Dieses Refactoring ist vor allem dann einzusetzen, wenn die Superklasse viele Attribute besitzt, die von der Subklasse nicht verwendet werden. Abbildung 7.16 zeigt ein Beispiel für die Anwendung dieses Refactorings (von links nach rechts).

Die Klasse `Location` ist hierbei eine Subklasse der Klasse `Address`. Nach Anwendung des Refactorings existiert diese Vererbungsbeziehung nicht mehr, stattdessen ist `Location` durch eine Assoziation mit der Klasse `Address` verbunden.

Situation Subklasse nutzt nur wenige der Attribute und Methoden ihrer Superklasse.

Motivation Reduktion des Interfaces der Subklasse auf tatsächlich verwendete Methoden und Attribute.

Ziel/Lösung Ersetzen der Vererbungsbeziehung durch eine Delegationsbeziehung.

Realisierung Listing 7.18 zeigt einen Auszug aus der Transformationsregel für dieses Refactoring. Die Subklasse wird gematcht (vgl. Zeile 2). Anschließend wird die Vererbungsbeziehung entfernt (vgl. Zeile 2) und die Assoziation hinzugefügt (vgl. Zeile 4). Da CD4Code und CD4Analysis keine Mehrfachvererbung unterstützen, hat die Klasse nach Anwendung der Transformation keine Superklasse mehr.

```
1 classdiagram $_{                                                    CDTrans
2   class $sub extends [[ $super :- ]] ;
3
4   [[ :- association $sub -> $super; ]]
5 }
```

Listing 7.18: Auszug aus der Transformation Inheritance By Delegation.

Input Das Refactoring benötigt die Namen der beiden Klassen sowie das Modell.

Varianten -

Synonyme aus der Literatur

- Replace Inheritance with Delegation [Kos09, Are14]

Inverses Refactoring Replace Delegation By Inheritance (vgl. Abschnitt 7.1.10)

7.1.12 Encapsulate Attribute

Encapsulate Attribute ist eine Refactoring, bei dem der öffentliche (public) Zugriff auf Attribute einer Klasse durch explizite Zugriffsmethoden ersetzt wird. Durch dieses Refactoring werden die Daten gekapselt. Die Klasse behält die Kontrolle über die Zugriffe auf die entsprechenden Attribute. Abbildung 7.19 zeigt eine Anwendung dieses Refactorings.

Abbildung 7.19: Kapselung von Attributen.

Die Klasse `Address` hat zwei öffentliche Attribute `int zipcode` und `String city`. Nach Anwendung des Refactorings sind die Attribute verborgen, indem die Sichtbarkeit auf private geändert wurde. Außerdem existieren jeweils eine Methode zur Abfrage und zum Setzten der Attribute.

Situation Attribute einer Klasse sind öffentlich (`public`).

Motivation Kapselung der Daten.

Ziel/Lösung Verstecken der Attribute durch Änderung der Sichtbarkeit und hinzufügen expliziter Zugriffsmethoden.

Realisierung Listing 7.20 zeigt einen Auszug aus der Transformationsregel für dieses Refactoring. In Zeile 1-6 wird eine Klasse gematcht, die ein öffentliches Attribute hat (vgl. Zeile 2). Anschließend wird die Sichtbarkeit des Attributes von `public` auf `private` gesetzt (Zeile 2) und Zugriffsmethoden zum Auslesen (Zeile 4) und Setzen des Attributes (Zeile 5) hinzugefügt. Die Namen der Zugriffsmethoden werden im Zuweisungsblock (Zeile 8-11) berechnet.

CDTrans

```
class $_ {
  [[public  :- private]] $type $attrname;

  [[ :-  public $type $getName(); ]]
  [[ :-  public void $setName($type $attrname); ]]
}

assign {
  $getName = "get" +  capitalize($attrname);
  $setName = "set" +  capitalize($attrname);
}
```

Listing 7.20: Auszug aus der Transformation *Encapsulate Attributes*.

Input Das Refactoring benötigt lediglich das Modell.

Varianten

Boolean Attribute: Das Refactoring erzeugt die Zugriffsmethode mit dem Präfix `is`.
Für bestimmte Attribute: Das Refactoring benötigt zusätzlich die Namen der Attribute.

Synonyme aus der Literatur

- Encapsulate Attribute [Por03, Por05]
- Encapsulate Field [Fow99, SFP17]
- Hide Property [Are14]
- Create Accessors for a Variable [RBJ97]

Inverses Refactoring -

7.1.13 Replace Attribute by Association

Replace Attribute by Association ist ein Refactoring, welches gerichtete Assoziationen mit der Multiplizität 1 durch entsprechende Attribute ersetzt [Lan05]. Dieses Refactoring kann beispielsweise ein vorbereitender Schritt Richtung Implementierung sein, wenn die Zielsprache keine Assoziationen unterstützt.

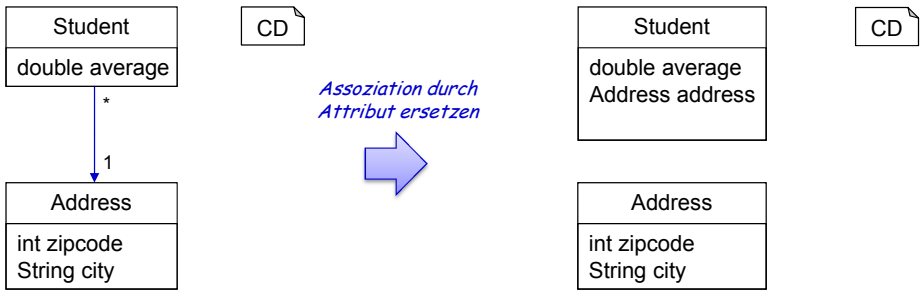


Abbildung 7.21: Ersetzung einer Assoziation durch ein Attribut.

Situation Zwischen zwei Klassen gibt es eine gerichtete Assoziation mit Multiplizität 1.

Motivation Die Zielsprache für die Implementierung unterstützt keine Assoziationen.

Ziel/Lösung Ersetzen der Assoziation durch ein Attribut.

Realisierung Listing 7.22 zeigt einen Auszug aus der Transformationsregel für dieses Refactoring. Das Refactoring matcht eine Klasse (Zeile1-3) sowie eine ausgehende gerichtete Assoziation mit der Multiplizität 1 (Zeile 5). Anschließend wird die Assoziation entfernt und der Klasse ein entsprechendes Attribute hinzugefügt (Zeile 2). Eine weitere Transformationsregel behandelt Assoziationen mit Rollennamen. In diesem Fall wird nicht der kleingeschriebene Klassenname sondern der Rollenname verwendet.

CDTrans

```
1 class $source {
2   [[ :- $target $attr; ]]
3 }
4
5 [[ association $source -> $target  [1] ; :- ]]
6
7 assign {
8   $attr = uncapitalize($target);
9 }
```

Listing 7.22: Auszug aus der Transformation *Replace Association By Attribute*.

Input Das Refactoring benötigt lediglich das Modell.

Varianten *Für spezielle Klasse:* Das Refactoring benötigt zusätzlich die Namen der beteiligten Klassen.

Synonyme aus der Literatur

- Equivalence of Attribute and Association [Lan05]

Inverses Refactoring -

7.1.14 Basistransformationen

Neben den zuvor beschriebenen Transformationen enthält die CDTrans Transformationsbibliothek außerdem eine Reihe von Basis-Transformationen. Diese können ebenfalls durch den Modellierer verwendet werden. Darüber hinaus werden sie als Bausteine für die zuvor beschriebenen Transformationen verwendet. Die CDTrans Transformationsbibliothek umfasst folgende Transformationen:

Rename Attribute Benennt ein Attribut innerhalb eines Klassendiagramms um. Zusätzlich werden die betroffenen Zugriffsmethoden umbenannt. Die Transformation benötigt den neuen und alten Namen des Attributes.

Rename Method Benennt eine Methode innerhalb eines Klassendiagramms um. Die Transformation benötigt den neuen und alten Namen der Methode.

Rename Class Benennt eine Klasse innerhalb eines Klassendiagramms um. Zusätzlich werden die betroffenen Referenzen (z.B. Assoziationen) angepasst. Die Transformation benötigt den neuen und alten Namen der Klasse.

Move Attribute Verschiebt ein Attribute von einer Klasse in eine andere. Die Transformation benötigt den Namen der neuen und alten Klasse sowie den des Attributs.

Move Method Verschiebt eine Methode von einer Klasse in eine andere. Die Transformation benötigt den Namen der neuen und alten Klasse sowie den der Methode.

Remove Attribute Entfernt ein Attribut aus einer Klasse. Zusätzlich werden die betroffenen Zugriffsmethoden entfernt. Die Transformation benötigt den Namen des Attributes und der Klasse, aus der das Attribut entfernt wird.

Remove Method Entfernt eine Methode aus einer Klasse. Die Transformation benötigt den Namen der Methode und der Klasse, aus der die Methode entfernt wird.

Remove Class Entfernt eine Klasse innerhalb eines Klassendiagramms. Zusätzlich werden die Referenzen auf diese Klasse entfernt (z.B. Assoziationen, Vererbungsbeziehungen). Die Transformation benötigt den Namen der Klasse.

7.2 Transformationsbibliothek für MontiArc

In diesem Abschnitt wird die Bibliothek von MontiArc-Transformationen, die mittels der DSTL MATrans entwickelt wurde, vorgestellt. Sie stellt dem Modellierer zur Anwendung und dem Transformationsentwickler zur Wiederverwendung Transformationen zur Normalisierung und zum Bearbeiten von MontiArc-Modellen sowie zur besseren Interoperabilität mit dem Robot Operating System (ROS) [QGC+09] zur Verfügung. Die Entwicklung basiert zum Teil auf der betreuten Vorarbeit [Som16].

7.2.1 Normalisierungen zur Codegenerierung

Eine Normalisierung von Modellen [MVG06] bildet Syntactic Sugar einer Sprache auf Basiskonzepte ab. Ein Beispiel hierfür ist die Vereinfachung von hierarchischen Statecharts [Rum16, MVG06]. Auch der MontiArc-Generator [Hab16] verwendet zunächst normalisierende Transformationen bevor die eigentliche Codegenerierung Template-basiert geschieht. Diese Normalisierungen sind jedoch unabhängig von dem nachgelagerten Codegenerator und können daher auch unabhängig von dieser Codegenerierung zur Normalisierung von MontiArc-Modellen verwendet werden. In diesem Abschnitt wird die Realisierung dieser Transformationen mittels der DSTL MATrans als Teil der MATrans Transformationsbibliothek vorgestellt. Dazu wird jeweils die Idee der Normalisierung anhand eines Beispiels verdeutlicht, anschließend wird die Ausgangssituation sowie das Ziel erläutert und schließlich kurz auf die jeweilige Realisierung eingegangen.

Name Implicitly Named Subcomponents

In MontiArc können Subkomponenten explizit benannt werden. Wird dies nicht gemacht, erhalten Subkomponenten einen impliziten Namen, der sich aus dem Typ der Subkomponente ableitet. Die Normalisierung *Name Implicitly Named Subcomponents* normalisiert MontiArc-Modelle, indem sie implizite zu expliziten Namen macht.

Abbildung 7.23: Beispiel der Normalisierung *Name Implicitly Named Subcomponents*.

Abbildung 7.23 zeigt ein Beispiel für diese Normalisierung. Im linken Teil der Abbildung, also vor der Normalisierung, gibt es eine Systemkomponente, die eine innere Monitorsubkomponente hat. Die Subkomponente ist nicht explizit benannt. Nach der Normalisierung wurde `monitor` als Instanzname für die Subkomponente ergänzt.

Situation Im MontiArc-Modell gibt es Subkomponenten, die nicht benannt sind.

Ziel Der implizite Name (kleingeschriebener Name des Typs) wird den Subkomponenten explizit hinzugefügt, sodass alle Subkomponenten einen expliziten Namen haben.

MATrans

```
1 component $type [[ not[[ $_ ]] :- $name ]];
2
3 assign {
4   $name = uncapitalize($type);
5 }
```

Listing 7.24: Auszug der Transformation *Name Implicitly Named Subcomponents*.

Realisierung Listing 7.24 zeigt einen Auszug aus der Transformationsregel, die diese Normalisierung realisiert. In Zeile 1 wird eine Subkomponente beschrieben. Das negative Element stellt sicher, dass diese keinen expliziten Instanznamen hat. Ist dies gegeben, führt die Transformation einen expliziten Instanznamen ein, der sich aus dem Typ der Subkomponente ableitet (Zeile 3-5).

Name Implicitly Named Ports

In MontiArc können Ports explizit benannt werden. Wird dies nicht gemacht, erhalten Ports einen impliziten Namen, der sich aus dem Typ des Ports ableitet. Die Normalisierung *Name Implicitly Named Ports* normalisiert MontiArc-Modelle, indem sie diese impliziten Namen zu expliziten Namen macht.

Abbildung 7.25: Beispiel für die Normalisierung *Name Implicitly Named Ports*.

Abbildung 7.25 zeigt ein Beispiel für diese Normalisierung. Im linken Teil der Abbildung, also vor der Normalisierung, gibt es eine Monitorkomponente, die einen eingehenden Signalport hat. Der Port ist nicht explizit benannt. Nach der Normalisierung wurde `signal` als Name für den Port ergänzt.

Situation Im MontiArc-Modell gibt es Ports, die nicht benannt sind.

Ziel Der implizite Name (kleingeschriebener Name des Typs) wird den Ports explizit hinzugefügt, sodass alle Ports einen expliziten Namen haben.

Realisierung Listing 7.26 zeigt einen Auszug aus der Transformationsregel, die diese Normalisierung realisiert. In Zeile 1 wird der Port beschrieben, dem durch diese Normalisierung ein neuer Name hinzugefügt wird. Der Name wird in Zeile 3-5 berechnet.

```
1 port  $_ $type [[ :- $name]];                                    MATrans
2
3 assign {
4   $name = uncapitalize($type);
5 }
```

Listing 7.26: Auszug aus der Transformation *Name Implicitly Named Ports*.

Instantiation of Named Inner Component Definitions

MontiArc erlaubt es dem Nutzer innere Komponenten mit Instanznamen zu versehen [Hab16]. Diese Angabe ist äquivalent zur Angabe einer inneren Komponente und einer Subkomponente vom Typ der inneren Komponente mit dem gewählten Instanznamen. Die Normalisierung *Instantiation of Named Inner Component Definitions* [Hab16] formt alle benannten, innere Komponenten in innere Komponenten und Subkomponenten um.

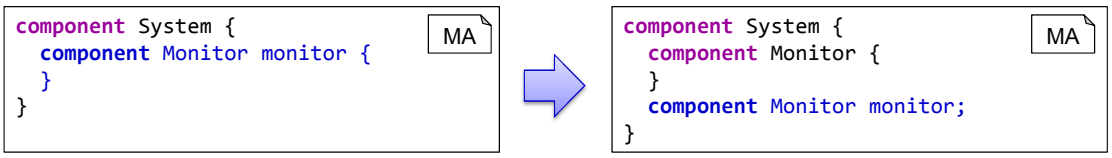


Abbildung 7.27: Beispiel für die Anwendung der Normalisierung *Instantiation of Named Inner Component Definitions*.

Abbildung 7.27 zeigt ein Beispiel für diese Normalisierung. Das Ausgangsmodell besteht aus einer Systemkomponente mit einer inneren Monitorkomponente. Dieser Komponente wurde der Instanzname `monitor` gegeben (vgl. Abbildung 7.27). Nach Anwendung der Normalisierung hat die Systemkomponente weiterhin eine innere Monitorkomponente, allerdings hat diese nun keinen Instanznamen mehr. Zusätzlich gibt es eine Subkomponente vom Typ dieser Komponente mit dem Namen `monitor`.

Situation Im MontiArc-Modell sind innere Komponenten mit Instanznamen versehen.

Ziel Erstellen einer Subkomponente als Instanz der inneren Komponente und entfernen des Instanznamen.

Realisierung Listing 7.28 zeigt einen Auszug aus der Transformationsregel, die diese Normalisierung realisiert. Zeile 1 bis 3 beschreiben eine Komponente und deren innere Komponente. Hierbei wird der Name der Komponente an die Schemavariable `$name` und der Name der Instanz an die Schemavariable `$instance` gebunden. In Zeile 5 wird sichergestellt, dass es keine Subkomponente als Instanz der inneren Komponente mit diesem Namen gibt. Ist dies der Fall, wird eine solche Subkomponente der äußeren Komponente hinzugefügt (Zeile 4) und der Instanzname gelöscht (Zeile 2).

MATrans

```
1 component $_ {
2   component $name [[ $instance :- ]] {
3   }
4   [[ :- component $name $instance; ]]
5   not [[ component $name $instance; ]]
6 }
```

Listing 7.28: Auszug aus der Transformationsregel für *Instantiation of Named Inner Component*.

Expand Autoinstantiate

Ein weiteres Feature von MontiArc, das normalisiert und damit auf Basiskonzepte von MontiArc abgebildet werden kann, ist die automatische Instantiierung von inneren Komponenten. Ähnlich wie die Angabe eines Instanznamen an inneren Komponenten kann auch dieses Feature genutzt werden, um innere Komponenten automatisch zu instantiieren. Das heißt, es werde für diese Komponenten automatisch Subkomponenten erstellt. In diesem Fall wird jedoch der Name automatisch aus der inneren Komponente abgeleitet, indem der kleingeschriebene Name der Komponente verwendet wird. Die Normalisierung *Expand Autoinstantiate* führt für alle nicht instantiierten, inneren Komponenten Subkomponenten als Instanz dieser Komponenten ein.

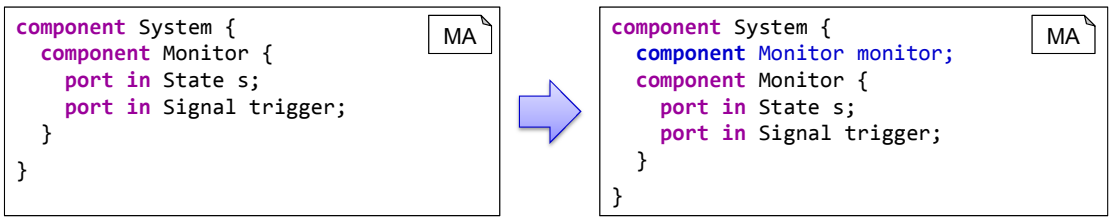


Abbildung 7.29: Beispiel für die Anwendung der Normalisierung der automatischen Instantiierung von inneren Komponenten.

Abbildung 7.29 zeigt ein Beispiel für die Normalisierung der automatischen Instantiierung. Vor der Normalisierung (links) gibt es eine Systemkomponente, die eine innere

Monitorkomponente hat. Für die Systemkomponente gibt es keine Subkomponente. Nach der Normalisierung (rechts) gibt es eine Subkomponente als Instanz der inneren Komponente mit dem Namen `monitor`, der sich aus dem Namen der Komponente ableitet.

Situation Im MontiArc-Modell gibt nicht instantiierte, innere Komponenten.

Ziel Für nicht instantiierte, innere Komponenten werden Subkomponenten hinzugefügt.

Realisierung Listing 7.30 zeigt einen Auszug aus der Transformationsregel, die Normalisierung *Expand Autoinstantiate* realisiert. Das Pattern besteht aus einer Komponente mit innerer Komponente. Es gibt keine Subkomponente als Instanz der inneren Komponente (Zeile 3). Die Normalisierung fügt eine Subkomponente als Instanz explizit ein (Zeile 4). Der Instanzname wird über die automatische Benennung von Subkomponenten hinzugefügt.

MATrans

```
1 component $_ {
2   component $type {}
3   not [[ component $type; ]]
4   [[ :- component $type; ]]
5 }
```

Listing 7.30: Auszug aus der Transformation *Expand Autoinstantiate*.

Qualify Subcomponent Connectors

MontiArc erlaubt es außerdem Konrektoren zwischen Subkomponenten zu modellieren [Hab16]. Dies ist eine verkürzte Schreibweise für eine Modellierung einzelner Konnektoren zwischen den Ports der Subkomponenten. Die Normalisierung *Qualify Subcomponent Connectors* [Hab16] ersetzt alle Konnektoren, die zwischen Subkomponenten definiert sind, durch einzelne Konnektoren zwischen den Ports der Subkomponenten.

MA

```
component Controller {
  port out State s;
  port out Signal t;
}
```

MA

```
component Monitor {
  port in State s;
  port in Signal trigger;
}
```

Abbildung 7.31: Zwei MontiArc-Modelle, die die beiden Komponenten `Monitor` und `Controller` zeigen.

Abbildung 7.31 und Abbildung 7.32 (links) zeigen ein Beispiel für die Verwendung eines Konnektors zwischen Subkomponenten. In Abbildung 7.32 (links) ist eine Systemkomponente mit zwei Subkomponenten `monitor` und `control` sowie einem Konnektor zwischen diesen Subkomponenten abgebildet. Hierbei ist `monitor` eine Instanz der Komponente `Monitor` und `control` eine Instanz der Komponente `Controller` (vgl. Abbildung 7.31). `Controller` hat zwei ausgehende Ports: `s` vom Typ `State` und `t` vom Typ `Signal`. `Monitor` hat zwei eingehende Ports `s` vom Typ `State` und

`trigger` vom Typ `Signal`. Abbildung 7.32 (rechts) zeigt die Systemkomponente nach Anwendung der Normalisierung. Der Konnektor zwischen den Subkomponenten ist nicht mehr vorhanden. Stattdessen gibt es zwei Konnektoren. Der eine Konnektor verbindet die Stateports und der andere Konnektor verbindet die Signalports. Die Monitor- und die Controllerkomponente bleiben unverändert.

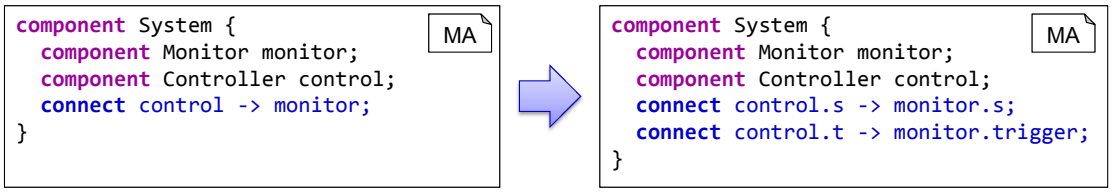


Abbildung 7.32: Ersetzung eines Konnektors zwischen zwei Subkomponenten durch einzelne Konnektoren zwischen den Ports der Subkomponenten.

Situation Im MontiArc-Modell sind Konnektoren zwischen Subkomponenten anstatt zwischen Ports beschrieben.

Ziel Ersetzen der Konnektoren zwischen den Subkomponenten durch einzelne Konnektoren, die die Ports dieser Subkomponenten verbinden.

Realisierung Listing 7.33 zeigt einen Auszug aus der Transformationsregel, die die Normalisierung *Qualify Subcomponent Connectors* realisiert. Zeile 7-14 beschreiben eine Komponente mit zwei Subkomponenten (Zeile 8, 9) und einem Konnektor zwischen diesen Subkomponenten (Zeile 10). In Zeile 1-3 sowie 4-6 werden die beiden Komponenten, deren Instanzen die Subkomponenten sind, beschrieben. Die eine Komponente hat einen ausgehenden Port dessen Typ mit dem Typ eines eingehenden Ports der anderen Komponente übereinstimmt. Das negative Element in Zeile 12 drückt aus, dass es noch keinen Konnektor zwischen den beiden Ports gibt. Ist diese Situation gegeben, wird der Komponente mit dem Subkomponentenkonnektor ein Konnektor zwischen den beiden beschriebenen Ports hinzugefügt. Sind alle Ports verbunden, wird der Subkomponentenkonnektor in einer nachgelagerten Transformationsregel entfernt.

Expand Autoconnect

Das `autoconnect`-Feature erlaubt Ports in MontiArc-Modellen automatisch verbinden zu lassen [Hab16]. Es gibt zwei Varianten. `autoconnect port` verbindet Ports, falls deren Name und deren Typ übereinstimmen. `autoconnect type` hingegen verbindet Ports, deren Typ übereinstimmt. Eine Übereinstimmung des Portnamens ist hierbei nicht erforderlich. Diese Feature kann normalisiert werden, indem stattdessen die Ports explizit verbunden werden. Dies macht die Normalisierung *Expand Autoconnect*.

Abbildung 7.34 zeigt für beide Varianten jeweils ein Beispiel. Im oberen Bereich a) ist `autoconnect port` abgebildet. Das Beispiel zeigt eine Systemkomponente mit den

```
1  component $sType {                                          MATrans
2      port  out $pType $sPort ;
3  }
4  component $tType {
5      port  in $pType $tPort ;
6  }
7  component $_ {
8    component $sType  $source ;
9    component $tType  $target ;
10   connect $source -> $target;
11
12    not [[ connect $source.$sPort -> $target.$tPort; ]]
13   [[ :- connect $source.$sPort -> $target.$tPort; ]]
14 }
```

Listing 7.33: Auszug aus der Transformation *Qualify Subcomponent Connectors*.

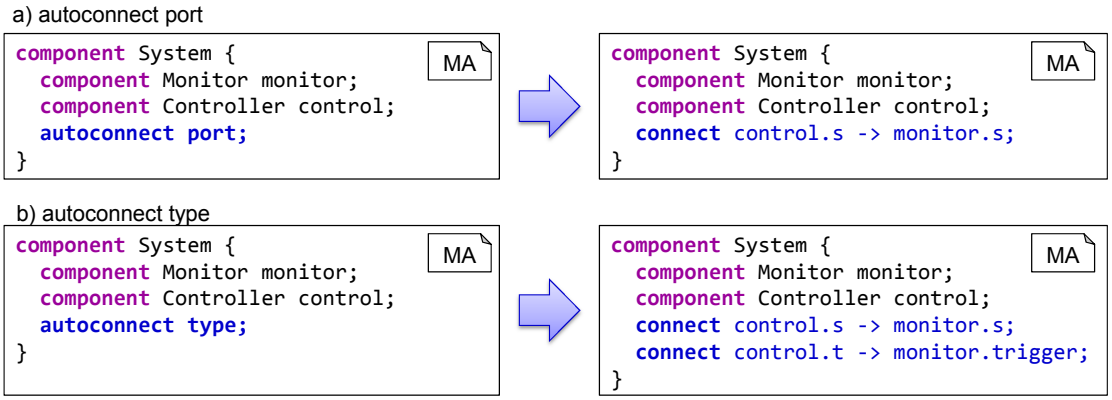


Abbildung 7.34: Beispiel für die Normalisierung des a) `autoconnect port`-Features und b) des `autoconnect type`-Features.

Subkomponenten `control` und `monitor` als Instanzen der Komponenten `Controller` und Monitor. Nach Anwendung der Normalisierung sind die Ports vom Typ `State` mit dem übereinstimmenden Namen `s` verbunden. Im unteren Fall b) (`autoconnect type`) ist zusätzlich der Signalport der beiden Subkomponenten verbunden.

Situation Im MontiArc-Modell ist das `autoconnect`-Feature verwendet.

Ziel Im Fall von `autoconnect port`: Es sollen alle Ports, die den gleichen Typ und den gleichen Namen haben, durch Konnektoren verbunden werden. Im Fall von `autoconnect type`: Es sollen alle Port, die den gleichen Typ haben, durch Konnektoren verbunden werden, unabhängig von deren Namen.

Realisierung Listing 7.35 zeigt einen Auszug aus der Transformationsregel, die die Normalisierung für `autoconnect port` realisiert. In Zeile 7-12 ist eine Komponente beschrieben, die `autoconnect port` nutzt (Zeile 8). Diese Komponente hat zwei

Subkomponenten als Instanzen der ebenfalls beschriebenen Komponenten (Zeilen 1-3 bzw. 4-6). Die eine Komponenten hat einen ausgehenden Port der im Namen und Typ mit einem eingehenden Port der anderen Komponente übereinstimmt. Die Ports der Subkomponenten werden verbunden (Zeile 11). Sobald alle Ports verbunden sind, kann das `autoconnect`-Statement entfernt werden. `autoconnect type` ist analog realisiert, wobei hier nur der Typ der Ports übereinstimmen muss.

```
1  component $sourceComp {                                    MATrans
2    port out $type $name;
3  }
4  component $targetComp {
5    port in $type $name;
6  }
7  component $_ {
8    autoconnect port;
9    component $sourceComp $source;
10   component $targetComp  $target ;
11   [[ :- connect $source.$name -> $target.$name; ]]
12 }
```

Listing 7.35: Auszug aus der Transformation *Autoconnect Port*.

Expand Simple Connectors

Ein weiteres Feature von MontiArc sind simple Konnektoren. Diese werden direkt an einer Subkomponente in eckigen Klammern angegeben. Äquivalent dazu ist ein „normaler“ Konnektor zwischen den angegebenen Ports. Die Normalisierung *Expand Simple Connectors* ersetzt simple durch normale Konnektoren.

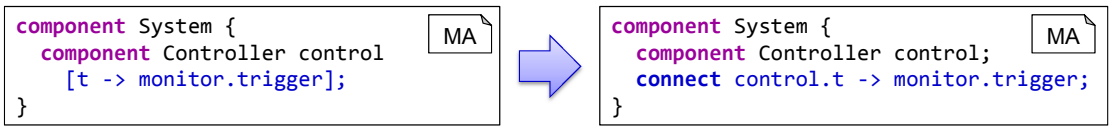


Abbildung 7.36: Beispiel für die Normalisierung *Expand Simple Connectors*.

Abbildung 7.36 zeigt ein Beispiel für diese Normalisierung. Vor der Normalisierung (links) hat die Subkomponente `control` einen simplen Konnektor. Dieser verbindet den ausgehenden Port `t` mit dem Port `trigger`-Port der Subkomponente `monitor` (in der Abbildung ausgelassen). Nach der Normalisierung (rechts) existiert ein normaler Konnektor zwischen den Ports.

Situation Im MontiArc-Modell gibt es simple Konnektoren.

Ziel Der simple Konnektor wird durch einen normalen Konnektor ersetzt.

Realisierung Listing 7.37 zeigt einen Auszug aus der Transformationsregel, die diese Normalisierung realisiert. In Zeile 2-4 wird eine Subkomponente mit simplem Konnektor

beschrieben. Ist dies gegeben, entfernt die Transformation diesen simplen Konnektor und fügt einen normalen Konnektor ein (Zeile 5).

```
1 component $_ {                                                    MATrans
2   component $_ $name [
3         [[  $source -> QualifiedName $TARGET  :- ]]
4                     ];
5   [[ :- connect $name.$source -> QualifiedName $TARGET; ]]
6 }
```

Listing 7.37: Auszug aus der Transformation *Expand Simple Connectors*.

7.2.2 Wrapper für Porttypen

Eine weitere wiederverwendbare Transformation ist die `Wrap Ports` Transformation. Bei dieser Transformation werden alle Porttypen gekapselt. Dies ist beispielsweise für die Interoperabilität mit dem Robot Operating System (ROS) [QGC+09] hilfreich [AHRW17a], da ROS jegliche Kommunikation durch Nachrichten (Messages) kapselt. Die Transformation nutzt das Feature der Parameterübergabe an Transformationen, sodass der Nutzer angeben kann, durch welchen Wrapper die Ports gekapselt werden sollen.

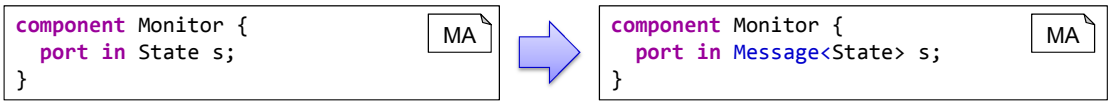


Abbildung 7.38: Beispiel für die Anwendung der Transformation *Wrap Ports*.

Abbildung 7.38 zeigt ein Beispiel für diese Transformation. Die Komponente `Monitor` hat zwei Ports: einen vom Typ `State` und einen vom Typ `Signal`. Im Beispiel wurde `Message` als Wrapper zur Kapselung der Ports gewählt. Als Ergebnis der Transformation sind die Porttypen jeweils durch `Message` gekapselt.

Ziel Kapselung der Porttypen durch einen Wrapper.

Realisierung Listing 7.39 zeigt die Transformationsregel, die diese Transformation realisiert. In Zeile 1-3 wird ein Port beschrieben. Der Porttyp wird mit dem Replacement Operators durch einen Wrapper, der den Typ beinhaltet, ersetzt (Zeile 3).

```
1 port $_ [[ SimpleReferenceType $SRT                               MATrans
2            :-
3            wrapper<SimpleReferenceType $SRT> ]] $_;
```

Listing 7.39: Auszug aus der Transformation *Wrap Ports*.

7.2.3 Vereinfachung von MontiArc-Modellen

In MontiArc können Komponenten atomar sein, d.h. es gibt keine Subkomponenten innerhalb der Komponenten, oder dekomponiert, d.h. es gibt Subkomponenten innerhalb der Komponenten [Hab16]. Ähnlich wie bei Statecharts [Rum16] können dekomponierte Komponenten vereinfacht werden, indem die Hierarchie entfernt wird. Auch diese Transformation ist beispielsweise im ROS-Kontext hilfreich, da ROS keine Hierarchien unterstützt. Die `Flatten`-Transformation realisiert diese Vereinfachung.

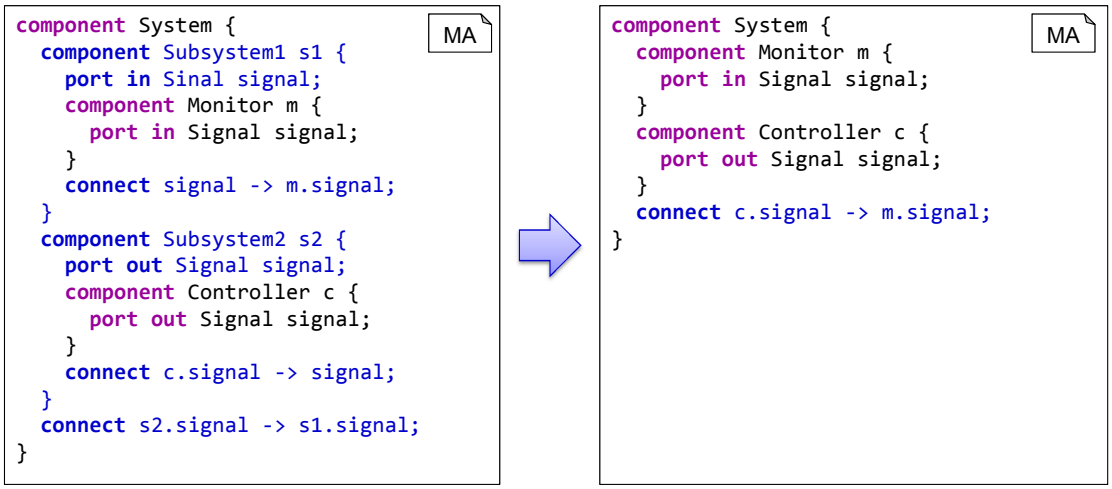


Abbildung 7.40: Beispiel für die Anwendung der Normalisierung *Flatten*.

Abbildung 7.40 zeigt ein Beispiel für diese Transformation. Das System besteht vor Anwendung der Transformation (links) aus zwei dekomponierten Komponenten `Subsystem1` und `Subsystem2`, wobei `Subsystem1` eine Monitorsubkomponente und `Subsystem2` eine Controllersubkomponente hat. Alle Komponenten haben einen Signalport. Der ausgehende Signalport der Controllerkomponente ist mit dem ausgehenden Port der `Subsystem2`-Komponente verbunden. Dieser ist mit dem eingehenden Signalport der `Subsystem1`-Komponente verbunden. Der eingehenden Signalport der `Subsystem1`-Komponente ist mit dem eingehenden Signalport der Monitorkomponente verbunden. Ein Signal fließt vom `Controller` über das `Subsystem2` zum `Subsystem1` zum `Monitor`. Nach der Vereinfachung (Abbildung 7.40 rechts) besteht das System nur noch aus den zwei atomaren Komponenten `Monitor` und `Controller`. Der Signalport der Controllerkomponente ist direkt mit dem Signalport der Monitorkomponente verbunden.

Situation Im MontiArc-Modell gibt es hierarchische (dekomponierte) Komponenten.

Ziel Vereinfachung des MontiArc-Modells durch Entfernung der Hierarchie.

Realisierung Listing 7.41 zeigt einen Auszug einer Transformationsregel, die diese Transformation realisiert. Die gezeigte Transformationsregel ändert die Quelle eines Konnektors von der äußeren hin zur inneren Komponente. Dadurch "überspringt" der

Konnektor eine Zwischenschicht. Im oberen Beispiel würde also `s2.signal` durch `c.signal` ersetzt. Das Pattern beschreibt drei Komponenten: die äußere in Zeile 1-5, die mittlere in Zeile 11-15 und die innere in Zeile 7-9. Es gibt einen Konnektor, der den ausgehenden Port der inneren mit dem ausgehenden Port der mittleren Komponente verbindet (Zeile 14) und einen Konnektor, der den ausgehenden Port der mittleren mit einem beliebigen anderen Port verbindet (Zeile 3-4).In diesem Fall wird der Port, der die mittlere mit einer beliebigen Komponente verbindet umgeleitet, sodass er die innere mit der gleichen beliebigen anderen Komponente verbindet und damit die mittlere überspringt. Dies wird wiederholt bis alle Konnektoren an atomaren Komponenten beginnen und die Transformationsregel nicht mehr anwendbar ist. Für eingehende Port wird analog vorgegangen, bis alle Ports an atomaren Komponenten enden. Anschließend werden die nicht länger benötigten mittleren Komponenten entfernt.

MATrans

```
1  component $_ {
2    component $source $outer;
3    connect [[$outer.$portO :- $instance.$portI]]
4                                -> QualifiedName $_;
5  }
6
7  component $inner {
8    port out $portType $portI;
9  }
10
11 component $source {
12   port out $portType $portO;
13   component $inner $instance;
14   connect $instance.$portI -> $portO;
15 }
```

Listing 7.41: Auszug aus der Transformation *Flatten*.

7.2.4 Eliminierung generischer Komponenten

Eine weitere wiederverwendbare Transformation für MontiArc-Modelle ist `Eliminate Generic Components`. Diese Transformation ersetzt generische Komponenten durch normale Komponenten, wobei der generische Parameter entsprechend eingesetzt wird. Genau wie *Wrap Ports* und *Flatten* ist auch diese Transformation beispielsweise im ROS-Kontext von Interesse, da ROS keine Generics unterstützt.

Abbildung 7.42 zeigt ein Beispiel für die Ersetzung von generischen durch nicht generische Komponenten. In diesem Beispiel ist die Monitorkomponente vor der Transformation (links in der Abbildung) generisch. Der generische Parameter bestimmt den Typ sowohl des eingehenden als auch des ausgehenden Ports. Die Systemkomponente hat eine Subkomponente als Instanz der Monitorkomponente, wobei `State` für den generischen Parameter der Monitorkomponent verwendet wird. Nach Anwendung der Transforma-

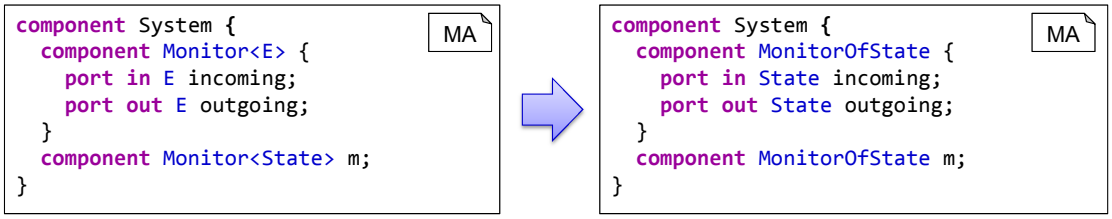


Abbildung 7.42: Beispiel für die Normalisierung *Eliminate Generic Components*.

```
1  component $type<$_> ComponentBody $body                    MATrans
2
3  component $_ {
4    component    [[ $type :- $compName ]] [[ <$param>  :- ]] $_;
5    [[ :- component $compName ComponentBody $body ]]
6  }
7
8  assign {
9    $compName = $type + "Of" + $param;
10 }
```

Listing 7.43: Auszug aus der Transformation *Eliminate Generic Components*.

tion (rechts) gibt es die generische Monitorkomponente nicht mehr, dafür wurde ein `MonitorOfState`-Komponente erstellt, deren Ports den Typ `State` verwendet. Die Subkomponente wurde durch eine Subkomponente von diesem Typ ersetzt.

Situation Im MontiArc-Modell gibt es generische Kompoonenten.

Ziel Eliminierung der generischen Komponenten.

Realisierung Listing 7.41 zeigt einen Auszug einer Transformationsregel, die diese Transformation realisiert. Die Transformationsregel fügt nicht generische für generische Komponenten ein und ändert den Typ der Subkomponente entsprechend. Das Pattern beschreibt eine generische Komponente (Zeile 1) und eine Komponente, die eine Subkomponente als Instanz der generischen Komponente hat (Zeile 3-6). Ist diese Situation gegeben, ändert die Transformation den Typ der Subkomponente auf den neuen Typ (Zeile 4). Der neue Typ berechnet sich aus dem Namen der generischen Komponente und dem konkret verwendeten generischen Parameter (Zeile 8-10). Im oberen Beispiel also `MonitorOfState`. Außerdem entfernt die Transformationsregel den generischen Parameter der Subkomponente und führt die neue nicht-generische Komponente als Kopie der generischen ohne den generischen Parameter und mit dem neuen Namen ein (Zeile 5). In nachgelagerten Transformationsregeln werden noch die generischen Parameter innerhalb der Kopie auf den konkreten Typ geändert. Im Beispiel waren dies die Typen der beiden Ports.

7.3 Diskussion und verwandte Arbeiten

In diesem Kapitel wurden ergänzend zu den in Kapitel 5 vorgestellten DSTLs Bibliotheken wiederverwendbarer Modelltransformationen für CD4Code-, CD4Analysis- und MontiArc-Modelle vorgestellt (vgl. TR 1 und TR 2). Zum einem wurde damit die Möglichkeit geschaffen, dass Transformationsentwickler auf diese zurückgreifen, um neue Transformationen zu entwickeln und Modellierer, um entwickelte Modelle zu transformieren. Zum anderen können die hier entwickelten Transformationen auch als Vorlage und Orientierung bei der Entwicklung neuer Transformationen dienen. Zusätzlich demonstrieren sie die Verwendbarkeit der im Rahmen dieser Arbeit entwickelten DSTLs.

Soweit der Bedarf erkennbar war, enthalten die Bibliotheken neben den eigentlichen Transformationen auch Varianten der einzelnen Transformationen (vgl. TR 3). Die Refactorings, Normalisierungen und ROS-Interoperabilitätstransformationen sind CD4Analysis, CD4Code bzw. MontiArc-spezifisch jedoch jeweils möglichst allgemein verwendbar entwickelt worden. Dennoch kann es vorkommen, dass diese allgemeinen Formen nicht exakt zu dem gegebenen Transformationsbedarf passen. Ist keine der Transformationen und Varianten verwendbar, können diese als Vorbild oder Grundlage zur Entwicklung projektspezifischer Transformationen dienen (vgl. TR 4). Der Transformationsentwickler kann sich an den Lösungsideen orientieren oder die vorhandenen Transformationen mit eigenen kombinieren, um passende Transformationen zu entwickeln.

In der Literatur finden sich zwar viele Beschreibungen von Modelltransformationen für Klassendiagramme bzw. solche, die sich auf Klassendiagramme übertragen lassen [RBJ97, Fow99, Rob99, Chr05, ZLG05, ALC08, WR09], die Umsetzung ist jedoch abhängig von der verwendeten Modellierungs- und Transformationssprache und können somit nicht in Kombination mit CD4Code, CD4Analysis oder MontiArc verwendet werden. Wie auch für die Softwareentwicklung gibt es auch im Bereich der Modellierung und Modelltransformation Ideen für wiederverwendbare Modelltransformationen [ISH08, SMM+12, dLG14, CGdL15, CMIC15]. Mit den in diesem Kapitel vorgestellten Transformationen wurde eine Basis an wiederverwendbare Transformationen für CD4Code-, CD4Analysis- und MontiArc-Modelle geschaffen.

Neben den in dieser Arbeit vorgestellten Bibliotheken bieten auch der Model Refactoring Browser [ZLG05] basierend auf dem Constraint-Specification Aspect Weaver C-SAW [ZLG05], der Refactoring Browser [BSF03] und EMF Refactor [Are14, AT13] wiederverwendbare Modelltransformationen für Klassendiagramme. Keines dieser Tools eignet sich jedoch für die Transformation von CD4Code- bzw. CD4Analysis-Modellen. Refactorings für UML/P Modelle werden in [Rum17] beschrieben, wobei hierfür bisher keine Implementierung für CD4Analysis bzw. CD4Code Modelle existiert.

Für Softwarearchitekturen existieren eine Reihe von Pattern sowie Lösungen für Anti-Pattern und Beschreibungen konkreter Transformationen [MKMG97, KKB+99, PR99, Fie00, Fow02, SW01, SW03]. Umsetzungen in Form von Sammlungen oder Tools existieren hierfür jedoch nicht.

Kapitel 8

Generierung der Transformationen

In den vorangegangenen Kapiteln wurden die Notation sowie die Operatoren der DSTLs vorgestellt (vgl. Kapitel 4). Darauf aufbauend wurden DSTLs für CD4Analysis-, CD4-Code- und MontiArc-Modelle (vgl. Kapitel 5) sowie wiederverwendbare Transformationen für Modelle dieser Sprachen vorgestellt (vgl. Kapitel 7). Außerdem wurde die systematische Ableitung sowie Generierung von DSTLs erläutert (vgl. Kapitel 6). In diesem Kapitel wird die Generierung von ausführbaren Java-Implementierungen aus modellierten Transformationsregeln vorgestellt. Damit schließt dieses Kapitel die Lücke zwischen der Modellierung und der Verwendung von Transformationen.

Die DSTLs in MontiCore erlauben eine deklarative Beschreibung von Transformationsregeln. Hierbei liegt der Fokus daher nicht auf der Beschreibung wie die Transformation des Modells von statten geht, sondern darauf, was das Modell erfüllen muss und welchen Effekt die Transformationsregel haben soll. Damit der Transformationsentwickler die Transformationsregeln nicht nur beschreiben, sondern auch anwenden kann, muss diese ausführbar werden. In [Wei12] wurde hierfür ein generativer Ansatz vorgestellt. Dieser wurde im Rahmen dieser Arbeit übernommen, für die neue MontiCore Version angepasst und weiterentwickelt. Für die in Kapitel 4 beschriebenen Erweiterungen des Collection Operators, den in dieser Arbeit entwickelten Optional Operator sowie den Zuweisungsblock und den Anweisungsblock wurde die auf Objektdiagrammen basierende ODRules-Notation von Transformationsregeln erweitert. Außerdem wurde der Pattern Matching Algorithmus für die Operatoren erweitert. Zusätzlich wurde die Performance des Pattern Matching Algorithmus durch eine automatische Modularisierung und dezentrale Überprüfung des Application Constraints verbessert.

Abbildung 8.1 zeigt erneut die bereits in Abbildung 1.1 vorgestellte dreischichtige Darstellung der Transformationssprachen- und Transformationsentwicklung sowie Transformationsanwendung. Die in diesem Kapitel relevante mittlere Schicht ist hervorgehoben. Die Generierung von Transformationen befindet sich in der mittleren Schicht und adressiert den Transformationsentwickler. Der Transformationsentwickler modelliert Transformationen und nutzt den Transformationsgenerator, um ausführbare, in Java implementierte Transformationen zu erhalten. Der Transformationsgenerator ist spezifisch für eine DSTL und wird daher zusammen mit der DSTL generiert. Aus Nutzersicht bekommt der Generator eine modellierte Transformation konform zu der zugehörigen DSTL als Input und generiert eine dazu passende Java-Implementierung. Abbildung 8.2 zeigt die Details

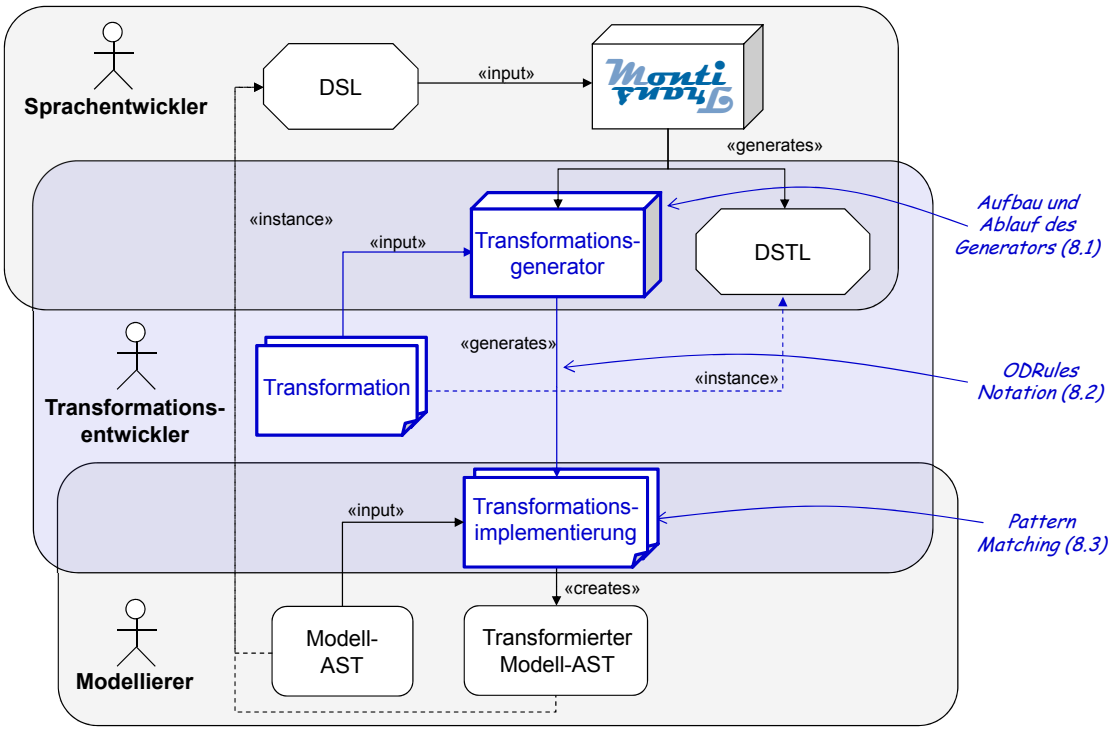


Abbildung 8.1: Übersicht über die verschiedenen Aspekte von MontiTrans mit Fokus auf den in diesem Kapitel vorgestellten Teil (Generierung von Java-Implementierungen aus Transformationen).

und beteiligten Artefakte der Generierung von Java-Implementierungen für Transformationsregeln am Beispiel der DSTL CDTrans. Eine Erklärung dieser Abbildung erfolgt im nachfolgenden Abschnitt.

Der Fokus dieses Kapitels liegt auf der Generierung von Java-Implementierungen aus mithilfe von DSTLs modellierten Transformationsregeln. Nachdem die Einordnung in das Schaubild vorgenommen wurde, wird zunächst der Aufbau des Generators erläutert. Anschließend werden der Ablauf der Generierung und die daran beteiligten Artefakte vorgestellt. Schließlich werden noch eine der Generierung als Zwischenstruktur dienende, auf Objektdiagrammen basierende Notation von Transformationsregeln sowie der Pattern Matching Algorithmus vorgestellt. Abschließend werden die Unterschiede des Aufbaus der Generatoren sowie des Pattern Matching Algorithmus im Vergleich zu der in [Wei12] präsentierten Umsetzung diskutiert.

Die wichtigsten Ergebnisse dieses Kapitels sind:

- Eine Beschreibung des Ablaufs der Generierung von Java-Implementierungen für Transformationsregeln und der daraus resultierenden Artefakte.
- Die Vorstellung einer Objektdiagramm-basierten Notation als modellierungsspra-

chenunabhängige Zwischenstruktur zur Codegenerierung für Transformationsregeln in domänenspezifischer Notation.

- Die Vorstellung des für die weiterentwickelten und neu entwickelten Operatoren der DSTLs erweiterten suchplanbasierten Pattern Matching Algorithmus.
- Die automatische Modularisierung von Application Constraints und die dezentrale Überprüfung der Constraints zur Performanceverbesserung des Pattern Matching.

Eine Methodik zur Entwicklung neuer Transformationen wird im nachfolgenden Kapitel 9 beschrieben. Zusätzlich wird dort die Verwendung von Transformationen erklärt.

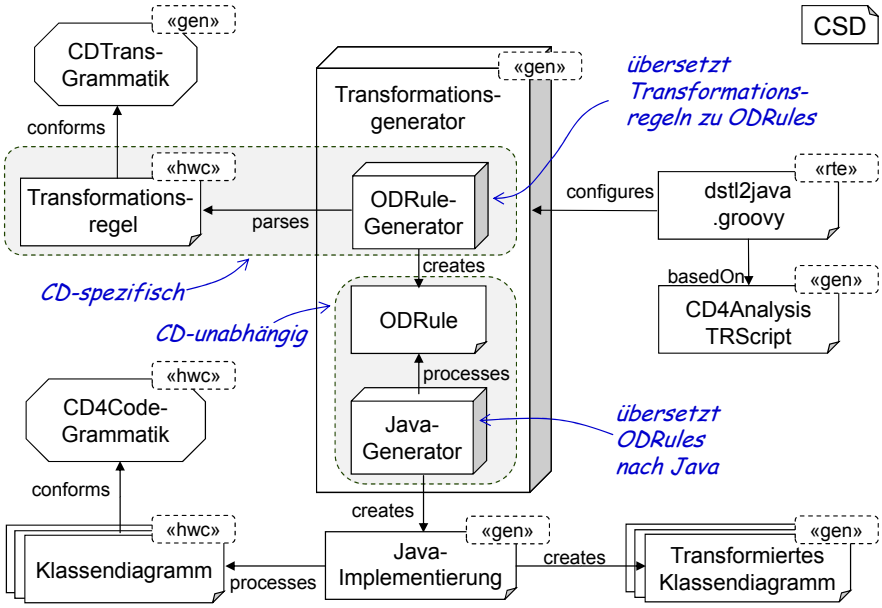


Abbildung 8.2: Erstellung von Transformationsimplementierung (Detailsicht).

8.1 Aufbau des Generators und Ablauf der Generierung

Der Transformationsgenerator ist zweigeteilt und besteht aus einem Java-Generator und einem ODRule-Generator (vgl. Abbildung 8.2). Der Transformationsgenerator wird – wie auch der MontiTrans DSTL-Generator – über ein Groovy-Skript gesteuert. Das benötigte Groovy-Skript *dstl2java* wird von MontiTrans zur Verfügung gestellt und gibt den Ablauf der Generierung vor. Das Skript ist in Listing 9.16 dargestellt und wird in Kapitel 9 beschrieben. Außerdem wird dort auch die Entwicklung und Verwendung eigener Groovy-Skripte erklärt. Der Standardablauf der Generierung wird im weiteren Verlauf dieses Kapitels detailliert. Die zugehörige Base Class ist erneut DSTL-spezifisch und wird zusammen mit der DSTL generiert. Die Base Class für CDTrans ist in Abbildung 8.4 dargestellt und wird im weiteren Verlauf dieses Kapitels erläutert. Der Transformationsgenerator bekommt eine Transformationsregel als Input und erzeugt eine Java-Klasse, die diese Transformation implementiert. Die Implementierung besteht hierbei

im wesentlichen aus dem Pattern Matching und der Modifikation der Eingabemodelle. Der ODRule-Generator bildet den ersten Teil des Transformationsgenerators. Dieser Generator bekommt die Transformationsregel in DSTL-Notation als Eingabe und produziert basierend auf dieser eine Transformationsregel in Objektdiagramm-basierter Notation, die ODRule. Die Notation wird im späteren Verlauf dieses Kapitels detailliert. Die ODRule bildet die Eingabe für den Java-Generator. Dieser generiert basierend auf der ODRule die Java-Klasse als Implementierung der Transformationsregel. Durch die Zweiteilung des Generator in einen modellierungssprachenabhängigen und einen modellierungssprachenunabhängigen Teil kann der unabhängige zweite Teil des Generators für alle DSTLs verwendet werden. Außerdem kann er auch für andere Frontends wie beispielsweise Delta-Sprachen [HHK+15, HHK+13] oder andere konkrete Syntaxen für Modelltransformationen wiederverwendet werden.

Die Steuerung des Ablaufs der Generierung von Java-Implementierungen für Transformationsregeln in DSTL-Notation erfolgt durch das von MontiTrans bereitgestellte *dstl2java* Groovy-Skript. Abbildung 8.3 stellt den Ablauf der Generierung in Form eines Aktivitätsdiagramms dar, während Abbildung 6.14 die wichtigsten Klassen des Transformationsgenerators am Beispiel CDTrans darstellt. Der Generator liest zunächst die Transformationsregeln, für die eine Java-Implementierung generiert werden soll, ein. Anschließend werden die Kontextbedingungen der DSTL (vgl. Abschnitt 6.4) überprüft. Ist die Transformationsregel nicht wohlgeformt, gibt der Generator die entsprechende Meldung der angeschlagenen Kontextbedingung aus und terminiert.

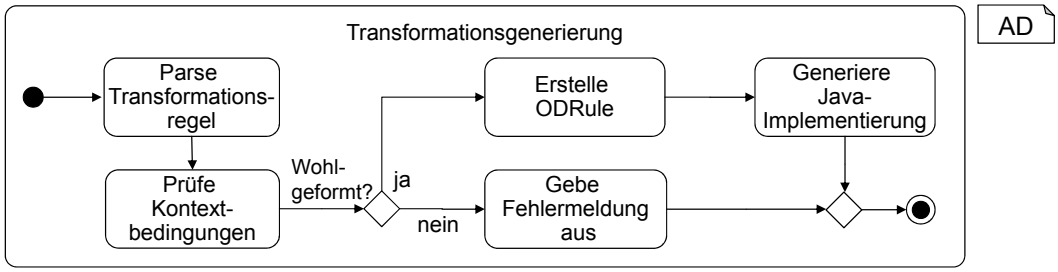


Abbildung 8.3: Ablauf der Generierung von Transformationsimplementierung.

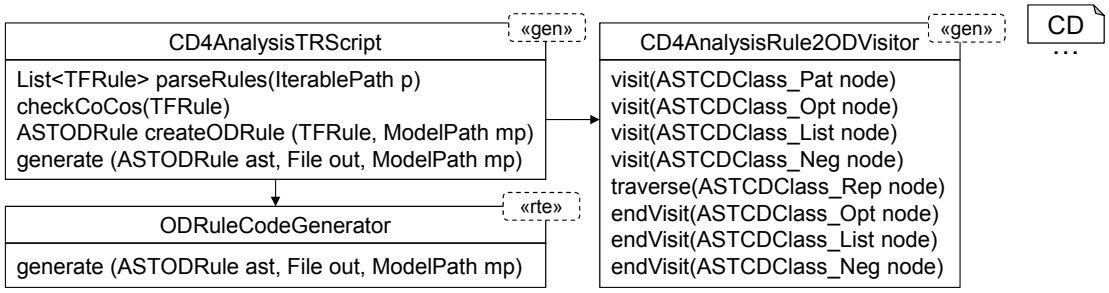


Abbildung 8.4: Struktur des Transformationsgenerators für CDTrans.

Ist die Transformationsregel wohlgeformt, wird sie weiter verarbeitet und an den ODRule-Generator weitergereicht. Dieser erstellt basierend auf der Transformationsregel eine ODRule. Die Übersetzung entspricht einer exogenen M2M-Transformation und ist durch einen Visitor – im Fall von CDTrans dem `CD4AnalysisRule2ODVisitor` – realisiert. Dieser traversiert die AST-Struktur der Transformationsregel und erstellt dabei die AST-Struktur der ODRule. Das Mapping wird in Abschnitt 8.2 vorgestellt. Die so erstellte ODRule wird danach an den Java-Generator (repräsentiert durch die Klasse `ODRuleCodeGenerator` in Abbildung 8.4) weitergereicht. Dieser generiert templatebasiert anhand der ODRule eine Java-Implementierung der Transformationsregel. Im Folgenden wird zunächst die angesprochene objektdiagrammbasierte ODRules-Notation sowie das Mapping von der DSTL-Notation hin zur ODRules-Notation und weiter zur generierten Java-Implementierung erläutert (Abschnitt 8.2). Anschließend wird in Abschnitt 8.3 der Pattern Matching Algorithmus detailliert.

8.2 Transformationsregeln in ODRules-Notation

Die Objektstruktur eines Modells kann nach dem Parsen durch ein Objektdiagramm dargestellt werden [Wei12], das einen Spezialfall eines attributierten Graphen darstellt. Diese Abbildung von Modellen auf Objektdiagramme und auf gerichtete, attributierte Graphen kann auch auf Modellteile angewendet werden [Wei12]. Dies stellt die Grundlage für die Anwendung von Graphersetzungsregeln dar. Die Generierung von Java-Implementierungen für Transformationsregeln nutzt die ODRules-Notation als Zwischenstruktur. Für die in Kapitel 4 vorgestellten neuen Features der DSTLs wurde die Notation erweitert. Diese erweiterte Notation wird nachfolgend vorgestellt.

Eine Graphersetzungsregel besteht in der Regel aus zwei Graphen: einer linken Regelseite (LHS) und einer rechten Regelseite (RHS) [Nag79, EEPT06, EM12, Sch91]. Die linke Regelseite entspricht dem Pattern und damit dem Zustand des Graphen vor Anwendung der Transformation. Die rechte Regelseite umfasst den gleichen Graphen nach Anwendung der Transformation. Die Modifikation ergibt sich aus dem Unterschied zwischen den beiden Graphen. Die ODRules-Notation (OD-Notation) nutzt zur Darstellung der LHS und RHS jeweils ein Objektdiagramm [Rum16, Sch12]. Obwohl eine konkrete Syntax für diese Sprache existiert, dient die OD-Notation primär als Zwischenrepräsentation, die im Speicher aufgebaut und normalerweise nicht als Datei erzeugt wird.

Abbildung 8.5 demonstriert das Mapping zwischen DSTL und OD-Notation am Beispiel einer CDTrans-Transformationsregel. Das Mapping wird im Folgenden erläutert. Im linken Teil der Abbildung ist eine Transformationsregel in CDTrans und auf der rechten Seite die gleiche Regel in OD-Notation dargestellt. Die Transformationsregel beschreibt ein Pattern, das aus einer Klasse namens `Person` und einem beliebigen Attribut besteht. Als Modifikation wird durch die Transformationsregel das Attribut aus der Klasse entfernt und die Klasse von `Person` zu `Human` umbenannt. Im linken Teil der Abbildung wurde hierfür die konkrete Syntax der Klasse und eine Schemavariable für das Attribut verwendet. Außerdem wurde der Replacement Operator verwendet, um das Attribut aus

CDTrans

```
class [[ Person :- Human ]] {
  [[  CDAttribute $A :- ]]
}
```

ODRule

```
1  pattern objectdiagram lhs {
2    cDClass_1:ASTCDClass{
3      String name = "Person";
4    }
5    modifier_1:ASTModifier;
6    $A:ASTCDAttribute;
7
8    composition cDClass_1 -- (cDAttributes) $A ;
9    composition cDClass_1 -- (modifier) modifier_1 ;
10 }
11 replacement objectdiagram rhs {
12   cDClass_1:ASTCDClass{
13     String name = "Human";
14   }
15   modifier_1:ASTModifier;
16
17   composition cDClass_1 -- (modifier) modifier_1 ;
18 }
```

Abbildung 8.5: Gegenüberstellung von DSTL- und OD-Notation: Transformationsregel in CDTrans (links), die gleiche Regel in OD-Notation (rechts).

der Klasse zu entfernen. Auf der rechten Seite ist die gleiche Regel durch zwei Objektdiagramme ausgedrückt. Das erste Objektdiagramm – eingeleitet durch das Schlüsselwort `pattern` – beschreibt das Pattern der Transformationsregel (Zeile 1-10) während das zweite Objektdiagramm (Zeile 11 - 16) durch das Schlüsselwort `replacement` eingeleitet wird und den gleichen Modellausschnitt nach Anwendung der Transformation zeigt. Die Modifikation, die die Transformationsregel vornimmt, ergibt sich aus dem Unterschied zwischen dem Pattern-Objektdiagramm und dem Replacement-Objektdiagramm. Eine graphische Darstellung der beiden Objektdiagramme ist in Abbildung 8.6 gezeigt.

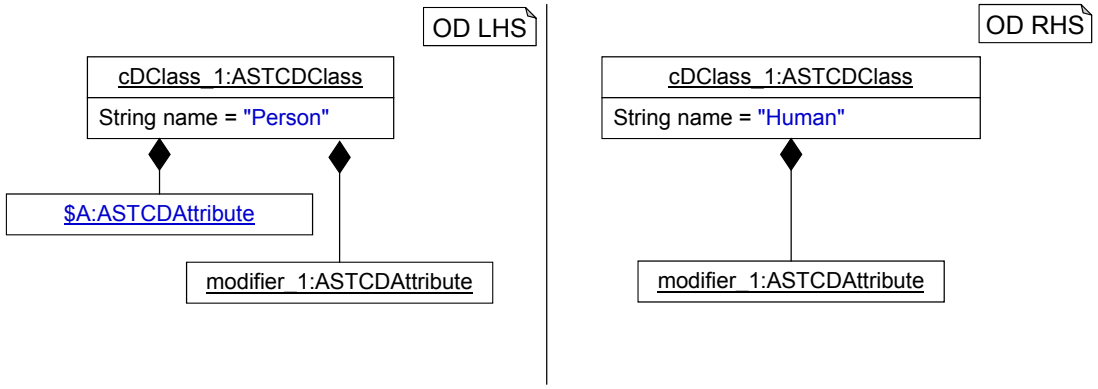


Abbildung 8.6: Graphische Darstellung der OD-Notation aus Abbildung 8.5.

Die Objektdiagramme beschreiben den AST des Modellausschnitts vor und nach der Transformation. Im Beispiel sind Objekte für die im Pattern beschriebene Klasse (Zeile

2-4), den Modifier (dieser bildet die default Sichtbarkeit der Klasse ab) und das Attribut. Der Typ dieser Objekte entspricht der AST-Klasse, dessen Instanz die Objekte sind. In diesem Beispiel also `ASTCDClass`, `ASTModifier` und `ASTCDAttribute`. Der Name eines Objekts entspricht dem Namen der Schemavariable, die in der Transformationsregel für das Element verwendet wird, falls eine Schemavariable verwendet wird. Im gezeigten Beispiel wurde für das Attribut die Schemavariable `$A` verwendet. Dadurch ist der Name des `ASTCDAttibute`-Objekts ebenfalls `$A`. Wird in der Transformationsregel für ein Patternelement keine Schemavariable verwendet, wird der Name automatisch generiert. In diesem Fall wird als Name der Typ des Objekts mit kleinem Anfangsbuchstaben verwendet. Da mehr als ein Objekt den gleichen Typ haben kann, wird ein Unterstrich sowie eine Zahl zur Unterscheidung angehängt. Das Attribut `name` der Klasse hat den Wert `"Person"` und sowohl der Modifier als auch das Attribut sind Teile der Klasse. Dies wird durch die beiden Kompositionen abgebildet. Der Rollenname an den Kompositionen dient hierbei der Identifikation, welche Komposition des ASTs abgebildet wird. Die Notation bietet neben Kompositionen durch Links in den Objektdiagrammen außerdem die Möglichkeit zur Beschreibung weiterer Beziehungen von AST-Knoten untereinander [Wei12]. Das Replacement-Objektdiagramm enthält ebenfalls die Klasse und den Modifier. Das Attribut hingegen fehlt im Replacement-Objektdiagramm. Ein weiterer Unterschied zwischen dem Pattern- und dem Replacement-Objektdiagramm ist der Wert des `name`-Attributes der Klasse. Im Pattern-Objektdiagramm hat dieses den Wert `"Person"` während im Replacement-Objektdiagramm das Attribut den Wert `"Human"` hat. Aus diesen Unterschieden ergibt sich als Modifikation der Transformation die Entfernung des Attributs sowie die Änderung des Namens der Klasse von `Person` zu `Human`.

Neben den „normalen“ Patternelementen kann das Pattern einer Transformationsregel durch die Features negative Elemente, Optional Operator und Collection Operator auch negative, optionale und mengenwertige Patternelemente enthalten. In der OD-Notation werden diese Elemente mit Stereotypen getaggt und bei der Generierung der Java-Implementierung entsprechend berücksichtigt. Hiervon ist insbesondere das Pattern Matching (vgl. Abschnitt 8.3) betroffen. Abbildung 8.7 verdeutlicht dies am Beispiel des Optional Operators. Erneut ist die DSTL-Notation (links) der OD-Notation (rechts) gegenübergestellt. Die Transformationsregel beschreibt eine Klasse, die ein Attribut haben kann. Ausgedrückt wird dies mittels des Optional Operators angewendet auf das Attribut. Eine Modifikation wird durch die Transformationsregel nicht vorgenommen.

Im rechten Teil von Abbildung 8.7 ist das entsprechende Pattern in OD-Notation abgebildet. Genau wie zuvor gibt es ein Objekt des Typs `ASTCDClass` und eines vom Typ `ASTModifier`, die die Klasse im Pattern repräsentieren. Da in diesem Fall kein Name der Klasse im Pattern angegeben ist, sondern eine (anonyme) Schemavariable, wird diesmal auch kein Wert für das `name`-Attribut des `CDClass`-Objekts gefordert. Auch ein Objekt vom Typ `ASTCDAttribute` ist erneut Teil des Patterns. Diesmal ist es ein inneres Objekt eines kapselnden hierarchischen Objekts, das den Stereotyp `optional` hat. Alle Objekte, die innerhalb eines solchen optionalen Objekts vorkommen, bilden einen optionalen Teil des Patterns. Da dieser Teil optional ist, muss dieser für einen gül-

CDTrans

```
class $_ {
  opt [[ CDAttribute $A ]]
}
```

ODRule

```
pattern objectdiagram lhs {
  cDClass_1:ASTCDClass;
  modifier_1:ASTModifier;
  <<optional>>  optional_1 {
    $A:ASTCDAttribute;
  }
  composition cDClass_1  --  (cDAttributes) $A ;
  composition cDClass_1  --  (modifier) modifier_1 ;
}
```

Abbildung 8.7: Mapping zwischen DSTL und OD-Notation: Transformationsregel in CDTrans (links) inklusive Optional Operator, die gleiche Regel in OD-Notation (rechts).

tigen Match des Pattern nicht gefunden werden. Der optionale Teil gilt jedoch nur dann als gefunden, wenn alle innerhalb eines optionalen Objekts enthaltenen Objekte mit all ihren angegebenen Kompositionen, Links und Attributen gefunden werden. Die Kapselung durch ein hierarchischen Objekt erlaubt es mehrere optionale Teile innerhalb eines Pattern zu definieren, die alle unabhängig voneinander gefunden oder nicht gefunden werden können. Negative Elemente und der Collection Operator werden entsprechend durch den Stereotyp `not` bzw. `list` markiert. Für weitere Details zur technischen Realisierung sei an dieser Stelle auf die betreuten Vorarbeiten [Wla16] für den Optional Operator und [Fra16] für den Collection Operator verwiesen.

MCG

```
ODRule =
  "pattern" lhs:ODDefinition
  ("replacement" rhs:ODDefinition)?
  ("folding" "{" FoldingSet* "}")?
  ("where" "{" constraint:Expression "}")?
  ("assign" "{" Assignment* "}")?
  ("do"  doBlock:BlockStatement )?;
```

Listing 8.8: Die Produktion `ODRule` als Startregel der ODRules-Sprache, die die OD-Notation definiert.

Listing 8.8 zeigt die Startregel der ODRules-Sprache. Eine Transformationsregel besteht mindestens aus einem Objektdiagramm, das das Pattern beschreibt. Optional kann außerdem ein Replacement-Objektdiagramm angegeben werden. Ist kein Replacement-Objektdiagramm gegeben, ist dies gleichbedeutend mit einem Replacement-Objektdiagramm, das identisch zum Pattern-Objektdiagramm ist. Anschließend folgen jeweils optional das Folding für nicht-isomorphe Matches und der Bedingungsblock für den Application Constraint, ein Zuweisungsblock für Zuweisungen von Schemavariablen und der Anweisungsblock für abschließende Aktionen wie Reporting bzw. Logging oder Modellanalysen. Das Folding und der Application Constraint verwenden die gleiche Syntax wie die DSTLs für das Folding bzw. den Application Constraint [Wei12]. Die OD-Notation wurde dahingehend erweitert, dass die OD-Notation ebenfalls Zuweisungen und einen Anweisungsblock erlaubt. Da beide Features bereits in der DSTL unabhängig von der

zugehörigen Modellierungssprache sind, wurde für die OD-Notation die gleiche Syntax wie für die DSTLs gewählt (vgl. Abschnitt 4.10 und 4.12).

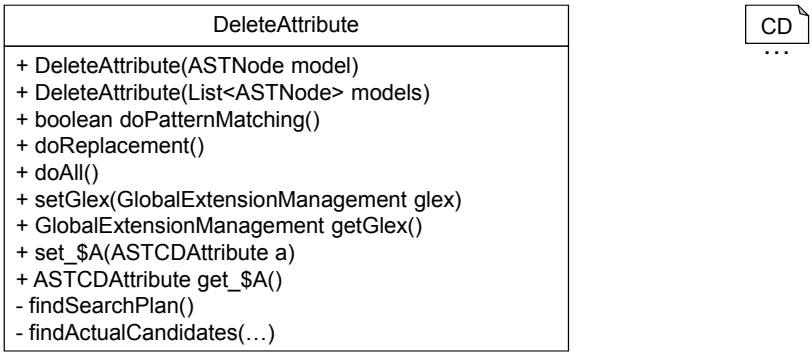


Abbildung 8.9: Methoden einer generierten Java-Implementierung für die Transformationsregel aus Abbildung 8.5.

Aus einer solchen Transformationsregel in Form einer ODRule wird schließlich eine Java-Implementierung generiert. Diese umfasst sowohl den Pattern Matching Algorithmus als auch die Modifikation. Abbildung 8.9 zeigt die Struktur einer generierten Java-Implementierung für eine ODRule am Beispiel der Transformationsregel aus Abbildung 8.5. Hierbei realisieren die Methoden `doPatternMatching()` und `doReplacement()` das Pattern Matching bzw. die Modifikation des Modells. Das Pattern Matching wird aus dem Pattern-Objektdiagramm generiert, während die Modifikation aus dem Unterschied zwischen dem Pattern- und dem Replacement-Objektdiagramm generiert wird. Die Methode `doAll()` führt zunächst das Pattern Matching und anschließend die Modifikation aus. Der Konstruktor erlaubt die Übergabe eines oder mehrerer Modelle. Außerdem bieten die Transformationen die Möglichkeit der Transformation eine Instanz der `GlobalExtensionManagement`s zu übergeben. Nutzt die Transformationsregel Templateerweiterung [Rot17] wird diese Instanz verwendet. Wird der Transformation keine Instanz übergeben, erstellt die Transformation diese selbst. Die get-Methode bietet dann die Möglichkeit diese Instanz von der Transformation zu erhalten, um sie in nachgelagerten Transformationen oder zur templatebasierten Codegenerierung zu nutzen. Die bisher beschriebenen Methoden werden für alle Transformationsregeln generiert. Die an Schemavariablen gebundenen Patternelemente können außerdem als Parameter der Transformation betrachtet werden. Hierfür bieten die generierten Java-Implementierungen für jedes Patternelement, das in der Transformationsregel mit einer Schemavariablen versehen wurde, Zugriffsmethoden an. Die Zugriffsmethoden für Schemavariablen sind spezifisch für jede Transformationsregel und werden abhängig von der Transformationsregel generiert. Durch diese Methoden werden die Schemavariablen einer Transformation als Eingabe- bzw. Ausgabeparameter genutzt. Wird vor Ausführung der Transformation ein Modellelement für ein Patternelement gesetzt, dann betrachtet die Transformation dieses als gegeben und matcht die restlichen Patternelemente passend zu den bereits festgesetzten Modellelement(en). Damit unterstützen die DSTLs zusätz-

lich zu dem automatischen Pattern Matching auch eine Form von Pivoting [GBA14], also die Festlegung eines Ausgangspunkts für einen Match. Modellelemente können zudem im Anschluss an die Transformation von dieser abgefragt werden. Hierdurch wird es möglich Modellelemente, die durch die Transformationsregel gefunden oder erzeugt werden, an eine nachgelagerte Transformationsregel weiterzureichen. Die Verwendung von generierten Java-Implementierungen ist in Kapitel 9 erläutert.

8.3 Pattern Matching

Eine Mustersuche in einem Graphen lässt sich auf das Problem der Teilgraphisomorphie zurückführen, welches NP-vollständig ist [Coo71, Wei12]. Eine naive Umsetzung führt leicht zu Transformationsimplementierungen, die eine schlechte Laufzeit aufweisen. Das liegt daran, dass bei einer naiven Implementierung beispielsweise jedes gesuchte Patternelement mit jedem im Modell vorhandenen Modellelement verglichen wird. Das ergibt bei n gesuchten Patternelementen in einem Modell bestehend aus m Modellelementen m^n viele Möglichkeiten. Listing 8.10 zeigt eine Transformationsregel, die einen Teil der Pull Up Attribute Funktionalität darstellt. Die Transformationsregel beschreibt ein Pattern, das aus zwei Klassen besteht, die ein gemeinsames Attribut haben. Zu Demonstrationszwecken wird hierbei als Name der einen Klasse `Person` gefordert, indem es direkt im Pattern angegeben wird. Für die andere Klasse wird als Name `Group` gefordert. In diesem Fall ist dies durch den Application Constraint angegeben. Die Transformationsregel umfasst 10 Patternelemente: 2 Klassen, 2 Attribute, 2 `SimpleReferenceType`s und 4 `Modifier`. Das Klassendiagramm in Listing 8.11 dient der Demonstration des Pattern Matching Algorithmus und besteht aus 3 Klassen: `Profile`, `Person` und `Group`. `Person`, `Profile` und `Group` haben ein gemeinsames Attribut `String profileName`. Außerdem hat `Person` ein Attribute `String firstname`, das in keiner der anderen Klassen vorkommt. Das Klassendiagramm besteht aus 22 Modellelementen: 3 Klassen, 4 Attributen, 6 `SimpleReferenceType`s, 1 `CDDefinition`, 1 `CDCompilationUnit` und 7 `Modifiern`. Verwendet man die Transformationsregel aus Listing 8.10 und das Modell aus Listing 8.11, dann ergeben sich 10 gesuchte Elemente in 22 Modellelementen, also $22^{10} = 26.559.922.791.424$ Möglichkeiten. Werden für jedes gesuchte Patternelement nur Modellelemente des passenden Typs betrachtet, ergeben sich $\#CDClass^2 \cdot \#CDAttribute^2 \cdot \#SimpleReferenceType^2 \cdot \#Modifier^4 = 3^2 \cdot 3^2 \cdot 6^2 \cdot 7^4 =$ 142.884 Möglichkeiten. Dadurch reduziert sich die Anzahl der Möglichkeiten zwar bereits deutlich, die Suche nach einem gültigen Match kann jedoch weiter verbessert werden. Dazu existieren verschiedene Pattern Matching Strategien [GBA14, KC15].

In [Wei12] wurde für die DSTLs in MontiCore ein modellsensitiver, suchplangesteuerter Pattern Matching Algorithmus [BKG08, GBA14] ausgewählt und umgesetzt. Das bedeutet, dass für das Pattern Matching ein Suchplan berechnet wird, der es ermöglicht, gegenüber der naiven Implementierung deutlich effizienter einen Match zu finden oder zu dem Ergebnis zu kommen, dass kein Match im Modell vorhanden ist. Modellsensitiv bedeutet, dass neben den Informationen aus dem Pattern auch der Aufbau des gegebenen Modells die Berechnung des Suchplans beeinflusst, wodurch das Pattern Matching für das

```
CDTrans
$C1 [[ class Person {
         $A1 [[ $type $name;]]
       } ]]

$C2 [[ class $_ {
         $A2 [[ $type $name;]]
       }]]

where {
  $C2.getName().equals("Group")
}
```

Listing 8.10: Transformationsregel in CDTrans zur Demonstration des Pattern Machting Algorithmus.

```
CD
classdiagram SocNet {
  class Profile {
    String profileName;
  }
  class Person {
    String profileName;
    String firstName;
  }
  class Group  {
    String profileName;
  }
}
```

Listing 8.11: Klassendiagramm, um den Pattern Machting Algorithmus zu demonstrieren.

aktuelle Modell optimiert wird. Der suchplangesteuerte Ansatz wurde anderen Ansätzen wie beispielsweise einer Abbildung auf das Constraint Satisfaction Problem (CSP) und der Verwendung von Constraint Solvern [Rud00, LV02] oder die Abbildung auf ein relationales Modell [Vos00], um die Optimierungen im Bereich der Datenbankabfrage [ZCÖ09, CYD+08, GBG+06] auszunutzen, vorgezogen. Entscheidend beeinflusst hat die Entscheidung hierbei das schlechtere Abschneiden dieser Ansätze in vergleichenden Benchmarks [TBB+08]. Auf eine Abbildung auf das CSP setzen beispielsweise die Transformationstools AGG [Run17, Tae04] und VIATRA [VIA17, BDH+15, VB07].

In [Wei12] wurde der für die Transformationsregeln gewählte und umgesetzte Pattern Matching Algorithmus beschrieben. Im folgenden wird der Ansatz daher lediglich kurz zusammengefasst, um anschließend auf die Erweiterungen des Ansatzes im Rahmen dieser Arbeit eingehen zu können. Durch die in Kapitel 4 beschriebenen Veränderungen an den vorhandenen Operatoren sowie die Einführung neuer Operatoren, musste auch das Pattern Matching entsprechend angepasst werden. Zusätzlich hat der Einsatz der DSTLs in realistischen Szenarien wie beispielsweise dem Data Explorer gezeigt, dass eine zentralisierte Application Constraint Auswertung zu Laufzeitproblemen führt. Im Folgenden wird zunächst der allgemeine bereits in [Wei12] erläuterte Algorithmus kurz vorgestellt. Anschließend wird auf die notwendigen Änderungen für die einzelnen Operatoren detaillierter eingegangen.

8.3.1 Suchplangesteuertes Pattern Matching

Die Idee der suchplangesteuerten Pattern Matching Strategien ist es, vor dem eigentlichen Matching einen Suchplan zu berechnen, der eine effiziente Suche nach dem gegebenen Muster im gegebenen Modell erlaubt [GBA14, BKG08]. Neben MontiTrans setzen auch PROGRES [Zün96a, Zün96b, SWZ99, SWZ95], FUJABA [SZG06, FNTZ00, NNZ00] und GrGen.NET [GrG17, JBK10] auf diese Art des Pattern Matchings.

Der Suchplan legt die Reihenfolge fest, in der die Elemente des Pattern gesucht werden. Er lässt sich über einen Stack von der Reihe nach zu besetzenden Patternelementen abbilden. Für das Beispiel aus Listing 8.10 wäre

$$[\$C1, \$C2, \$A1, \$A2, sRType\_1, sRType\_2, modifier\_1, .., modifier\_4]$$

ein möglicher Suchplan[1]. Hierbei ist die Spitze des Stacks das linke Ende der Liste. Der Algorithmus versucht in der Reihenfolge, die der Suchplan vorgibt, Modellelemente als Match für die Patternelemente zu finden. Sind Matches für die Patternelemente gefunden, wird der Application Constraint ausgewertet und überprüft, ob es sich um einen gültigen Match handelt. Ist der Application Constraint erfüllt, dann wird dieser Match gespeichert und die Transformation kann angewendet werden. Verwirft der Application Constraint den Match, dann wird versucht einen weiteren Match zu finden, der den Application Constraint erfüllt. Wird an einer Stelle für das nächste Patternelement keine passendes Modellelement gefunden oder, falls das letzte Patternelement zugeordnet ist, verwirft der Application Constraint den Match. In diesem Fall wechselt die Transformation in das Backtracking und sucht ein anderes Modellelement für das zuletzt zugeordnete Patternelement. Das Pattern Matching scheitert, wenn für das erste Patternelement des Suchplans alle Kandidaten ausprobiert wurden und keines davon zu einem gültigen Match geführt hat.

Abbildung 8.12 zeigt einen Ausschnitt aus dem Suchbaum für das angegebene Pattern bzw. die ersten vier Patternelemente des exemplarischen Suchplans. Die fettgedruckten Pfeile stellen hierbei das Match dar. Dünne Pfeile zeigen ausprobierte Pfade, die jedoch zu keinem gültigen Match geführt haben. Die angedeuteten Linien bedeuten einen weiteren nicht dargestellten Abstieg in diesen Teilbaum. Da es das gleiche Attribute in den Klassen `Person`, `Profile` und `Group` gibt, sind die entsprechenden Attribute in der Darstellung mit einer entsprechenden Markierung versehen. Die grau eingefärbten Modellelemente sind Modellelemente, die nach dem naiven zuvor beschrieben Ansatz ebenfalls betrachtet würden, in der optimierten Variante aus [Wei12] jedoch nicht betrachtet werden müssen, da sie bereits frühzeitig als Kandidaten für das entsprechende Patternelement ausgeschlossen werden. Das Pattern Matching startet mit der Suche nach einem Match für `$C1`. Hierfür ergibt sich nur 1 Kandidat, da es nur eine Klasse gibt, deren Name `Person` ist. `Profile` und `Group` werden sofort verworfen. Als nächstes sucht der Algorithmus einen Match für `$C2`. Es gibt zwei Kandidaten für `$C2`, die Klasse `Profile` und die Klasse `Group`. `Person` ist hier kein Kandidat, da `Person` bereits für `$C1` gemacht wurde und die Transformation nicht erlaubt, dass das gleiche Modellelement für beide Patternelemente verwendet wird. Soll dies erlaubt werden, kann der Folding Operator verwendet werden (vgl. Abschnitt 4.9). Der Suchbaum unterhalb von `Profile` wird ausprobiert, allerdings findet sich auf diesem Weg kein gültiger Match, da der Application Constraint nicht erfüllt ist. Ähnlich wie bei der Suche nach einem Kandidaten für `$C1`, könnte dieser Teilbaum deutlich früher verworfen werden. Dies

[1] Zur besseren Lesbarkeit wurde $simpleReferenceType$ durch $sRType$ abgekürzt.

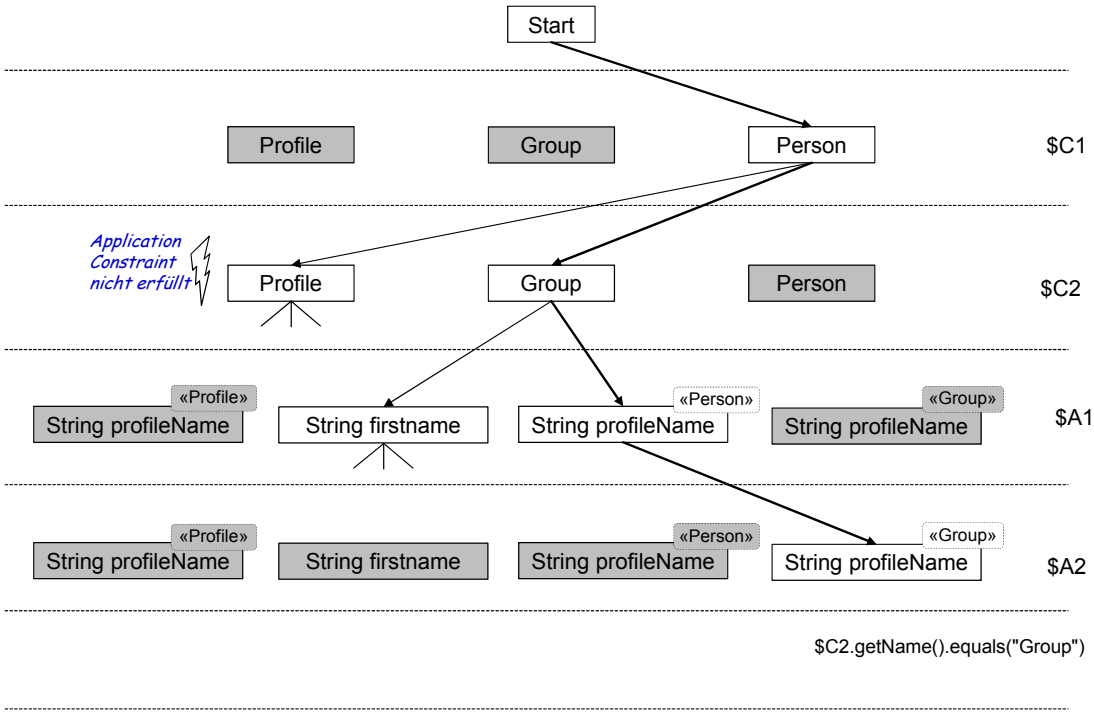


Abbildung 8.12: Darstellung des Suchraums und des Verlaufs des Pattern Matching für die Transformationsregel aus Listing 8.10 angelehnt an die Darstellung in [Wei12].

wird im späteren Verlauf dieses Kapitels (vgl. Abschnitt 8.3.2) adressiert. Aus diesem Grund wird für `$C2` die Klasse `Group` zugeordnet. Als nächstes wird ein Match für `$A1` gesucht. `$A1` muss ein Attribut der Klasse `Person` sein. Hier gibt es zwei Kandidaten `String firstName` und `String profileName`. Der Suchbaum unterhalb von `String firstName` wird ausprobiert, führt allerdings zu keinem gültigen Match, da es für `String firstName` kein Pendant in der Klasse `Group` gibt. Als Match wird daher `String profileName` gefunden. Als nächstes wird ein Match für `$A2` gesucht. `$A2` muss ein Attribut der Klasse `Group` sein. Hier gibt es nur einen Kandidaten. Dieser wird als Match angenommen. Anschließend wird der Application Constraint überprüft. Dieser ist erfüllt und das Pattern Matching hat einen gültigen Match gefunden.

In [Wei12] werden zwei Arten der Optimierung des Pattern Matching Algorithmus beschrieben. Zum einen kann basierend auf der Transformation optimiert werden. Zum anderen kann basierend auf dem Eingabemodell optimiert werden. Im Sinne der ersten Kategorie werden beispielsweise Attributwerte der Kandidaten wie beispielsweise der Name der Klasse direkt bei der Suche nach einem Kandidaten überprüft und nicht erst nach Zuordnung aller Kandidaten. Außerdem werden die Kandidaten nach jedem zugeordneten Modellelement aktualisiert. So werden in Abbildung 8.12 für `$A1` als Kandidaten nur noch Attribute der Klasse `Person` betrachtet. Die zweite Kategorie wirkt sich auf

die Berechnung des Suchplans aus. Um den Suchplan zu optimieren, gibt es verschiedene Strategien [GBA14]. So können beispielsweise Heuristiken über typische Modelle oder Metamodellinformationen herangezogen werden. Für MontiCore wurde ein modellsensitiver Ansatz gewählt. Dadurch wird der Suchplan zu Beginn einer jeden Transformation dynamisch anhand des Eingabemodells berechnet und ist für jedes Eingabemodell individuell optimiert. Der Suchplan berechnet sich anhand der Kosten für das Pattern Matching der einzelnen Patternelemente. Dem zugrunde liegt die Formel

$$c(S) = \sum_{k=1}^{n} \prod_{i=1}^{k} c_i \tag{8.1}$$

wobei n die Anzahl der Patternelemente ist und c_i die Kosten eines zu matchenden Patternelements darstellt. Die Formel summiert also für jedes Suchplanelement die Kosten des Produkts aller bisher gematchten Patternelemente auf. Je früher ein Patternelement gematcht wird, desto größer ist der Einfluss auf die Gesamtkosten des Suchplans. Die Kosten eines zu matchenden Patternelements berechnen sich anhand der möglichen Kandidaten für das Patternelement, also dem Kantengrad der Elemente im Suchraum (vgl. Abbildung 8.12) [Wei12]. Je weniger Kandidaten für ein Patternelement existieren desto geringer sind die Kosten und desto früher sollte ein Match für dieses Patternelement gesucht werden. Entsprechend dieser Formel ergibt sich der oben gezeigte Suchplan. Für `$C1` gibt es 1 Kandidaten, für `$C2` 2, für `$A1` und `$A2` jeweils 3, für die `SimpleReferenceTyp`s jeweils 6 und für die `Modifier` sogar jeweils 7. Für den Suchplan werden zunächst die günstigen Patternelemente gesucht. Nach jedem Schritt werden zudem die Kandidaten für die restlichen Patternelemente aktualisiert, wodurch die Kosten für nachfolgende Patternelemente weiter reduziert werden können. So sinkt beispielsweise die Anzahl der Kandidaten für den `SimpleReferenceTyp`s des Attributs `$A1` nachdem ein Match für `$A1` gefunden wurde auf 1.

8.3.2 Modularisierung und Überprüfung des Application Constraints

Der Application Constraint wird in dem in [Wei12] entwickelten Ansatz zentralisiert nach Zuordnung aller Patternelemente überprüft. Das Beispiel in Abbildung 8.12 verdeutlicht, dass die zentralisierte Überprüfung einen starken Einfluss auf die Laufzeit des Pattern Matching haben kann. Im diesem Beispiel wird der komplette Suchbaum unterhalb der Klasse Profile gematcht, bevor entschieden werden kann, dass bereits die Klasse `Profile` kein gültiger Match für das Patternelement `$C2` ist. Aus diesem Grund wurde die Überprüfung des Application Constraints in das Pattern Matching integriert, indem der Application Constraint zunächst modularisiert und schließlich dezentral überprüft wird. Das Konzept der Modularisierung und dezentralen Constraint-Überprüfung wird im Folgenden erläutert. Die technische Realisierung erfolgte in der betreuten Vorarbeit [Wil17]. Die Details der Realisierung können dort nachgelesen werden.

Listing 8.13 zeigt eine Abwandlung der Transformationsregel aus Listing 8.10. Erneut werden in dem Klassendiagramm aus Listing 8.11 die beiden Klassen `Person` und `Group`

```
$C1 [[ class $_ {                                                    CDTrans
  $A1 [[ $type $name;]]
} ]]

$C2 [[ class $_ {
  $A2 [[ $type $name;]]
}]]

where {
  $C1.getName().equals("Person") && $C2.getName().equals("Group")
}
```

Listing 8.13: Transformationsregel mit Application Constraint.

sowie deren gemeinsame Attribute `String profileName` durch die Transformation gematcht. Diesmal wurde jedoch auch der Name der Klasse `Person` in den Application Constraint verlagert. Dadurch verschlechtert sich die Laufzeit der Transformation noch weiter. Für `$C1` werden nun alle drei Klassen als Kandidaten betrachtet und zwei weitere Teile des Suchbaum unnötigerweise durchlaufen. Diese beiden simplen Beispiele verdeutlichen, dass der Constraint ebenfalls frühest möglich ausgewertet werden sollte, um ein spätes und damit teures Verwerfen von gematchten Modellelementen zu vermeiden.

Eine mögliche Maßnahme zur Verbesserung ist, den Constraint zu überprüfen, sobald alle von dem Constraint betroffenen Patternelemente gematcht sind. In beiden betrachteten Fällen würde dies zu einer verbesserten Laufzeit führen. Im ersten Fall (Transformationsregel aus Listing 8.10) würde der Constraint bereits beim Zuordnen von `$C2` überprüft. Im zweiten Fall (Transformationsregel aus Listing 8.13) würde der Constraint ebenfalls beim Matchen von `$C2` überprüft. In diesem Fall allerdings werden für `$C1` bereits nicht gültige Matches gefunden und diese erst bei der Zuordnung von `$C2` verworfen. Es würden also die Paare (`Profile`, `Group`), (`Profile`, `Person`), (`Group`, `Profile`), (`Group`, `Person`) und (`Person`, `Profile`) gefunden, bevor das Pattern Matching schließlich mit `Person` und `Group` fortgesetzt werden kann. Betrachtet man den Constraint genauer, fällt auf, dass er zwar beide Patternelemente `$C1` und `$C2` betrifft, die Teilbedingungen aber unabhängig voneinander sind. Aus diesem Grund wurde für MontiTrans eine Modularisierung des Constraints in Subconstraints entwickelt.

Der Application Constraint ist ein Ausdruck, der zu einem booleschen Wert auswertet und angibt, ob die Transformation für den gegebenen Match anwendbar ist. Ein solcher Application Constraint besteht in der Regel aus einer Menge von Teilausdrücken, die miteinander über Konjunktionen oder Disjunktionen verknüpft sind. Eine Konjunktion der Teilausdrücke ist genau dann erfüllt, wenn alle ihre Teilausdrücke zu wahr auswerten. Diese Eingenschaft der Konjunktion macht sich die Modularisierung von Constraints zu Nutzen. Ein Application Constraint kann in seine konjugierten Teilausdrücke (Subconstraints) zerlegt werden, diese können anschließend separat überprüft werden. Der Constraint im obigen Beispiel wird in die Subconstraints `$C1.getName().equals("Person")` und `$C2.getName().equals("Group")`

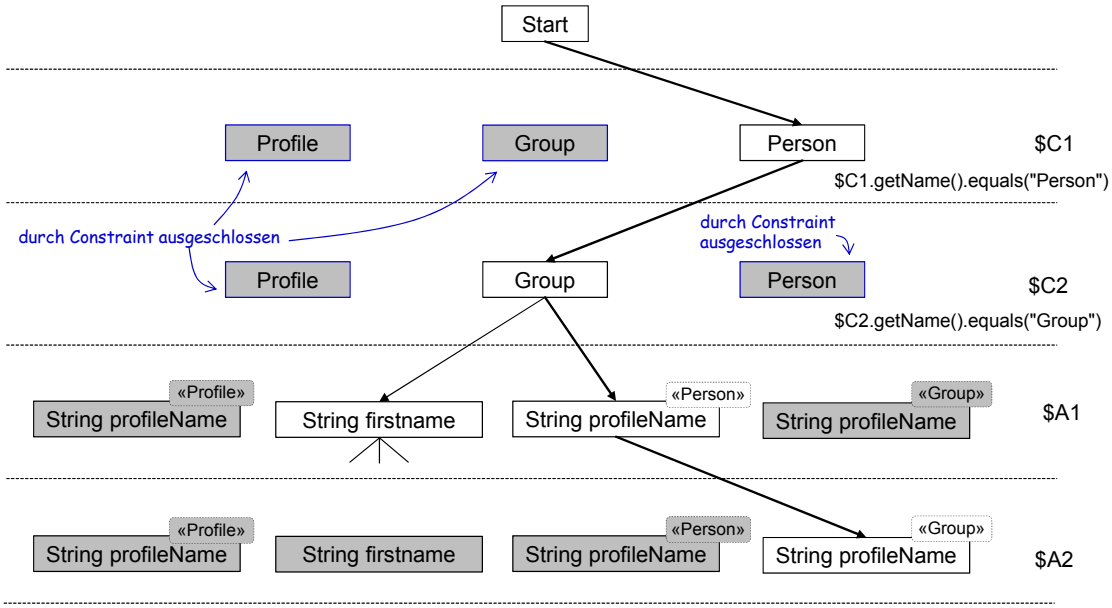


Abbildung 8.14: Pattern Matching mit dezentraler Constraintauswertung für die Transformationsregel aus Abbildung 8.5.

zerlegt. Die Subconstraints können separat geprüft werden. Eine Modularisierung alleine würde noch keine Verbesserung der Laufzeit mit sich bringen, solange die Subconstraint-Überprüfung weiterhin zentralisiert nach Matchen aller Patternelemente stattfindet. Daher wurde die zentralisierte durch eine dezentralisierte Überprüfung ersetzt.

Die dezentralisierte Überprüfung der Constraint verlagert die Überprüfung der Subconstraints in das Pattern Matching. Dazu werden die Subconstraints jeweils auf die von ihnen betroffenen Patternelemente analysiert. Sobald alle Patternelemente eines Subconstraints gematcht sind, kann dieser überprüft werden und, falls er nicht erfüllt ist, das weitere Matching an dieser Stelle verhindert werden. Dadurch wird bereits frühzeitig erkannt, dass in diesem Teil des Suchbaum kein gültiger Match gefunden werden kann. Zusätzlich beeinflusst die Modularisierung und dezentralisierte Überprüfung auch die Berechnung des Suchplans. Betrifft ein Subconstraint nur ein Patternelement, wird dieser wie eine Attributeigenschaft – wie beispielsweise der Name einer Klasse – behandelt und reduziert bereits die maximal mögliche Anzahl der Kandidaten für dieses Patternelement. Zusätzlich ist es für das Pattern Matching von Vorteil Patternelemente, die in vielen Subconstraint vorkommen, frühzeitig zu matchen, da so Teile des Suchbaum frühzeitig verworfen werden können. Aus diesem Grund reduziert ein Vorkommen in vielen Subconstraints die Kosten eines Patternelements, sodass es bei der Berechnung des Suchplans bevorzugt wird.

Abbildung 8.14 zeigt den Verlauf des Pattern Matching für die Transformationsregel aus Listing 8.13. Der Suchplan bleibt hierbei der gleiche, wie in Abbildung 8.12, da die beiden von den Subconstraints betroffenen Patternelemente bereits die ersten beiden

zu matchenden Patternelemente darstellen. Für `$C1` existiert durch den Subconstraint nur 1 Kandidat, die Klasse `Person`. Für `$C2` existiert – bedingt durch den zweiten Subconstraint – als Kandidat nur noch die Klasse `Group`. Damit werden die 5 Klassenkombinationen, wie es im Fall der zentralisierten Auswertung der Fall war, nicht länger betrachtet. Die gültige (`Person`, `Group`)-Kombination wird direkt gefunden. Das restliche Matching ist nicht von dem Application Constraint betroffen. Das bedeutet, die restlichen Patternelemente werden von keinem Subconstraint eingeschränkt.

Bislang wurde davon ausgegangen, dass der Constraint eine Konjunktion darstellt. Ein Constraint kann jedoch auch Disjunktionen enthalten. Disjunktionen sind wahr, wenn mindestens einer der Terme der Disjunktion wahr ist. Damit lässt sich eine Disjunktion im Gegensatz zu einer Konjunktion nicht modularisieren. Ein Constraint der Form `(a || b)` könnte daher nicht in weitere Subconstraints zerlegt werden. Die Modularisierung greift hier somit nicht und in diesem Fall findet lediglich eine dezentrale Überprüfung statt, sobald alle von dem Constraint betroffenen Patternelemente gematcht wurden. Eine Klammerung von Konjunktionen sowie eine Konjunktion von Disjunktionen kann jedoch modularisiert werden. Das heißt, ein Constraint der Form `a &&((b || c) && d)` wird in 3 Subconstraints zerlegt: `a`, `(b || c)` und `d`. Das Pattern Matching profitiert daher von einem geschickt formulierten Application Constraint, der zum Beispiel in konjunktiver Normalform (KNF) vorliegt. Diese Optimierung des Constraint obliegt derzeit dem Transformationsentwickler, es wäre jedoch denkbar MontiTrans so zu erweitern, dass eine Überführung in KNF automatisiert geschieht.

In [Wil17] wurde zudem eine Performanceanalyse durchgeführt. Hierbei wurde die Laufzeit des Pattern Matching in verschiedenen Szenarien gegenübergestellt. Es wurde insbesondere die Laufzeit der Pattern Matching mit modularisiertem Application Constraint und dezentraler Überprüfung der nicht-modularisierten, zentralisierten Variante gegenübergestellt. Insgesamt wurde die Laufzeit von 32 Transformationen gemessen. Hierbei wurden 16 Transformationen, die einmal mit und einmal ohne Modularisierung ausgeführt wurden, betrachtet. Die 16 Transformationen teilen sich in 4 verschiedene Fälle, die jeweils in 4 Patterngrößen betrachtet wurden.

Die folgenden Fälle wurden betrachtet:

Fall 1 kein Match im Modell vorhanden, der Constraint ist modularisierbar

Fall 2 kein Match im Modell vorhanden, der Constraint ist nicht-modularisierbar

Fall 3 Match im Modell vorhanden, der Constraint ist modularisierbar

Fall 4 Match im Modell vorhanden, der Constraint ist nicht-modularisierbar

Für die Patterngröße wurden 2, 3, 4, bzw. 5 Klassen gewählt, wobei über den Constraint jeweils die 2.-x. Klasse von der ersten abhängt. Listing 8.15 zeigt ein Beispiel mit 2 Klassen, bei denen der Name der einen Klasse der Name der anderen Klasse plus das Suffix `Impl` ist. Um die Modularisierbarkeit auszuhebeln, wurde der Constraint in eine Disjunktion umgeformt, indem der Constraint mit sich selbst Oder-verknüpft wurde. Gesucht wurden die Klassen jeweils in einem Klassendiagramm, das aus 50 Klassen besteht,

innerhalb derer es genau ein bzw. keinen Match gibt. Tabelle 8.16 zeigt die Ergebnisse der Laufzeitmessung. Jede Transformation wurde 10 mal durchgeführt, die Laufzeit gemessen und die durchschnittliche Laufzeit des Pattern Matching berechnet. Der Effekt des *lazy class loading* [LB98] wurde ausgehebelt, indem zunächst vorweg eine weitere Ausführung der Transformation stattfand, die nicht in die Berechnung mit eingeflossen ist. Alle Messungen wurden auf demselben Testsystem durchgeführt. Das Testsystem verfügt über eine *Intel 4770k* CPU mit 4 × 3.5 GHz, 16 GB Arbeitsspeicher und eine *Samsung EVO 850* SSD mit einer Kapazität von 250 GB. Windows 10 wurde als Betriebssystem und IntelliJ [Jet17] als Entwicklungsumgebung verwendet. Die Messung wurde mithilfe der `Stopwatch` der Google Guava Bibliothek [Bej13] vorgenommen. Es wurde die Laufzeit der `doPatternMatching`-Methode gemessen. Zum Zeitpunkt der Messung wurden auf dem System nur das Betriebssystem, die Java-Anwendung und die Entwicklungsumgebung ausgeführt. Weitere Details zur Messung der Laufzeiten können in [Wil17] nachgelesen werden.

CDTrans

```
1 class $className;
2 class $classNameImpl;
3
4 where{
5   $classNameImpl.equals($className+"Impl")
6 }
```

Listing 8.15: Transformationsregel mit einem Pattern aus zwei Klassen, die über den Application Constraint verknüpft sind.

Die Ergebnisse der Analyse bestätigen, dass durch die Modularisierung eine deutlich verbesserte Laufzeit erreicht wurde. Im teuersten, betrachteten Fall, – es werden 5 Klassen gesucht und es ist kein Match vorhanden – wurde für einen gut modularisierbaren Constraint ein Verbesserung um den Faktor 245.000 erreicht. Für einen nicht-modularisierbaren Constraint wurde immerhin eine Verbesserung um den Faktor 100 erreicht. Das verdeutlicht, dass selbst die dezentrale Auswertung des Constraints bereits einen merklichen Einfluss auf die Laufzeit hat. Zusätzlich wird jedoch auch deutlich, dass die Formulierung des Constraints eine wichtige Rolle spielt. Weiter hat die Analyse gezeigt, dass sowohl mit als auch ohne Modularisierung die Laufzeit mit zunehmenden Patternelementen exponentiell wächst.

8.3.3 Pattern Matching für den Optional und den Collection Operator

In Kapitel 4 wurden der Collection Operator und der Optional Operator für die DSTLs beschrieben. Es wurde erläutert, dass der Collection Operator im Vergleich zu [Wei12] so erweitert wurde, dass er auf Teilpattern anwendbar ist. Ein Teilpattern besteht aus einem Wurzelpatternelement und beliebig vielen Patternelementen, die dieses Patternelement als Wurzel haben. Ein Beispiel hierfür ist eine Klasse inklusive ihrer Attribute. Damit ist der Collection Operator für Teilbäume des ASTs und nicht nur Patternele-

Tabelle 8.16: Ergebnisse der Performanceanalyse aus [Wil17] (Laufzeiten in ms).

Variante	Match?	modul.-bar?	2 Kl.	3 Kl.	4 Kl.	5 Kl.
modular	✗	✓	1,0	1,0	2,3	2,3
modular	✗	✗	1,0	10,6	141,9	5.066,2
modular	✓	✓	1,9	1,8	1,8	2,1
modular	✓	✗	2,3	5,5	8,7	4.514,0
zentral	✗	✓	5,8	168,0	10.366,7	564.064,1
zentral	✗	✗	10,8	175,8	10.734,2	551.944,7
zentral	✓	✓	1,2	26,3	203,6	441.204,5
zentral	✓	✗	3,3	25,6	204,6	443.990,9

mente anwendbar. Für den Optional Operator und für die Erweiterung des Collection Operators musste auch das Pattern Matching erweitert werden, sodass die Pattern Matching Strategie die Semantik des Operators umsetzt. Beide Fälle haben gemeinsam, dass der Patternteil innerhalb des Operators einer anderen Pattern Matching Strategie bedarf. Die Strategie der beiden Operatoren ist dennoch sehr unterschiedlich. Im Folgenden wird das Pattern Matching sowohl für den Optional Operator als auch für den Collection Operator konzeptionell erläutert. Die technische Realisierung für den Collection Operator erfolgte in der betreuten Vorarbeit [Fra16] und für den Optional Operator in [Wla16]. Die Details der technischen Realisierung können dort nachgelesen werden.

Der Optional Operator wird verwendet, um einen Teil des Pattern als optional zu markieren. Dies erlaubt die Zusammenfassung von Transformationen, die beide Fälle (mit und ohne den optionalen Teil) abdecken (vgl. Abschnitt 4.8). Für die Anwendbarkeit der Transformationsregel gilt, dass, falls es keinen gültigen Match für den optionale Teil gibt, die Transformation dennoch anwendbar ist. Um einen optionalen Teil des Pattern zu matchen, muss ein gültiger Match für alle Patternelemente des optionalen Patternteil gefunden werden. Anders als das Pattern Matching „normaler“, verpflichtender Patternelemente, darf die Transformation nicht abbrechen, falls kein Match gefunden wird. Abbildung 8.17 illustriert den allgemeinen Ablauf des Pattern Matchings. Nach Erstellung des Suchplans werden zunächst Matches für die verpflichtenden Patternelemente gesucht. Können nicht für alle Patternelemente entsprechende Matches gefunden werden, bricht die Transformation ab und meldet zurück, dass die Transformation nicht anwendbar ist. Wurden Matches für die verpflichtenden Patternelemente gefunden, wird versucht Matches für die optionalen Patternelemente zu finden. Hierbei wird nicht abgebrochen, falls es keinen Match gibt. Anschließend werden Matches für Patternelemente gesucht, auf die der Collection Operator angewendet wird. Auf das Pattern Matching hierfür wird im späteren Verlauf dieses Abschnitts eingegangen. Kann kein Match gefunden werden, wird die Transformation abgebrochen und der Fehlschlag zurückgemeldet. War das Matching für die Patternelemente des Collection Operators erfolgreich, wird

versucht Matches für die negativen Patternelemente zu finden. Anders als bei den anderen Patternelementen darf für den negativen Teil des Pattern kein Match gefunden werden. Wird ein Match gefunden, bricht die Transformation ab und melden den Fehlschlag. Andernfalls ist das Pattern Matching mit Erfolg beendet und die Transformation geht zur Modifikation des Modells über.

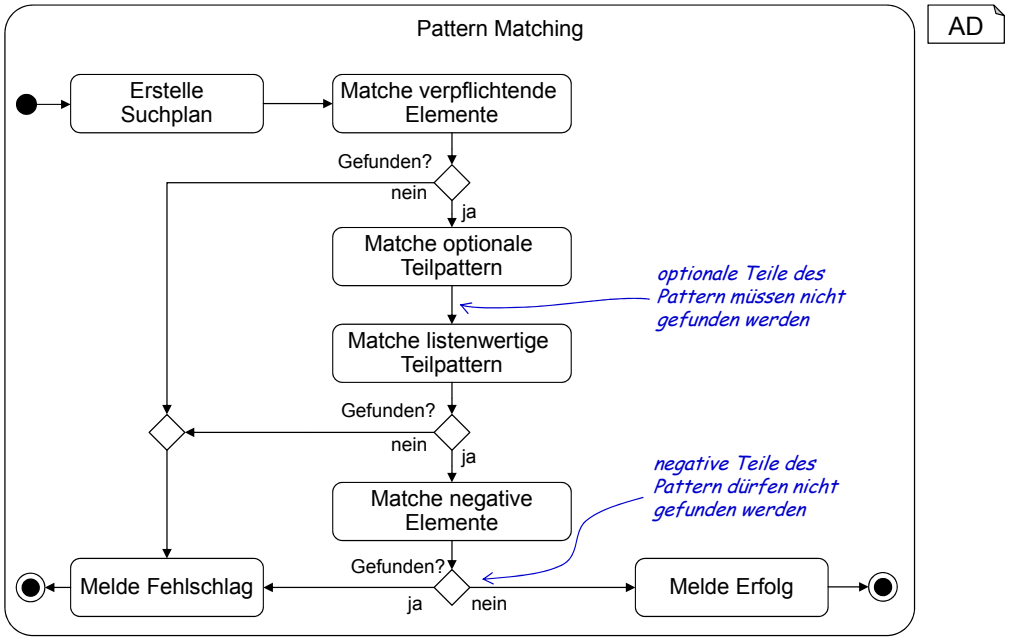


Abbildung 8.17: Übersicht über den Ablauf des Pattern Matchings.

Der Optional Operator kann geschachtelt werden, sodass es innerhalb des Matches für einen optionalen Patternteil erneut einen optionalen Patternteil geben kann. In diesem Fall kann der äußere optionale Teil auch ohne den inneren optionalen Teil einen gültigen Match haben. Des Weiteren kann ein Pattern mehrere Optional Operatoren beinhalten, das heißt, es kann mehrere optionale Patternteile geben, die unabhängig voneinander gefunden oder nicht gefunden werden können. Für das Pattern Matching bedeutet dies, dass jeder optionale Teil eines Pattern – auch falls es sich um einen optionalen Teil innerhalb eines optionalen Teils handelt – eines eigenen Pattern Matchings bedarf. Um dies zu ermöglichen wird für jeden optionalen Patternteil ein eigenes Teilpattern Matching generiert. Abbildung 8.18 zeigt das Mapping zwischen einer Transformationsregel bestehend aus einer Klasse mit einem optionalen Attribut und der dazu generierten Java-Klasse.

Abbildung 8.18 zeigt, dass es durch den Optional Operator zwei neue Methoden gibt: `doPatternMatching_optional_1` und `splitSearchplan`. Die Methode `doPatternMatching_optional_1` realisiert das Teilpattern Matching für den optionalen Patternteil. Ähnlich wie in PROGRES [SWZ95] werden verpflichtende Patternelemente, also Patternelemente, die weder innerhalb des Optional noch innerhalb des Collection Operators vorkommen, gegenüber optionalen Patternelementen priorisiert. Für

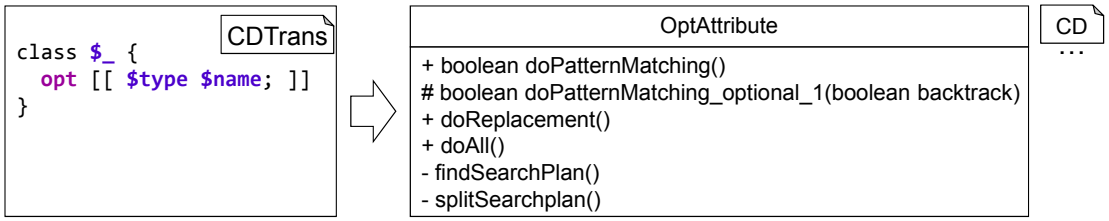


Abbildung 8.18: Mapping zwischen Transformationsregel mit Optional Operator und generierter Java Klasse.

die Suchplanberechnung heißt dies, dass zunächst die verpflichtenden und erst danach die optionalen Patternelemente gematcht werden. Die Methode `spiltSearchplan` sorgt anschließend dafür, dass es eigene Teilsuchpläne für die optionalen Patternteile gibt, die von der jeweils zugehörigen `doPatternMatching_optional_`-Methode verwendet werden. Das Pattern Matching der `doPatternMatching_optional_`-Methode verläuft analog zum Matching des normalen, verpflichtenden Pattern. Es unterschiedet sich lediglich in der Tatsache, dass die Methode auch dann Erfolg zurückmeldet, wenn kein gültiger Match gefunden wurde. Falls ein gültiger Match für einen optionalen Teil gefunden wird, wird dieser dem gesamten Match hinzugefügt. Die Behandlung des Collection Operators erfolgt analog. Für jeden Collection Operator wird ebenso ein eigenes Teilpattern Matching generiert. Im Gegensatz zum Teilpattern Matching für optionale Patternteile, ist dieses Teilpattern Matching jedoch nur dann erfolgreich, wenn mindesten 1 gültiger Match gefunden wird. Zusätzlich arbeitet dieses Teilpattern Matching „greedy“. Das bedeutet, es werden solange weitere Matches für dieses Teilpattern gesucht, bis kein weiteres gefunden werden kann.

8.4 Diskussion und verwandte Arbeiten

In diesem Kapitel wurde die Generierung von Java-Implementierungen aus Transformationsregeln vorgestellt. Im Vergleich zur Umsetzung des Generators, der in [Wei12] vorgestellt wurde, wurde die Generatorarchitektur dahingehend verändert, dass die Generierung – entsprechend der neuen Generatorarchitektur für MontiCore-Generatoren – nicht länger Workflow-basiert [Kra10] geschieht, sondern durch ein Groovy-Skript gesteuert wird. Für die Weiterentwicklung der Collection Operators sowie die Entwicklung des Optional Operators wurde zudem sowohl die objektdiagrammbasierte Zwischenrepräsentation als auch der Pattern Matching Algorithmus erweitert (vgl. GR 10 und GR 7.2). Zusätzlich erlauben die Transformationen nun die Übergabe eines oder mehrerer Modelle, innerhalb derer das Pattern Matching stattfindet (vgl. GR 4 bzw. GR 4.1 und GR 4.2). Außerdem kann der generierten Transformation eine Instanz der `GlobalExtensionManagement` Klasse übergeben werden, die das Anhängen von Templates an Modellelemente innerhalb der Transformation ermöglicht. Dies dient der Integration von Transformations- und templatebasierter Generierung (vgl. GR 14). Durch die Gene-

rierung einzelner Java-Implementierungen für die verschiedenen Transformationsregeln und Komposition dieser zu komplexen Transformationen ist durch diese Modularität eine Wiederverwendung der Transformationsregeln gegeben (GR 6).

Die Zweiteilung in einen auf Objektdiagrammen basierenden Generator und einen DSTL-spezifischen Generator erweckt zunächst den Eindruck einer unnötigen Komplexität. Diese Teilung hat jedoch den Vorteil, dass der zweite Generator sowohl für die verschiedenen DSTLs als auch für andere Frontends verwendet werden kann, die sich auf eine linke und rechte Regelseite abbilden lassen. Ein Beispiel hierfür sind Deltasprachen [HHK+13, HHK+15]. Darüber hinaus lassen sich auch verschiedene konkrete und abstrakte Syntaxen für Transformationssprachen auf diese Zwischenrepräsentation abbilden und die Generierung der Transformationsimplementierungen wiederverwenden.

Ein ähnlicher, generativer Ansatz wird in [RKR+06] verfolgt, wobei hier im Gegensatz zu MontiTrans ein Generator nicht automatisiert erzeugt wird, sondern ein Framework zur Entwicklung solcher Generatoren vorgestellt wird. Darüber hinaus wurde in den Generierungsablauf eine Überprüfung der in Kapitel 4 beschriebenen Kontextbedingungen integriert. Für die neuen bzw. weiterentwickelten Operatoren wurde außerdem die objektdiagrammbasierte Notation für Transformationsregeln erweitert. Zusätzlich wurde für diese Operatoren auch das Pattern Matching um Teilpattern Matching für den Optional und den Collection Operator erweitert (vgl. GR 7.2 und GR 10). Neben MontiTrans bieten auch Fujaba [FNTZ00], PROGRES [SWZ99] und ModGraph [Win12] optionale Patternelemente, wobei die letzten beiden ebenfalls auf ein suchplangesteuertes Pattern Matching setzen. Im Gegensatz zu MontiTrans wird jedoch keine Schachtelung von optionalen Elementen unterstützt. Ähnlich wie MontiTrans bieten auch andere Tools Collection Operatoren [GKMP13, FT08, MH08, BNN+07, Ren06, HJvE06, dLETE04]. Diese sind jedoch auf eine Darstellung auf Basis der abstrakten Syntax beschränkt und zum Teil sogar auf einzelne Patternelemente (wie in Fujaba oder PROGRES).

Schließlich wurde noch die Performance des Pattern Matching Algorithmus der DSTLs durch eine dezentrale Auswertung des Application Constraint in Kombination mit automatischer Modularisierung des Constraints verbessert. Eine Alternative zur automatisierten Modularisierung wäre eine manuelle Modularisierung durch den Transformationsentwickler gewesen. Dies würde jedoch bedeuten, dass der Transformationsentwickler selbst verantwortlich für die Optimierung der Laufzeit wäre. Dies wiederum erfordert umfassendes Verständnis des Pattern Matching Algorithmus und widerspricht dem Gedanken, dass auch Domänenexperten, die die Modellierungssprache beherrschen, in die Lage versetzt werden sollen, Modelltransformationen zu entwickeln. Abgesehen von MontiTrans bieten auch AGG [Run17, Tae04] und PROGRES die Möglichkeit Constraints für ein Pattern anzugeben. In Progress hat der Transformationsentwickler die Möglichkeit selbst modulare Constraints für Attribute zu formulieren. Dies entspricht in etwa den Subconstraints in MontiTrans, allerdings müssen diese manuell entwickelt werden. Für diese findet ebenfalls eine dezentrale Überprüfung zum frühest möglichen Zeitpunkt statt. AGG erlaubt die Formulierung von Application Conditions [HHT96]. AGG bildet im Gegensatz zu MontiTrans das Pattern Matching auf das Constraint Sa-

tisfaction Problem ab. Die Überprüfung der Application Conditions findet bei AGG statt, nachdem ein Match gefunden wurde. Dies entspricht der zentralisierten Variante, die die DSTLs vor Einführung der dezentralen modularisierten Überprüfung unterstützt haben. Die Performanceanalyse hat außerdem gezeigt, dass trotz Modularisierung und dezentraler Überprüfung der Application Constraints die Formulierung des Constraints eine Modularisierung begünstigen oder verhindern kann. Eine mögliche weitere Optimierungen für das Pattern Matching wäre eine automatisierte Umformung des Constraints in konjunktive Normalform. In der aktuellen MontiCore Version bilden die ASTs der Modelle keine Graphen mit spannendem Baum, sondern echte Bäume, die um eine Symboltabelle [MSN17] ergänzt werden. Diese wird auf Basis des ASTs berechnet und muss im Anschluss an Transformationen aktualisiert bzw. neu berechnet werden. Der aktuelle Ansatz erlaubt hingegen die Transformation von Graphen. Wird die Baumstruktur für MontiCore beibehalten, kann basierend auf den Ergebnissen dieser Arbeit die Performance möglicherweise weiter verbessert werden, indem dieses Wissen über die zu transformierenden Modelle ebenfalls in die Berechnung des Suchplan einfließt. Darüber hinaus kann untersucht werden, ob eine Ersetzung der aktuellen Graphpattern Matching Strategie durch eine baumbasierte Strategie wie das Subtree Matching [ST97] sich günstig auf die Performance auswirkt.

Kapitel 9

MontiTrans-basierte Methodiken

In den vorherigen Kapiteln wurde MontiTrans sowie verschiedene DSTLs und basierend auf diesen wiederverwendbare Transformationen vorgestellt. In diesem Kapitel wird nun ergänzend die Methodik zur Entwicklung neuer DSTLs sowie neuer Transformationsregeln und Transformationen vorgestellt. Hierzu werden insbesondere die Verwendungs- und die Konfigurationsmöglichkeiten von MontiTrans aus Sicht des Sprachentwicklers, des Transformationsentwicklers und des Transformationsnutzers bzw. Modellierers erläutert (vgl. Abbildung 9.1).

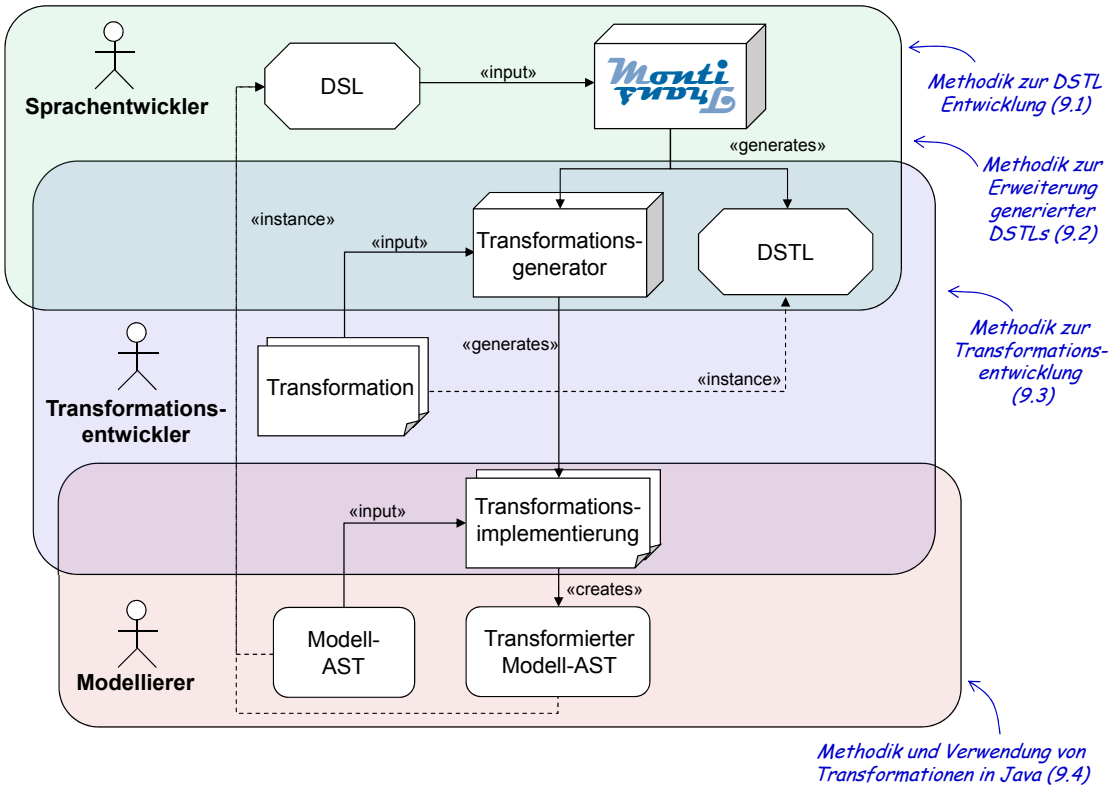


Abbildung 9.1: Übersicht über die verschiedenen Aspekte von MontiTrans mit Fokus auf den in diesem Kapitel vorgestellten Teil (Methodik).

Das restliche Kapitel ist wie folgt aufgeteilt. Zunächst wird die Methodik, sowie die mögliche Konfiguration und Ausführung des MontiTrans DSTL-Generators beschrieben. Anschließend wird eine Methodik zur Entwicklung neuer Transformationen erläutert und die Generierung von Java-Implementierungen für Transformationsregeln am Beispiel der DSTL CDTrans erklärt.

Die wichtigsten Ergebnisse dieses Kapitels sind:

- Eine Methodik zur DSTLs Entwicklung.
- Eine Methodik zur handgeschriebenen Erweiterung von generierten DSTLs.
- Eine Methodik zur Transformationsregel- und Modelltransformationsentwicklung.
- Eine Methodik zur Verwendung und Komposition von Transformationen in Java.
- Die Erläuterung der Konfiguration und Ausführung des Generators zur Generierung von DSTLs.
- Die Erläuterung der Konfiguration und Ausführung des Generators zur Generierung von Java-Implementierungen für Transformationsregeln.

9.1 Entwicklung einer neuen DSTL

In Abschnitt 3.1 wurden drei verschiedene Szenarien vorgestellt. Das erste Szenario betraf die Entwicklung einer neuen Sprache und entsprechend einer neuen DSTL für diese Sprache. In diesem Abschnitt wird die Methodik zur Neuentwicklung einer DSTL vorgestellt. Es wird beschrieben, welche Entscheidungen der Sprachentwickler treffen muss und wie er vorgehen sollte. Außerdem wird erklärt, welche Konfigurationsmöglichkeiten MontiTrans für die Generierung von DSTLs bietet.

9.1.1 Methodik

Zur Generierung von DSTLs stellt MontiTrans einen Generator bereit. Wie in Kapitel 6 vorgestellt, generiert der MontiTrans DSTL-Generator aus einer gegebenen DSL-Grammatik die DSTLs inklusive Kontextbedingungen und einem Generator zur Übersetzung der Transformationsregeln nach Java. Außerdem wurde erläutert, dass für modular definierte DSLs die DSTLs ebenfalls modular aufgebaut werden. Das heißt, dass für jede DSL Grammatik einer modularen Sprachdefinition auch eine DSTL generiert wird. Abbildung 9.2 zeigt das methodische Vorgehen des Transformationsentwickler inklusive zu treffender Entscheidungen.

Die DSTL-Entwicklung erfolgt wie auch die Entwicklung neuer DSLs modular. Das heißt, dass DSLs auf Basis existierender DSLs erstellt werden können. Hierzu bietet MontiCore die Konzepte der Sprachvererbung und Spracheinbettung. Diese Mechanismen sind in [LNPR+13, HLMSN+15b, HLMSN+15a] erläutert. Die erste Entscheidung bei der Entwicklung einer neuen DSTL ist die, ob für alle verwendeten Supersprachen DSTLs generiert werden sollen oder nur für die aktuelle Grammatik. Existieren bereits DSTLs für die Supersprachen, können diese wiederverwendet werden. Für die von Monti-

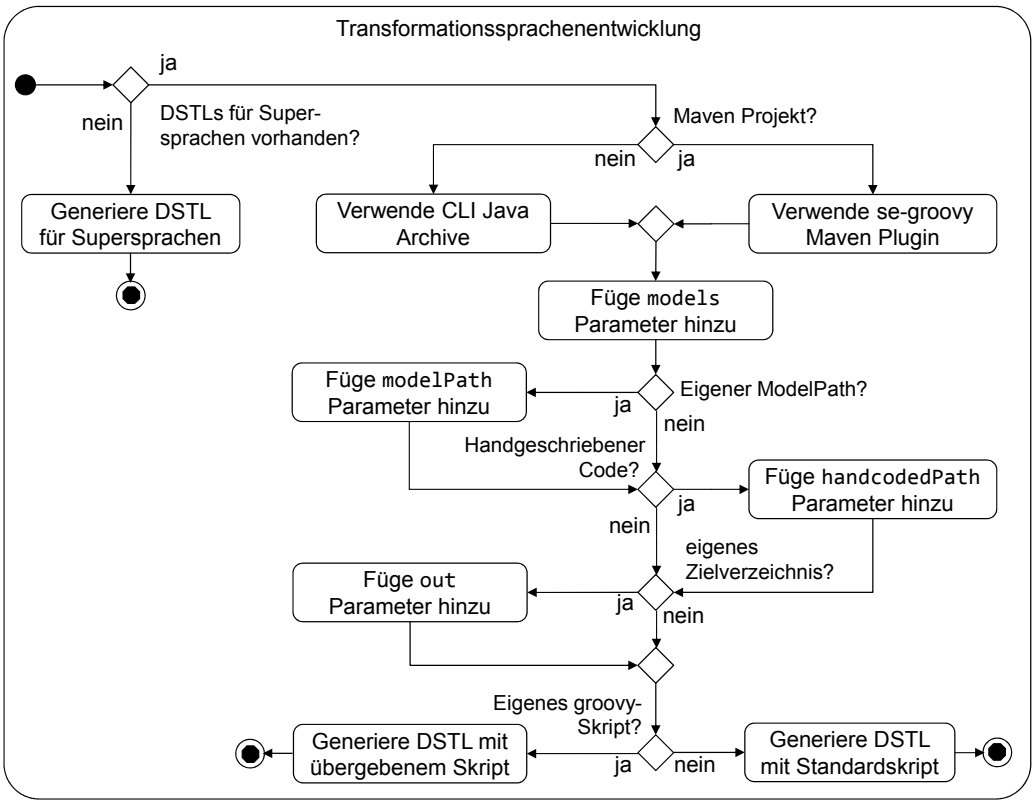


Abbildung 9.2: Vorgehen zur Entwicklung einer neuen DSTL mittels MontiTrans.

Core bereitgestellten Basissprachen wurden – wie in Kapitel 5 vorstellt – bereits DSTLs entwickelt. Ist dies nicht der Fall, müssen DSTLs für die Supersprachen entwickelt bzw. generiert werden.

Die nächste Entscheidung betrifft den eigenen Entwicklungsprozess. Wird Maven als Buildtool verwendet, kann die DSTL-Generierung in den eigenen Buildprozess integriert werden, indem das *se-groovy* Maven Plugin in Kombination mit dem MontiTrans DSTL-Generator verwendet wird (siehe Abschnitt 9.1.2). Andernfalls kann das ausführbare Command Line Interface (CLI) Java Archive verwendet werden (siehe hierzu Abschnitt 9.1.2). Die Vorgehensweise der nächsten Schritte der Entwickelung einer neuen DSTL ist unabhängig von der Entscheidung für Maven oder CLI. Aus diesem Grund wird die weitere Vorgehensweise des Sprachentwicklers unabhängig von der Wahl der Ausführung beschrieben. Im späteren Verlauf dieses Kapitels wird auf die Unterschiede bei der Konfiguration des CLI Java Archives und des Maven Plugins eingegangen. In beiden Fällen muss eine Konfiguration der Eingabe erfolgen. Dies geschieht über den notwendigen Konfigurationsparameter `models` (Erklärung in Abschnitt 9.1.2).

Die weiteren Entscheidungen betreffen die im nächsten Abschnitt detailliert vorgestellten Konfigurationsmöglichkeiten. Für das Zielverzeichnis, den Modellpfad (`ModelPath`)

und den Pfad für handgeschriebenen Code (`HandcodedPath`) ist jeweils zu entscheiden, ob die Generierung auf Basis einer eigenen Angabe oder unter Verwendung des Standardwertes erfolgen soll. Schließlich kann noch entschieden werden, ob das von MontiTrans zur Verfügung gestellte Groovy-Skript zur Generierung verwendet werden soll oder ob der Sprachentwickler ein eigenes Skript an den Generator übergibt, welches dieser basierend auf der in Abschnitt 6.4 vorgestellten Base Class ausführt.

9.1.2 Konfiguration und Ausführung des DSTL-Generators

Nachdem in Abschnitt 6.4 der Ablauf der Generierung und im vorangegangenen Abschnitt die Vorgehensweise des Sprachentwicklers beschrieben wurde, wird in diesem Abschnitt auf die mögliche Konfiguration des Generators und die Ausführung des Generators eingegangen. Dazu werden zunächst die Konfigurationsmöglichkeiten sowie Standardwerte der Konfiguration erläutert. Im Anschluss werden die beiden Möglichkeiten der Generatorausführung (per Kommandozeile (CLI) und per Maven Plugin [ABC+14]) erläutert. Die Parameter werden hier allgemein erläutert. Die Syntax zur Verwendung der Parameter ist jedoch verschieden, je nachdem, ob per Aufruf des CLI oder des Maven Plugins generiert wird. Die Syntax wird jeweils in den einzelnen Unterabschnitten zur Generierung per CLI bzw. Maven Plugin erläutert. Der MontiTrans DSTL-Generator bietet folgende Konfigurationsmöglichkeiten:

Eingabegrammatiken Der DSTL-Generator generiert basierend auf Grammatiken im MontiCore-Grammatikformat DSTLs inklusive einer Infrastruktur zur Verarbeitung von Modellen dieser DSTLs. Die einzige notwendige Angabe für die Ausführung des Generators ist die Angabe, welche Eingabemodelle der Generator verwenden soll. Hier kann sowohl eine einzelne Datei als auch ein Verzeichnis mit ein oder mehreren enthaltenden Grammatiken angegeben werden. Der Konfigurationsparameter hierfür ist `models` bzw. in Kurzform `m`.

ModelPath Der DSTL-Generator kann sowohl für monolithisch als auch für modular definierte Sprachen verwendet werden. Bei modularen Sprachdefinitionen müssen die Supersprachen der zu verarbeitenden Sprachen ebenfalls während der Generierung nachgeladen werden können. Falls sich nicht alle Sprachen der Sprachhierarchie innerhalb der Eingabegrammatiken befinden, bietet der ModelPath-Parameter die Möglichkeit den Ort zu spezifizieren, an dem weitere Grammatiken zum Nachzuladen gefunden werden können. Über den optionalen Parameter `modelPath` bzw. die Kurzform `mp` können Java Archives (JAR-Dateien) bzw. Verzeichnisse dem ModelPath hinzugefügt werden. Standardwert für den ModelPath ist der leere ModelPath bzw. im Fall des CLI das CLI Java Archive.

Zielverzeichnis Der DSTL-Generator produziert Ausgabeartefakte, die im Dateisystem abgelegt werden. Der Generator erlaubt die Konfiguration des Zielverzeichnisses über den optionalen Parameter `out` bzw. die Kurzform `o`. Wird kein Zielverzeichnis durch den Nutzer explizit angegeben wird `out` als Standardwert verwendet.

HandcodedPath Wie bereits in Abschnitt 6.4 beschrieben verwendet der MontiTrans DSTL-Generator den TOP-Mechanismus zur Integration von handgeschriebenem Code. Für die Überprüfung, ob handgeschriebener Code berücksichtigt werden muss, verwendet der Generator den `HandcodedPath`. Über den optionalen Parameter `handcodedPath` bzw. die Kurzform `hcp` können Verzeichnisse dem `HandcodedPath` hinzugefügt werden. Der Standardwert ist der leere `HandcodedPath`.

Dateiendung Der DSTL-Generator generiert neben der eigentlichen DSTL auch Infrastruktur zur Verarbeitung von Modellen (Transformationsregeln) der DSTLs. Die Dateiendung der Modell- bzw. Transformationsregeldateien kann durch den Sprachentwickler vorgegeben werden. Dazu kann der Konfigurationsparameter `fileExtension` bzw. die Kurzform `fe` verwendet werden. Ist keine Dateiendung angegeben wird als Standardwert `mtr` verwendet.

Groovy Skript Der DSTL-Generator wird durch ein Groovy-Skript [KKL+15], das auf der in Abschnitt 6.4 beschriebenen Base Class basiert, gesteuert. Das Skript realisiert den dort beschriebenen Ablauf und wird durch MontiTrans zur Verfügung gestellt. Soll der Generator ein anderes, handgeschriebenes Groovy-Skript basierend auf der in Abschnitt 6.4 beschriebenen Base Class nutzen, kann dies über den Parameter `script` bzw. in der Kurzform `s` angegeben werden.

```groovy
grammars = parseGrammars(models)

for (g in grammars){
  initSymbolTable(g, modelPath)
  init(g, out, handcodedPath, fileExtension)
  generateDSTL(g)
  generateCoCos(g)
  generateTranslator(g)
  generateScript(g)
}
```

Listing 9.3: Beispielausführung der DSTL-Generierung in Groovy.

Listing 9.3 zeigt das von MontiTrans zur Verfügung gestellte Groovy-Skript. Dieses realisiert den in Abschnitt 6.4 vorgestellten Ablauf der DSTL-Generierung. Das Skript zeigt, dass die konfigurierbaren Parameter hier in Form von Variablen Verwendung finden und die Generierung entsprechend beeinflussen. Die Variablen sind hierbei wie folgt getypt: `models` und `handcodedPath` sind vom Typ `IterablePath`, `out` ist vom Typ `File` und `modelPath` vom Typ `ModelPath`. Die Übersetzung der Konfigurationsparameter in getypte Variablen übernimmt MontiTrans. Diese Variablen können auch in von Nutzern erstellten Groovy-Skripten ohne weitere Übersetzung verwendet werden.

Im Folgenden werden die beiden Möglichkeiten zur Ausführung des Generators erläutert. Zunächst wird hierzu die Verwendung des CLI Java Archives erklärt und anschließend wird die Konfiguration und Verwendung des se-groovy Maven Plugins [ABC+14] zur Generierung einer DSTL erläutert.

Generierung mittels CLI und Groovy-Skript

Eine der beiden Möglichkeiten zur Ausführung des MontiTrans DSTL-Generators ist die Verwendung per CLI. Dazu wird das Java Archive `montitrans-cli.jar` – im Folgenden als MontiTrans CLI bezeichnet – verwendet und mit entsprechenden Konfigurationsparametern aufgerufen. Tabelle 9.4 gibt eine Übersicht über die Konfigurationsparameter. Die erste Spalte nimmt hierbei Bezug auf die im vorangegangenen Abschnitt beschriebenen Konfigurationsparameter. In Spalte zwei ist die Syntax des Parameters angegeben, während in Spalte drei die alternativ zu verwendende Kurzform des Parameters angegeben ist. Schließlich ist ein Beispiel für die Verwendung des Konfigurationsparameters in Spalte vier gezeigt.

Tabelle 9.4: Übersicht über die Konfigurationsparameter des MontiTrans CLIs.

Parameter	Syntax	Kurz	Beispiel
Eingabe	`-models <path>`	`-m`	`-m Automaton.mc4`
ModelPath	`-modelPath <path1> ... <pathn>`	`-mp`	`-mp supergrammars/ montitrans-cli.jar`
Zielverzeichnis	`-out <path>`	`-o`	`-o target/`
HandcodedPath	`-handcodedlPath <path1> ... <pathn>`	`-hcp`	`-hcp java/`
Dateiendung	`-fileExtension <ext>`	`-fe`	`-fe auttrans`
Skript	`-script <file>`	`-s`	`-s myscript.groovy`

Listing 9.5 zeigt einen exemplarischen Aufruf des MontiTrans CLIs. Durch `java -jar montitrans-cli.jar` wird das Java Archive ausgeführt. In dem Aufruf folgt als nächstes der notwendige Konfigurationsparameter für die Eingabegrammatiken (`-models Automaton.mc4`). Hierdurch wird die Grammatik `Automaton.mc4` als Eingabe angegeben. Das heißt, dass für diese Grammatik eine DSTL generiert wird. `-mp supergrammars montitrans-cli.jar` legt fest, dass der ModelPath aus dem Verzeichnis `supergrammars` sowie dem Java Archive `montitrans-cli.jar` bestehen soll. In diesem Verzeichnis und diesem Java Archive werden nachzuladende Grammatiken gesucht. Durch `-hcp java/` wird festgelegt, dass handgeschriebener Code im Verzeichnis `java` berücksichtigt werden soll. Schließlich wird das Ausgabeverzeichnis durch `-o target/` auf das Verzeichnis `target` festgelegt.

```
1 java -jar montitrans-cli.jar -m Automaton.mc4 -o target/          CLI
    -mp supergrammars/ montitrans-cli.jar -hcp java/ -fe atrans
```

Listing 9.5: Beispielaufruf des MontiTrans CLIs inklusive Konfiguration.

Generierung mittels Maven Plugin und Groovy-Skript

Die zweite Möglichkeit zur Ausführung des DSTL-Generators ist die Integration des *se-groovy Plugins* in einen Maven Buildprozess [ABC+14]. Das se-groovy Maven Plugin ermöglicht unter Angabe eines Groovy-Skripts sowie einer Base Class die Ausführung von Groovy-Skripten innerhalb eines Maven Buildprozesses. In Listing 9.6 ist die Verwendung des Plugins innerhalb des `<plugins>...</plugins>` Bereichs eines Project Objects Models (POM)s dargestellt. Mit Hilfe dieser Koordinaten kann das se-groovy Plugin in beliebige Maven-Projekte integriert werden. Zur Konfiguration des Plugins wird der `<configuration>...</configuration>` Block (Zeile 4-6) innerhalb des `<plugin>...</plugin>` Blocks verwendet. Abhängigkeiten können über den `<dependencies>...</dependencies>` Block angegeben werden.

pom.xml

```
1 <plugin>
2   <groupId>de.se_rwth.maven</groupId>
3   <artifactId>se-groovy-maven-plugin</artifactId>
4     <!-- Konfiguration -->
5     <!-- Dependencies -->
6 </plugin>
```

Listing 9.6: Verwenung des se-groovy Maven Plugins.

Listing 9.7 zeigt eine Konfiguration des se-groovy Plugins zur Verwendung des MontiTrans DSTL-Generators. Die Angaben innerhalb des `script`- und des `baseClass`-Tags sind notwendige Angaben für die Ausführung des se-groovy Plugins. Die Angabe `dstlgen.groovy` entspricht dem von MontiTrans bereitgestellten Groovy-Skript und `DSTLGenScript` der bereitgestellten Base Class. Die Angaben innerhalb der `arguments`-Tags werden als Konfiguration an den DSTL-Generator weitergegeben.

pom.xml

```
1  <configuration>
2    <script>mc/tfcs/dstlgen.groovy</script>
3    <baseClass>mc.tfcs.DSTLGenScript</baseClass>
4    <arguments>
5      <models>grammars</models>
6      <modelPath>supergrammars</modelPath>
7      <out>target</out>
8      <handcodedPath>java</handcodedPath>
9    </arguments>
10 </configuration>
```

Listing 9.7: Konfiguration des se-groovy Plugins zur Generierung einer DSTL.

Diese Argumente entsprechen den zuvor in Abschnitt 9.1.2 beschriebenen Konfigurationsparametern: `models` entspricht dem Eingabegrammatiken-Parameter, `modelPath` dem ModelPath-Parameter, `out` entspricht dem Zielverzeichnisparameter, `handcodedPath` dem HandcodedPath-Parameter und `script` dem Groovy-Skript-Parameter.

Durch die in Listing 9.7 angegebene Konfiguration wird das Verzeichnis `grammars` als Eingabeverzeichnis angegeben. Alle in diesem Verzeichnis abgelegten Grammatiken dienen dem Generator daher als Eingabe. Das heißt, für diese Grammatiken werden DSTLs generiert. Außerdem legt der Aufruf fest, dass der ModelPath aus dem Verzeichnis `supergrammars` bestehen soll. In diesem Verzeichnis und diesem Java Archive werden nachzuladende Grammatiken gesucht. Es wird außerdem festgelegt, dass handgeschriebener Code im Verzeichnis `java` berücksichtigt werden soll. Zusätzlich wird das Ausgabeverzeichnis auf das Verzeichnis `target` festgelegt. Schließlich muss der MontiTrans DSTL-Generator dem se-groovy Plugin noch als Abhängigkeit mitgegeben werden. Die Koordinaten sind in Listing 9.8 dargestellt. Ein ausführliches Konfigurationsbeispiel des se-groovy Plugins zur Generierung von DSTLs ist in Anhang C.5 zu finden.

pom.xml

```
<dependency>
  <groupId>de.monticore.tf</groupId>
  <artifactId>montitrans-dstlgen</artifactId>
</dependency>
```

Listing 9.8: Koordinaten des MontiTrans DSTL Generators.

9.2 Methodik zur Erweiterung der generierten DSTLs

Ein weiterer Aspekt des ersten Szenarios aus Abschnitt 3.1 ist die Anpassung der DSTL an ihre Nutzer. MontiTrans generiert sowohl die Grammatik der DSTLs, die Kontextbedingungen zur Überprüfung der Wohlgeformtheit der Transformationsregeln als auch die Infrastruktur zur Übersetzung der Transformationsregeln nach Java. Zur Adaption von generiertem Code durch den Nutzer gibt es verschiedene Möglichkeiten handgeschriebenen Code mit dem generierten zu kombinieren [GHK+15b]. Für die DSTL-Infrastruktur wird der auch in MontiCore verwendete *TOP-Mechanismus* – eine Variante des Generation Gap Pattern [VS13, Fow10, Vli98] – verwendet.

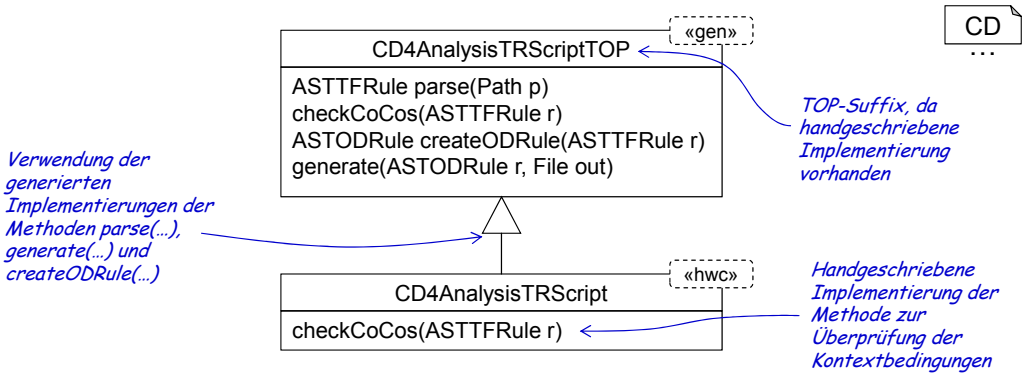


Abbildung 9.9: Beispiel zur Verwendung des TOP-Mechanismus zur Einbindung einer handgeschriebenen Implementierung der Base Class.

Abbildung 9.9 zeigt ein Beispiel für die Verwendung des TOP-Mechanismus. Hier wurde eine handgeschriebene Base Class `CD4AnalysisTRScript` erstellt, um eine eigene Implementierung der Methode `checkCoCos(...)` zu verwenden. Der Generator reagiert auf die Existenz der handgeschriebenen Klasse und generiert stattdessen die Klasse `CD4AnalysisTRScriptTOP`. Die handgeschriebene Klasse erbt von der generierten TOP-Klasse. Dadurch werden die Methoden `parse(...)`, `generate(...)` und `createODRule(...)` unverändert übernommen. Die Methode `checkCoCos(...)` wird überschrieben, wodurch die handgeschriebene Implementierung verwendet wird. Alle Stellen, an denen die Base Class verwendet wird, verweisen dadurch automatisch auf die handgeschriebene Klasse.

Der Generator prüft zur Laufzeit, ob die zu generierenden Klassen bereits handgeschrieben existieren. Falls keine handgeschriebene Version der zu generierenden Klasse vorliegt, wird sie wie üblich generiert. Andernfalls wird die Klasse mit dem Namenszusatz `TOP` generiert (z.B. `CD4AnalysisTRScript` wird zu `CD4AnalysisTRScriptTOP`). Die TOP-Klasse bietet genau die Funktionalität und Signaturen, die die generierte Klasse bieten würde und kann der handgeschriebenen Klasse als Superklasse dienen. Alle Stellen, an denen die generierte Klasse verwendet wird, bleiben ohne Namenszusatz und zeigen so automatisch auf die handgeschriebene Implementierung. Der Mechanismus kann für alle generierten Klassen verwendet werden. Dadurch können die Kontextbedingungen, der Generator sowie die Base Class mit handgeschriebenen Code ergänzt oder durch diesen ersetzt werden.

Um die Syntax der DSTL anzupassen gibt es zwei Möglichkeiten. Zum einen kann eine Grammatik erstellt werden, die von der generierten Grammatik erbt (vgl. Abschnitt 2.1.4). Hierdurch ist es sowohl möglich der DSTLs weitere Syntax hinzuzufügen, indem neue Nichtterminale hinzugefügt werden, als auch die bestehende Syntax zu verändern, indem Nichtterminale überschrieben werden. Für die neuen und redefinierten Nichtterminale ist es zusätzlich notwendig, die generierte Infrastruktur zur Übersetzung nach Java anzupassen. Dies ist insbesondere der Fall, da die von MontiCore generierte Visitor-Infrastruktur erfordert, dass Visitoren für die Sprache verwendet werden, für die sie generiert wurden, [HMSNRW16]. Damit kann ein Visitor nicht für eine Subsprache wiederverwendet werden. Für diese Anpassungen eignet sich der oben beschriebene TOP-Mechanismus.

Alternativ zur Bildung einer Subsprache kann die generierte Grammatik als Vorlage für eine handgeschriebene genutzt werden. Hierbei definiert die generierte Grammatik die von der generierten Infrastruktur erwartete Nichtterminalstruktur. Diese kann in die handgeschriebene Grammatik übernommen und angepasst werden. Der Nachteil dieser Methodik ist, dass die DSTL-Grammatik manuell gepflegt werden muss. Dies ist insbesondere ein Nachteil, wenn sich die Modellierungssprache noch in Entwicklung befindet und somit Änderungen unterliegt oder eine neuere Version des DSTL-Generators verwendet wird. In beiden Fällen wird durch die Generierung eine andere Grammatik generiert als es zuvor der Fall war. Diese Änderungen müssen anschließend manuell in die handgeschriebene Grammatik übernommen werden. Wird bei dieser Form der An-

passung die Nichtterminalstruktur und deren Semantik beibehalten, ist im Vergleich zur oberen Möglichkeit keine Anpassung der generierten Infrastruktur notwendig. Dies ist der Fall, wenn lediglich die konkrete Syntax verändert wird.

9.3 Entwicklung von Transformationsregeln

In Abschnitt 3.1 wurden drei verschiedene Szenarien vorgestellt. Das zweite Szenario betraf die Entwicklung von Transformationen. Die Entwicklung einer DSTL zur Spezifikation von Transformationen durch einen Sprachentwickler ist also bereits geschehen. Diese DSTL kann der Transformationsentwickler nutzen, um neue Transformationen zu entwickeln. In diesem Abschnitt wird die Methodik zur Neuentwicklung von Transformationen vorgestellt. Es wird erläutert, welche Entscheidungen der Transformationsentwickler treffen muss und wie er dabei vorgehen sollte. Außerdem wird erklärt, welche Konfigurationsmöglichkeiten die Generatoren der Java-Implementierungen bieten.

9.3.1 Methodik

Im Rahmen dieser Arbeit und insbesondere bei der Entwicklung der wiederverwendbaren Transformationen hat sich eine iterative Herangehensweise als sinnvoll herausgestellt. Hierbei werden Transformationsregeln zunächst sehr modellspezifisch erstellt und schrittweise verallgemeinert. Zusätzlich wurden drei Prinzipien identifiziert, die die Entwicklung und insbesondere Weiterentwicklung und Wartung von Transformationsregeln und Transformationen erleichtern:

1. Angabe der kleinstmöglichen Änderung,
2. eine modellorientierte Formatierung und
3. ein sparsamer Umgang mit Schemavariablen.

Im Folgenden wird zunächst die iterative Vorgehensweise erklärt. Anschließend werden die drei Prinzipien erläutert.

Iteratives Vorgehen

Die empfohlene iterative Vorgehensweise für Transformationsentwickler, um neue Transformationsregeln zu entwickeln, inklusive zu treffender Entscheidungen ist in Abbildung 9.10 dargestellt. Angelehnt an diese Vorgehensweise wurden auch die Operatoren der DSTLs in Kapitel 4 sowie die Syntax der entwickelten DSTLs in Kapitel 5 vorgestellt. Für die Vorgehensweise wird die Annahme getroffen, dass dem Transformationsentwickler wenigstens ein Modell vorliegt, anhand dessen er die Transformation entwickeln kann. Als erstes übernimmt der Transformationsentwickler die Modellelemente in das Pattern, die für die Transformation von Interesse sind. Dies entspricht bereits einem Pattern

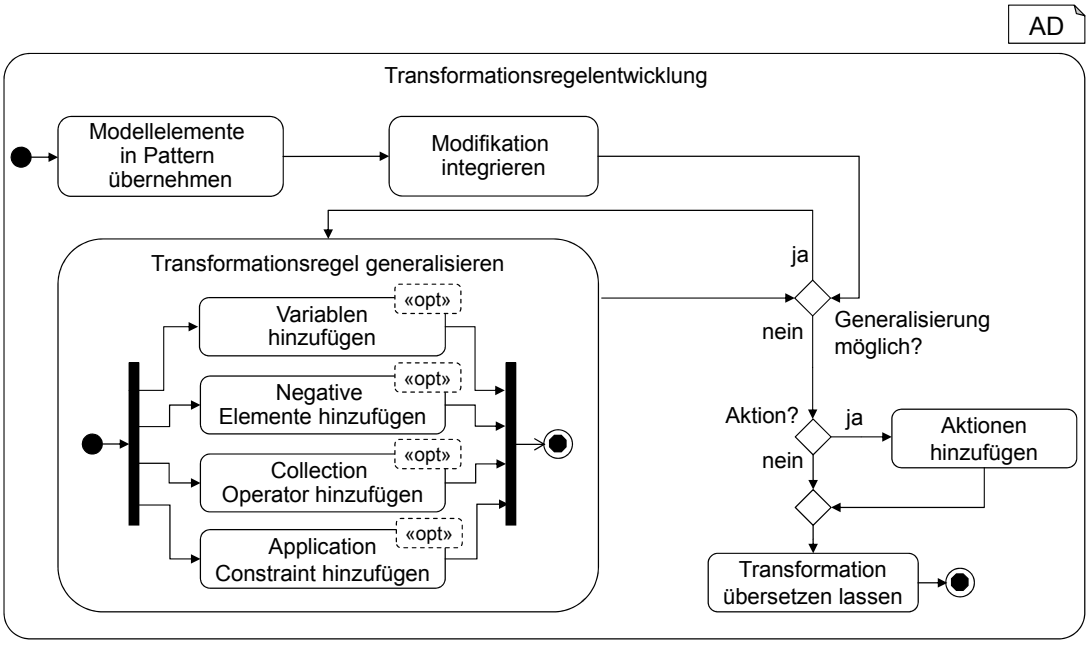


Abbildung 9.10: Vorgehen zur Entwicklung einer Transformationsregel.

für genau dieses Eingabemodell. Als nächstes wird das so entstandene Pattern um eine Modifikation ergänzt, die aus dem vorliegenden Modell das gewünschte Zielmodell erstellt. Damit wäre eine erste Version der angestrebten Transformation bereits fertig. Anschließend kann sich der Transformationsentwickler entscheiden, ob die Transformationsregel in der vorliegenden Form ausreicht oder die Transformation verallgemeinert werden sollte, um auf verschiedene Modells anwendbar zu sein. Soll die Transformationsregel verallgemeinert werden, dann kann dies mittels Variablen, negativen Elementen, dem Optional oder dem Collection Operator geschehen. In Abbildung 9.10 sind diese daher innerhalb der *Generalisieren*-Aktivität als parallele Aktivitäten angegeben, wobei jede Aktivität zusätzlich optional innerhalb des Generalisierungsschritts ist (markiert durch den Stereotyp «`opt`»). Eine Beschreibung der Operatoren ist in Kapitel 4 gegeben. Da es sich um ein iteratives Vorgehen handelt, kann die Generalisierung innerhalb eines Generalisierungsschrittes klein gehalten werden. Nach Abschluss eines Generalisierungsschrittes wird erneut geprüft, ob die Transformationsregel weiter verallgemeinert werden kann und soll. Es werden solange Operatoren, Variablen und Constraints eingefügt bis die Transformationsregel die aktuell erforderliche Generalität erreicht hat. Wird zu einem späteren Zeitpunkt festgestellt, dass die Transformation nicht allgemein genug anwendbar ist, kann das iterative Vorgehen hier fortgesetzt werden. Optional kann die Transformationsregel noch um Aktionen ergänzt werden wie beispielsweise Reporting, Logging oder das Anhängen von Templates an Modellelemente mittels `GlobalExtensionManagement` (vgl. Abschnitt 4.12 bzw. 4.13.4). Schließlich wird der Generator zur Generierung von Java-Implementierungen für Transformationsregeln – im Folgenden als Java-Generatoren bezeichnet – verwendet, um eine ausführbare Implementie-

rung der Transformationsregel zu erhalten. Die Konfiguration und Ausführung dieses Generators wird in Abschnitt 9.3.2 beschrieben. Die Verwendung der generierten Java-Implementierungen wird in Abschnitt 9.4 erläutert.

Kleinstmögliche Änderung angeben

Eines der Prinzipien, die der Transformationsentwickler befolgen sollte, ist immer nur die kleinstmögliche Änderung zu spezifizieren. Pattern lassen bewusst Modellinformationen, die für die Transformationsregel irrelevant sind, aus. Dadurch wird nichts darüber ausgesagt, wie diese Modellinformationen im Modell aussehen. Betrachtet man beispielsweise die Transformationsregeln in Listing 9.11 und Listing 9.12, könnte zunächst der Eindruck entstehen, dass beide Transformationsregeln auf verschiedene Weise die Umbenennung einer Klasse `A` zu `B` beschreiben. Das Pattern beschreibt in beiden Fällen eine Klasse mit dem Namen `A`. In Listing 9.11 wird der Replacement Operator angewendet, um den Namen `A` durch `B` zu ersetzen. Hierbei handelt es sich also tatsächlich um eine Umbenennung. In Listing 9.12 wird der Replacement Operator auf die Klasse `A` angewendet. Diese wird durch eine Klasse `B` ersetzt. Obwohl die Klasse im Pattern ohne Attribute angegeben wird, kann das Patternelement während des Pattern Matching durch eine Klasse besetzt werden, die zum Beispiel Attribute, Methoden und einen Modifier enthält. Wird die Transformationsregel aus Listing 9.11 angewendet, bleibt all dies erhalten. Im zweiten Fall jedoch wird die gesamte Klasse `A` samt Inhalt entfernt und eine neue, leere Klasse `B` erzeugt und hinzugefügt. In diesem Fall müssen durch den Transformationsentwickler alle Eigenschaften der Klasse `A` auf die Klasse `B` übertragen werden, was zu einer unnötig komplizierten und unverständlichen Transformationsregel führen würde. Aus diesem Grund sollte das Prinzip der kleinstmöglichen Änderung beachtet werden.

```
1 class [[A :- B]]:                                                CDTrans
```

Listing 9.11: Transformationsregel zur Umbebnennung einer Klasse.

```
1 [[ class A ; :- class B; ]]                                      CDTrans
```

Listing 9.12: Transformationsregel zur Ersetzung einer Klasse.

Formatierung der Transformationsregel an die Modellformatierung anlehnen

Wie bereits in [Wei12] erläutert profitieren Transformationsregel von einer sinnvollen Formatierung. Da sich die Syntax der DSTL an der Syntax der zugehörigen DSL orientiert, sollte sich auch die Formatierung der Transformationsregel an dieser orientieren. Genau wie die Verwendung der Modellsyntax verbessert dies die Integration von Domänenexperten in die Transformationsentwicklung, da die Transformationsregel sich so noch stärker an den Modellen, die durch die Transformationsregel transformiert werden

sollen, orientiert. Die beiden Transformationen zur Umbenennung einer Klasse `A` mit dem Attribut `String name` zu `B` in Listing 9.13 und Listing 9.14 verdeutlichen dies. Während die Struktur der Transformationsregel in Listing 9.13, also ein Klassendiagramm mit einer Klasse `A`, die ein Attribute `String name` hat, erkennbar ist, verschleiert die Formatierung in Listing 9.14 diese Struktur.

CDTrans

```
1 classdiagramm $_ {
2   class [[A :- B]] {
3     String name;
4   }
5 }
```

Listing 9.13: Transformationsregel zur Umbebnennung einer Klasse.

CDTrans

```
1 classdiagramm $_ { class
2 [[A :- B]]
3  {String name;} }
```

Listing 9.14: Transformationsregel zur Umbebnennung einer Klasse.

Schemavariablen nur wenn sie benötigt werden

Neben irritierender Formatierung kann auch ein übermäßiger Gebrauch von Schemavariablen für Patternelemente eine Transformationsregel schwer verständlich machen. Schemavariable sollten nur dort verwendet werden, wo sie entweder als Ein-/Ausgabeparameter, für die Zuweisungen innerhalb des Zuweisungsblock, innerhalb des Anweisungsblocks oder des Application Constraints gebraucht werden. Listing 9.15 zeigt die gleiche Transformationsregel wie Listing 9.13, allerdings wurde hier exzessiver Gebrauch von Schemavariablen gemacht, die keinen Nutzen für die Transformationsregel haben. Auch wenn sich die Formatierung der Transformationsregel an der Formatierung von Klassendiagrammen orientiert, hemmen die vielen, unnötigen Schemavariablen die Lesbarkeit.

CDTrans

```
1 $CD[[ Modifier $MCD classdiagramm $_ {
2   $C [[ Modifier $MCA class [[A :- B]] {
3     $A [[ Modifier $MA $T [[String]] name;
4   }
5 } ]]
```

Listing 9.15: Transformationsregel zur Umbebnennung einer Klasse.

9.3.2 Konfiguration und Ausführung des Java-Generators

Nachdem in Abschnitt 6.4 der Ablauf der Generierung und im vorangegangenen Abschnitt die Vorgehensweise des Transformationsentwicklers beschrieben wurde, wird in diesem Abschnitt auf die mögliche Konfiguration des Generators für Java-Implementierungen und die Ausführung dieses Generators eingegangen. Die Ausgangssituation für

die Verwendung des Generators ist eine Menge von Transformationsregeln, die konform zu einer bereits entwickelten DSTL formuliert wurden. Zunächst werden die Konfigurationsmöglichkeiten sowie Standardwerte der Konfiguration erläutert. Im Anschluss werden die beiden Möglichkeiten der Generatorausführung (per Kommandozeile (CLI) und per Maven Plugin [ABC+14]) erläutert. Die Parameter werden hier allgemein erläutert. Die Syntax zur Verwendung der Parameter ist jedoch verschieden, je nachdem, ob per CLI oder per Maven Plugin generiert wird. Die Syntax wird jeweils in den einzelnen Unterabschnitten zur Generierung per CLI bzw. Maven Plugin erläutert. Die Generatoren für Java-Implementierungen zu Transformationsregeln – im Folgenden als Java-Generatoren bezeichnet – bieten folgende Konfigurationsmöglichkeiten:

Eingabetransformationen Die Java-Generatoren generieren basierend auf Transformationsregeln Java-Implementierungen dieser Transformationsregeln. Die einzige notwendige Angabe für die Ausführung dieser Generatoren ist die Angabe, welche Eingabemodelle der Generator verwenden soll. Hier kann sowohl eine einzelne Datei als auch ein Verzeichnis mit ein oder mehreren enthaltenden Transformationsregeln angegeben werden. Der Konfigurationsparameter hierfür ist `models` bzw. in Kurzform `m`.

ModelPath Genau wie für den DSTL-Generator kann auch für die diese Generatoren der ModelPath konfiguriert werden. Über den optionalen Parameter `modelPath` bzw. die Kurzform `mp` können Java Archives (JAR-Dateien) bzw. Verzeichnisse dem ModelPath hinzugefügt werden. Standardwert für den ModelPath ist der leere ModelPath bzw. im Fall des CLI das CLI Java Archive.

Zielverzeichnis Die Java-Generatoren produzieren Ausgabeartefakte, die im Dateisystem abgelegt werden. Der Generator erlaubt die Konfiguration des Zielverzeichnisses über den optionalen Parameter `out` bzw. die Kurzform `o`. Wird kein Zielverzeichnis durch den Nutzer explizit angegeben, wird `out` als Standardwert verwendet.

Groovy Skript Die Generierung durch die Java-Generatoren wird mithilfe ein Groovy-Skripts [KKL+15] gesteuert, das auf der in Kapitel 8 beschriebenen Base Class basiert. Das Skript realisiert den dort beschriebenen Ablauf und wird durch MontiTrans zur Verfügung gestellt. Soll der Generator ein anderes Groovy-Skript basierend auf der bereitgestellten Base Class nutzen, kann dies über den Parameter `script` bzw. in der Kurzform `s` angegeben werden.

```groovy
astRules = parseRules(models)

for (rule in astRules){
  checkCoCos(rule)
  odrule = createODRule(rule, modelPath)
  generate(odrule, out, modelPath)
}
```

Listing 9.16: Groovy-Skript zur Java-Generierung für Transformationsregeln.

Listing 9.16 zeigt das von MontiTrans zur Verfügung gestellte Groovy-Skript zur Generierung von Transformationsimplementierungen. Dieses realisiert den in Kapitel 8 vor-

gestellten Ablauf der Generierung. Das Skript zeigt, dass die konfigurierbaren Parameter hier in Form von Variablen Verwendung finden und die Generierung entsprechend beeinflussen. Die Variablen sind hierbei wie folgt getypt: `models` ist vom Typ `IterablePath`, `out` ist vom Typ `File` und `modelPath` vom Typ `ModelPath`. Die Übersetzung der Konfigurationsparameter in getypte Variablen übernimmt MontiTrans. Diese Variablen können auch in vom Transformationsentwickler erstellten Groovy-Skripten ohne weitere Übersetzung verwendet werden.

Generierung mittels CLI und Groovy-Skript

Eine der beiden Möglichkeiten zur Ausführung der Java-Generatoren ist die Verwendung per CLI. Dazu wird ein ausführbares Java Archive aus dem Generat des MontiTrans DSTL-Generators erzeugt und mit entsprechenden Konfigurationsparametern aufgerufen. Dies wird im Folgenden als CLI bezeichnet. Die Main-Klasse hiefür wird ebenfalls durch MontiTrans generiert. Tabelle 9.17 gibt eine Übersicht über die Konfigurationsparameter. Die erste Spalte nimmt hierbei Bezug auf die im vorangegangenen Abschnitt beschriebenen Konfigurationsparameter. In Spalte zwei ist die Syntax des Parameters angegeben, während in Spalte drei die alternativ zu verwendende Kurzform des Parameters angegeben ist. Schließlich ist ein Beispiel für die Verwendung des Konfigurationsparameters in Spalte vier gezeigt.

Tabelle 9.17: Übersicht über die Konfigurationsparameter des CLIs.

Parameter	Syntax	Kurz	Beispiel
Eingabe	`-models <path>`	`-m`	`-m transformations/`
ModelPath	`-modelPath <path1> ... <pathn>`	`-mp`	`-mp models/`
Zielverzeichnis	`-out <path>`	`-o`	`-o target/`
Skript	`-script <file>`	`-s`	`-s myscript.groovy`

Listing 9.18 zeigt einen exemplarischen Aufruf für das CLI der DSTL CDTrans. Durch `java -jar cdtrans-cli.jar` wird das Java Archive ausgeführt. In dem Aufruf folgt als nächstes der notwendige Konfigurationsparameter für die Eingabetransformationen (`-m transformations/`). Hierdurch werden alle CDTrans-Transformationsregeln im Verzeichnis `transformations` als Eingabe verwendet. Es werden Java-Implementierungen für diese Transformationsregeln generiert. Schließlich wird das Ausgabeverzeichnis durch `-o target/` auf das Verzeichnis `target` festgelegt.

```
1 java -jar cdtrans-cli.jar -m transformations/ -o target/
```
CLI

Listing 9.18: Beispielaufruf des MontiTrans CLIs inklusive Konfiguration.

Generierung mittels Maven Plugin und Groovy-Skript

Die zweite Möglichkeit zur Ausführung der Java-Generatoren ist die Integration des se-groovy Plugins in einen Maven Buildprozess [ABC+14]. Das se-groovy Maven Plugin ermöglicht unter Angabe eines Groovy-Skripts sowie einer Base Class die Ausführung von Groovy-Skripten innerhalb eines Maven Buildprozesses. In Listing 9.6 in Abschnitt 9.1.2 ist die Verwendung des Plugins innerhalb des `<plugin>...</plugin>` Bereichs eines Project Object Model (POM) dargestellt. Mit Hilfe dieser Koordinaten kann das se-groovy Plugin in beliebige Maven-Projekte integriert werden. Zur Konfiguration des Plugins wird der `<configuration>...</configuration>` Block innerhalb des `<plugin>...</plugin>` Blocks verwendet.

pom.xml

```
<configuration>
  <script>script/dstl2java.groovy</script>
  <baseClass>script.CD4ATransScript</baseClass>
  <arguments>
    <models>transformations</models>
    <out>/target</out>
  </arguments>
</configuration>
```

Listing 9.19: Konfiguration des Maven Plugins zur Generierung einer Java-Implementierung für eine CDTrans-Transformationsregel.

Listing 9.19 zeigt eine mögliche Konfiguration des se-groovy Plugins zur Verwendung des Java-Generators für CDTrans-Transformationsregeln. Hierbei sind die Angaben innerhalb des `script`- und des `baseClass`-Tags notwendige Angaben für die Ausführung des se-groovy Maven Plugins. `dstl2java.groovy` entspricht hierbei dem von MontiTrans bereitgestellten Groovy-Skript und `CD4ATransScript` der Base Class für CDTrans. Die Angaben innerhalb der `arguments`-Tags werden als Konfiguration an den Java-Generator weitergegeben.

pom.xml

```
<dependency>
  <groupId>de.monticore.lang</groupId>
  <artifactId>cd4a-trans</artifactId>
</dependency>
```

Listing 9.20: Koordinaten des CDTrans Java-Generators.

Diese Argumente entsprechen den zuvor beschriebenen Konfigurationsparametern: `models` entspricht dem Eingabetransformationen-Parameter, `out` entspricht dem Zielverzeichnisparameter und `script` dem Groovy-Skript-Parameter. Hierdurch wird das Verzeichnis `transformations` als Eingabeverzeichnis angegeben. Alle in diesem Verzeichnis abgelegten CDTrans-Transformationsregeln dienen dem Java-Generator daher als Eingabe. Für diese Transformationsregeln werden Java-Implementierungen generiert. Außerdem wird das Ausgabeverzeichnis auf das Verzeichnis `target` festgelegt. Schließ-

lich muss der Java-Generator dem se-groovy Plugin noch als Abhängigkeit mitgegeben werden. Die Koordinaten sind in Listing 9.20 dargestellt. Ein ausführliches Konfigurationsbeispiel des se-groovy Plugins zur Generierung von Java-Implementierungen ist in Anhang C.6 zu finden.

9.4 Methodik zur Verwendung und Entwicklung komplexer Transformationen in Java

In den vorangegangenen Abschnitten wurde beschrieben wie aus modellierten Transformationsregeln Java-Implementierungen generiert werden können. In diesem Abschnitt liegt der Fokus auf der Verwendung der generierten Transformationen sowie auf der Entwicklung von komplexen Transformationen aus Transformationsregeln. Dies entspricht dem dritten in Abschnitt 3.1 vorgestellten Szenario.

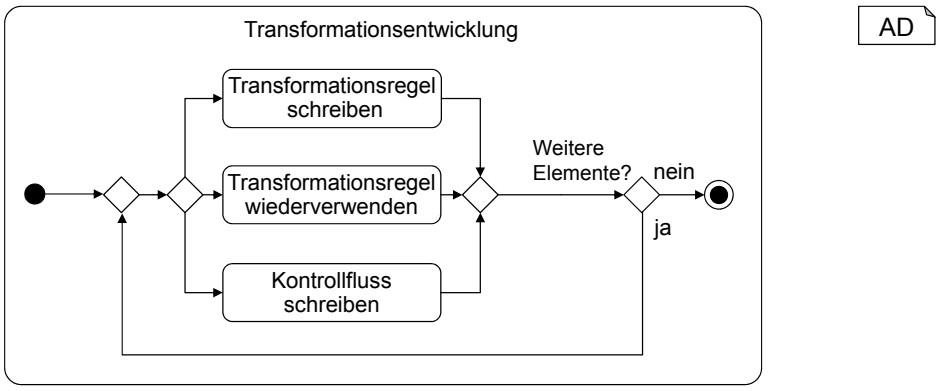


Abbildung 9.21: Methodik zur Entwicklung einer komplexen Transformation.

Komplexe Transformationen bestehen aus vielen einzelnen in der Regel simplen Transformationsregeln, die durch eine Anwendungsstrategie miteinander kombiniert werden. Die einfachste komplexe Transformation setzt sich daher aus einer einzelnen Transformationsregel sowie einmaliger Anwendung dieser Regel zusammen. Für die Entwicklung komplexer Transformationen empfiehlt sich erneut ein iteratives Vorgehen. Abbildung 9.21 verdeutlicht dieses Vorgehen. In jedem Schritt erstellt der Transformationsentwickler wahlweise eine Transformationsregel (vgl. Abschnitt 9.3), verwendet eine vorhandene Transformationsregel – beispielsweise aus einer Bibliothek von Transformationsregeln – oder erweitert den Kontrollfluss, der definiert, wann welche Transformationsregel angewendet wird. Dies wird solange fortgesetzt bis die komplexe Transformation die gewünschte Modelltransformation realisiert. Im Folgenden wird erläutert, wie Transformationsregeln verwendet und über einen in Java beschriebenen Kontrollfluss zu komplexen Transformation zusammengesetzt werden können.

9.4.1 Anwendung und Kombination von Transformationsregeln

Neben der in [Wei12] vorgestellten Kontrollflusssprache können die Transformationsregeln mittels Java kombiniert werden. Beide Ansätze überlassen die Definition der Anwendungsstrategie dem Transformationsentwickler. Anders als beispielsweise bei der strategischen Programmierung [Vis01a, Vis01b, LVV03] werden keine vordefinierten Strategien kombiniert, sondern die Anwendungsstrategie imperativ formuliert. Für die Anwendung einer Transformation wird die generierte Java-Implementierung verwendet. Listing 9.22 zeigt ein Beispiel für die Anwendung der `RenameClass`-Transformation aus Listing 9.11. Hierbei wird in Zeile 3 eine Transformationsregel erzeugt und der ASTeines Modells an die Regel übergeben. An dieser Stelle können beliebig viele ASTs kommasepariert übergeben werden. Das Pattern Matching findet in allen der Transformation übergebenen Modellen statt. Die `doAll`-Methode in Zeile 4 führt zunächst das Pattern Matching (realisiert durch die Methode `doPatternMatching()`) und anschließend die Modifikation (realisiert durch die Methode `doReplacement()`) durch. Eine sequentielle Kombination von Transformationsregeln wird erreicht, indem weitere Transformationsregeln erzeugt und angewendet werden. Die Anwendung von lokal definierten und aus Bibliotheken importierten Transformationsregeln unterscheidet sich nicht. In beiden Fällen wird die generierte Java-Implementierung importiert und auf die beschriebene Weise instanziiert und ausgeführt.

```java
1 /* Erstellen des ASTs */
2 // Anwenden der Transformationsregel
3 RenameClass renameClass = new RenameClass(ast);
4 renameClass.doAll();
```

Listing 9.22: Anwendung einer Transformationsregel.

Die beiden Methoden `doPatternMatching` und `doReplacement` können auch einzeln verwendet werden, um eine feinere Steuerung des Ablaufs zu ermöglichen, beispielsweise um eine Transformationsregel auszuführen, falls eine andere ebenfalls anwendbar ist. Dies ist in Listing 9.23 dargestellt. Hier wird, falls die Transformationsregel `RenameClass` anwendbar ist, auch die Transformationsregel `PullUpAttributes` (vgl. Abschnitt 7.1.1) angewendet. Dies entspricht dem `if`-Statement der in [Wei12] vorgestellten Kontrollflusssprache. Dadurch können Transformationsregeln in Abhängigkeit zueinander gesetzt werden und somit komplexe Transformationen erstellt werden, die auf die Anwendbarkeit der Transformationsregeln reagieren und andere Transformationsregeln ausführen. Über einen `else`-Teil kann darauf reagiert werden, falls eine Transformationsregel nicht anwendbar ist. Die ASTs werden in-place transformiert. Das bedeutet, es wird keine Kopie der ASTs erzeugt, sondern die ASTs direkt modifiziert.

Listing 9.24 zeigt eine Umsetzung des `loop`-Statements der Kontrollflusssprache aus [Wei12] in Java. Die `while`-Schleife wird hierbei solange ausgeführt, solange es ein Match für das angegebene Pattern gibt. Die bisher vorgestellten Kontrollflusselemente erlauben es beliebig komplexe Transformationen aus Transformationsregeln zu erstellen.

```java
RenameClass renameClass = new RenameClass(ast);
if(renameClass.doPatternmatching()){
  renameClass.doReplacement();
  PullUpAttributes pullup = new PullUpAttributes(ast);
  pullup.doAll()
}
```

Listing 9.23: Anwendung einer Transformationsregel, falls eine andere ebenfalls anwendbar ist.

```java
RenameClass renameClass = new RenameClass(ast);
while(renameClass.doPatternmatching()){
  renameClass.doReplacement();
  renameClass = new RenameClass(ast);
}
```

Listing 9.24: Anwendung einer Transformationsregel solange diese anwendbar ist.

Mit den bisher vorgestellten Möglichkeiten für den Kontrollfluss wird jede Transformationsregel für sich angewendet. Ein Austausch zwischen den Transformationsregeln findet nicht statt. Zwar kann jede Transformationsregel die von ihr benötigten Elemente entweder im Modell finden oder, falls sie hinzugefügt werden sollen, erzeugen, dies führt aber zu unnötig komplexen Transformationsregeln. Zusätzlich wirken sich zusätzliche Patternelemente negativ auf die Laufzeit des Pattern Matching aus. Aus diesem Grund ist ein Austausch von Modellelementen zwischen Transformationsregeln wünschenswert. Die DSTLs ermöglichen dies durch Schemavariablen. Patternelemente, die an Schemavariablen gebunden werden, können sowohl als Eingabe- als auch als Ausgabeparameter fungieren. Dies wird im nächsten Abschnitt erläutert.

9.4.2 Verwendung von Parametern

Um Modellelemente zwischen Transformationsregeln auszutauschen, können Patternelemente, die innerhalb des Pattern mit einer Schemavariable versehen sind, als Eingabe- bzw. Ausgabeparameter der Transformationsregel verwendet werden. Listing 9.25 zeigt eine Transformationsregel bei der einer Klasse mit dem Namen `B` – gebunden an die Schemavariable `$C` – ein Attribut, das an die Variable `$A` gebunden ist, hinzugefügt wird. Diese Transformationsregel soll nun so genutzt werden, dass `$A` der Transformationsregel übergeben wird und die Klasse `$C` von der Transformationsregel gefunden und zurückgegeben wird.

Listing 9.26 zeigt die Verwendung der zuvor beschriebenen Transformationsregel. Wie zuvor beschrieben wird die Transformationsregel instanziiert und das Modell an die Transformationsregel übergeben. Bevor jedoch die Transformationsregel ausgeführt wird, wird der Transformationsregel noch über die generierte `set_$A`-Methode das hinzuzufügende Attribut übergeben. Dadurch ist das Attribut `$A` innerhalb der Transformati-

```
$C  [[ class B {                                                    CDTrans
          [[ :- CDAttribute $A ]]
       } ]]
```

Listing 9.25: AddAttribute: Transformationsregel, die einer Klasse B ein Attribut hinzufügt.

onsregel verfügbar und wird von der Transformationsregel der Klasse `B` hinzugefügt. Auf die gleiche Weise können auch Teile des Pattern der Transformationsregel mitgegeben werden. In diesem Fall betrachtet die Transformationsregel diese Teile als fixiert und führt kein Pattern Matching für dieses Patternelement durch.

```
/* Erstellen des ASTs und des Attributs attr */                      Java
// Verwenden von Parametern der Transformationsregel
AddAttribute addAttribute = new AddAttribute(ast);
addAttribute.set_$A(attr);
addAttribute.doAll();
ASTCDClass c = addaddAttribute.get_$C();
```

Listing 9.26: Verwendung von Schemavariablen einer Transformationsregel als Ein- bzw. Ausgabeparameter.

Im Anschluss an die Ausführung der Transformationsregel wird die `get_$C-Methode` verwendet, um die Klasse `B`, als Match für das Patternelement `$C` von der Transformationsregel zu erhalten. Auf diese Weise können Modellelemente von einer Transformationsregel erhalten und an eine andere Transformationsregel übergeben werden.

9.4.3 Verwendung des Templateerweiterungsmechanismus

Wie in Abschnitt 4.13.4 beschrieben, kann innerhalb von Transformationsregeln durch die Variable `glex` als Instanz der Klasse `GlobalExtensionManagement` der Templateerweiterungsmechanismus von MontiCore verwendet werden. Dadurch können beispielsweise Templates an neu erzeugte Modellelemente angehängt werden. Listing 9.27 zeigt eine Transformationsregel, bei der einer Klasse mit dem Namen `B` ein Attribut `String name`, das an die Schemavariable `$A` gebunden ist, hinzugefügt wird. Außerdem wird innerhalb des Anweisungsblocks für das hinzugefügte Attribute das zu verwendende Template ausgetauscht. Damit diese Ersetzung sinnvoll funktioniert, benötigt die Transformationsregel eine Instanz der Klasse `GlobalExtensionManagement`. Listing 9.28 zeigt die Verwendung einer Transformationsregel, die den Templateerweiterungsmechanismus verwendet. Der Transformationsregel wird eine eigene Instanz der Klasse `GlobalExtensionManagement` übergeben.

Listing 9.28 erzeugt zunächst wie zuvor beschrieben eine Instanz der Transformationsregel und übergibt dieser das Modell. Bevor die Transformationsregel ausgeführt wird, wird der Transformationsregel über die `setGlex`-Methode die zu verwendende `Glo-`

```
CDTrans
class B {
  [[ :-  $A [[ String name; ]] ]]
}
do {
  glex.replaceTemplate("cd.DefaultAttr", $A, "cd.NameAttr")
}
```

Listing 9.27: AddAttribute: Transformationsregel, die einer Klasse B ein Attribut hinzufügt.

balExtensionManagement-Instanz übergeben. Intern verwendet die Transformationsregel die durch den Nutzer übergebene GlobalExtensionManagement-Instanz. Die Transformationsregel bieten außerdem eine getGlex-Methode, über die auf die verwendete GlobalExtensionManagement-Instanz zugegriffen werden kann. Wird der Transformationsregel keine GlobalExtensionManagement-Instanz mitgegeben, erstellt die Transformationsregel dieses Instanz selbst. Anschließend kann der Nutzer durch die Zugriffsmethode getGlex auf diese Instanz zugreifen.

```
Java
/* Erstellen des ASTs und GlobalExtensionManagement glex */
// Transformationsregel mit GlobalExtensionManagement
RenameClass renameClass = new RenameClass(ast);
renameClass.setGlex(glex);
renameClass.doAll();
glex = renameClass.getGlex();
```

Listing 9.28: Anwendung einer Transformationsregel, der eine Instanz der Klasse GlobalExtensionManagement übergeben wird, um den Templateerweiterungsmechanismus zu nutzen.

9.5 Zusammenfassung

In diesem Kapitel wurde eine Methodik zur modellgetriebenen Entwicklung von und mit Modelltransformationen mit MontiTrans vorgestellt. Dazu wurde für die verschiedenen Einsatzmöglichkeiten bzw. aus Sicht der möglichen Nutzer entsprechend der in Kapitel 3 vorgestellten Rollen und entlang der ebenfalls dort vorgestellten Szenarien eine Methodik zur Verwendung von MontiTrans erläutert. Hierzu wurde für jedes Szenario – Erstellen neuer DSTLs, Erstellen neuer Transformationsregeln und Verwenden von Transformationen– beschrieben, wie MontiTrans in diesem Szenario verwendet werden kann. Zusätzlich wurden für den MontiTrans DSTL-Generator und für die Java-Generatoren sowie für die Verwendung der generierten Java-Implementierungen jeweils erläutert, welche Möglichkeiten zur Konfiguration bzw. Ausführung MontiTrans bietet. Es wurde insbesondere auch erläutert, wie Transformationsregeln auf mehrere Modelle angewendet werden können (vgl. GR 4) und wie die Integration mit dem Templa-

teerweiterungsmechanismus aus [Rot17] funktioniert (vgl. GR 14). Außerdem wurde gezeigt, wie Transformationen modular entwickelt und durch mittels Kontrollflusselementen miteinander kombiniert werden können (vgl. GR 6). Zusätzlich wurde gezeigt, wie der Austausch von Modellelementen oder ganzen Modellen zwischen Transformationsregeln funktioniert (vgl. GR 4.3). Schließlich wurden für die Entwicklung von Transformationsregeln Prinzipien entwickelt, die die Entwicklung und insbesondere Wartung und Weiterentwicklung von Transformationsregeln erleichtern. Diese wurden ebenfalls in diesem Kapitel vorgestellt.

Teil IV

Epilog

Kapitel 10

Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde MontiTrans vorgestellt. MontiTrans bietet einen DSTL-Generator, der aus einer gegebenen Grammatik einer DSL eine zugehörige DSTL plus Infrastruktur zur Verarbeitung der Transformationen, die mit dieser DSTL entwickelt wurden, erzeugt. Darüber hinaus wurde MontiTrans zur Entwicklung der DSTLs CD-Trans, MATrans und MACDTrans verwendet. Mithilfe dieser DSTLs wurden schließlich Bibliotheken von wiederverwendbaren Transformationen für MontiArc-Modelle und Klassendiagramme entwickelt.

In Kapitel 6 wurde eine Methodik zur Ableitung von DSTLs sowie deren Umsetzung in Form des DSTL-Generators von MontiTrans beschrieben. Dazu wurde für die verschiedenen Grammatikkonzepte gezeigt, wie diese durch die Ableitung berücksichtigt werden. Damit beantworten die Ableitung und der Generator die Forschungsfrage RQ1.1, also die Frage danach, welche Grammatikkonzepte durch die DSTL-Generierung berücksichtigt werden müssen. In Kapitel 4 wurden die Syntax und die Semantik der verschiedenen Operatoren und Features der DSTLs beschrieben, die entsprechend der in dieser Arbeit vorgestellten Ableitung abgeleitet bzw. mittels MontiTrans generiert werden. Zusätzlich wurden hier die Kontextbedingungen, die Transformationen erfüllen müssen, damit sie wohlgeformt sind, erklärt. Somit wird durch die Ergebnisse, die in diesem Kapitel beschrieben wurden, die Forschungsfrage RQ1.2, also die Frage danach, welche Features eine generierte DSTL bieten muss, beantwortet. Des Weiteren wurden in Kapitel 6 die Erweiterungsmöglichkeiten generierter DSTLs bzw. der zugehörigen ebenfalls generierten Java-Generatoren vorgestellt. Die Generierung durch diese Java-Generatoren wurde in Kapitel 8 erklärt. Zur Erweiterung dieser generierten Generatoren wurde der TOP-Mechanismus, der auch innerhalb von MontiCore verwendet wird, für das Generat umgesetzt, sodass das Generat um handgeschriebenen Code erweitert oder durch diesen ersetzt werden kann. Damit beantworten diese Kapitel die Forschungsfrage RQ1.3 und RQ2, also die Fragen danach, wie generierte DSTLs oder deren zugehörige Infrastruktur an die Bedürfnisse der Nutzer angepasst werden können. Zusammengenommen beantworten die Kapitel 4, 6 und 8 damit die Forschungsfrage RQ1, also die Frage danach, wie die Generierung von DSTLs verbessert werden kann.

In Kapitel 5 wurden die DSTLs CDTrans, MATrans und MACDTrans zur Definition von Transformationen für Klassendiagramme der Modellierungssprachen CD4Analysis und CD4Code und MontiArc-Modelle vorgestellt. Diese wurden mithilfe von Monti-

Trans entwickelt und handgeschrieben an die zugrundeliegenden Modellierungssprachen angepasst. Darüber hinaus wurden hier auch DSTLs für die Basissprachen von MontiCore vorgestellt und erklärt wie diese zur Entwicklung von CDTrans und MATrans verwendet wurden. Schließlich wurde gezeigt, wie MACDTrans durch die Kombination von CDTrans und MATrans entwickelt wurde. In Kapitel 7 wurden wiederverwendbare Modelltransformationen für Klassendiagramme und MontiArc-Modelle vorgestellt. Diese wurden mithilfe der DSTLs CDTrans und MATrans entwickelt und zu Bibliotheken zusammengefasst. Die Bibliotheken enthalten Transformationen zum Refactoring von Klassendiagrammen, zur Normalisierung von MontiArc-Modellen sowie zur besseren Interoperabilität mit ROS. Diese Transformationen dienen sowohl Modellierern zur Verwendung als auch Transformationsentwicklern zur Entwicklung neuer Transformationen. Hierbei können sie entweder zu neuen Transformationen kombiniert werden oder als Vorlage zur Entwicklung eigener Transformationen dienen. Damit beantworten die Kapitel 5 und 7 die Forschungsfrage, RQ3 und RQ4, also die Fragen, wie DSTLs für die Sprachen MontiArc, CD4Analysis und CD4Code entwickelt und deren Verwendung durch wiederverwendbare Modelltransformationen unterstützt werden kann.

In Kapitel 9 wurden Methodiken für die verschiedenen Ebenen der Transformationssprachen- und Transformationsentwicklung vorgestellt. Für den Sprachentwickler wurde ein Methodik zur Verwendung des MontiTrans DSTL-Generators entwickelt, für den Transformationsentwickler hingegen eine Methodik zur Entwicklung neuer Transformationen. Für den Modellierer wurde erläutert, wie die Transformationen verwendet und kombiniert werden können. Damit schließt dieses Kapitel eine Klammer über alle vorhergehenden Kapitel. Insbesondere wurde hier auch gezeigt, wie die Integration mit der templatebasierten Generierung durch das Füllen von Hookpoints und die Verwendung der Templateerweiterung funktioniert. In Kombination mit der Syntax zur Verwendung der Templateerweiterung und der Hookpoints, die in Kapitel 4 vorgestellt wurde, beantwortet dieses Kapitel die letzte Forschungsfrage RQ5.

Nachdem alle Teilforschungsfragen RQ1–RQ5, wie zuvor beschrieben, in den verschiedenen Kapiteln beantwortet wurden, kann damit die übergeordnete Fragestellung beantwortet werden. Modellierer können innerhalb der modellgetriebenen Entwicklung durch domänenspezifische Transformationen unterstützt werden, indem ihnen ausführbare Versionen der Transformationen bereitgestellt werden. Zur Dokumentation eignen sich die mithilfe der DSTLs entwickelten Transformationen, da diese die konkrete Syntax der Modellierungssprache – die Syntax, die dem Modellierer bekannt ist – angereichert mit Transformationsoperatoren verwenden. Transformationsentwickler können innerhalb eines agilen, modellgetriebenen Entwicklungsprozesses durch DSTLs unterstützt werden, indem sie von Sprachentwicklern zu den verwendeten Modellierungssprachen entsprechende DSTLs bekommen. Durch Verwendung der Modellsyntax verringert sich der Aufwand zur Einarbeitung in die Transformationssprache, da die interne Modellrepräsentation nicht zunächst tiefgehend verstanden werden muss. Außerdem verringert sich der Dokumentationsaufwand, da die Transformationsregeln in konkreter Syntax bereits eine gute Grundlage zur Dokumentation bieten. Schließlich kann der Sprachentwickler

durch die Generierung von DSTLs unterstützt werden, da zu jeder neuen DSL auch eine DSTL entwickelt werden muss. Dieser Aufwand reduziert sich sowohl durch die Systematik der Ableitung, aber insbesondere durch die Generierung von DSTLs. Zusätzlich hat er durch die Erweiterungsmechanismen von MontiTrans die Möglichkeit die generierte DSTL noch stärker an die Bedürfnisse der Nutzer anzupassen.

Es wurde gezeigt, dass MontiTrans ein umfassendes Werkzeug zur Entwicklung von DSTLs sowie für die Entwicklung von Modelltransformationen innerhalb modellgetriebener Softwareentwicklungsprojekte ist. MontiTrans erleichtert sowohl Sprachentwicklern die Entwicklung neuer DSLs und zugehöriger DSTLs als auch Transformationsentwicklern die Definition und Anwendung neuer Transformationen.

MontiTrans bildet damit die Grundlage zur Beantwortung bzw. Untersuchung weiterer Fragestellungen im Bereich der Modelltransformations- und Transformationssprachenentwicklung. Im Folgenden werde einige mögliche Anknüpfungspunkte vorgestellt.

Entwicklung weiterer Transformationssprachen

In dieser Arbeit wurden einige auf der Ableitung von DSTLs basierende Transformationssprachen vorgestellt. Dennoch gibt es noch eine Vielzahl von Sprachen, die auf MontiCore basieren, zu denen es noch keine zugehörige DSTL gibt. Aufbauend auf den Ergebnissen dieser Arbeit können daher zu bereits existierenden Modellierungssprachen ebenfalls Transformationssprachen mit MontiTrans entwickelt werden. Des Weiteren kann die Generierung von Transformationssprachen in den Generierungsablauf für MontiCore Sprachen integriert werden. Dadurch bekommt der Sprachentwickler neben Lexer, Parser, Symboltabelle und Infrastruktur für Kontextbedingungen auch direkt eine passende Transformationssprache für die neu entwickelte Sprache geliefert.

Neben der automatischen Generierung von endogenen DSTLs für neu entwickelte Sprachen sollte auch der Bedarf an exogenen DSTLs ermittelt werden. Hierbei wäre zu prüfen, für welche Kombinationen vom Modellierungssprachen DSTLs benötigt werden und inwiefern es zielführend ist, DSTLs für Kombinationen aus mehr als zwei Modellierungssprachen bereits zustellen.

Ausbau der Bibliotheken von wiederverwendbaren Transformationen

Mithilfe der DSTLs CDTrans und MATrans wurden im Rahmen dieser Arbeit wiederverwendbare Transformationen entwickelt. Diese dienen dem Refactoring, der Normalisierung und der besseren Interoperabilität mit ROS. Bei diesem Transformationen handelt es sich um semantikerhaltende Transformationen. Mithilfe der DSTLs lassen sich jedoch auch Evolution, Refinement, Abstraktion oder Migration mittels Transformationen beschreiben. Nachfolgende Arbeiten könnten untersuchen, welche Transformationen aus diesem Bereich zur Wiederverwendung geeignet sind und diese entsprechen den Bibliotheken hinzufügen. Sofern weitere DSTLs mit MontiTrans entwickelt werden, können zusätzlich auch weitere Bibliotheken, die zu diesen DSTLs gehören, entwickelt werden.

Mappings für Exogene Transformationen

Exogene Transformationssprachen können verschiedene Ziele haben. So kann beispielsweise die Koevolution von Modellen unterstützt werden. Exogene DSTLs können andererseits auch zur Übersetzung von Modellen zwischen Sprachen genutzt werden. Hierbei kann sowohl zwischen grundlegend verschiedenen Sprachen wie beispielsweise Klassendiagrammen und Entity-Relationship Diagrammen übersetzt werden oder zwischen ähnlichen Sprachen, beispielsweise weil eine Sprache weiterentwickelt wurde und existierende Modelle auf die neue Sprachversion migriert werden müssen. Neben der beschriebenen Bedarfsanalyse und Entwicklung von exogenen DSTLs sollte daher auch analysiert werden, wie die Übersetzung zwischen den involvierten Sprachen realisiert werden kann. Basierend darauf können anschließend ein oder mehrere Mappings zur Übersetzung zwischen den Sprachen zur Wiederverwendung bereitgestellt werden.

Kardinalitäten für den Collection Operator

Wie in Abschnitt 4.7 beschrieben, erlaubt der Collection Operator das Matchen von beliebig vielen ähnlichen Teilgraphen. Hierbei wurde festgelegt, dass der Collection Operator das Pattern um die Möglichkeit erweitert, statt nur einem mehrere ähnliche Teilgraphen zu finden. Diese Entscheidung wurde auf Grundlage von Erfahrungswerten aus der Entwicklung von Transformationen für die Transformationsbibliotheken getroffen. Dennoch sollte untersucht werden, ob eine Erweiterung des Operators um Kardinalitäten [GKMP13], die die Angabe der minimal und maximal zu findenden Teilgraphen erlaubt, die Benutzerfreundlichkeit für Transformationsentwickler weiter erhöhen kann. Dazu muss sowohl die konkrete Syntax der Transformationssprache erweitert werden als auch das Pattern Matching entsprechend angepasst werden.

Transitive Hülle und Sternoperator für DSTLs

Für rekursive oder hierarchische Strukturen kann es von Interesse sein, Hierarchien unbekannter Tiefe anzugeben, ähnlich wie es für verbindende Elemente wie Assoziationen oder Konnektoren von Interesse sein kann, Pfade unbekannter Länge zu spezifizieren. Dies kann beispielsweise genutzt werden, um die Verbindung durch Assoziationen zwischen zwei Klassen matchen zu können. Mittels der DSTLs kann dies bislang nur umständlich ausgedrückt werden, indem der Constraint oder die Möglichkeit zur direkten Manipulation genutzt werden. Hier sollte untersucht werden, wie die DSTLs erweitert werden können, um solche Pattern leichter und domänenspezifisch beschreiben zu können. Eine Möglichkeit, die hierfür in Betracht gezogen werden kann, ist der Sternoperator [LLP07], der es erlaubt hierarchische Strukturen zu matchen. Insbesondere sollte hierbei darauf geachtet werden, dass die Komplexität der Sprache weiterhin für Domänenexperten und Transformationsentwickler beherrschbar bleibt.

Weitere Optimierung der Performance des Pattern Matching

MontiTrans verwendet einen modellsensitiven Pattern Matching Ansatz, der auf einem im Vorfeld der Suche berechneten, modellabhängigen Suchplan basiert. In dieser Arbeit wurde die Performance des Pattern Matching verbessert, indem die zentralisierte Überprüfung von Constraints durch eine dezentralisierte Überprüfung sowie eine automatische Modularisierung ersetzt wurde. Wie in Kapitel 8 diskutiert, hängt die Modularisierung von der Formulierung des Constraint ab. Eine mögliche Erweiterung der Modularisierung könnte daher eine automatisierte Umformung des Constraint in konjunktive Normalform sein, da diese in ihre konjugierten Terme zerlegt und damit modularisiert werden kann. Dieser Schritt obliegt in der bisherigen Umsetzung dem Transformationsentwickler. Darüber hinaus kann die Performance möglicherweise weiter verbessert werden, indem das Pattern Matching noch stärker an die Modellstrukturen angepasst wird. In der aktuellen MontiCore Version bilden die AST-Instanzen nicht wie in vorherigen Versionen Graphen mit spannendem Baum sondern echte Baumstrukturen. Wird dies für MontiCore beibehalten, kann untersucht werden, inwiefern baumoptimierte Matching Strategien wie beispielsweise das Subtree Matching [ST97] zu einer Performanceverbesserung führen.

Umstellung der Grammatikableitung auf Transformationen

Die Implementierung der Grammatikgenerierung durch MontiTrans ist in Java vorgenommen. Angelehnt an Grammatikadaptions- bzw. Grammatiktransformationsansätze [Läm01, DCMS02, KLV05] könnte eine mögliche Erweiterung von MontiTrans daher sein, diese Ableitung ebenfalls durch Transformationen zu lösen. Die MontiCore Grammatiksprache ist ebenfalls durch eine MontiCore Grammatik realisiert. Daher kann mithilfe von MontiTrans eine Grammatiktransformationssprache entwickelt werden. Anschließend kann diese DSTL genutzt werden, um die in Kapitel 6 beschriebenen Ableitungsregeln mittels Transformationen zu realisieren. Hierdurch ergibt sich ein Bootstrappingansatz für die Entwicklung von MontiTrans ähnlich wie es bei MontiCore der Fall ist. Der eigentlichen Umsetzung sollte daher eine Evaluation vorausgehen, inwiefern sich dies auf die Wartbarkeit und Evolution von MontiTrans auswirkt und gegebenenfalls eine Methodik zur Evolution erarbeitet werden.

Literaturverzeichnis

[ABC+14] Lorenz Anardu, Roberto Baldi, Umberto A. Cicero, Riccardo Giomi und Giacomo Veneri. *Maven Build Customization.* Community experience distilled. Packt Publishing, 2014.

[ABJ+10] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause und Gabriele Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In *Proceedings of MoDELS'10*, LNCS 6394, Seiten 121–135. Springer, 2010.

[AHRW17a] Kai Adam, Katrin Hölldobler, Bernhard Rumpe und Andreas Wortmann. Engineering Robotics Software Architectures with Exchangeable Model Transformations. In *International Conference on Robotic Computing (IRC'17)*, Seiten 172–179. IEEE, April 2017.

[AHRW17b] Kai Adam, Katrin Hölldobler, Bernhard Rumpe und Andreas Wortmann. Modeling Robotics Software Architectures with Modular Model Transformations. *Journal of Software Engineering for Robotics (JOSER)*, 8(1):3–16, 2017.

[ALC08] Bastien Amar, Hervé Leblanc und Bernard Coulette. A Traceability Engine Dedicated to Model Transformation for Software Engineering. *ECMDA Traceability Workshop (ECMDA-TW) 2008*, Seiten 7–16, 2008.

[Are14] Thorsten Arendt. *Quality Assurance of Software Models - A Structured Quality Assurance Process Supported by a Flexible Tool Environment in the Eclipse Modeling Project.* Dissertation, Philipps-Universität Marburg, 2014.

[ASS16] Vlad Acreţoaie, Harald Störrle und Daniel Strüber. VMTL: A language for end-user model transformation. *Software & Systems Modeling*, 2016.

[AT13] Thorsten Arendt und Gabriele Taentzer. A tool environment for quality assurance based on the Eclipse Modeling Framework. *Automated Software Engineering*, 20(2):141–184, 2013.

[BBB+85] Friedrich Ludwig Bauer, Rudolf Berghammer, Manfred Broy, Walter Dosch, Franz Geiselbrechtinger, Rupert Gnatz, E. Hangel, Wolfgang Hesse, Bernd Krieg-Brückner, Alfred Laut, Thomas Matzner, Bernhard Möller, Friederike Nickl, Helmuth Partsch, Peter Pepper, Klaus Samelson, Martin Wirsing und Hans Wössner. *The Munich Project CIP: Volume I: The Wide Spectrum Language CIP-L.* LNCS. Springer, 1985.

[BBB+01] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries et al. Agile Manifesto. `http://agilemanifesto.org/`, 2001. [Online; abgerufen am: 02.10.2017].

[BCW12] Marco Brambilla, Jordi Cabot und Manuel Wimmer. Model-driven Software Engineering in Practice. *Synthesis Lectures on Software Engineering*, 1(1):1–182, 2012.

[BDH+15] Gábor Bergmann, István Dávid, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi und Dániel Varró. Viatra 3: A Reactive Model Transformation Platform. In Dimitris Kolovos und Manuel Wimmer, Editoren, *Theory and practice of model transformations*, LNCS 9152, Seiten 101–110. Springer, 2015.

[BEH+87] Friedrich L. Bauer, Herbert Ehler, A. Horsch, Bernhard Möller, Helmuth Partsch, O. Paukner und Peter Pepper. *The Munich Project CIP, Volume II: The Program Transformation System CIP-S*, LNCS 292. Springer, 1987.

[Bej13] Bill Bejeck. *Getting Started with Google Guava*. Packt Publishing, 2013.

[BEK+07] Enrico Biermann, Karsten Ehrig, Christian Köhler, Günter Kuhns, Gabriele Taentzer und Eduard Weiss. EMF Model Refactoring based on Graph Transformation Concepts. *Electronic Communications of the EASST*, 3, 2007.

[BGS14] Jeffrey M. Barnes, David Garlan und Bradley Schmerl. Evolution styles: foundations and models for software architecture evolution. *Software & Systems Modeling*, 13(2):649–678, 2014.

[BH02] Luciano Baresi und Reiko Heckel. Tutorial Introduction to Graph Transformation: A Software Engineering Perspective. In Andrea Corradini, Editor, *Graph transformation*, LNCS 2505, Seiten 402–429. Springer, 2002.

[Bie10] Matthias Biehl. Literature Study on Model Transformations. Technical Report ISRN/KTH/MMK/R-10/07-SE, Royal Institute of Technology, 2010.

[BKG08] Gernot Veit Batz, Moritz Kroll und Rubino Geiß. A First Experimental Evaluation of Search Plan Driven Graph Pattern Matching. In *Applications of Graph Transformations with Industrial Relevance*, LNCS 5088, Seiten 471–486. Springer, 2008.

[BKVV08] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas und Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of computer programming*, 72(1):52–70, 2008.

[BLMDL08] Olivier Barais, Anne Françoise Le Meur, Laurence Duchien und Julia Lawall. Software Architecture Evolution. In *Software Evolution*, Seiten 233–262. Springer, 2008.

[BLS+09] Petra Brosch, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, Gerti Kappel, Werner Retschitzegger und Wieland Schwinger. An Example Is Worth a Thousand Words: Composite Operation Modeling By-Example. In Andy Schürr und Bran Selic, Editoren, *Model Driven Engineering Languages and Systems: 12th International Conference*, Seiten 271–285. Springer, 2009.

[BNN+07] Daniel Balasubramanian, Anantha Narayanan, Sandeep Neema, Feng Shi, Ryan Thibodeaux und Gabor Karsai. A Subgraph Operator for Graph Transformation Languages. *Electronic Communications of the EASST*, 6, 2007.

[BNvBK06] Daniel Balasubramanian, Anantha Narayanan, Christopher P. van Buskirk und Gabor Karsai. The Graph Rewriting and Transformation Language: GReAT. *ECEASST*, 1, 2006.

[BR07] Manfred Broy und Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007.

[Bro98] William J. Brown. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley computer publishing. Wiley & Sons, 1998.

[BRW16] Arvid Butting, Bernhard Rumpe und Andreas Wortmann. Embedding Component Behavior DSLs into the MontiArcAutomaton ADL. In *Globalization of Modeling Languages Workshop (GEMOC'16)*, CEUR Workshop Proceedings 1731, Saint Malo, France, October 2016.

[BS01] Manfred Broy und Ketil Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Monographs in Computer Science. Springer, 2001.

[BSF03] Marko Boger, Thorsten Sturm und Per Fragemann. Refactoring Browser for UML. In Mehmet Aksit, Mira Mezini und Rainer Unland, Editoren, *Objects, Components, Architectures, Services, and Applications for a Networked World*, LNCS 2591, Seiten 366–377. Springer, 2003.

[Bur14] Erik Burger. *Flexible Views for View-based Model-driven Development*, The Karlsruhe Series on Software Design and Quality 15. KIT Scientific Publishing, 2014.

[BW07] Thomas Baar und Jon Whittle. On the Usage of Concrete Syntax in Model Transformation Rules. In *Perspectives of Systems Informatics*, LNCS. Springer, 2007.

[CDEP08] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo und Alfonso Pierantonio. Automating Co-evolution in Model-Driven Engineering. In *12th International IEEE Enterprise Distributed Object Computing Conference*, Seiten 222–231. IEEE, 2008.

[CE00] Krzysztof Czarnecki und Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[CFH+09] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr und James F. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In *Theory and Practice of Model Transformations, Second International Conference, ICMT*, Seiten 260–283, 2009.

[CGdL15] Jesús Sánchez Cuadrado, Esther Guerra und Juan de Lara. Reusable Model Transformation Components with bentō. In Dimitris Kolovos und Manuel Wimmer, Editoren, *Theory and Practice of Model Transformations*, LNCS 9152, Seiten 59–65. Springer, 2015.

[CH05] Anthony Cleve und Jean-Luc Hainaut. Co-transformations in Database Applications Evolution. In *Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE*, Seiten 409–421, 2005.

[CH06] Krzysztof Czarnecki und Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[Chr05] Alexander Christoph. Describing Horizontal Model Transformations with Graph Rewriting Rules. In Uwe Aßmann und Mehmet Aksit, Editoren, *Model Driven Architecture*, LNCS 3599, Seiten 93–107. Springer, 2005.

[CMIC15] Javier Criado, Salvador Martínez, Luis Iribarne und Jordi Cabot. Enabling the Reuse of Stored Model Transformations Through Annotations. In Dimitris Kolovos und Manuel Wimmer, Editoren, *Theory and Practice of Model Transformations*, LNCS 9152, Seiten 43–58. Springer, 2015.

[Coo71] Stephen A. Cook. The Complexity of Theorem-Proving Procedures. In Michael A. Harrison, Ranan B. Banerji und Jeffrey D. Ullman, Editoren, *Proceedings of the third annual ACM symposium on Theory of computing - STOC '71*, Seiten 151–158. ACM Press, 1971.

[CSE16] Jonathan Corley, Eugene Syriani und Huseyin Ergin. Evaluating the cloud architecture of AToMPM. In *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, Seiten 339–346, 2016.

[Cua12] Jesús Sánchez Cuadrado. Towards a Family of Model Transformation Languages. In Zhenjiang Hu und Juan de Lara, Editoren, *Theory and practice of model transformations*, SpringerLink Bücher 7307, Seiten 176–191. Springer, 2012.

[CV07] Alcino Cunha und Joost Visser. Strongly Typed Rewriting For Coupled Software Transformation. *Electronic Notes in Theoretical Computer Science*, 174(1):17–34, 2007.

[CYD+08] Jiefeng Cheng, Jeffrey Xu Yu, Bolin Ding, Philip S. Yu und Haixun Wang. Fast Graph Pattern Matching. In *IEEE 24th International Conference on Data Engineering, 2008*, Seiten 913–922. IEEE, 2008.

[DCMS02] Thomas R. Dean, James R. Cordy, Andrew J. Malton und Kevin A. Schneider. Grammar programming in txl. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation*, SCAM '02, Seiten 93–. IEEE Computer Society, 2002.

[DJ06] Danny Dig und Ralph Johnson. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):83–107, 2006.

[dLETE04] Juan de Lara, Claudia Ermel, Gabriele Taentzer und Karsten Ehrig. Parallel Graph Transformation for Model Simulation applied to Timed Transition Petri Nets. *Electronic Notes in Theoretical Computer Science*, 109:17–29, 2004.

[dLG14] Juan de Lara und Esther Guerra. Towards the flexible reuse of model transformations: A formal approach based on graph transformation. *Journal of Logical and Algebraic Methods in Programming*, 83(5-6):427–458, 2014.

[Eck16] Kathrin Eckhardt. Eine Bibliothek wiederverwendbarer, domänenspezifischer Modelltransformationen für Klassendiagramme. Masterarbeit, RWTH Aachen, Software Engineering, Deutschland, 2016.

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange und Gabriele Taentzer. *Fundamentals of algebraic graph transformation*. Monographs in theoretical computer science. Springer, 2006.

[EGK+02] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North und Gordon Woodhull. Graphviz — Open Source Graph Drawing Tools. In *Graph Drawing*, Seiten 483–484, 2002.

[EH86] Hartmut Ehrig und Annegret Habel. Graph Grammars with Application Conditions. In *The Book of L*, Seiten 87–100. Springer, 1986.

[EHR+13] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo und Hüseyin Ergin. AToMPM: A Web-based Modeling Environment. In *MODELS'13: Invited Talks, Demos, Posters, and ACM SRC*, Seiten 21–25, 2013.

[EM12] Hartmut Ehrig und Bernd Mahr. *Fundamentals of algebraic specification 1: Equations and initial semantics*. Springer, 2012.

[ERK99] Hartmut Ehrig, Grzegorz Rozenberg und Hans-Jörg Kreowski. *Handbook of Graph Grammars and Computing by Graph Transformation*. Concurrency, parallelism, and distribution. World Scientific, 1999.

[FHFB10] Jay Fields, Shane Harvie, Martin Fowler und Kent Beck. *Refactoring*. Addision-Wesley professional Ruby series. Addison-Wesley, ruby ed. Edition, 2010.

[Fie00] Roy Thomas Fielding. Architectural Styles and the Design of Network-Based Software Architectures. *Dissertation UCI*, 2000.

[FK05] William B. Frakes und Kyo Kang. Software Reuse Research: Status and Future. *IEEE Transactions on Software Engineering*, 31(7):529–536, 2005.

[FLB09] Andrew Forward, Timothy C. Lethbridge und Dusan Brestovansky. Improving Program Comprehension by Enhancing Program Constructs: An Analysis of the Umple language. In *IEEE 17th International Conference on Program Comprehension, 2009*, Seiten 311–312. IEEE, 2009.

[FNTZ00] Thorsten Fischer, Jörg Niere, Lars Torunski und Albert Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In *Theory and Application of Graph Transformations*, LNCS 1764, Seiten 296–309. Springer, 2000.

[For13] Charles Forsythe. *Instant FreeMarker Starter*. Packt Publishing, 2013.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[Fow06] Martin Fowler. *Analysis Patterns: Reusable Object Models*. The Addison-Wesley object technology series. Addison-Wesley, 19. Edition, 2006.

[Fow10] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st Edition, 2010.

[FR11] Martin Fowler und David Rice. *Patterns of Enterprise Application Architecture*. The Addison-Wesley signature series. Addison-Wesley, 17. Edition, 2011.

[Fra16] Linus Franke. Weiterentwicklung des Collection Operators domänenspezifischer Transformationssprachen in MontiCore zur Unterstützung hierarchischer Strukturen. Bachelorarbeit, RWTH Aachen, Software Engineering, Deutschland, 2016.

[Fre17] Freemarker. `http://freemarker.org/`, 2017. [Online; abgerufen am: 17.07.2017].

[FT08] Christian Fuss und Verena E. Tuttlies. Simulating Set-Valued Transformations with Algorithmic Graph Transformation Languages. In Andy Schürr, Manfred Nagl und Albert Zündorf, Editoren, *Applications of graph transformations with industrial relevance*, LNCS 5088, Seiten 442–455. Springer, 2008.

[GBA14] Cláudio Gomes, Bruno Barroca und Vasco Amaral. Classification of Model Transformation Tools: Pattern Matching Techniques. In Jürgen Dingel, Editor, *Model-driven engineering languages and systems*, LNCS 8767, Seiten 619–635. Springer, 2014.

[GBG+06] Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack und Adam Szalkowski. GrGen: A Fast SPO-Based Graph Rewriting Tool. In Andrea Corradini, Editor, *Graph Transformation*, LNCS 4178, Seiten 383–397. Springer, 2006.

[GdL07] Esther Guerra und Juan de Lara. Adding Recursion to Graph Transformation. *Electronic Communications of the EASST*, 6, 2007.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1995.

[GHK+15a] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler und Andreas Wortmann. A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, Seiten 74–85. SciTePress, 2015.

[GHK+15b] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler und Andreas Wortmann. Integration of Handwritten and Generated Object-Oriented Code. In *Model-Driven Engineering and Software Development*, Communications in Computer and Information Science 580, Seiten 112–132. Springer, 2015.

[GKMP13] Roy Grønmo, Stein Krogdahl und Birger Møller-Pedersen. A Collection Operator for Graph Transformation. *Software & Systems Modeling*, 12(1):121–144, 2013.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler und Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, Seiten 925–926, 2008.

[GLRR15] Timo Greifenberg, Markus Look, Sebastian Roidl und Bernhard Rumpe. Engineering Tagging Languages for DSLs. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, Seiten 34–43. ACM/IEEE, 2015.

[GMP09] Roy Grønmo und Birger Møller-Pedersen. Concrete Syntax-based Graph Transformation, 2009. Research Report 389.

[GMPO09a] Roy Grønmo, Birger Møller-Pedersen und GÃ¸ranK. Olsen. Comparison of Three Model Transformation Languages. In *Model Driven Architecture - Foundations and Applications*, LNCS 5562, Seiten 2–17. Springer, 2009.

[GMPO09b] Roy Grønmo, Birger Møller-Pedersen und Gøran K. Olsen. Comparison of Three Model Transformation Languages. In *Model Driven Architecture - Foundations and Applications*, LNCS 5562, Seiten 2–17. Springer, 2009.

[GrG17] GrGen.NET. `http://www.info.uni-karlsruhe.de/software/grgen/`, 2017. [Online; abgerufen am: 31.08.2017].

[Grø09] Roy Grønmo. *Using concrete syntax in graph-based model transformations*. Dissertation, University of Oslo, 2009.

[Gru05] Lars Grunske. Formalizing Architectural Refactorings as Graph Transformation Systems. *SNPD/SAWN '05*, Seiten 324–329, 2005.

[GZ06] Leif Geiger und Albert Zündorf. Tool Modeling with Fujaba. *Electronic Notes in Theoretical Computer Science*, 148(1):173–186, 2006.

[Hab16] Arne Haber. *MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems*. Aachener Informatik-Berichte, Software Engineering, Band 24. Shaker Verlag, September 2016.

[HBJ08] Markus Herrmannsdoerfer, Sebastian Benz und Elmar Juergens. Automatability of Coupled Evolution of Metamodels and Models in Practice. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl und Markus Völter, Editoren, *Model Driven Engineering Languages and Systems*, LNCS 5301, Seiten 645–659. Springer, 2008.

[HHK+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe und Ina Schaefer. Engineering Delta Modeling Languages. In *Software Product Line Conference (SPLC'13)*, Seiten 22–31. ACM, 2013.

[HHK+15] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, Ina Schaefer und Christoph Schulze. Systematic Synthesis of Delta Modeling Languages. *Journal on Software Tools for Technology Transfer (STTT)*, 17(5):601–626, October 2015.

[HHRW15] Lars Hermerschmidt, Katrin Hölldobler, Bernhard Rumpe und Andreas Wortmann. Generating Domain-Specific Transformation Languages for Component & Connector Architecture Descriptions. In *Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp'15)*, CEUR Workshop Proceedings 1463, Seiten 18–23, 2015.

[HHT96] Annegret Habel, Reiko Heckel und Gabriele Taentzer. Graph Grammars with Negative Application Conditions. *Fundamenta Informaticae*, 26(3):287–313, 1996.

[HJvE06] Berthold Hoffmann, Dirk Janssens und Niels van Eetvelde. Cloning and Expanding Graph Transformation Rules for Refactoring. *Electronic Notes in Theoretical Computer Science*, 152:53–67, 2006.

[HKR+11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe und Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *Software Architecture Conference (ECSA'11)*, Seiten 6:1–6:10. ACM, 2011.

[HKR+16] Robert Heim, Oliver Kautz, Jan Oliver Ringert, Bernhard Rumpe und Andreas Wortmann. Retrofitting Controlled Dynamic Reconfiguration into the Architecture Description Language MontiArcAutomaton. In *Software Architecture - 10th European Conference (ECSA'16)*, LNCS 9839, Seiten 175–182, Copenhagen, Denmark, December 2016. Springer.

[HKV12] Mark Hills, Paul Klint und Jurgen J. Vinju. Scripting a refactoring with Rascal and Eclipse. In Peter Sommerlad, Editor, *Proceedings of the Fifth Workshop on Refactoring Tools*, Seiten 40–49. ACM, 2012.

[HLMSN+15a] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel und Andreas Wortmann. Composition of Heterogeneous Modeling Languages. In *Model-Driven Engineering and Software Development*, Communications in Computer and Information Science 580, Seiten 45–66. Springer, 2015.

[HLMSN+15b] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel und Andreas Wortmann. Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, Seiten 19–31. SciTePress, 2015.

[HMSNRW16] Robert Heim, Pedram Mir Seyed Nazari, Bernhard Rumpe und Andreas Wortmann. Compositional Language Engineering using Generated, Extensible, Static Type Safe Visitors. In *Conference on Modelling Foundations and Applications (ECMFA)*, LNCS 9764, Seiten 67–82. Springer, July 2016.

[HP05] Annegret Habel und Karl-Heinz Pennemann. Nested Constraints and Application Conditions for High-Level Structures. In Hans-Jörg Kreowski, Ugo Montanari, Fernando Orejas, Grzegorz Rozenberg und Gabriele Taentzer, Editoren, *Formal Methods in Software and Systems Modeling: Essays Dedicated to Hartmut Ehrig on the Occasion of His 60th Birthday*, Seiten 293–308. Springer, 2005.

[HR04] David Harel und Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004.

[HRR10] Arne Haber, Jan Oliver Ringert und Bernhard Rumpe. Towards Architectural Programming of Embedded Systems. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter-Systeme VI*, Informatik-Bericht 2010-01, Seiten 13 – 22. fortiss GmbH, Germany, 2010.

[HRR12] Arne Haber, Jan Oliver Ringert und Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012.

[HRRW17] Katrin Hölldobler, Alexander Roth, Bernhard Rumpe und Andreas Wortmann. Advances in Modeling Language Engineering. In *International Conference on Model and Data Engineering*, Seiten 3–17. Springer, 2017.

[HRW11] John Hutchinson, Mark Rouncefield und Jon Whittle. Model-driven Engineering Practices in Industry. In *33rd International Conference on Software Engineering (ICSE)*, Seiten 633–642, 2011.

[HRW15] Katrin Hölldobler, Bernhard Rumpe und Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, Seiten 136–145. ACM/IEEE, 2015.

[HVW11] Markus Herrmannsdoerfer, Sander D. Vermolen und Guido Wachsmuth. An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models. In Brian Malloy, Steffen Staab und Mark van den Brand, Editoren, *Software Language Engineering*, LNCS 6563, Seiten 163–182. Springer, 2011.

[HWRK11] John Hutchinson, Jon Whittle, Mark Rouncefield und Steinar Kristoffersen. Empirical Assessment of MDE in Industry. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, Seiten 471–480. ACM, 2011.

[ISH08] Maria-Eugenia Iacob, Maarten W. A. Steen und Lex Heerink. Reusable Model Transformation Patterns. In *12th Enterprise Distributed Object Computing Conference workshops*, Seiten 1–10. IEEE, 2008.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin und Ivan Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1):31–39, 2008.

[JBK10] Edgar Jakumeit, Sebastian Buchwald und Moritz Kroll. GrGen.NET. *Software Tools for Technology Transfer*, 12(3-4):263–271, 2010.

[Jet17] JetBrains. IntelliJ IDEA: The Java IDE for Professional Developers by JetBrains. `https://www.jetbrains.com/idea/`, 2017. [Online; abgerufen am: 27.10.2017].

[JK06] Frédéric Jouault und Ivan Kurtev. Transforming models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, LNCS, Seiten 128–138. Springer, 2006.

[KBC05] Audris Kalnins, Janis Barzdins und Edgars Celms. Model Transformation Language MOLA. In Uwe Aßmann und Mehmet Aksit, Editoren, *Model Driven Architecture*, LNCS 3599, Seiten 62–76. Springer, 2005.

[KC15] Nafiseh Kahani und James R. Cordy. Comparison and Evaluation of Model Transformation Tools. Technical Report 2015-627, School of Computing, Queen's University, 2015.

[KK04] Günter Kniesel und Helge Koch. Static composition of refactorings. *Science of Computer Programming*, 52:9–51, 2004.

[KKB+99] Mark Klein, Rick Kazman, Len Bass, Jeromy Carriere, Mario Barbacci und Howard Lipson. Attribute-Based Architecture Styles: Attribute-Based Architecture Styles. In *Proceedings of WICSA*, Seiten 225–243. Springer, 1999.

[KKL+15] Dierk Knig, Paul King, Guillaume Laforge, Hamlet D'Arcy, Cédric Champeau, Erik Pragt und Jon Skeet. *Groovy in Action*. Manning Publications Co, 2015.

[KLR+12] Gerti Kappel, Philip Langer, Werner Retschitzegger, Wieland Schwinger und Manuel Wimmer. Model Transformation By-Example: A Survey of the First Wave. In *Conceptual Modelling and Its Theoretical Foundations*, LNCS 7260, Seiten 197–215. Springer, 2012.

[KLV05] Paul Klint, Ralf Lämmel und Chris Verhoef. Toward an Engineering Discipline for Grammarware. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(3):331–380, 2005.

[KMS+10] Thomas Kühne, Gergely Mezei, Eugene Syriani, Hans Vangheluwe und Manuel Wimmer. Explicit Transformation Modeling. In *Models in Software Engineering*, LNCS 6002, Seiten 240–255. Springer, 2010.

[Kos09] Piotr Kosiuczenko. Redesign of UML class diagrams: A formal approach. *Software & Systems Modeling*, 8(2):165–183, 2009.

[KR05] Vinay Kulkarni und Sreedhar Reddy. Model-driven Development of Enterprise Applications. In *UML Modeling Languages and Applications*, Seiten 118–128. Springer, 2005.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, März 2010.

[KRLP+14] Shekoufeh Kolahdouz-Rahimi, Kevin Lano, Suresh Pillay, Javier Troya und Pieter Van Gorp. Evaluation of model transformation approaches for model refactoring. *Science of Computer Programming*, 85:5–40, 2014.

[KRV08] Holger Krahn, Bernhard Rumpe und Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe'08)*, LNBIP 11, Seiten 297–315. Springer, 2008.

[KRV10] Holger Krahn, Bernhard Rumpe und Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

[KTRK16] Timo Kehrer, Gabriele Taentzer, Michaela Rindt und Udo Kelter. Automatically Deriving the Specification of Model Editing Operations from Meta-Models. In *Theory and Practice of Model Transformations: 9th International Conference, ICMT 2016*, Seiten 173–188. Springer, 2016.

[Küh05] Thomas Kühne. What is a Model? In Jean Bezivin und Reiko Heckel, Editoren, *Language Engineering for Model-Driven Software Development*, Nummer 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), 2005.

[Küh06] Thomas Kühne. Matters of (Meta-) Modeling. *Software & Systems Modeling*, 5(4):369–385, 2006.

[KV10] Lennart Kats und Eelco Visser. The Spoofax Language Workbench. In William R. Cook, Editor, *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, Seite 444. ACM, 2010.

[KvdSV09] Paul Klint, Tijs van der Storm und Jurgen Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, Seiten 168–177. IEEE, 2009.

[KWB03] Anneke Kleppe, Jos B. Warmer und Wim Bast. *MDA explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[Läm01] Ralf Lämmel. Grammar Adaptation. In *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe*, Seiten 550–570, 2001.

[Läm02] Ralf Lämmel. Towards Generic Refactoring. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Rule-based Programming*, RULE '02, Seiten 15–28. ACM, 2002.

[Läm04] Ralf Lämmel. Transformations everywhere. *Science of Computer Programming*, 52:1–8, 2004.

[Läm16] Ralf Lämmel. Coupled software transformations revisited. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam*, Seiten 239–252, 2016.

[Lan05] Kevin Lano. *Advanced Systems Design with Java, UML and MDA*. Butterworth-Heinemann, 2005.

[LAS14] Erhan Leblebici, Anthony Anjorin und Andy Schürr. Developing eMoflon with eMoflon. In *Theory and Practice of Model Transformations*, LNCS 8568, Seiten 138–145. Springer, 2014.

[LB98] Sheng Liang und Gilad Bracha. Dynamic class loading in the Java virtual machine. *ACM SIGPLAN Notices*, 33(10):36–44, 1998.

[LKR10] Kevin Lano und Shekoufeh Kolahdouz-Rahimi. Specification and Verification of Model Transformations Using UML-RSDS. In Dominique Méry und Stephan Merz, Editoren, *Integrated Formal Methods: 8th International Conference*, Seiten 199–214. Springer, 2010.

[LLMC05] Tihamér Levendovszky, László Lengyel, Gergely Mezei und Hassan Charaf. A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS. *Electronic Notes in Theoretical Computer Science*, 127(1):65–75, 2005.

[LLP07] Johan Lindqvist, Torbjörn Lundkvist und Ivan Porres. A Query Language With the Star Operator. *Electronic Communications of the EASST*, 6, 2007.

[LNPR+13] Markus Look, Antonio Navarro Pérez, Jan Oliver Ringert, Bernhard Rumpe und Andreas Wortmann. Black-box Integration of Heterogeneous Modeling Languages for Cyber-Physical Systems. In *Globalization of Modeling Languages Workshop (GEMOC'13)*, CEUR Workshop Proceedings 1102, 2013.

[Loo17] Markus Look. *Modellgetriebene, agile Entwicklung und Evolution mehrbenutzerfähiger Enterprise Applikationen mit MontiEE.* Aachener Informatik-Berichte, Software Engineering, Band 27. Shaker Verlag, March 2017.

[LV02] Javier Larrosa und Gabriel Valiente. Constraint satisfaction algorithms for graph pattern matching. *Mathematical Structures in Computer Science*, 12(04), 2002.

[LVV03] Ralf Lämmel, Eelco Visser und Joost Visser. Strategic programming meets adaptive programming. In *Proceedings of the 2Nd International Conference on Aspect-oriented Software Development*, AOSD '03, Seiten 168–177. ACM, 2003.

[LWK10] Philip Langer, Manuel Wimmer und Gerti Kappel. Model-to-Model Transformations By Demonstration. In Laurence Tratt und Martin Gogolla, Editoren, *Theory and Practice of Model Transformations*, LNCS 6142, Seiten 153–167. Springer, 2010.

[Mac06] Lori MacVittie. *XAML in a Nutshell.* O'Reilly, 2006.

[MB07] Slaviša Marković und Thomas Baar. Refactoring OCL annotated UML class diagrams. *Software & Systems Modeling*, 7(1):25–47, 2007.

[MBK91] Yoëlle S. Maarek, Daniel M. Berry und Gail E. Kaiser. An Information Retrieval Approach For Automatically Constructing Software Libraries. *IEEE Transactions on Software Engineering*, 17(8):800–813, 1991.

[MC12] Iman Hemati Moghadam und Mel Ó. Cinnéide. Automated Refactoring Using Design Differencing. In Tom Mens, Editor, *16th European Conference on Software Maintenance and Reengineering (CSMR)*, Seiten 43–52. IEEE, 2012.

[MH08] Mark Minas und Berthold Hoffmann. An Example of Cloning Graph Transformation Rules for Programming. *Electronic Notes in Theoretical Computer Science*, 211:241–250, 2008.

[MHS05] Marjan Mernik, Jan Heering und Anthony M. Sloane. When and How to Develop Domain-specific Languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[MKMG97] Robert Monroe, Andrew Kompanek, Ralph Melton und David Garlan. Architectural Styles, Design Patterns, and Objects. *IEEE Software*, 14(1):43–52, 1997.

[ML05] Dave Minter und Jeff Linwood. *Pro Hibernate 3*. Apress, 2005.

[MMBJ09] Naouel Moha, Vincent Mahé, Olivier Barais und Jean-Marc Jézéquel. Generic Model Refactorings. In Andy Schürr und Bran Selic, Editoren, *Model driven engineering languages and systems*, LNCS 5795, Seiten 628–643. Springer, 2009.

[Mol06] Anthony Molinaro. *SQL Cookbook*. O'Reilly, 2006.

[MSN17] Pedram Mir Seyed Nazari. *MontiCore: Efficient Development of Composed Modeling Language Essentials*. Aachener Informatik-Berichte, Software Engineering, Band 29. Shaker Verlag, June 2017.

[MSNRR15] Pedram Mir Seyed Nazari, Alexander Roth und Bernhard Rumpe. Mixed Generative and Handcoded Development of Adaptable Datacentric Business Applications. In *Domain-Specific Modeling Workshop (DSM'15)*, Seiten 43–44. ACM, 2015.

[MT00] Nenad Medvidovic und Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[MVG06] Tom Mens und Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125 – 142, 2006.

[Nag79] Manfred Nagl. *Graph-Grammatiken: Theorie Anwendungen Implementierung*. Studienbücher Informatik. Vieweg+Teubner Verlag, 1979.

[NMSS09] Mangala Gowri Nanda, Senthil Mani, Vibha Singhal Sinha und Saurabh Sinha. Demystifying Model Transformations: An Approach Based on Automated Rule Inference. In Shail Arora, Editor, *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, Seite 341. ACM, 2009.

[NNZ00] Ulrich Nickel, Jörg Niere und Albert Zündorf. The FUJABA environment. In Carlo Ghezzi, Mehdi Jazayeri und Alexander L. Wolf, Editoren, *Proceedings of the 2000 International Conference on Software Engineering*, Seiten 742–745. ACM, 2000.

[NTVW15] Pierre Neron, Andrew Tolmach, Eelco Visser und Guido Wachsmuth. A Theory of Name Resolution. In Jan Vitek, Editor, *Programming Languages and Systems*, LNCS 9032, Seiten 205–231. Springer, 2015.

[OMG15] Object Management Group. Unified Modeling Language (UML), Version 2.5. `http://www.omg.org/spec/UML/2.5/PDF/`, 2015. [Online; abgerufen am: 12.10.2017].

[Opd92] William F. Opdyke. *Refactoring Object-oriented Frameworks*. Dissertation, University of Illinois at Urbana-Champaign, 1992.

[Ora17] Oracle. Java Language and Virtual Machine Specifications. `https://docs.oracle.com/javase/specs/`, 2017.

[Par14] Terence Parr. *The definitive ANTLR 4 reference.* Pragmatic programmers. The Pragmatic Programmers, 2014.

[Por03] Ivan Porres. Model Refactorings as Rule-Based Update Transformations. In Perdita Stevens, Jon Whittle und Grady Booch, Editoren, *«UML» 2003 - The Unified Modeling Language. Modeling Languages and Applications*, LNCS 2863, Seiten 159–174. Springer, 2003.

[Por05] Ivan Porres. Rule-based update transformations and their application to model refactorings. *Software & Systems Modeling*, 4(4):368–385, 2005.

[PR99] Jan Philipps und Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In *Congress on Formal Methods in the Development of Computing System (FM'99)*, LNCS 1708, Seiten 96–115. Springer, 1999.

[PR01] Jan Philipps und Bernhard Rumpe. Roots of Refactoring. In Kilov, H. and Baclavski, K., Editor, *Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15.* Northeastern University, 2001.

[QGC+09] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler und Andrew Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.

[RBJ97] Don Roberts, John Brant und Ralph Johnson. A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997.

[Ren06] Arend Rensink. Nested Quantification in Graph Transformation Rules. In Andrea Corradini, Editor, *Graph Transformation*, LNCS 4178, Seiten 1–13. Springer, 2006.

[RKK14] Michaela Rindt, Timo Kehrer und Udo Kelter. Automatic Generation of Consistency-Preserving Edit Operations for MDE Tools. In *Proceedings of the Demonstrations Track of the ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2014)*, 2014.

[RKR+06] Thomas Reiter, Elisabeth Kapsammer, Werner Retschitzegger, Wieland Schwinger und Markus Stumptner. A Generator Framework for Domain-Specific Model Transformation Languages. In *8th International Conference on Enterprise Information Systems (ICEIS)*, Seiten 27–35, 2006.

[Rob99] Donald B. Roberts. *Practical Analysis for Refactoring.* Dissertation, University of Illinois at Urbana-Champaign, 1999.

[Rot17] Alexander Roth. *Adaptable Code Generation of Consistent and Customizable Data-Centric Applications with MontiDEx.* Dissertation, RWTH Aachen, 2017.

[RR11] Jan Oliver Ringert und Bernhard Rumpe. A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. *International Journal of Software and Informatics*, 2011.

[RRW15] Jan Oliver Ringert, Bernhard Rumpe und Andreas Wortmann. Tailoring the MontiArcAutomaton Component & Connector ADL for Generative Development. In *MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*, Seiten 41–47. ACM, 2015.

[Rud00] Michael Rudolf. Utilizing Constraint Satisfaction Techniques for Efficient Graph Pattern Matching. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski und Grzegorz Rozenberg, Editoren, *Theory and Application of Graph Transformations*, LNCS 1764, Seiten 238–251. Springer, 2000.

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.

[Rum17] Bernhard Rumpe. *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International, May 2017.

[Run17] Olga Runge. The <AGG> Homepage. `http://www.tfs.tu-berlin.de/agg`, 2017. [Online; abgerufen am: 31.08.2017].

[RW11] Bernhard Rumpe und Ingo Weisemöller. A Domain Specific Transformation Language. In *Workshop on Models and Evolution (ME)*, 2011.

[SAL+03] Jonathan Sprinkle, Aditya Agrawal, Tíhamer Levendovszky, Feng Shi und Gabor Karsai. Domain Model Translation Using Graph Transformations. In *10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems. Proceedings.*, Seiten 159–168, 2003.

[SBG+17] Daniel Strüber, Kristopher Born, Kanwal Daud Gill, Raffaela Groner, Timo Kehrer, Manuel Ohrndorf und Matthias Tichy. *Henshin: A Usability-Focused Framework for EMF Model Transformation Development*, Seiten 196–208. Springer, 2017.

[SBPM09] David Steinberg, Frank Budinsky, Marcelo Paternostro und Ed Merks. *Eclipse modeling framework: EMF*. The eclipse series. Addison-Wesley, 2. Edition, 2009.

[SCGdL14] Jesús Sánchez Cuadrado, Esther Guerra und Juan de Lara. Towards the Systematic Construction of Domain-Specific Transformation Languages. In *Modelling Foundations and Applications*, LNCS 8569, Seiten 196–212. Springer, 2014.

[Sch91] Andreas Schürr. *Operationales Spezifizieren mit Programmierten Graphersetzungssystemen: Formale Definitionen Anwendungsbeispiele and Werkzeugunterstützung.* Wiesbaden: Deutscher Universitäts-Verlag, 1991.

[Sch95] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In Ernst W. Mayr, Editor, *Graph-theoretic concepts in computer science*, LNCS 903, Seiten 151–163. Springer, 1995.

[Sch07] Markus Schmidt. Transformations of UML 2 Models Using Concrete Syntax Patterns. In *Rapid Integration of Software Engineering Techniques*. Springer, 2007.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P.* Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[SD11] Jim Steel und Robin Drogemuller. Domain-Specific Model Transformation in Building Quantity Take-Off. In *Model Driven Engineering Languages and Systems*, LNCS 6981, Seiten 198–212. Springer, 2011.

[Sei03] Ed Seidewitz. What Models Mean. *IEEE Software*, 20(5):26–32, 2003.

[SFP17] Alexander Shevets, Gerhard Frey, und Marina Pavlova. SourceMaking - Refactoring. `https://sourcemaking.com/refactoring`, 2017. [Online; abgerufen am: 01.03.2017].

[SGV13] Eugene Syriani, Jeff Gray und Hans Vangheluwe. Modeling a Model Transformation Language. In Iris Reinhartz-Berger, Arnon Sturm, Tony Clark, Sholom Cohen und Jorn Bettin, Editoren, *Domain Engineering*, Seiten 211–237. Springer, 2013.

[SGW15] Yu Sun, Jeff Gray und Jules White. A demonstration-based model transformation approach to automate model scalability. *Software & Systems Modeling*, 14(3):1245–1271, 2015.

[SK03] Shane Sendall und Wojtek Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45, 2003.

[SKSL11] Hagen Schink, Martin Kuhlemann, Gunter Saake und Ralf Lämmel. Hurdles in Multi-language Refactoring of Hibernate Applications. In *ICSOFT 2011 - Proceedings of the 6th International Conference on Software and Data Technologies*, Seiten 129–134, 2011.

[SLH+17] Simon Schauss, Ralf Lämmel, Johannes Härtel, Marcel Heinz, Kevin Klein, Lukas Härtel und Thorsten Berger. A Chrestomathy of DSL Implementations. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2017*, Seiten 103–114, 2017.

[SMM+12] Sagar Sen, Naouel Moha, Vincent Mahé, Olivier Barais, Benoit Baudry und Jean-Marc Jézéquel. Reusable model transformations. *Software & Systems Modeling*, 11(1):111–125, 2012.

[Som16] Peter Sommerhoff. Refactoring, Refining, And Evolving MontiArc Software Architecture Descriptions: A Model Transformation Library. Bachelorarbeit, RWTH Aachen, Software Engineering, Deutschland, 2016.

[SPLJ01] Gerson Sunyé, Damien Pollet, Yves Le Traon und Jean-Marc Jézéquel. Refactoring UML Models. In Martin Gogolla und Cris Kobryn, Editoren, *≪UML≫ 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, LNCS 2185, Seiten 134–148. Springer, 2001.

[SS11] Christopher Schmitt und Kyle Simpson. *HTML5 Cookbook*. O'Reilly, 2011.

[ST97] Ron Shamir und Dekel Tsur. Faster Subtree Isomorphism. In *Proceedings of the Fifth Israeli Symposium on Theory of Computing and Systems*, Seiten 126–131, 1997.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer-Verlag, Wien, 1973.

[Sta06] Miroslaw Staron. Adopting Model Driven Software Development in Industry - A Case Study at Two Companies. In *Model Driven Engineering Languages and Systems*, Seiten 57–72. Springer, 2006.

[Ste15] Friedrich Steimann. From well-formedness to meaning preservation: Model refactoring for almost free. *Software & Systems Modeling*, 14(1):307–320, 2015.

[SVC06] Thomas Stahl, Markus Voelter und Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley and Sons, 2006.

[SVGJ05] Hans Schippers, Pieter Van Gorp und Dirk Janssens. Leveraging UML Profiles to Generate Plugins From Visual Model Transformations. *Electronic Notes in Theoretical Computer Science*, 127(3):5–16, 2005.

[SVL13] Eugene Syriani, Hans Vangheluwe und Brian LaShomb. T-Core: a framework for custom-built model transformation engines. *Software & Systems Modeling*, Seiten 1–29, 2013.

[SVM+13] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo und Hüseyin Ergin. AToMPM: A Web-based Modeling Environment. In *MODELS'13: Invited Talks, Demos, Posters, and ACM SRC.*, Seiten 21–25, 2013.

[SW01] Connie Smith und Lloyd Williams. Software Performance Antipatterns; Common Performance Problems and Their Solutions. In *International CMG Conference*, Seiten 797–806, 2001.

[SW03] Connie Smith und Lloyd Williams. More New Software Performance Antipatterns: Even More Ways to Shoot Yourself in the Foot. In *Computer Measurement Group Conference*, Seiten 717–725, 2003.

[SWG09] Yu Sun, Jules White und Jeff Gray. Model Transformation by Demonstration. In Andy Schürr und Bran Selic, Editoren, *Model Driven Engineering Languages and Systems: 12th International Conference*, Seiten 712–726. Springer, 2009.

[SWS+13] Martin Schmidt, Arif Wider, Markus Scheidgen, Joachim Fischer und Sebastian von Klinski. Refactorings in Language Development with Asymmetric Bidirectional Model Transformations. In Ferhat Khendek, Maria Toeroe, Abdelouahed Gherbi und Rick Reed, Editoren, *SDL 2013: Model-Driven Dependability Engineering*, LNCS 7916, Seiten 222–238. Springer, 2013.

[SWZ95] Andy Schürr, Andreas J. Winter und Albert Zündorf. Graph grammar engineering with PROGRES. In Wilhelm Schäfer, Editor, *Software Engineering - ESEC '95*, LNCS 989, Seiten 219–234. Springer, 1995.

[SWZ99] Andy Schürr, Andreas J. Winter und Albert Zündorf. The PROGRES Approach: Language and Environment. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski und Grzegorz Rozenberg, Editoren, *Handbook of Graph Grammars and Computing by Graph Transformation*, Seiten 487–550. World Scientific Publishing Co., Inc, 1999.

[SZG06] Mirko Stölzel, Steffen Zschaler und Leif Geiger. Integrating OCL and Model Transformations in Fujaba. *ECEASST*, 5, 2006.

[Tae04] Gabriele Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In John L. Pfaltz, Manfred Nagl und Boris Böhlen, Editoren, *Applications of Graph Transformations with Industrial Relevance*, LNCS 3062, Seiten 446–453. Springer, 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 und Tamás Vajk. Generation of Sierpinski Triangles: A Case Study for Graph Transformation Tools. In Andy Schürr, Manfred Nagl und Albert Zündorf, Editoren, *Applications of Graph Transformations with Industrial Relevance. AGTIVE 2007*, Seiten 514–539. Springer, 2008.

[Ull12] Christian Ullenboom. *Java ist auch eine Insel: Das umfassende Handbuch*. Galileo computing. Galileo Press, 10., aktualisierte und überarbeitete Auflage Edition, 2012.

[UML17a] UML-RSDS Model Transformation and Model-Driven Development Tools. `https://nms.kcl.ac.uk/kevin.lano/uml2web/`, 2017. [Online; abgerufen am: 08.09.2017].

[UML17b] UMLX Home page. lhttps://www.eclipse.org/gmt/umlx/, 2017. [Online; abgerufen am: 08.09.2017].

[VB07] Dániel Varró und András Balogh. The model transformation language of the VIATRA2 framework. *Science of computer programming*, 68(3):214–234, 2007.

[vdBHKO02] Mark G. J. van den Brand, Jan Heering, Paul Klint und Pieter A Olivier. Compiling language definitions: the ASF+ SDF compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(4):334–368, 2002.

[VGSMD03] Pieter Van Gorp, Hans Stenten, Tom Mens und Serge Demeyer. Towards Automating Source-Consistent UML Refactorings. In Perdita Stevens, Jon Whittle und Grady Booch, Editoren, *«UML» 2003 - The Unified Modeling Language. Modeling Languages and Applications*, LNCS 2863, Seiten 144–158. Springer, 2003.

[VIA17] VIATRA Project. http://www.eclipse.org/viatra/, 2017. [Online; abgerufen am: 31.08.2017].

[Vis01a] Eelco Visser. A Survey of Rewriting Strategies in Program Transformation Systems. *Electronic Notes in Theoretical Computer Science*, 57:109 – 143, 2001.

[Vis01b] Eelco Visser. Stratego: A Language for Program Transformation Based on Rewriting Strategies System Description of Stratego 0.5. In Aart Middeldorp, Editor, *Rewriting Techniques and Applications*, Seiten 357–361. Springer Berlin Heidelberg, 2001.

[Vis02] Eelco Visser. Meta-programming with Concrete Object Syntax. In Don S. Batory, Editor, *Generative programming and component engineering*, LNCS 2487, Seiten 299–315. Springer, 2002.

[Vis04] Eelco Visser. Program Transformation with Stratego/XT. In Christian Lengauer, Don Batory, Charles Consel und Martin Odersky, Editoren, *Domain-Specific Program Generation*, LNCS 3016, Seiten 216–238. Springer, 2004.

[Vli98] John Vlissides. *Pattern Hatching: Design Patterns Applied*. Addison-Wesley Longman Publishing Co., Inc, 1998.

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.

[Vos00] Gottfried Vossen. *Datenmodelle, Datenbanksprachen und Datenbankmanagementsysteme*. Oldenbourg, 4. Edition, 2000.

[VS13] Markus Völter und Thomas Stahl. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley Software Patterns Series. John Wiley and Sons, 2013.

[Wac07] Guido Wachsmuth. Metamodel Adaptation and Model Co-adaptation. In Erik Ernst, Editor, *ECOOP 2007 – Object-Oriented Programming*, LNCS 4609, Seiten 600–624. Springer, 2007.

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012.

[WHR14] Jon Whittle, John Hutchinson und Mark Rouncefield. The State of Practice in Model-Driven Engineering. *Software, IEEE*, 31(3):79–85, 2014.

[Wil17] Alexander Wilts. Performance-Optimierung und Modularisierung von Constraints für domänenspezifische Modelltransformationen. Masterarbeit, RWTH Aachen, Software Engineering, Deutschland, 2017.

[Win12] Sabine Winetzhammer. ModGraph - Generating Executable EMF Models. *ECEASST*, 54, 2012.

[WJE+09] Jon Whittle, Praveen Jayaraman, Ahmed Elkhodary, Ana Moreira und João Araújo. MATA: A Unified Approach for Composing UML Aspect Models Based on Graph Transformation. In Shmuel Katz, Harold Ossher, Robert France und Jean-Marc Jézéquel, Editoren, *Special issue on aspects and model-driven engineering*, LNCS 5560, Seiten 191–237. Springer, 2009.

[Wla16] David Wlazlo. Designing and Implementing an Optional Operator for Domain-specific Transformation Languages in MontiCore. Bachelorarbeit, RWTH Aachen, Software Engineering, Deutschland, 2016.

[Wor16] Andreas Wortmann. *An Extensible Component & Connector Architecture Description Infrastructure for Multi-Platform Modeling*. Aachener Informatik-Berichte, Software Engineering, Band 25. Shaker Verlag, November 2016.

[WR09] William C. Wake und Kevin Rutherford. *Refactoring in Ruby*. Addison-Wesley Professional, 1st Edition, 2009.

[WWM+07] Thomas Weigert, Frank Weil, Kevin Marth, Paul Baker, Clive Jervis, Paul Dietz, Yexuan Gui, Aswin Van Den Berg, Kim Fleer, David Nelson et al. Experiences in Deploying Model-Driven Engineering. In *SDL 2007: Design for Dependable Systems*, Seiten 35–53. Springer, 2007.

[ZCÖ09] Lei Zou, Lei Chen und M. Tamer Özsu. Distance-join: Pattern Match Query in a Large Graph Database. *Proceedings of the VLDB Endowment*, 2(1):886–897, 2009.

[ZLG05] Jing Zhang, Yuehua Lin und Jeff Gray. Generic and Domain-Specific Model Refactoring Using a Model Transformation Engine. In Sami Beydeda, Matthias Book und Volker Gruhn, Editoren, *Model-Driven Software Development*, Seiten 199–217. Springer, 2005.

[Zün96a] Albert Zündorf. Graph pattern matching in PROGRES. In *Graph Grammars and Their Application to Computer Science*, LNCS 1073, Seiten 454–468. Springer, 1996.

[Zün96b] Albert Zündorf. *PROgrammierte GRaphErsetzungsSysteme: Spezifikation, Implementierung und Anwendung einer integrierten Entwicklungsumgebung.* Deutscher Universitätsverlag, Wiesbaden, 1996.

Teil V

Anhänge

Anhang A

Markierungen in Abbildungen und Listings

Dieser Anhang beschreibt die verwendeten Markierungen in Abbildungen und Listings. Tabelle A.1 beschreibt die Markierungen, die die verwendete Sprache angeben. Tabelle A.2 beschreibt die verwendeten Stereotypen.

Markierung	**Erklärung**
AD	Aktivitätsdiagramm
CD	Klassendiagramm
CDTrans	CDTrans-Transformation
CLI	CLI-Aufruf
CSD	Komponentendiagramm
Groovy	Groovy Quellcode
Java	Java Quellcode
LHS	Linke Regelseite einer Transformation
MA	MontiArc-Modell
MATrans	MATrans-Transformation
MACDTrans	MACDTrans Transformation
MCG	MontiCore Grammatik
OD LHS	Objektdiagramm der linken Regelseite
OD RHS	Objektdiagramm der rechten Regelseite
ODRule	ODRule-Modell
pom.xml	Project Object Model (POM) Datei
RHS	Rechte Regelseite einer Transformation

Tabelle A.1: Erklärung der verwendeten Tags in Abbildungen und Listings.

Tag	Description
«gen»	Generiertes Element
«hwc»	Handgeschriebenes Element
«opt»	Optionales Element
«rte»	Durch das Runtime Environment (RTE) bereitgestelltes Element

Tabelle A.2: Erklärung der verwendeten Stereotypen.

Anhang B

Abkürzungen

AST	Abstrakter Syntaxbaum
AD	Aktivitätsdiagramm
ADL	Architekturbeschreibungssprache
ANTLR	Another Tool for Language Recognition
ATL	Atlas Transformation Language
CD	Klassendiagramm
CLI	Command Line Interface
CRUD	Create, Read, Update, Delete
CSP	Constraint Satisfaction Problem
C&C	Component & Connector
DSL	Domänenspezifische Sprache
DSML	Domänenspezifische Modellierungssprache
DSTL	Domänenspezifische Transformationssprache
EBNF	Erweiterte Backus-Naur Form
EMF	Eclipse Modeling Framework
Fujaba	From UML to Java and back again
GPL	General Purpose Language
GPML	General Purpose Modeling Language
GPTL	General Purpose Transformationssprache

GReAT	Graph Rewriting and Transformation
HTML	Hypertext Markup Language
HQL	Hibernate Query Language
KNF	Konjunktive Normalform
LHS	linke Regelseite
M2M	Model-to-Model
M2T	Model-to-Text
OCL	Object Constraint Language
OD	Objektdiagramm
PROGRES	PROgrammed Graph REwriting Systems
POM	Project Object Model
RHS	rechte Regelseite
ROS	Robot Operation System
RTE	Runtime Environment
SC	Statechart
SD	Sequenzdiagramm
SQL	Structured Query Language
TGG	Triple Graph Grammars
UML	Unified Modeling Language
VIATRA	Visual Automated Model Transformation
XAML	Extensible Application Markup Language

Anhang C

Modelle und Grammatiken

C.1 Grammatik der DSLs CD4Analysis und CD4Code

MCG

```
grammar CD4Analysis extends de.monticore.types.Types {

  CDCompilationUnit = ("package" package:(Name || ".")+ ";")?
    (ImportStatement)* CDDefinition;

  CDDefinition = "classdiagram" Name
    "{" (cDClasses:CDClass | CDInterface | CDEnum | CDAssociation)* "}";

  CDClass  = Modifier?  "class" Name
  ( "extends"  superclass:ReferenceType)?
  ( "implements" interfaces:(ReferenceType || ",")+ )?
  ( "{" (CDAttribute | CDConstructor | CDMethod)* "}"  | ";" );

  CDInterface = Modifier? "interface" Name
    ( "extends" interfaces:(ReferenceType || ",")+ )?
    ( "{" ( CDAttribute | CDMethod )* "}" | ";" );

  CDEnum  = Modifier? "enum" Name
    ( "implements" interfaces:(ReferenceType || ",")+ )?
    ( "{" CDEnumConstant || ",")* ";"
    (CDConstructor | CDMethod)* "}" | ";" );

  CDAttribute     = Modifier? Type Name ("=" Value)? ";";

  CDEnumConstant  = Name ( "(" (CDEnumParameter || ",")+ ")" )?;
  CDEnumParameter = Value;

  CDConstructor =
    Modifier Name "(" (CDParameter || ",")* ")"
    ("throws" exceptions:(QualifiedName || "," )+)?
    ";";

  CDMethod = Modifier ReturnType Name "(" (CDParameter || ",")* ")"
    ("throws" exceptions:(QualifiedName || ",")+)? ";";

  CDAssociation =
    Stereotype? (["association"] | ["composition"])
    ([Derived:"/"])?  Name?
    leftModifier:Modifier? leftCardinality:Cardinality?
    leftReferenceName:QualifiedName
    leftQualifier:CDQualifier? ("(" leftRole:Name ")")?
    (leftToRight:["->"]     | rightToLeft:["<-"]|
```

```
43        bidirectional:["<->"] | unspecified:["--"] )
44      ("(" rightRole:Name ")")? rightQualifier:CDQualifier?
45      rightReferenceName:QualifiedName
46      rightCardinality:Cardinality? rightModifier:Modifier? ";" ;
47
48    Modifier = Stereotype? (["abstract"] | ["final"]     | ["static"]
49         | ["private"]   | [private:"-"] | ["protected"] | [protected:"#"]
50         | ["public"]    | [public:"+"]  | ["derived"]   | [derived:"/"] )*;
51
52    Cardinality = many:["[*]"] | one:["[1]"]
53        | oneToMany:["[1..*]"] | optional:["[0..1]"];
54
55    CDParameter = Type (Ellipsis:["..."])? Name;
56    CDQualifier  = "[[" Name "]]" | "[" Type "]";
57    Stereotype   = "<<" values:(StereoValue || ",")+ ">" ">";
58    StereoValue  = Name ("=" value:String)?;
59    Value        = SignedLiteral;
60  }
```

Listing 3.1: Grammatik der DSLs CD4Analysis und CD4Code.

C.2 Grammatik der DSTL CDTrans

MCG

```
1  package de.monticore.umlcd4a.tr;
2
3  grammar CD4AnalysisTR extends de.monticore.types.tr.TypesTR {
4
5    TFRule =
6      (CDCompilationUnit | CDDefinition | CDClass          | CDInterface
7      | CDEnum           | CDAttribute  | CDEnumConstant  | CDEnumParameter
8      | CDConstructor    | CDMethod     | CDParameter     | CDAssociation
9      | Cardinality      | CDQualifier  | Stereotype      | StereoValue
10     | Value)* TFFolding? TFAssignments? TFWhere? TFDo?;
11
12   interface Value              astextends mc.tfcs.ast.ITFElement;
13   interface StereoValue        astextends mc.tfcs.ast.ITFElement;
14   interface Stereotype         astextends mc.tfcs.ast.ITFElement;
15   interface CDQualifier        astextends mc.tfcs.ast.ITFElement;
16   interface Cardinality        astextends mc.tfcs.ast.ITFElement;
17   interface ITFModifier        astextends mc.tfcs.ast.ITFElement;
18   interface CDAssociation      astextends mc.tfcs.ast.ITFElement;
19   interface CDParameter        astextends mc.tfcs.ast.ITFElement;
20   interface CDMethod           astextends mc.tfcs.ast.ITFElement;
21   interface CDConstructor      astextends mc.tfcs.ast.ITFElement;
22   interface CDEnumParameter    astextends mc.tfcs.ast.ITFElement;
23   interface CDEnumConstant     astextends mc.tfcs.ast.ITFElement;
24   interface CDAttribute        astextends mc.tfcs.ast.ITFElement;
25   interface CDEnum             astextends mc.tfcs.ast.ITFElement;
26   interface CDInterface        astextends mc.tfcs.ast.ITFElement;
27   interface CDClass            astextends mc.tfcs.ast.ITFElement;
28   interface CDDefinition       astextends mc.tfcs.ast.ITFElement;
29   interface CDCompilationUnit astextends mc.tfcs.ast.ITFElement;
30   interface Optional_Constant;
31   interface OneToMany_Constant;
32   interface One_Constant;
33   interface Many_Constant;
34   interface Public_Constant;
35   interface Protected_Constant;
36   interface Private_Constant;
37   interface Static_Constant;
38   interface Final_Constant;
39   interface Abstract_Constant;
40   interface Unspecified_Constant;
41   interface Bidirectional_Constant;
42   interface RightToLeft_Constant;
43   interface LeftToRight_Constant;
44   interface Derived_Constant;
45   interface Composition_Constant;
46   interface Association_Constant;
47   interface Ellipsis_Constant;
48
49   CDCompilationUnit_List implements CDCompilationUnit
50                                          astimplements mc.tfcs.ast.IList =
51     "list" ("<CDCompilationUnit>")? schemaVarName:Name? "[["
52       CDCompilationUnit
53     "]]";
54
55   CDCompilationUnit_Neg implements CDCompilationUnit
56                                       astimplements mc.tfcs.ast.INegation =
57     "not" schemaVarName:Name? "[[" CDCompilationUnit "]]";
```

```
58
59  CDCompilationUnit_Opt implements CDCompilationUnit
60                                        astimplements mc.tfcs.ast.IOptional =
61    "opt" "[[" CDCompilationUnit "]]";
62
63  CDCompilationUnit_Rep implements CDCompilationUnit
64                                      astimplements mc.tfcs.ast.IReplacement =
65    ("[[" lhs:CDCompilationUnit? ":-" rhs:CDCompilationUnit? "]]");
66
67  CDCompilationUnit_Pat implements CDCompilationUnit
68                                        astimplements mc.tfcs.ast.IPattern =
69    (("package" Package:TfIdentifier ("." Package:TfIdentifier)* ";")?
70    (ImportStatements:ITFImportStatement)* CDDefinition)
71    | "CDCompilationUnit" schemaVarName:Name
72    | (("CDCompilationUnit" schemaVarName:Name? |
73        "CDCompilationUnit"? schemaVarName:Name )  "[["
74          (("package" Package:TfIdentifier ("." Package:TfIdentifier)* ";")?
75          (ImportStatements:ITFImportStatement)* CDDefinition)
76        "]]");
77
78  CDDefinition_List implements CDDefinition astimplements mc.tfcs.ast.IList =
79    "list" ("<CDDefinition>")? schemaVarName:Name? "[["
80      CDDefinition
81    "]]";
82
83  CDDefinition_Neg implements CDDefinition astimplements mc.tfcs.ast.INegation =
84    "not" "[[" CDDefinition "]]";
85
86  CDDefinition_Opt implements CDDefinition astimplements mc.tfcs.ast.IOptional =
87    "opt" "[[" CDDefinition "]]";
88
89  CDDefinition_Rep implements CDDefinition astimplements mc.tfcs.ast.IReplacement =
90    ("[[" lhs:CDDefinition? ":-" rhs:CDDefinition? "]]");
91
92  CDDefinition_Pat implements CDDefinition astimplements mc.tfcs.ast.IPattern =
93    ("classdiagram" Name:TfIdentifier "{"
94      (cDClasses:CDClass | cDInterfaces:CDInterface
95       | cDEnums:CDEnum | cDAssociations:CDAssociation)*
96    "}")
97    | "CDDefinition" schemaVarName:Name
98    | (("CDDefinition" schemaVarName:Name? |
99        "CDDefinition"? schemaVarName:Name) "[["
100         ("classdiagram" Name:TfIdentifier "{"
101           (cDClasses:CDClass | cDInterfaces:CDInterface
102             | cDEnums:CDEnum | cDAssociations:CDAssociation)*
103         "}")
104     "]]");
105
106 CDClass_List implements CDClass astimplements mc.tfcs.ast.IList =
107   "list" ("<CDClass>")? schemaVarName:Name? "[[" CDClass "]]";
108
109 CDClass_Neg implements CDClass astimplements mc.tfcs.ast.INegation =
110   "not" "[[" CDClass "]]";
111
112 CDClass_Opt implements CDClass astimplements mc.tfcs.ast.IOptional =
113   "opt" "[[" CDClass "]]";
114
115 CDClass_Rep implements CDClass astimplements mc.tfcs.ast.IReplacement =
116   ("[[" lhs:CDClass? ":-" rhs:CDInterfaceOrClassOrEnum? "]]");
117
118 CDClass_Pat implements CDInterfaceOrClassOrEnum, CDClass
119                                       astimplements mc.tfcs.ast.IPattern =
```

```
120   (ITFModifier? "class" Name:TfIdentifier ("extends" superclass:ITFReferenceType)?
121   ("implements" interfaces:ITFReferenceType ("," interfaces:ITFReferenceType)*)?
122   ("{"
123     (cDAttributes:CDAttribute | cDConstructors:CDConstructor | cDMethods:CDMethod)*
124   "}" | ";"))
125   | "CDClass" schemaVarName:Name
126   | (( "CDClass" schemaVarName:Name? | "CDClass"? schemaVarName:Name) "[["
127     (ITFModifier? "class" Name:TfIdentifier
128     ("extends" superclass:ITFReferenceType)?
129     ("implements" interfaces:ITFReferenceType ("," interfaces:ITFReferenceType)*)?
130     ("{"
131      (cDAttributes:CDAttribute | cDConstructors:CDConstructor | cDMethods:CDMethod)*
132     "}" | ";")) "]]");
133
134   interface CDInterfaceOrClassOrEnum astextends mc.tfcs.ast.IPattern;
135
136   CDInterface_List implements CDInterface astimplements mc.tfcs.ast.IList =
137     "list" ("<CDInterface>")? schemaVarName:Name? "[[" CDInterface "]]";
138
139   CDInterface_Neg implements CDInterface astimplements mc.tfcs.ast.INegation =
140     "not" "[[" CDInterface "]]";
141
142   CDInterface_Opt implements CDInterface astimplements mc.tfcs.ast.IOptional =
143     "opt" "[[" CDInterface "]]";
144
145   CDInterface_Rep implements CDInterface astimplements mc.tfcs.ast.IReplacement =
146     ("[[" lhs:CDInterface? ":-" rhs:CDInterfaceOrClassOrEnum? "]]" );
147
148   CDInterface_Pat implements  CDInterfaceOrClassOrEnum, CDInterface
149                                         astimplements mc.tfcs.ast.IPattern =
150     (ITFModifier? "interface" Name:TfIdentifier
151     ("extends" interfaces:ITFReferenceType ("," interfaces:ITFReferenceType)*)? ("{"
152       (cDAttributes:CDAttribute | cDMethods:CDMethod)*
153     "}" | ";"))
154     | "CDInterface" schemaVarName:Name
155     | (("CDInterface" schemaVarName:Name? | "CDInterface"? schemaVarName:Name) "[["
156       (ITFModifier? "interface" Name:TfIdentifier
157       ("extends" interfaces:ITFReferenceType ("," interfaces:ITFReferenceType)*)?
158       ("{" (cDAttributes:CDAttribute | cDMethods:CDMethod)* "}" | ";")) "]]");
159
160   CDEnum_List implements CDEnum astimplements mc.tfcs.ast.IList =
161     "list" ("<CDEnum>")? schemaVarName:Name? "[[" CDEnum "]]";
162
163   CDEnum_Neg implements CDEnum astimplements mc.tfcs.ast.INegation =
164     "not" "[[" CDEnum "]]";
165
166   CDEnum_Opt implements CDEnum astimplements mc.tfcs.ast.IOptional =
167     "opt" "[[" CDEnum "]]";
168
169   CDEnum_Rep implements CDEnum astimplements mc.tfcs.ast.IReplacement =
170     ("[[" lhs:CDEnum? ":-" rhs:CDInterfaceOrClassOrEnum? "]]");
171
172   CDEnum_Pat implements CDInterfaceOrClassOrEnum, CDEnum
173                                           astimplements mc.tfcs.ast.IPattern =
174     (ITFModifier? "enum" Name:TfIdentifier
175     ("implements" interfaces:ITFReferenceType ("," interfaces:ITFReferenceType)*)?
176     ("{"
177     (cDEnumConstants:CDEnumConstant ("," cDEnumConstants:CDEnumConstant)*
178     (cDConstructors:CDConstructor | cDMethods:CDMethod)* ";")? "}" | ";"))
179     | "CDEnum" schemaVarName:Name
180     | (("CDEnum" schemaVarName:Name? | "CDEnum"? schemaVarName:Name ) "[["
181       (ITFModifier? "enum" Name:TfIdentifier
```

```
182       ("implements" interfaces:ITFReferenceType ("," interfaces:ITFReferenceType)*)?
183       ("{"
184         (cDEnumConstants:CDEnumConstant ("," cDEnumConstants:CDEnumConstant)*
185         (cDConstructors:CDConstructor | cDMethods:CDMethod)* ";")? "}" | ";"))
186     "]]");
187
188   CDAttribute_List implements CDAttribute astimplements mc.tfcs.ast.IList =
189     "list" ("<CDAttribute>")? schemaVarName:Name? "[[" CDAttribute "]]";
190
191   CDAttribute_Neg implements CDAttribute astimplements mc.tfcs.ast.INegation =
192     "not" "[[" CDAttribute "]]";
193
194   CDAttribute_Opt implements CDAttribute astimplements mc.tfcs.ast.IOptional =
195     "opt" "[[" CDAttribute "]]";
196
197    interface CDAttributeOrMethod astextends mc.tfcs.ast.IPattern;
198
199    CDAttribute_Rep implements CDAttribute astimplements mc.tfcs.ast.IReplacement =
200     "[[" lhs:CDAttribute? ":-" rhs:CDAttributeOrMethod? "]]";
201
202   CDAttribute_Pat implements CDAttributeOrMethod, CDAttribute  =
203     (ITFModifier? ITFType Name:TfIdentifier ("=" Value)? ";")
204     | "CDAttribute" schemaVarName:Name
205     |(("CDAttribute" schemaVarName:Name? | "CDAttribute"? schemaVarName:Name ) "[["
206       (ITFModifier? ITFType Name:TfIdentifier ("=" Value)? ";")
207     "]]");
208
209   CDEnumConstant_List implements CDEnumConstant astimplements mc.tfcs.ast.IList =
210     "list" ("<CDEnumConstant>")? schemaVarName:Name? "[[" CDEnumConstant "]]";
211
212   CDEnumConstant_Neg implements CDEnumConstant astimplements mc.tfcs.ast.INegation =
213     "not" "[[" CDEnumConstant "]]";
214
215   CDEnumConstant_Opt implements CDEnumConstant astimplements mc.tfcs.ast.IOptional =
216     "opt" "[[" CDEnumConstant "]]";
217
218   CDEnumConstant_Rep implements CDEnumConstant
219                                       astimplements mc.tfcs.ast.IReplacement =
220     ("[[" lhs:CDEnumConstant? ":-" rhs:CDEnumConstant? "]]");
221
222   CDEnumConstant_Pat implements CDEnumConstant astimplements mc.tfcs.ast.IPattern =
223     (Name:TfIdentifier ("("
224       cDEnumParameters:CDEnumParameter ("," cDEnumParameters:CDEnumParameter)*
225     ")")?)
226     | "CDEnumConstant" schemaVarName:Name
227     | (("CDEnumConstant" schemaVarName:Name? |
228         "CDEnumConstant"? schemaVarName:Name) "[["
229         (Name:TfIdentifier ("("
230           cDEnumParameters:CDEnumParameter ("," cDEnumParameters:CDEnumParameter)*
231         ")")?)
232       "]]");
233
234   CDEnumParameter_List implements CDEnumParameter astimplements mc.tfcs.ast.IList =
235     "list" ("<CDEnumParameter>")? schemaVarName:Name? "[[" CDEnumParameter "]]";
236
237   CDEnumParameter_Neg implements CDEnumParameter
238                                         astimplements mc.tfcs.ast.INegation =
239     "not" "[[" CDEnumParameter "]]";
240
241   CDEnumParameter_Opt implements CDEnumParameter
242                                          astimplements mc.tfcs.ast.IOptional =
243     "opt" "[[" CDEnumParameter "]]";
```

```
244
245   CDEnumParameter_Rep implements CDEnumParameter
246                                       astimplements mc.tfcs.ast.IReplacement =
247     ("[[" lhs:CDEnumParameter? ":-" rhs:CDEnumParameter? "]]");
248
249   CDEnumParameter_Pat implements CDEnumParameter
250                                       astimplements mc.tfcs.ast.IPattern =
251     (Value)
252     | "CDEnumParameter" schemaVarName:Name
253     | (("CDEnumParameter" schemaVarName:Name? |
254         "CDEnumParameter"? schemaVarName:Name) "[["
255         (Value)
256     "]]");
257
258   CDConstructor_List implements CDConstructor astimplements mc.tfcs.ast.IList =
259     "list" ("<CDConstructor>")? schemaVarName:Name? "[[" CDConstructor "]]";
260
261   CDConstructor_Neg implements CDConstructor astimplements mc.tfcs.ast.INegation =
262     "not" "[[" CDConstructor "]]";
263
264   CDConstructor_Opt implements CDConstructor astimplements mc.tfcs.ast.IOptional =
265     "opt" "[[" CDConstructor "]]";
266
267   CDConstructor_Rep implements CDConstructor
268                                       astimplements mc.tfcs.ast.IReplacement =
269     ("[[" lhs:CDConstructor? ":-" rhs:CDConstructor? "]]");
270
271   CDConstructor_Pat implements CDConstructor
272                                       astimplements mc.tfcs.ast.IPattern =
273     (ITFModifier Name:TfIdentifier
274     "(" (cDParameters:CDParameter ("," cDParameters:CDParameter)*)? ")"
275     ("throws" exceptions:ITFQualifiedName ("," exceptions:ITFQualifiedName)*)? ";")
276     | "CDConstructor" schemaVarName:Name
277     | (("CDConstructor" schemaVarName:Name? |
278        "CDConstructor"? schemaVarName:Name) "[["
279          (ITFModifier Name:TfIdentifier
280          "(" (cDParameters:CDParameter ("," cDParameters:CDParameter)*)? ")"
281          ("throws" exceptions:ITFQualifiedName ("," exceptions:ITFQualifiedName)*)?
282          ";")
283     "]]");
284
285   CDMethod_List implements CDMethod astimplements mc.tfcs.ast.IList =
286     "list" ("<CDMethod>")? schemaVarName:Name? "[[" CDMethod "]]";
287
288   CDMethod_Neg implements CDMethod astimplements mc.tfcs.ast.INegation =
289     "not" "[[" CDMethod "]]";
290
291   CDMethod_Opt implements CDMethod astimplements mc.tfcs.ast.IOptional =
292     "opt" "[[" CDMethod "]]";
293
294   CDMethod_Rep implements CDMethod astimplements mc.tfcs.ast.IReplacement =
295     ("[[" lhs:CDMethod? ":-" rhs:CDAttributeOrMethod? "]]");
296
297   CDMethod_Pat implements CDAttributeOrMethod, CDMethod =
298     (ITFModifier ITFReturnType Name:TfIdentifier
299     "(" (cDParameters:CDParameter ("," cDParameters:CDParameter)*)? ")"
300     ("throws" exceptions:ITFQualifiedName ("," exceptions:ITFQualifiedName)*)?
301     ";")
302     | "CDMethod" schemaVarName:Name
303     | (("CDMethod" schemaVarName:Name? | "CDMethod"? schemaVarName:Name )"[["
304      (ITFModifier ITFReturnType Name:TfIdentifier
305      "(" (cDParameters:CDParameter ("," cDParameters:CDParameter)*)? ")"
```

```
306       ("throws" exceptions:ITFQualifiedName ("," exceptions:ITFQualifiedName)*)? ";")
307     "]]");
308
309   CDParameter_List implements CDParameter astimplements mc.tfcs.ast.IList =
310     "list" ("<CDParameter>")? schemaVarName:Name? "[[" CDParameter "]]";
311
312   CDParameter_Neg implements CDParameter astimplements mc.tfcs.ast.INegation =
313     "not" "[[" CDParameter "]]";
314
315   CDParameter_Opt implements CDParameter astimplements mc.tfcs.ast.IOptional =
316     "opt" "[[" CDParameter "]]";
317
318   CDParameter_Rep implements CDParameter
319                                       astimplements mc.tfcs.ast.IReplacement =
320     ("[[" lhs:CDParameter? ":-" rhs:CDParameter? "]]");
321
322   CDParameter_Pat implements CDParameter astimplements mc.tfcs.ast.IPattern =
323     (ITFType (Ellipsis_Constant)? Name:TfIdentifier)
324     | "CDParameter" schemaVarName:Name
325     |(( "CDParameter" schemaVarName:Name? | "CDParameter"? schemaVarName:Name ) "[["
326       (ITFType (Ellipsis_Constant)? Name:TfIdentifier)
327     "]]");
328
329   CDAssociation_List implements CDAssociation astimplements mc.tfcs.ast.IList =
330     "list" ("<CDAssociation>")? schemaVarName:Name? "[[" CDAssociation "]]";
331
332   CDAssociation_Neg implements CDAssociation astimplements mc.tfcs.ast.INegation =
333     "not" "[[" CDAssociation "]]";
334
335   CDAssociation_Opt implements CDAssociation astimplements mc.tfcs.ast.IOptional =
336     "opt" "[[" CDAssociation "]]";
337
338   CDAssociation_Rep implements CDAssociation
339                                       astimplements mc.tfcs.ast.IReplacement =
340     ("[[" lhs:CDAssociation? ":-" rhs:CDAssociation? "]]");
341
342   CDAssociation_Pat implements CDAssociation astimplements mc.tfcs.ast.IPattern =
343     (Stereotype?
344     (Association_Constant |Composition_Constant)
345     (Derived_Constant)? Name:TfIdentifier? leftModifier:ITFModifier?
346     leftCardinality:Cardinality? leftReferenceName:ITFQualifiedName
347     ("[" leftQualifier:CDQualifier "]")? ("(" leftRole:TfIdentifier ")")?
348     (LeftToRight_Constant    | RightToLeft_Constant
349     | Bidirectional_Constant | Unspecified_Constant)
350     ("(" rightRole:TfIdentifier ")")? ("[" rightQualifier:CDQualifier "]")?
351     rightReferenceName:ITFQualifiedName rightCardinality:Cardinality?
352     rightModifier:ITFModifier? ";")
353     | "CDAssociation" schemaVarName:Name
354     | (("CDAssociation" schemaVarName:Name? |
355       "CDAssociation"? schemaVarName:Name ) "[["
356         (Stereotype? (Association_Constant | Composition_Constant)
357         (Derived_Constant)?
358         Name:TfIdentifier? leftModifier:ITFModifier? leftCardinality:Cardinality?
359         leftReferenceName:ITFQualifiedName ("[" leftQualifier:CDQualifier "]")?
360         ("(" leftRole:TfIdentifier ")")?
361         (LeftToRight_Constant    | RightToLeft_Constant
362         | Bidirectional_Constant | Unspecified_Constant)
363         ("(" rightRole:TfIdentifier ")")? ("[" rightQualifier:CDQualifier "]")?
364         rightReferenceName:ITFQualifiedName rightCardinality:Cardinality?
365         rightModifier:ITFModifier? ";")
366     "]]");
367
```

```
368  Modifier_List implements ITFModifier astimplements mc.tfcs.ast.IList =
369    "list" ("<Modifier>")? schemaVarName:Name? "[[" Modifier "]]";
370
371  Modifier_Neg implements ITFModifier astimplements mc.tfcs.ast.INegation =
372    "not" "[[" Modifier "]]";
373
374  Modifier_Opt implements ITFModifier astimplements mc.tfcs.ast.IOptional =
375    "opt" "[[" Modifier "]]";
376
377  Modifier_Rep implements ITFModifier astimplements mc.tfcs.ast.IReplacement =
378    ("[[" lhs:ITFModifier? ":-" rhs:Modifier? "]]");
379
380  Modifier_Pat implements ITFModifier astimplements mc.tfcs.ast.IPattern =
381    (Stereotype?
382     (Abstract_Constant   | Final_Constant  | Static_Constant | Private_Constant
383     | Protected_Constant | Public_Constant | Derived_Constant)*)
384    | "Modifier" schemaVarName:Name
385    | (("Modifier" schemaVarName:Name? | "Modifier"? schemaVarName:Name ) "[["
386      (Stereotype?
387      (Abstract_Constant   | Final_Constant  | Static_Constant | Private_Constant
388      | Protected_Constant | Public_Constant | Derived_Constant)*)
389    "]]");
390
391  Cardinality_List implements Cardinality astimplements mc.tfcs.ast.IList =
392    "list" ("<Cardinality>")? schemaVarName:Name? "[[" Cardinality "]]";
393
394  Cardinality_Neg implements Cardinality astimplements mc.tfcs.ast.INegation =
395    "not" "[[" Cardinality "]]";
396
397  Cardinality_Opt implements Cardinality astimplements mc.tfcs.ast.IOptional =
398    "opt" "[[" Cardinality "]]";
399
400  Cardinality_Rep implements Cardinality astimplements mc.tfcs.ast.IReplacement =
401    ("[[" lhs:Cardinality? ":-" rhs:Cardinality? "]]");
402
403  Cardinality_Pat implements Cardinality astimplements mc.tfcs.ast.IPattern =
404    (Many_Constant | One_Constant | OneToMany_Constant | Optional_Constant)
405    | "Cardinality" schemaVarName:Name
406    | (("Cardinality" schemaVarName:Name? | "Cardinality"? schemaVarName:Name) "[["
407    (Many_Constant | One_Constant | OneToMany_Constant | Optional_Constant)
408    "]]");
409
410  CDQualifier_List implements CDQualifier astimplements mc.tfcs.ast.IList =
411    "list" ("<CDQualifier>")? schemaVarName:Name? "[[" CDQualifier "]]";
412
413  CDQualifier_Neg implements CDQualifier astimplements mc.tfcs.ast.INegation =
414    "not" "[[" CDQualifier "]]";
415
416  CDQualifier_Opt implements CDQualifier astimplements mc.tfcs.ast.IOptional =
417    "opt" "[[" CDQualifier "]]";
418
419  CDQualifier_Rep implements CDQualifier astimplements mc.tfcs.ast.IReplacement =
420    ("[[" lhs:CDQualifier? ":-" rhs:CDQualifier? "]]");
421
422  CDQualifier_Pat implements CDQualifier astimplements mc.tfcs.ast.IPattern =
423    ("[[" Name:TfIdentifier "]]" | "[" ITFType "]")
424    | "CDQualifier" schemaVarName:Name
425    | (("CDQualifier" schemaVarName:Name? | "CDQualifier"? schemaVarName:Name) "[["
426      ("[[" Name:TfIdentifier "]]" | "[" ITFType "]")
427    "]]");
428
429  Stereotype_List implements Stereotype astimplements mc.tfcs.ast.IList =
```

```
430     "list" ("<Stereotype>")? schemaVarName:Name? "[[" Stereotype "]]";
431
432   Stereotype_Neg implements Stereotype astimplements mc.tfcs.ast.INegation =
433     "not" "[[" Stereotype "]]";
434
435   Stereotype_Opt implements Stereotype astimplements mc.tfcs.ast.IOptional =
436     "opt" "[[" Stereotype "]]";
437
438   Stereotype_Rep implements Stereotype astimplements mc.tfcs.ast.IReplacement =
439     ("[[" lhs:Stereotype? ":-" rhs:Stereotype? "]]");
440
441   Stereotype_Pat implements Stereotype astimplements mc.tfcs.ast.IPattern =
442     ("<<" values:StereoValue ("," values:StereoValue)* ">>")
443     | "Stereotype" schemaVarName:Name
444     | (("Stereotype" schemaVarName:Name? | "Stereotype"? schemaVarName:Name )"[["
445       ("<<" values:StereoValue ("," values:StereoValue)* ">>")
446     "]]");
447
448   StereoValue_List implements StereoValue astimplements mc.tfcs.ast.IList =
449     "list" ("<StereoValue>")? schemaVarName:Name? "[[" StereoValue "]]";
450
451   StereoValue_Neg implements StereoValue astimplements mc.tfcs.ast.INegation =
452     "not" "[[" StereoValue "]]";
453
454   StereoValue_Opt implements StereoValue astimplements mc.tfcs.ast.IOptional =
455     "opt" "[[" StereoValue "]]";
456
457   StereoValue_Rep implements StereoValue astimplements mc.tfcs.ast.IReplacement =
458     ("[[" lhs:StereoValue? ":-" rhs:StereoValue? "]]");
459
460   StereoValue_Pat implements StereoValue astimplements mc.tfcs.ast.IPattern =
461     (Name:TfIdentifier ("=" value:String)?)
462     | "StereoValue" schemaVarName:Name
463     | (("StereoValue" schemaVarName:Name? |"StereoValue"? schemaVarName:Name ) "[["
464       (Name:TfIdentifier ("=" value:String)?)
465     "]]");
466
467   Value_List implements Value astimplements mc.tfcs.ast.IList =
468     "list" ("<Value>")? schemaVarName:Name? "[[" Value "]]";
469
470   Value_Neg implements Value astimplements mc.tfcs.ast.INegation =
471     "not" "[[" Value "]]";
472
473   Value_Opt implements Value astimplements mc.tfcs.ast.IOptional =
474     "opt" "[[" Value "]]";
475
476   Value_Rep implements Value astimplements mc.tfcs.ast.IReplacement =
477     ("[[" lhs:Value? ":-" rhs:Value? "]]");
478
479   Value_Pat implements Value astimplements mc.tfcs.ast.IPattern =
480     (ITFSignedLiteral)
481     | "Value" schemaVarName:Name
482     | (("Value" schemaVarName:Name? | "Value"? schemaVarName:Name ) "[["
483      (ITFSignedLiteral)
484     "]]");
485
486     Optional_Constant_Neg implements Optional_Constant
487                                 astimplements mc.tfcs.ast.IAttributeNegation =
488     "not" "[[" Optional_Constant_Pat "]]";
489
490   Optional_Constant_Opt implements Optional_Constant
491                                 astimplements mc.tfcs.ast.IAttributeOptional =
```

```
492     "opt" "[[" Optional_Constant_Pat "]]";
493
494   Optional_Constant_Rep implements Optional_Constant
495                                 astimplements mc.tfcs.ast.IAttributeReplacement =
496     (("[[" lhs:Optional_Constant_Pat ":-" "]]")
497      |
498     ("[[" ":-" rhs:Optional_Constant_Pat "]]"));
499
500   Optional_Constant_Pat implements Optional_Constant
501                                  astimplements mc.tfcs.ast.IAttributePattern =
502     optional:["[0..1]"];
503
504   OneToMany_Constant_Neg implements OneToMany_Constant
505                                 astimplements mc.tfcs.ast.IAttributeNegation =
506     "not" "[[" OneToMany_Constant_Pat "]]";
507
508   OneToMany_Constant_Opt implements OneToMany_Constant
509                                 astimplements mc.tfcs.ast.IAttributeOptional =
510     "opt" "[[" OneToMany_Constant_Pat "]]";
511
512   OneToMany_Constant_Rep implements OneToMany_Constant
513                                 astimplements mc.tfcs.ast.IAttributeReplacement =
514     (("[[" lhs:OneToMany_Constant_Pat ":-" "]]")
515      |
516     ("[[" ":-" rhs:OneToMany_Constant_Pat "]]"));
517
518   OneToMany_Constant_Pat implements OneToMany_Constant
519                                  astimplements mc.tfcs.ast.IAttributePattern =
520     oneToMany:["[1..*]"];
521
522   One_Constant_Neg implements One_Constant
523                                 astimplements mc.tfcs.ast.IAttributeNegation =
524     "not" "[[" One_Constant_Pat "]]";
525
526   One_Constant_Opt implements One_Constant
527                                 astimplements mc.tfcs.ast.IAttributeOptional =
528     "opt" "[[" One_Constant_Pat "]]";
529
530   One_Constant_Rep implements One_Constant
531                                 astimplements mc.tfcs.ast.IAttributeReplacement =
532     (("[[" lhs:One_Constant_Pat ":-" "]]")
533      |
534     ("[[" ":-" rhs:One_Constant_Pat "]]"));
535
536   One_Constant_Pat implements One_Constant
537                                  astimplements mc.tfcs.ast.IAttributePattern =
538     one:["[1]"];
539
540   Many_Constant_Neg implements Many_Constant
541                                 astimplements mc.tfcs.ast.IAttributeNegation =
542     "not" "[[" Many_Constant_Pat "]]";
543
544   Many_Constant_Opt implements Many_Constant
545                                 astimplements mc.tfcs.ast.IAttributeOptional =
546     "opt" "[[" Many_Constant_Pat "]]";
547
548   Many_Constant_Rep implements Many_Constant
549                                 astimplements mc.tfcs.ast.IAttributeReplacement =
550     (("[[" lhs:Many_Constant_Pat ":-" "]]")
551      |
552     ("[[" ":-" rhs:Many_Constant_Pat "]]"));
553
```

```
554  Many_Constant_Pat implements Many_Constant
555                                  astimplements mc.tfcs.ast.IAttributePattern =
556    many:["[*]"];
557
558  Public_Constant_Neg implements Public_Constant
559                                  astimplements mc.tfcs.ast.IAttributeNegation =
560    "not" "[[" Public_Constant_Pat "]]";
561
562  Public_Constant_Opt implements Public_Constant
563                                  astimplements mc.tfcs.ast.IAttributeOptional =
564    "opt" "[[" Public_Constant_Pat "]]";
565
566  Public_Constant_Rep implements Public_Constant
567                               astimplements mc.tfcs.ast.IAttributeReplacement =
568    (("[[" lhs:Public_Constant_Pat ":-" "]]")
569     |
570    ("[[" ":-" rhs:Public_Constant_Pat "]]"));
571
572  Public_Constant_Pat implements Public_Constant
573                                  astimplements mc.tfcs.ast.IAttributePattern =
574    Public:["public"];
575
576  Protected_Constant_Neg implements Protected_Constant
577                                  astimplements mc.tfcs.ast.IAttributeNegation =
578    "not" "[[" Protected_Constant_Pat "]]";
579
580  Protected_Constant_Opt implements Protected_Constant
581                                  astimplements mc.tfcs.ast.IAttributeOptional =
582    "opt" "[[" Protected_Constant_Pat "]]";
583
584  Protected_Constant_Rep implements Protected_Constant
585                               astimplements mc.tfcs.ast.IAttributeReplacement =
586    (("[[" lhs:Protected_Constant_Pat ":-" "]]")
587     |
588    ("[[" ":-" rhs:Protected_Constant_Pat "]]"));
589
590  Protected_Constant_Pat implements Protected_Constant
591                                  astimplements mc.tfcs.ast.IAttributePattern =
592    Protected:["protected"];
593
594  Private_Constant_Neg implements Private_Constant
595                                  astimplements mc.tfcs.ast.IAttributeNegation =
596    "not" "[[" Private_Constant_Pat "]]";
597
598  Private_Constant_Opt implements Private_Constant
599                                  astimplements mc.tfcs.ast.IAttributeOptional =
600    "opt" "[[" Private_Constant_Pat "]]";
601
602  Private_Constant_Rep implements Private_Constant
603                               astimplements mc.tfcs.ast.IAttributeReplacement =
604    (("[[" lhs:Private_Constant_Pat ":-" "]]")
605     |
606    ("[[" ":-" rhs:Private_Constant_Pat "]]"));
607
608  Private_Constant_Pat implements Private_Constant
609                                  astimplements mc.tfcs.ast.IAttributePattern =
610    Private:["private"];
611
612  Static_Constant_Neg implements Static_Constant
613                                  astimplements mc.tfcs.ast.IAttributeNegation =
614    "not" "[[" Static_Constant_Pat "]]";
615
```

```
616  Static_Constant_Opt implements Static_Constant
617                                  astimplements mc.tfcs.ast.IAttributeOptional =
618    "opt" "[[" Static_Constant_Pat "]]";
619
620  Static_Constant_Rep implements Static_Constant
621                               astimplements mc.tfcs.ast.IAttributeReplacement =
622    (("[[" lhs:Static_Constant_Pat ":-" "]]")
623     |
624    ("[[" ":-" rhs:Static_Constant_Pat "]]"));
625
626  Static_Constant_Pat implements Static_Constant
627                                   astimplements mc.tfcs.ast.IAttributePattern =
628    Static:["static"];
629
630  Final_Constant_Neg implements Final_Constant
631                                  astimplements mc.tfcs.ast.IAttributeNegation =
632    "not" "[[" Final_Constant_Pat "]]";
633
634  Final_Constant_Opt implements Final_Constant
635                                  astimplements mc.tfcs.ast.IAttributeOptional =
636    "opt" "[[" Final_Constant_Pat "]]";
637
638  Final_Constant_Rep implements Final_Constant
639                               astimplements mc.tfcs.ast.IAttributeReplacement =
640    (("[[" lhs:Final_Constant_Pat ":-" "]]")
641     |
642    ("[[" ":-" rhs:Final_Constant_Pat "]]"));
643
644  Final_Constant_Pat implements Final_Constant
645                                   astimplements mc.tfcs.ast.IAttributePattern =
646    Final:["final"];
647
648  Abstract_Constant_Neg implements Abstract_Constant
649                                  astimplements mc.tfcs.ast.IAttributeNegation =
650    "not" "[[" Abstract_Constant_Pat "]]";
651
652  Abstract_Constant_Opt implements Abstract_Constant
653                                  astimplements mc.tfcs.ast.IAttributeOptional =
654    "opt" "[[" Abstract_Constant_Pat "]]";
655
656  Abstract_Constant_Rep implements Abstract_Constant
657                               astimplements mc.tfcs.ast.IAttributeReplacement =
658    (("[[" lhs:Abstract_Constant_Pat ":-" "]]")
659     |
660    ("[[" ":-" rhs:Abstract_Constant_Pat "]]"));
661
662  Abstract_Constant_Pat implements Abstract_Constant
663                                   astimplements mc.tfcs.ast.IAttributePattern =
664    Abstract:["abstract"];
665
666  Unspecified_Constant_Neg implements Unspecified_Constant
667                                  astimplements mc.tfcs.ast.IAttributeNegation =
668    "not" "[[" Unspecified_Constant_Pat "]]";
669
670  Unspecified_Constant_Opt implements Unspecified_Constant
671                                  astimplements mc.tfcs.ast.IAttributeOptional =
672    "opt" "[[" Unspecified_Constant_Pat "]]";
673
674  Unspecified_Constant_Rep implements Unspecified_Constant
675                               astimplements mc.tfcs.ast.IAttributeReplacement =
676    (("[[" lhs:Unspecified_Constant_Pat ":-" "]]")
677     |
```

```
678     ("[[" ":-" rhs:Unspecified_Constant_Pat "]]"));
679
680   Unspecified_Constant_Pat implements Unspecified_Constant
681                                astimplements mc.tfcs.ast.IAttributePattern =
682     unspecified:["--"];
683
684   Bidirectional_Constant_Neg implements Bidirectional_Constant
685                               astimplements mc.tfcs.ast.IAttributeNegation =
686     "not" "[[" Bidirectional_Constant_Pat "]]";
687
688   Bidirectional_Constant_Opt implements Bidirectional_Constant
689                               astimplements mc.tfcs.ast.IAttributeOptional =
690     "opt" "[[" Bidirectional_Constant_Pat "]]";
691
692   Bidirectional_Constant_Rep implements Bidirectional_Constant
693                              astimplements mc.tfcs.ast.IAttributeReplacement =
694     (("[[" lhs:Bidirectional_Constant_Pat ":-" "]]")
695      |
696     ("[[" ":-" rhs:Bidirectional_Constant_Pat "]]"));
697
698   Bidirectional_Constant_Pat implements Bidirectional_Constant
699                                astimplements mc.tfcs.ast.IAttributePattern =
700     bidirectional:["<->"];
701
702   RightToLeft_Constant_Neg implements RightToLeft_Constant
703                               astimplements mc.tfcs.ast.IAttributeNegation =
704     "not" "[[" RightToLeft_Constant_Pat "]]";
705
706   RightToLeft_Constant_Opt implements RightToLeft_Constant
707                               astimplements mc.tfcs.ast.IAttributeOptional =
708     "opt" "[[" RightToLeft_Constant_Pat "]]";
709
710   RightToLeft_Constant_Rep implements RightToLeft_Constant
711                              astimplements mc.tfcs.ast.IAttributeReplacement =
712     (("[[" lhs:RightToLeft_Constant_Pat ":-" "]]")
713      |
714     ("[[" ":-" rhs:RightToLeft_Constant_Pat "]]"));
715
716   RightToLeft_Constant_Pat implements RightToLeft_Constant
717                                astimplements mc.tfcs.ast.IAttributePattern =
718     rightToLeft:["<-"];
719
720   LeftToRight_Constant_Neg implements LeftToRight_Constant
721                               astimplements mc.tfcs.ast.IAttributeNegation =
722     "not" "[[" LeftToRight_Constant_Pat "]]";
723
724   LeftToRight_Constant_Opt implements LeftToRight_Constant
725                               astimplements mc.tfcs.ast.IAttributeOptional =
726     "opt" "[[" LeftToRight_Constant_Pat "]]";
727
728   LeftToRight_Constant_Rep implements LeftToRight_Constant
729                              astimplements mc.tfcs.ast.IAttributeReplacement =
730     (("[[" lhs:LeftToRight_Constant_Pat ":-" "]]")
731      |
732     ("[[" ":-" rhs:LeftToRight_Constant_Pat "]]"));
733
734   LeftToRight_Constant_Pat implements LeftToRight_Constant
735                                astimplements mc.tfcs.ast.IAttributePattern =
736     leftToRight:["->"];
737
738   Derived_Constant_Neg implements Derived_Constant
739                               astimplements mc.tfcs.ast.IAttributeNegation =
```

```
740     "not" "[[" Derived_Constant_Pat "]]";
741
742   Derived_Constant_Opt implements Derived_Constant
743                                 astimplements mc.tfcs.ast.IAttributeOptional =
744     "opt" "[[" Derived_Constant_Pat "]]";
745
746   Derived_Constant_Rep implements Derived_Constant
747                               astimplements mc.tfcs.ast.IAttributeReplacement =
748     (("[[" lhs:Derived_Constant_Pat ":-" "]]")
749      |
750     ("[[" ":-" rhs:Derived_Constant_Pat "]]"));
751
752   Derived_Constant_Pat implements Derived_Constant
753                                  astimplements mc.tfcs.ast.IAttributePattern =
754     Derived:[DERIVED:"/"];
755
756   Composition_Constant_Neg implements Composition_Constant
757                                 astimplements mc.tfcs.ast.IAttributeNegation =
758     "not" "[[" Composition_Constant_Pat "]]";
759
760   Composition_Constant_Opt implements Composition_Constant
761                                 astimplements mc.tfcs.ast.IAttributeOptional =
762     "opt" "[[" Composition_Constant_Pat "]]";
763
764   Composition_Constant_Rep implements Composition_Constant
765                               astimplements mc.tfcs.ast.IAttributeReplacement =
766     (("[[" lhs:Composition_Constant_Pat ":-" "]]")
767      |
768     ("[[" ":-" rhs:Composition_Constant_Pat "]]"));
769
770   Composition_Constant_Pat implements Composition_Constant
771                                  astimplements mc.tfcs.ast.IAttributePattern =
772     Composition:["composition"];
773
774   Association_Constant_Neg implements Association_Constant
775                                 astimplements mc.tfcs.ast.IAttributeNegation =
776     "not" "[[" Association_Constant_Pat "]]";
777
778   Association_Constant_Opt implements Association_Constant
779                                 astimplements mc.tfcs.ast.IAttributeOptional =
780     "opt" "[[" Association_Constant_Pat "]]";
781
782   Association_Constant_Rep implements Association_Constant
783                               astimplements mc.tfcs.ast.IAttributeReplacement =
784     (("[[" lhs:Association_Constant_Pat ":-" "]]")
785      |
786     ("[[" ":-" rhs:Association_Constant_Pat "]]"));
787
788   Association_Constant_Pat implements Association_Constant
789                                 astimplements mc.tfcs.ast.IAttributePattern =
790     Association:["association"];
791
792   Ellipsis_Constant_Neg implements Ellipsis_Constant
793                                 astimplements mc.tfcs.ast.IAttributeNegation =
794     "not" "[[" Ellipsis_Constant_Pat "]]";
795
796   Ellipsis_Constant_Opt implements Ellipsis_Constant
797                                 astimplements mc.tfcs.ast.IAttributeOptional =
798     "opt" "[[" Ellipsis_Constant_Pat "]]";
799
800   Ellipsis_Constant_Rep implements Ellipsis_Constant
801                               astimplements mc.tfcs.ast.IAttributeReplacement =
```

```
802     (("[[" lhs:Ellipsis_Constant_Pat ":-" "]]")
803      |
804     ("[[" ":-" rhs:Ellipsis_Constant_Pat "]]"));
805
806   Ellipsis_Constant_Pat implements Ellipsis_Constant
807                                 astimplements mc.tfcs.ast.IAttributePattern =
808     Ellipsis:["..."];
809 }
```

Listing 3.2: Grammatik der DSTL CDTrans.

C.3 Grammatik der DSTL MATrans

MCG

```
1  package de.monticore.lang.montiarc.tr;
2
3  grammar MontiArcTR extends de.monticore.lang.montiarc.tr.CommonTR {
4
5  TFRule =
6    ( Component       | MACompilationUnit    | ComponentBody        | Interface
7    | Port            | SubComponent         | SubComponentInstance | Connector
8    | SimpleConnector | MontiArcInvariant    | MontiArcAutoConnect  | Element
9    | MontiArcConfig)* TFFolding? TFAssignments? TFWhere? TFDo?;
10
11   interface Component extends Element         astextends mc.tfcs.ast.ITFElement;
12   interface MACompilationUnit                 astextends mc.tfcs.ast.ITFElement;
13   interface ComponentHead                     astextends mc.tfcs.ast.ITFElement;
14   interface ComponentBody                     astextends mc.tfcs.ast.ITFElement;
15   interface Port                              astextends mc.tfcs.ast.ITFElement;
16   interface Interface extends Element         astextends mc.tfcs.ast.ITFElement;
17   interface SubComponent extends Element      astextends mc.tfcs.ast.ITFElement;
18   interface SubComponentInstance              astextends mc.tfcs.ast.ITFElement;
19   interface Connector extends Element         astextends mc.tfcs.ast.ITFElement;
20   interface SimpleConnector                   astextends mc.tfcs.ast.ITFElement;
21   interface MontiArcInvariant extends Element astextends mc.tfcs.ast.ITFElement;
22   interface MontiArcAutoConnect extends MontiArcConfig
23                                               astextends mc.tfcs.ast.ITFElement;
24   interface MontiArcConfig extends Element    astextends mc.tfcs.ast.ITFElement;
25   interface Element                           astextends mc.tfcs.ast.ITFElement;
26   interface Outgoing_Constant;
27   interface Incoming_Constant;
28   interface Sync_Constant;
29   interface Port_Constant;
30   interface Causalsync_Constant;
31   interface Untimed_Constant;
32   interface Delayed_Constant;
33   interface Instant_Constant;
34   interface On_Constant;
35   interface Off_Constant;
36   interface Type_Constant;
37
38   MACompilationUnit_List implements  MACompilationUnit
39                                           astimplements mc.tfcs.ast.IList =
40     "list" ("<MACompilationUnit>")? schemaVarName:Name? "[[" MACompilationUnit "]]";
41
42   MACompilationUnit_Neg implements  MACompilationUnit
43                                        astimplements mc.tfcs.ast.INegation =
44     "not" "[[" MACompilationUnit "]]";
45
46   MACompilationUnit_Opt implements  MACompilationUnit
47                                     astimplements mc.tfcs.ast.IOptional =
48   "opt" "[[" MACompilationUnit "]]";
49
50   MACompilationUnit_Rep implements  MACompilationUnit
51                                   astimplements mc.tfcs.ast.IReplacement =
52     ("[[" lhs:MACompilationUnit? ":-" rhs:MACompilationUnit? "]]");
53
54   MACompilationUnit_Pat implements MACompilationUnit
55                                        astimplements mc.tfcs.ast.IPattern =
56     (("package" packagee:TfIdentifier ("." packagee:TfIdentifier)* ";")?
57     ( ITFImportStatement)* Component)
```

```
58      | "MACompilationUnit" schemaVarName:Name
59      | (("MACompilationUnit" schemaVarName:Name? |
60      "MACompilationUnit"? schemaVarName:Name )"[["
61        (("package" packagee:TfIdentifier ("." packagee:TfIdentifier)* ";")?
62        ( ITFImportStatement)* Component)
63      "]]");
64
65
66    Component_List implements  Component astimplements mc.tfcs.ast.IList =
67      "list" ("<Component>")? schemaVarName:Name? "[[" Component"]]";
68
69    Component_Neg implements  Component astimplements mc.tfcs.ast.INegation =
70      "not" "[[" Component"]]";
71
72    Component_Opt implements  Component astimplements mc.tfcs.ast.IOptional =
73      "opt" "[[" Component"]]";
74
75    Component_Rep implements  Component astimplements mc.tfcs.ast.IReplacement =
76      ("[[" lhs:Component? ":-" rhs:Component? "]]");
77
78    Component_Pat implements Component astimplements mc.tfcs.ast.IPattern =
79      (ITFStereotype? "component" Name:TfIdentifier ( instanceName:TfIdentifier)?
80      head:ComponentHead body:ComponentBody)
81      | "Component" schemaVarName:Name
82      | (("Component" schemaVarName:Name? | "Component"? schemaVarName:Name )"[["
83        (ITFStereotype? "component" Name:TfIdentifier ( instanceName:TfIdentifier)?
84        head:ComponentHead body:ComponentBody)
85      "]]");
86
87    ComponentHead_List implements ComponentHead astimplements mc.tfcs.ast.IList =
88      "list" ("<ComponentHead>")? schemaVarName:Name? "[[" ComponentHead "]]";
89
90    ComponentHead_Neg implements ComponentHead astimplements mc.tfcs.ast.INegation =
91      "not" "[[" ComponentHead "]]";
92
93    ComponentHead_Opt implements ComponentHead astimplements mc.tfcs.ast.IOptional =
94      "opt" "[[" ComponentHead "]]";
95
96    ComponentHead_Rep implements ComponentHead
97                                        astimplements mc.tfcs.ast.IReplacement =
98      ("[[" lhs:ComponentHead? ":-" rhs:ComponentHead? "]]");
99
100   ComponentHead_Pat implements ComponentHead astimplements mc.tfcs.ast.IPattern =
101     (( genericTypeParameters:ITFTypeParameters)?
102     ("[" ITFParameter ("," ITFParameter)* "]")?
103     ("extends" superComponent:ITFReferenceType)?)
104     | "ComponentHead" schemaVarName:Name
105     | (("ComponentHead" schemaVarName:Name?|
106         "ComponentHead"? schemaVarName:Name) "[["
107           (( genericTypeParameters:ITFTypeParameters)?
108           ("[" ITFParameter ("," ITFParameter)* "]")?
109           ("extends" superComponent:ITFReferenceType)?)
110     "]]");
111
112   ComponentBody_List implements  ComponentBody astimplements mc.tfcs.ast.IList =
113     "list" ("<ComponentBody>")? schemaVarName:Name? "[[" ComponentBody "]]";
114
115   ComponentBody_Neg implements  ComponentBody astimplements mc.tfcs.ast.INegation =
116     "not" "[[" ComponentBody "]]";
117
118   ComponentBody_Opt implements  ComponentBody astimplements mc.tfcs.ast.IOptional =
119     "opt" "[[" ComponentBody "]]";
```

```
120
121   ComponentBody_Rep implements  ComponentBody
122                                       astimplements mc.tfcs.ast.IReplacement =
123     ("[[" lhs:ComponentBody? ":-" rhs:ComponentBody? "]]");
124
125   ComponentBody_Pat implements ComponentBody astimplements mc.tfcs.ast.IPattern =
126     ("{" Element* "}")
127     | "ComponentBody" schemaVarName:Name
128     | (("ComponentBody" schemaVarName:Name? | "ComponentBody"? schemaVarName:Name)
129     "[[" ("{" Element* "}") "]]");
130
131   Interface_List implements  Interface astimplements mc.tfcs.ast.IList =
132     "list" ("<Interface>")? schemaVarName:Name? "[[" Interface"]]";
133
134   Interface_Neg implements Interface astimplements mc.tfcs.ast.INegation =
135     "not" "[[" Interface"]]";
136
137     Interface_Opt implements Interface astimplements mc.tfcs.ast.IOptional =
138     "opt" "[[" Interface"]]";
139
140   Interface_Rep implements  Interface astimplements mc.tfcs.ast.IReplacement =
141     ("[[" lhs:Interface? ":-" rhs:Interface? "]]");
142
143   Interface_Pat implements Interface astimplements mc.tfcs.ast.IPattern =
144     (ITFStereotype? ("port" | "ports") ports:Port ("," ports:Port)* ";")
145     | "Interface" schemaVarName:Name
146     | (("Interface" schemaVarName:Name? | "Interface"? schemaVarName:Name) "[["
147       (ITFStereotype? ("port" | "ports") ports:Port ("," ports:Port)* ";")
148     "]]");
149
150   Port_List implements  Port astimplements mc.tfcs.ast.IList =
151     "list" ("<Port>")? schemaVarName:Name? "[[" Port"]]";
152
153   Port_Neg implements  Port astimplements mc.tfcs.ast.INegation =
154     "not" "[[" Port"]]";
155
156   Port_Opt implements  Port astimplements mc.tfcs.ast.IOptional =
157     "opt" "[[" Port"]]";
158
159   Port_Rep implements  Port astimplements mc.tfcs.ast.IReplacement =
160     ("[[" lhs:Port? ":-" rhs:Port? "]]");
161
162   Port_Pat implements Port astimplements mc.tfcs.ast.IPattern =
163     (ITFStereotype? (Incoming_Constant | Outgoing_Constant| dC:TfIdentifier)
164      ITFType Name:TfIdentifier?)
165     | "Port" schemaVarName:Name
166     | (("Port" schemaVarName:Name?| "Port"? schemaVarName:Name) "[["
167       (ITFStereotype? (Incoming_Constant | Outgoing_Constant |  dC:TfIdentifier)
168       ITFType Name:TfIdentifier?)
169     "]]");
170
171
172   SubComponent_List implements  SubComponent astimplements mc.tfcs.ast.IList =
173     "list" ("<SubComponent>")? schemaVarName:Name? "[[" SubComponent "]]";
174
175   SubComponent_Neg implements  SubComponent astimplements mc.tfcs.ast.INegation =
176     "not" "[[" SubComponent "]]";
177
178   SubComponent_Opt implements  SubComponent astimplements mc.tfcs.ast.IOptional =
179     "opt" "[[" SubComponent "]]";
180
181   SubComponent_Rep implements  SubComponent astimplements mc.tfcs.ast.IReplacement =
```

```
182     ("[[" lhs:SubComponent? ":-" rhs:SubComponent? "]]");
183
184   SubComponent_Pat implements SubComponent astimplements mc.tfcs.ast.IPattern =
185     (ITFStereotype? "component" type:ITFReferenceType
186     ("(" arguments:Expression ("," arguments:Expression)* ")")?
187     (instances:SubComponentInstance ("," instances:SubComponentInstance)*)? ";")
188     | "SubComponent" schemaVarName:Name
189     | (("SubComponent" schemaVarName:Name? |
190         "SubComponent"? schemaVarName:Name) "[["
191           (ITFStereotype? "component" type:ITFReferenceType
192           ("(" arguments:Expression ("," arguments:Expression)* ")")?
193           (instances:SubComponentInstance ("," instances:SubComponentInstance)*)?
194           ";")
195     "]]");
196
197   SubComponentInstance_List implements  SubComponentInstance
198                                           astimplements mc.tfcs.ast.IList =
199     "list" ("<SubComponentInstance>")? schemaVarName:Name? "[["
200       SubComponentInstance
201     "]]";
202
203   SubComponentInstance_Neg implements  SubComponentInstance
204                                          astimplements mc.tfcs.ast.INegation =
205     "not" "[[" SubComponentInstance "]]";
206
207   SubComponentInstance_Opt implements  SubComponentInstance
208                                          astimplements mc.tfcs.ast.IOptional =
209     "opt" "[[" SubComponentInstance "]]";
210
211   SubComponentInstance_Rep implements  SubComponentInstance
212                                         astimplements mc.tfcs.ast.IReplacement =
213     ("[[" lhs:SubComponentInstance? ":-" rhs:SubComponentInstance? "]]");
214
215   SubComponentInstance_Pat implements SubComponentInstance
216                                           astimplements mc.tfcs.ast.IPattern =
217     (Name:TfIdentifier
218     ("[" connectors:SimpleConnector (";" connectors:SimpleConnector)* "]")?)
219     | "SubComponentInstance" schemaVarName:Name
220     | (("SubComponentInstance" schemaVarName:Name? |
221       "SubComponentInstance"? schemaVarName:Name) "[["
222         (Name:TfIdentifier
223         ("[" connectors:SimpleConnector (";" connectors:SimpleConnector)* "]")?)
224     "]]");
225
226   Connector_List implements  Connector astimplements mc.tfcs.ast.IList =
227     "list" ("<Connector>")? schemaVarName:Name? "[[" Connector"]]";
228
229   Connector_Neg implements  Connector astimplements mc.tfcs.ast.INegation =
230     "not" "[[" Connector "]]";
231
232   Connector_Opt implements  Connector astimplements mc.tfcs.ast.IOptional =
233     "opt" "[[" Connector "]]";
234
235   Connector_Rep implements  Connector astimplements mc.tfcs.ast.IReplacement =
236     ("[[" lhs:Connector? ":-" rhs:Connector? "]]");
237
238   Connector_Pat implements Connector astimplements mc.tfcs.ast.IPattern =
239     (ITFStereotype? "connect" source:ITFQualifiedName
240     "->" targets:ITFQualifiedName ("," targets:ITFQualifiedName)* ";")
241     | "Connector" schemaVarName:Name
242     |(("Connector" schemaVarName:Name? | "Connector"? schemaVarName:Name) "[["
243       (ITFStereotype? "connect" source:ITFQualifiedName
```

```
244         "->" targets:ITFQualifiedName ("," targets:ITFQualifiedName)* ";")
245     "]]");
246
247   SimpleConnector_List implements  SimpleConnector astimplements mc.tfcs.ast.IList =
248     "list" ("<SimpleConnector>")? schemaVarName:Name? "[[" SimpleConnector "]]";
249
250   SimpleConnector_Neg implements  SimpleConnector
251                                       astimplements mc.tfcs.ast.INegation =
252     "not" "[[" SimpleConnector "]]";
253
254   SimpleConnector_Opt implements  SimpleConnector
255                                       astimplements mc.tfcs.ast.IOptional =
256     "opt" "[[" SimpleConnector "]]";
257
258   SimpleConnector_Rep implements  SimpleConnector
259                                     astimplements mc.tfcs.ast.IReplacement =
260     ("[[" lhs:SimpleConnector? ":-" rhs:SimpleConnector? "]]");
261
262   SimpleConnector_Pat implements SimpleConnector
263                                        astimplements mc.tfcs.ast.IPattern =
264     (ITFStereotype? source:ITFQualifiedName
265     "->" targets:ITFQualifiedName ("," targets:ITFQualifiedName)*)
266     | "SimpleConnector" schemaVarName:Name
267     |(( "SimpleConnector" schemaVarName:Name? |
268       "SimpleConnector"? schemaVarName:Name) "[["
269         (ITFStereotype? source:ITFQualifiedName
270         "->" targets:ITFQualifiedName ("," targets:ITFQualifiedName)*)
271     "]]");
272
273   MontiArcInvariant_List implements  MontiArcInvariant
274                                          astimplements mc.tfcs.ast.IList =
275     "list" ("<MontiArcInvariant>")? schemaVarName:Name? "[[" MontiArcInvariant "]]";
276
277   MontiArcInvariant_Neg implements  MontiArcInvariant
278                                       astimplements mc.tfcs.ast.INegation =
279     "not" "[[" MontiArcInvariant "]]";
280
281   MontiArcInvariant_Opt implements  MontiArcInvariant
282                                        astimplements mc.tfcs.ast.IOptional =
283     "opt" "[[" MontiArcInvariant "]]";
284
285   MontiArcInvariant_Rep implements  MontiArcInvariant
286                                     astimplements mc.tfcs.ast.IReplacement =
287     ("[[" lhs:MontiArcInvariant? ":-" rhs:MontiArcInvariant? "]]");
288
289   MontiArcInvariant_Pat implements MontiArcInvariant
290                                        astimplements mc.tfcs.ast.IPattern =
291     (( kind:TfIdentifier)?
292     "inv" Name:TfIdentifier ":" invariantExpression:ITFInvariantContent ";")
293     | "MontiArcInvariant" schemaVarName:Name
294     | (("MontiArcInvariant" schemaVarName:Name? |
295       "MontiArcInvariant"? schemaVarName:Name) "[["
296         (( kind:TfIdentifier)?
297         "inv" Name:TfIdentifier ":" invariantExpression:ITFInvariantContent ";")
298     "]]");
299
300   MontiArcAutoConnect_List implements  MontiArcAutoConnect
301                                            astimplements mc.tfcs.ast.IList =
302     "list" ("<MontiArcAutoConnect>")? schemaVarName:Name? "[["
303       MontiArcAutoConnect
304     "]]";
305
```

```
306  MontiArcAutoConnect_Neg implements  MontiArcAutoConnect
307                                       astimplements mc.tfcs.ast.INegation =
308    "not" "[[" MontiArcAutoConnect "]]";
309
310  MontiArcAutoConnect_Opt implements  MontiArcAutoConnect
311                                       astimplements mc.tfcs.ast.IOptional =
312    "opt" "[[" MontiArcAutoConnect "]]";
313
314  MontiArcAutoConnect_Rep implements  MontiArcAutoConnect
315                                      astimplements mc.tfcs.ast.IReplacement =
316    ("[[" lhs:MontiArcAutoConnect? ":-" rhs:MontiArcAutoConnect? "]]");
317
318  MontiArcAutoConnect_Pat implements MontiArcAutoConnect
319                                       astimplements mc.tfcs.ast.IPattern =
320   ("autoconnect" ITFStereotype? (Type_Constant | Port_Constant | Off_Constant) ";")
321   | "MontiArcAutoConnect" schemaVarName:Name
322   | (("MontiArcAutoConnect" schemaVarName:Name? |
323      "MontiArcAutoConnect"? schemaVarName:Name) "[["
324        ("autoconnect" ITFStereotype? (Type_Constant | Port_Constant | Off_Constant)
325        ";")
326   "]]");
327
328  MontiArcConfig_Pat implements MontiArcConfig astimplements mc.tfcs.ast.IPattern =
329    "MontiArcConfig" schemaVarName:Name | "MontiArcConfig" schemaVarName:Name? "[["
330      MontiArcConfig
331    "]]";
332
333  MontiArcConfig_List implements  MontiArcConfig astimplements mc.tfcs.ast.IList =
334    "list" ("<MontiArcConfig>")? schemaVarName:Name? "[[" MontiArcConfig "]]";
335
336  MontiArcConfig_Neg implements  MontiArcConfig
337                                       astimplements mc.tfcs.ast.INegation =
338    "not" "[[" MontiArcConfig "]]";
339
340  MontiArcConfig_Opt implements  MontiArcConfig
341                                       astimplements mc.tfcs.ast.IOptional =
342    "opt" "[[" MontiArcConfig "]]";
343
344  MontiArcConfig_Rep implements  MontiArcConfig
345                                     astimplements mc.tfcs.ast.IReplacement =
346    ("[[" lhs:MontiArcConfig? ":-" rhs:MontiArcConfig? "]]");
347
348  Element_Pat implements Element astimplements mc.tfcs.ast.IPattern =
349    "Element" schemaVarName:Name | "Element" schemaVarName:Name? "[[" Element "]]";
350
351  Element_List implements  Element astimplements mc.tfcs.ast.IList =
352    "list" ("<Element>")? schemaVarName:Name? "[[" Element"]]";
353
354  Element_Neg implements  Element astimplements mc.tfcs.ast.INegation =
355    "not" "[[" Element"]]";
356
357  Element_Opt implements  Element astimplements mc.tfcs.ast.IOptional =
358    "opt" "[[" Element"]]";
359
360  Element_Rep implements  Element astimplements mc.tfcs.ast.IReplacement =
361    ("[[" lhs:Element? ":-" rhs:Element? "]]");
362
363  Port_Constant_Neg implements Port_Constant
364                                  astimplements mc.tfcs.ast.IAttributeNegation =
365    "not" "[[" Port_Constant_Pat "]]";
366
367  Port_Constant_Opt implements Port_Constant
```

```
368                                 astimplements mc.tfcs.ast.IAttributeOptional =
369     "opt" "[[" Port_Constant_Pat "]]";
370
371   Port_Constant_Rep implements Port_Constant
372                               astimplements mc.tfcs.ast.IAttributeReplacement =
373     (("[[" lhs:Port_Constant_Pat ":-" "]]")
374      |
375     ("[[" ":-" rhs:Port_Constant_Pat "]]"));
376
377   Port_Constant_Pat implements Port_Constant
378                                  astimplements mc.tfcs.ast.IAttributePattern =
379     port:["port"];
380
381   Sync_Constant_Neg implements Sync_Constant
382                                 astimplements mc.tfcs.ast.IAttributeNegation =
383     "not" "[[" Sync_Constant_Pat "]]";
384
385   Sync_Constant_Opt implements Sync_Constant
386                                 astimplements mc.tfcs.ast.IAttributeOptional =
387     "opt" "[[" Sync_Constant_Pat "]]";
388
389   Sync_Constant_Rep implements Sync_Constant
390                               astimplements mc.tfcs.ast.IAttributeReplacement =
391     (("[[" lhs:Sync_Constant_Pat ":-" "]]")
392      |
393     ("[[" ":-" rhs:Sync_Constant_Pat "]]"));
394
395   Sync_Constant_Pat implements Sync_Constant
396                                  astimplements mc.tfcs.ast.IAttributePattern =
397     Sync:["sync"];
398
399   Causalsync_Constant_Neg implements Causalsync_Constant
400                                 astimplements mc.tfcs.ast.IAttributeNegation =
401     "not" "[[" Causalsync_Constant_Pat "]]";
402
403   Causalsync_Constant_Opt implements Causalsync_Constant
404                                 astimplements mc.tfcs.ast.IAttributeOptional =
405     "opt" "[[" Causalsync_Constant_Pat "]]";
406
407   Causalsync_Constant_Rep implements Causalsync_Constant
408                               astimplements mc.tfcs.ast.IAttributeReplacement =
409     (("[[" lhs:Causalsync_Constant_Pat ":-" "]]")
410      |
411     ("[[" ":-" rhs:Causalsync_Constant_Pat "]]"));
412
413   Causalsync_Constant_Pat implements Causalsync_Constant
414                                  astimplements mc.tfcs.ast.IAttributePattern =
415     Causalsync:["causalsync"];
416
417   Untimed_Constant_Neg implements Untimed_Constant
418                                 astimplements mc.tfcs.ast.IAttributeNegation =
419     "not" "[[" Untimed_Constant_Pat "]]";
420
421   Untimed_Constant_Opt implements Untimed_Constant
422                                 astimplements mc.tfcs.ast.IAttributeOptional =
423     "opt" "[[" Untimed_Constant_Pat "]]";
424
425   Untimed_Constant_Rep implements Untimed_Constant
426                               astimplements mc.tfcs.ast.IAttributeReplacement =
427     (("[[" lhs:Untimed_Constant_Pat ":-" "]]")
428      |
429     ("[[" ":-" rhs:Untimed_Constant_Pat "]]"));
```

```
430
431   Untimed_Constant_Pat implements Untimed_Constant
432                                    astimplements mc.tfcs.ast.IAttributePattern =
433     Untimed:["untimed"];
434
435   Delayed_Constant_Neg implements Delayed_Constant
436                                   astimplements mc.tfcs.ast.IAttributeNegation =
437     "not" "[[" Delayed_Constant_Pat "]]";
438
439   Delayed_Constant_Opt implements Delayed_Constant
440                                   astimplements mc.tfcs.ast.IAttributeOptional =
441     "opt" "[[" Delayed_Constant_Pat "]]";
442
443   Delayed_Constant_Rep implements Delayed_Constant
444                                astimplements mc.tfcs.ast.IAttributeReplacement =
445     (("[[" lhs:Delayed_Constant_Pat ":-" "]]")
446      |
447     ("[[" ":-" rhs:Delayed_Constant_Pat "]]"));
448
449   Delayed_Constant_Pat implements Delayed_Constant
450                                    astimplements mc.tfcs.ast.IAttributePattern =
451     Delayed:["delayed"];
452
453   Instant_Constant_Neg implements Instant_Constant
454                                   astimplements mc.tfcs.ast.IAttributeNegation =
455     "not" "[[" Instant_Constant_Pat "]]";
456
457   Instant_Constant_Opt implements Instant_Constant
458                                   astimplements mc.tfcs.ast.IAttributeOptional =
459     "opt" "[[" Instant_Constant_Pat "]]";
460
461   Instant_Constant_Rep implements Instant_Constant
462                                astimplements mc.tfcs.ast.IAttributeReplacement =
463     (("[[" lhs:Instant_Constant_Pat ":-" "]]")
464      |
465     ("[[" ":-" rhs:Instant_Constant_Pat "]]"));
466
467   Instant_Constant_Pat implements Instant_Constant
468                                    astimplements mc.tfcs.ast.IAttributePattern =
469     Instant:["instant"];
470
471   On_Constant_Neg implements On_Constant
472                                   astimplements mc.tfcs.ast.IAttributeNegation =
473     "not" "[[" On_Constant_Pat "]]";
474
475   On_Constant_Opt implements On_Constant
476                                   astimplements mc.tfcs.ast.IAttributeOptional =
477     "opt" "[[" On_Constant_Pat "]]";
478
479   On_Constant_Rep implements On_Constant
480                                astimplements mc.tfcs.ast.IAttributeReplacement =
481     (("[[" lhs:On_Constant_Pat ":-" "]]")
482      |
483     ("[[" ":-" rhs:On_Constant_Pat "]]"));
484
485   On_Constant_Pat implements On_Constant
486                                    astimplements mc.tfcs.ast.IAttributePattern =
487     On:["on"];
488
489   Off_Constant_Neg implements Off_Constant
490                                   astimplements mc.tfcs.ast.IAttributeNegation =
491     "not" "[[" Off_Constant_Pat "]]";
```

```
492
493   Off_Constant_Opt implements Off_Constant
494                                  astimplements mc.tfcs.ast.IAttributeOptional =
495     "opt" "[[" Off_Constant_Pat "]]";
496
497   Off_Constant_Rep implements Off_Constant
498                                astimplements mc.tfcs.ast.IAttributeReplacement =
499     (("[[" lhs:Off_Constant_Pat ":-" "]]")
500      |
501     ("[[" ":-" rhs:Off_Constant_Pat "]]"));
502
503   Off_Constant_Pat implements Off_Constant
504                                   astimplements mc.tfcs.ast.IAttributePattern =
505     Off:["off"];
506
507   Type_Constant_Neg implements Type_Constant
508                                  astimplements mc.tfcs.ast.IAttributeNegation =
509     "not" "[[" Type_Constant_Pat "]]";
510
511   Type_Constant_Opt implements Type_Constant
512                                  astimplements mc.tfcs.ast.IAttributeOptional =
513     "opt" "[[" Type_Constant_Pat "]]";
514
515   Type_Constant_Rep implements Type_Constant
516                                astimplements mc.tfcs.ast.IAttributeReplacement =
517     (("[[" lhs:Type_Constant_Pat ":-" "]]")
518      |
519     ("[[" ":-" rhs:Type_Constant_Pat "]]"));
520
521   Type_Constant_Pat implements Type_Constant
522                                   astimplements mc.tfcs.ast.IAttributePattern =
523     Type:["type"];
524
525   Outgoing_Constant_Neg implements Outgoing_Constant
526                                  astimplements mc.tfcs.ast.IAttributeNegation =
527     "not" "[[" Outgoing_Constant_Pat "]]";
528
529   Outgoing_Constant_Opt implements Outgoing_Constant
530                                  astimplements mc.tfcs.ast.IAttributeOptional =
531     "opt" "[[" Outgoing_Constant_Pat "]]";
532
533   Outgoing_Constant_Rep implements Outgoing_Constant
534                                astimplements mc.tfcs.ast.IAttributeReplacement =
535     (("[[" lhs:Outgoing_Constant_Pat ":-" "]]")
536      |
537     ("[[" ":-" rhs:Outgoing_Constant_Pat "]]"));
538
539   Outgoing_Constant_Pat implements Outgoing_Constant
540                                   astimplements mc.tfcs.ast.IAttributePattern =
541     outgoing:["out"];
542
543   Incoming_Constant_Neg implements Incoming_Constant
544                                  astimplements mc.tfcs.ast.IAttributeNegation =
545     "not" "[[" Incoming_Constant_Pat "]]";
546
547   Incoming_Constant_Opt implements Incoming_Constant
548                                  astimplements mc.tfcs.ast.IAttributeOptional =
549     "opt" "[[" Incoming_Constant_Pat "]]";
550
551   Incoming_Constant_Rep implements Incoming_Constant
552                                astimplements mc.tfcs.ast.IAttributeReplacement =
553     (("[[" lhs:Incoming_Constant_Pat ":-" "]]")
```

```
554       |
555      ("[[" ":-" rhs:Incoming_Constant_Pat "]]"));
556
557    Incoming_Constant_Pat implements Incoming_Constant
558                                  astimplements mc.tfcs.ast.IAttributePattern =
559      incoming:["in"];
560  }
```

Listing 3.3: Grammatik der DSTL MATrans.

C.4 Abgeleitete Automatons DSTL

MCG

```
grammar AutomatonsTR extends mc.TFCommons {

  TFRule =
  (Automaton| State| Transition)*
  TFFolding? TFAssignments? TFWhere? TFDo?;


  interface Automaton;
  interface State;
  interface Transition;
  interface Initial;
  interface Final;

  Automaton_Pat implements Automaton =
    ("aut" name:TfIdentifier "{"
      (State| Transition)*
     "}")
    | "Automaton" schemaVarName:Name
    | ("Automaton"? schemaVarName:Name  "[["
      "aut" name:TfIdentifier "{"
        (State| Transition)*
      "}"
     "]]");

  State_Pat implements State  =
    ((Initial | Final)*
    "st" name:TfIdentifier ";")
    | "State" schemaVarName:Name
    | ("State"? schemaVarName:Name "[["
      (Initial| Final)* "st" name:TfIdentifier ";"
    "]]");

  Transition_Pat implements Transition  =
    (from:TfIdentifier "->" to:TfIdentifier ";")
    | "Transition" schemaVarName:Name
    | ("Transition"? schemaVarName:Name "[["
       from:TfIdentifier "->" to:TfIdentifier ";"
    "]]");

  Initial_Pat implements Initial =
   "initial";

  Final_Pat implements Final =
    "final";

  Automaton_Rep implements Automaton =
   "[[" lhs:Automaton? ":-" rhs:Automaton? "]]";

  State_Rep implements State =
   "[[" lhs:State? ":-" rhs:State? "]]";

  Transition_Rep implements Transition =
   "[[" lhs:Transition? ":-" rhs:Transition? "]]";

  Initial_Rep implements Initial =
   "[[" lhs:Initial_Pat ":-" "]]" | "[[" ":-" rhs:Initial_Pat "]]";

```

```
58    Final_Rep implements Final =
59      "[[" lhs:Final_Pat ":-" "]]" | "[[" ":-" rhs:Final_Pat "]]";
60
61    Automaton_Neg implements Automaton =
62      "not" "[[" Automaton "]]";
63
64    State_Neg implements Automaton =
65      "not" "[[" State "]]";
66
67    Transition_Neg implements Transition =
68      "not" "[[" Transition "]]";
69
70    Initial_Neg implements Initial =
71      "not" "[[" "initial" "]]";
72
73    Final_Neg implements Final =
74      "not" "[[" "final" "]]";
75
76    Automaton_List implements Automaton =
77      "list" "[[" Automaton "]]";
78
79    State_List implements Automaton =
80      "list" "[[" State "]]";
81
82    Transition_List implements Transition =
83      "list" "[[" Transition "]]";
84
85    Automaton_Opt implements Automaton =
86     "opt" "[[" Automaton "]]";
87
88    State_Opt implements Automaton =
89     "opt" "[[" State "]]";
90
91    Transition_Opt implements Transition =
92      "opt" "[[" Transition "]]";
93
94    Initial_Opt implements Initial =
95      "opt" "[[" "initial" "]]";
96
97    Final_Opt implements Final =
98      "opt" "[[" "final" "]]";
99  }
```

C.5 DSTL-Generierung mit MontiTrans

pom.xml

```
<!-- Generierung einer DSTL -->
<plugin>
  <groupId>de.se_rwth.maven</groupId>
  <artifactId>se-groovy-maven-plugin</artifactId>
  <version>${se-groovy.maven.version}</version>
  <executions>
    <execution>
      <configuration>
        <classifiers>${grammars.classifier}</classifiers>
        <script>mc/tfcs/dstlgen.groovy</script>
        <baseClass>mc.tfcs.DSTLGenScript</baseClass>
        <arguments>
          <models>${basedir}/src/main/grammars</models>
          <out>${basedir}/target/generated-sources/</out>
          <handcodedPath>${basedir}/src/main/java</handcodedPath>
        </arguments>
      </configuration>
      <goals>
        <goal>execute</goal>
      </goals>
    </execution>
  </executions>
  <dependencies>
    <dependency>
      <groupId>de.monticore.tf</groupId>
      <artifactId>montitrans-dstlgen</artifactId>
      <version>${tfengine.version}</version>
    </dependency>
  </dependencies>
</plugin>
```

Listing 3.4: DSTL-Generierung mit MontiTrans.

C.6 Generierung von Java-Implementierungen für Transformationsregeln

pom.xml

```
<!-- Generate Java for Transformations-->
<plugin>
  <groupId>de.se_rwth.maven</groupId>
  <artifactId>se-groovy-maven-plugin</artifactId>
  <version>${se-groovy.maven.version}</version>
  <executions>
    <execution>
      <configuration>
        <classifiers>${grammars.classifier}</classifiers>
        <script>script/dstl2java.groovy</script>
        <baseClass>script.CD4ATransScript</baseClass>
        <arguments>
          <models>${basedir}/src/main/transformations</models>
          <out>${basedir}/target/transformations</out>
        </arguments>
      </configuration>
      <goals>
        <goal>execute</goal>
      </goals>
    </execution>
  </executions>

  <dependencies>
    <dependency>
      <groupId>de.monticore.tf</groupId>
      <artifactId>tfruntime</artifactId>
      <version>${tf.version}</version>
    </dependency>
    <dependency>
      <groupId>de.monticore.lang</groupId>
      <artifactId>cd4a-trans</artifactId>
      <version>${project.version}</version>
    </dependency>
  </dependencies>
</plugin>
```

Listing 3.5: Generierung von Java-Implementierungen für Transformationsregeln.

Anhang D

Curriculum Vitae

Name	Hölldobler
Vorname	Katrin
Geburtstag	20.11.1987
Geburtsort	Haan
Staatsangehörigkeit	deutsch
seit 2016	Wissenschaftliche Mitarbeiterin am Lehrstuhl für Software Engineering RWTH Aachen
2013 - 2016	Stipendiatin des DFG Graduiertenkollegs AlgoSyn (Algorithmic Synthesis of Reactive and Discrete-continuous Systems) RWTH Aachen
2012 - 2013	Wissenschaftliche Mitarbeiterin am Lehrstuhl für Software Engineering RWTH Aachen
2012	Abschluss als Master of Science in Informatik
2010	Abschluss als Bachelor of Science in Informatik
2007 - 2012	Studium der Informatik an der RWTH Aachen
2007	Abitur
1998 - 2007	Carl-Fuhlrott-Gymnasium, Wuppertal
1994 - 1998	Grundschule Distelbeck, Wuppertal

Abbildungsverzeichnis

Listings

Tabellenverzeichnis

Related Interesting Work from the SE Group, RWTH Aachen

Agile Model Based Software Engineering

Agility and modeling in the same project? This question was raised in [Rum04]: "Using an executable, yet abstract and multi-view modeling language for modeling, designing and programming still allows to use an agile development process." Modeling will be used in development projects much more, if the benefits become evident early, e.g with executable UML [Rum02] and tests [Rum03]. In [GKRS06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum12] and [Rum16], the UML/P, a variant of the UML especially designed for programming, refactoring and evolution, is defined. The language workbench MontiCore [GKR+06, GKR+08] is used to realize the UML/P [Sch12]. Links to further research, e.g., include a general discussion of how to manage and evolve models [LRSS10], a precise definition for model composition as well as model languages [HKR+09] and refactoring in various modeling and programming languages [PR03]. In [FHR08] we describe a set of general requirements for model quality. Finally [KRV06] discusses the additional roles and activities necessary in a DSL-based software development project. In [CEG+14] we discuss how to improve reliability of adaprivity through models at runtime, which will allow developers to delay design decisions to runtime adaptation.

Generative Software Engineering

The UML/P language family [Rum12, Rum11, Rum16] is a simplified and semantically sound derivate of the UML designed for product and test code generation. [Sch12] describes a flexible generator for the UML/P based on the MontiCore language workbench [KRV10, GKR+06, GKR+08]. In [KRV06], we discuss additional roles necessary in a model-based software development project. In [GKRS06] we discuss mechanisms to keep generated and handwritten code separated. In [Wei12] demonstrate how to systematically derive a transformation language in concrete syntax. To understand the implications of executability for UML, we discuss needs and advantages of executable modeling with UML in agile projects in [Rum04], how to apply UML for testing in [Rum03] and the advantages and perils of using modeling languages for programming in [Rum02].

Unified Modeling Language (UML)

Starting with an early identification of challenges for the standardization of the UML in [KER99] many of our contributions build on the UML/P variant, which is described in the two books [Rum16] and [Rum12] implemented in [Sch12]. Semantic variation points of the UML are discussed in [GR11]. We discuss formal semantics for UML [BHP+98] and describe UML semantics using the "System Model" [BCGR09a], [BCGR09b], [BCR07b] and [BCR07a]. Semantic variation points have, e.g., been applied to define class diagram semantics [CGR08]. A precisely defined semantics for variations is applied, when checking variants of class diagrams [MRR11c] and objects diagrams [MRR11d] or the consistency of both kinds of diagrams [MRR11e]. We also apply these concepts to activity diagrams [MRR11b] which allows us to check for semantic differences of activity diagrams [MRR11a]. The basic semantics for ADs and their semantic variation points is given in [GRR10]. We also discuss how to ensure and identify model quality [FHR08], how models, views and the system under development correlate to each other [BGH+98] and how to use modeling in agile development projects [Rum04], [Rum02]. The question how to adapt and

extend the UML is discussed in [PFR02] describing product line annotations for UML and more general discussions and insights on how to use meta-modeling for defining and adapting the UML are included in [EFLR99], [FELR98] and [SRVK10].

Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use, but need appropriate tooling. The MontiCore language workbench [GKR+06, KRV10, Kra10, GKR+08] allows the specification of an integrated abstract and concrete syntax format [KRV07b] for easy development. New languages and tools can be defined in modular forms [KRV08, GKR+07, Völ11] and can, thus, easily be reused. [Wei12] presents a tool that allows to create transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11]. A successful application has been carried out in the Air Traffic Management domain [ZPK+11]. Based on the concepts described above, meta modeling, model analyses and model evolution have been discussed in [LRSS10] and [SRVK10]. DSL quality [FHR08], instructions for defining views [GHK+07], guidelines to define DSLs [KKP+09] and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

Software Language Engineering

For a systematic definition of languages using composition of reusable and adaptable language components, we adopt an engineering viewpoint on these techniques. General ideas on how to engineer a language can be found in the GeMoC initiative [CBCR15, CCF+15]. As said, the MontiCore language workbench provides techniques for an integrated definition of languages [KRV07b, Kra10, KRV10]. In [SRVK10] we discuss the possibilities and the challenges using metamodels for language definition. Modular composition, however, is a core concept to reuse language components like in MontiCore for the frontend [Völ11, KRV08] and the backend [RRRW15]]. Language derivation is to our believe a promising technique to develop new languages for a specific purpose that rely on existing basic languages. How to automatically derive such a transformation language using concrete syntax of the base language is described in [HRW15, Wei12] and successfully applied to various DSLs. We also applied the language derivation technique to tagging languages that decorate a base language [GLRR15] and delta languages [HHK+15a, HHK+13], where a delta language is derived from a base language to be able to constructively describe differences between model variants usable to build feature sets.

Modeling Software Architecture & the MontiArc Tool

Distributed interactive systems communicate via messages on a bus, discrete event signals, streams of telephone or video data, method invocation, or data structures passed between software services. We use streams, statemachines and components [BR07] as well as expressive forms of composition and refinement [PR99] for semantics. Furthermore, we built a concrete tooling infrastructure called MontiArc [HRR12] for architecture design and extensions for states [RRW13b]. MontiArc was extended to describe variability [HRR+11] using deltas [HRRS11, HKR+11] and evolution on deltas [HRRS12]. [GHK+07] and [GHK+08] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants. [MRR14] provides a precise technique to verify consistency of architectural views [Rin14, MRR13] against a complete

architecture in order to increase reusability. Co-evolution of architecture is discussed in [MMR10] and a modeling technique to describe dynamic architectures is shown in [HRR98].

Compositionality & Modularity of Models

[HKR+09] motivates the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07] and algebraically underpinned in [HKR+07]. Semantic and methodical aspects of model composition [KRV08] led to the language workbench MontiCore [KRV10] that can even be used to develop modeling tools in a compositional form. A set of DSL design guidelines incorporates reuse through this form of composition [KKP+09]. [Völ11] examines the composition of context conditions respectively the underlying infrastructure of the symbol table. Modular editor generation is discussed in [KRV07a]. [RRRW15] applies compositionality to Robotics control. [CBCR15] (published in [CCF+15]) summarizes our approach to composition and remaining challenges in form of a conceptual model of the "globalized" use of DSLs. As a new form of decomposition of model information we have developed the concept of tagging languages in [GLRR15]. It allows to describe additional information for model elements in separated documents, facilitates reuse, and allows to type tags.

Semantics of Modeling Languages

The meaning of semantics and its principles like underspecification, language precision and detailedness is discussed in [HR04]. We defined a semantic domain called "System Model" by using mathematical theory in [RKB95, BHP+98] and [GKR96, KRB96]. An extended version especially suited for the UML is given in [BCGR09b] and in [BCGR09a] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied to class diagrams in [CGR08]. To better understand the effect of an evolved design, detection of semantic differencing as opposed to pure syntactical differences is needed [MRR10]. [MRR11a, MRR11b] encode a part of the semantics to handle semantic differences of activity diagrams and [MRR11e] compares class and object diagrams with regard to their semantics. In [BR07], a simplified mathematical model for distributed systems based on black-box behaviors of components is defined. Meta-modeling semantics is discussed in [EFLR99]. [BGH+97] discusses potential modeling languages for the description of an exemplary object interaction, today called sequence diagram. [BGH+98] discusses the relationships between a system, a view and a complete model in the context of the UML. [GR11] and [CGR09] discuss general requirements for a framework to describe semantic and syntactic variations of a modeling language. We apply these on class and object diagrams in [MRR11e] as well as activity diagrams in [GRR10]. [Rum12] defines the semantics in a variety of code and test case generation, refactoring and evolution techniques. [LRSS10] discusses evolution and related issues in greater detail.

Evolution & Transformation of Models

Models are the central artifact in model driven development, but as code they are not initially correct and need to be changed, evolved and maintained over time. Model transformation is therefore essential to effectively deal with models. Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04], refinement [PR99, KPR97, PR94], refactoring [Rum12, PR03], translating models from one language into another [MRR11c, Rum12] and systematic model transformation language development [Wei12]. [Rum04] describes how compre-

hensible sets of such transformations support software development and maintenance [LRSS10], technologies for evolving models within a language and across languages, and mapping architecture descriptions to their implementation [MMR10]. Automaton refinement is discussed in [PR94, KPR97], refining pipe-and-filter architectures is explained in [PR99]. Refactorings of models are important for model driven engineering as discussed in [PR01, PR03, Rum12]. Translation between languages, e.g., from class diagrams into Alloy [MRR11c] allows for comparing class diagrams on a semantic level.

Variability & Software Product Lines (SPL)

Products often exist in various variants, for example cars or mobile phones, where one manufacturer develops several products with many similarities but also many variations. Variants are managed in a Software Product Line (SPL) that captures product commonalities as well as differences. Feature diagrams describe variability in a top down fashion, e.g., in the automotive domain [GHK+08] using 150% models. Reducing overhead and associated costs is discussed in [GRJA12]. Delta modeling is a bottom up technique starting with a small, but complete base variant. Features are additive, but also can modify the core. A set of commonly applicable deltas configures a system variant. We discuss the application of this technique to Delta-MontiArc [HRR+11, HRR+11] and to Delta-Simulink [HKM+13]. Deltas can not only describe spacial variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK+13] and [HRW15] describe an approach to systematically derive delta languages. We also apply variability to modeling languages in order to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02]. Furthermore, we specified a systematic way to define variants of modeling languages [CGR09] and applied this as a semantic language refinement on Statecharts in [GR11].

Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12] are complex, distributed systems which control physical entities. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12] and autonomous driving [BR12a] to processes and tools to improve the development as well as the product itself [BBR07]. In the aviation domain, a modeling language for uncertainty and safety events was developed, which is of interest for the European airspace [ZPK+11]. A component and connector architecture description language suitable for the specific challenges in robotics is discussed in [RRW13b, RRW14]. Monitoring for smart and energy efficient buildings is developed as Energy Navigator toolset [KPR12, FPPR12, KLPR12].

State Based Modeling (Automata)

Today, many computer science theories are based on statemachines in various forms including Petri nets or temporal logics. Software engineering is particularly interested in using statemachines for modeling systems. Our contributions to state based modeling can currently be split into three parts: (1) understanding how to model object-oriented and distributed software using statemachines resp. Statecharts [GKR96, BCR07b, BCGR09b, BCGR09a], (2) understanding the refinement [PR94, RK96, Rum96] and composition [GR95] of statemachines, and (3) applying statemachines for modeling systems. In [Rum96] constructive transformation rules for refining

automata behavior are given and proven correct. This theory is applied to features in [KPR97]. Statemachines are embedded in the composition and behavioral specification concepts of Focus [BR07]. We apply these techniques, e.g., in MontiArcAutomaton [RRW13a, RRW14] as well as in building management systems [FLP+11].

Robotics

Robotics can be considered a special field within Cyber-Physical Systems which is defined by an inherent heterogeneity of involved domains, relevant platforms, and challenges. The engineering of robotics applications requires composition and interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers broad propagation of robotics applications. The MontiArcAutomaton language [RRW13a] extends ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13b, RRW14, RRRW15] that perfectly fit Robotic architectural modelling. The LightRocks [THR+13] framework allows robotics experts and laymen to model robotic assembly tasks.

Automotive, Autonomic Driving & Intelligent Driver Assistance

Introducing and connecting sophisticated driver assistance, infotainment and communication systems as well as advanced active and passive safety-systems result in complex embedded systems. As these feature-driven subsystems may be arbitrarily combined by the customer, a huge amount of distinct variants needs to be managed, developed and tested. A consistent requirements management that connects requirements with features in all phases of the development for the automotive domain is described in [GRJA12]. The conceptual gap between requirements and the logical architecture of a car is closed in [GHK+07, GHK+08]. [HKM+13] describes a tool for delta modeling for Simulink [HKM+13]. [HRRW12] discusses means to extract a well-defined Software Product Line from a set of copy and paste variants. [RSW+15] describes an approach to use model checking techniques to identify behavioral differences of Simulink models. Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12a, BR12b], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in development and evolution of autonomous car functionality, and thus enables us to develop software in an agile way [BR12a]. [MMR10] gives an overview of the current state-of-the-art in development and evolution on a more general level by considering any kind of critical system that relies on architectural descriptions. As tooling infrastructure, the SSElab storage, versioning and management services [HKR12] are essential for many projects.

Energy Management

In the past years, it became more and more evident that saving energy and reducing CO2 emissions is an important challenge. Thus, energy management in buildings as well as in neighbourhoods becomes equally important to efficiently use the generated energy. Within several research projects, we developed methodologies and solutions for integrating heterogeneous systems at different scales. During the design phase, the Energy Navigators Active Functional Specification (AFS) [FPPR12, KPR12] is used for technical specification of building services already. We adapted the well-known concept of statemachines to be able to describe different states of a facility

and to validate it against the monitored values [FLP+11]. We show how our data model, the constraint rules and the evaluation approach to compare sensor data can be applied [KLPR12].

Cloud Computing & Enterprise Information Systems

The paradigm of Cloud Computing is arising out of a convergence of existing technologies for web-based application and service architectures with high complexity, criticality and new application domains. It promises to enable new business models, to lower the barrier for web-based innovations and to increase the efficiency and cost-effectiveness of web development [KRR14]. Application classes like Cyber-Physical Systems and their privacy [HHK+14, HHK+15b], Big Data, App and Service Ecosystems bring attention to aspects like responsiveness, privacy and open platforms. Regardless of the application domain, developers of such systems are in need for robust methods and efficient, easy-to-use languages and tools [KRS12]. We tackle these challenges by perusing a model-based, generative approach [NPR13]. The core of this approach are different modeling languages that describe different aspects of a cloud-based system in a concise and technology-agnostic way. Software architecture and infrastructure models describe the system and its physical distribution on a large scale. We apply cloud technology for the services we develop, e.g., the SSELab [HKR12] and the Energy Navigator [FPPR12, KPR12] but also for our tool demonstrators and our own development platforms. New services, e.g., collecting data from temperature, cars etc. can now easily be developed.

[BBR07] Christian Basarke, Christian Berger, and Bernhard Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 4(12):1158–1174, 2007.

[BCGR09a] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, November 2009.

[BCGR09b] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, November 2009.

[BCR07a] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, Germany, February 2007.

[BCR07b] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, Germany, February 2007.

[BGH+97] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. In *Object-oriented Behavioral Semantics Workshop (OOPSLA'97)*, Technical Report TUM-I9737, Germany, 1997. TU Munich.

[BGH+98] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, Germany, 1998.

[BHP+98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In *Workshop on Requirements Targeting Software and Systems Engineering (RTSE'97)*, LNCS 1526, pages 43–68. Springer, 1998.

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007.

[BR12a] Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Automotive Software Engineering Workshop (ASE'12)*, pages 789–798, 2012.

[BR12b] Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*, pages 243–271. Springer, Germany, 2012.

[CBCR15] Tony Clark, Mark van den Brand, Benoit Combemale, and Bernhard Rumpe. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*, LNCS 9400, pages 7–20. Springer, 2015.

[CCF+15] Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe, editors. *Globalizing Domain-Specific Languages*, LNCS 9400. Springer, 2015.

[CEG+14] Betty Cheng, Kerstin Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi

Müller, Patrizio Pelliccione, Anna Perini, Nauman Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In *Models@run.time*, LNCS 8378, pages 101–136. Springer, Germany, 2014.

[CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany, 2008.

[CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, LNCS 5795, pages 670–684. Springer, 2009.

[EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Meta-Modelling Semantics of UML. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 45–60. Kluver Academic Publisher, 1999.

[FELR98] Robert France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19(7):325–334, November 1998.

[FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008.

[FLP+11] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. State-based Modeling of Buildings and Facilities. In *Enhanced Building Operations Conference (ICEBO'11)*, 2011.

[FPPR12] M. Norbert Fisch, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Energy Efficiency in Commercial Buildings Conference(IEECB'12)*, 2012.

[GHK+07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based Modeling of Function Nets. In *Object-oriented Modelling of Embedded Real-Time Systems Workshop (OMER4'07)*, 2007.

[GHK+08] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, 2008.

[GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen*, Informatik Bericht 2008-01, pages 76–89. TU Braunschweig, 2008.

[GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TU Munich, Germany, July 1996.

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006.

[GKR+07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völ-

kel. Textbased Modeling. In *4th International Workshop on Software Language Engineering, Nashville*, Informatik-Bericht 4/2007. Johannes-Gutenberg-Universität Mainz, 2007.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008.

[GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, and Martin Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Modellierung 2006 Conference*, LNI 82, Seiten 67–81, 2006.

[GLRR15] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. Engineering Tagging Languages for DSLs. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 34–43. ACM/IEEE, 2015.

[GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TU Munich, Germany, October 1995.

[GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems*, LNCS 6662, pages 17–32. Springer, 2011.

[GRJA12] Tim Gülke, Bernhard Rumpe, Martin Jansen, and Joachim Axmann. High-Level Requirements Management and Complexity Costs in Automotive Development Projects: A Problem Statement. In *Requirements Engineering: Foundation for Software Quality (REFSQ'12)*, 2012.

[GRR10] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, LNCS 6394, pages 331–345. Springer, 2010.

[HHK+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Software Product Line Conference (SPLC'13)*, pages 22–31. ACM, 2013.

[HHK+14] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. User-driven Privacy Enforcement for Cloud-based Services in the Internet of Things. In *Conference on Future Internet of Things and Cloud (FiCloud'14)*. IEEE, 2014.

[HHK+15a] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, Ina Schaefer, and Christoph Schulze. Systematic Synthesis of Delta Modeling Languages. *Journal on Software Tools for Technology Transfer (STTT)*, 17(5):601–626, October 2015.

[HHK+15b] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. A comprehensive approach to privacy in the cloud-based Internet of Things. *Future Generation Computer Systems*, 56:701–718, 2015.

[HKM+13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-Class Variability Modeling in Matlab/Simulink. In *Variability Modelling of Software-intensive Systems Workshop (VaMoS'13)*, pages 11–18. ACM, 2013.

[HKR+07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In *Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'07)*, LNCS 4530, pages 99–113. Springer, Germany, 2007.

[HKR+09] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In *Conference on Software Engineeering in Research and Practice (SERP'09)*, pages 172–176, July 2009.

[HKR+11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *Software Architecture Conference (ECSA'11)*, pages 6:1–6:10. ACM, 2011.

[HKR12] Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Developing Tools as Plug-Ins Workshop (TOPI'12)*, pages 61–66. IEEE, 2012.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004.

[HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling Dynamic Component Interfaces. In *Technology of Object-Oriented Languages and Systems (TOOLS 26)*, pages 58–70. IEEE, 1998.

[HRR+11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Software Product Lines Conference (SPLC'11)*, pages 150–159. IEEE, 2011.

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012.

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VII*, pages 1 – 10. fortiss GmbH, 2011.

[HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208. Springer, 2012.

[HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering Conference (SE'12)*, LNI 198, Seiten 181–192, 2012.

[HRW15] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 136–145. ACM/IEEE, 2015.

[KER99] Stuart Kent, Andy Evans, and Bernhard Rumpe. UML Semantics FAQ. In A. Moreira and S. Demeyer, editors, *Object-Oriented Technology, ECOOP'99 Workshop Reader*, LNCS 1743, Berlin, 1999. Springer Verlag.

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schind-

ler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM'09)*, Techreport B-108, pages 7–13. Helsinki School of Economics, October 2009.

[KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Modelling of the Physical World Workshop (MOTPW'12)*, pages 2:1–2:6. ACM, October 2012.

[KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*, pages 284–297. IOS-Press, 1997.

[KPR12] Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In H. Lichter and B. Rumpe, Editoren, *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering, Band 14. Shaker Verlag, Aachen, Deutschland, 2012.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, März 2010.

[KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Workshop on Formal Methods for Open Object-based Distributed Systems*, IFIP Advances in Information and Communication Technology, pages 323–338. Chapmann & Hall, 1996.

[KRR14] Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe. *Trusted Cloud Computing*. Springer, Schweiz, December 2014.

[KRS12] Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, Seiten 113–116. VDI Verlag, 2012.

[KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Domain-Specific Modeling Workshop (DSM'06)*, Technical Report TR-37, pages 150–158. Jyväskylä University, Finland, 2006.

[KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Domain-Specific Modeling Workshop (DSM'07)*, Technical Reports TR-38. Jyväskylä University, Finland, 2007.

[KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'07)*, LNCS 4735, pages 286–300. Springer, 2007.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe'08)*, LNBIP 11, pages 297–315. Springer, 2008.

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for

Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

[LRSS10] Tihamer Levendovszky, Bernhard Rumpe, Bernhard Schätz, and Jonathan Sprinkle. Model Evolution and Management. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 241–270. Springer, 2010.

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, May 2010.

[MRR10] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. A Manifesto for Semantic Model Differencing. In *Proceedings Int. Workshop on Models and Evolution (ME'10)*, LNCS 6627, pages 194–203. Springer, 2010.

[MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Conference on Foundations of Software Engineering (ESEC/FSE '11)*, pages 179–189. ACM, 2011.

[MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011.

[MRR11c] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 592–607. Springer, 2011.

[MRR11d] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *Object-Oriented Programming Conference (ECOOP'11)*, LNCS 6813, pages 281–305. Springer, 2011.

[MRR11e] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 153–167. Springer, 2011.

[MRR13] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views. In Meyer, B. and Baresi, L. and Mezini, M., editor, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, pages 444–454. ACM New York, 2013.

[MRR14] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *Software Engineering Conference (ICSE'14)*, pages 95–105. ACM, 2014.

[NPR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, CEUR Workshop Proceedings 1118, pages 15–24, 2013.

[PFR02] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product Line Annotations with UML-F. In *Software Product Lines Conference (SPLC'02)*, LNCS 2379, pages 188–197. Springer, 2002.

[PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *Proceedings of the Industrial Benefit of Formal Methods (FME'94)*, LNCS 873, pages 154–174. Springer, 1994.

[PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In *Congress on Formal Methods in the Development of Computing System (FM'99)*, LNCS 1708, pages 96–115. Springer, 1999.

[PR01] Jan Philipps and Bernhard Rumpe. Roots of Refactoring. In Kilov, H. and Baclavski, K., editor, *Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15*. Northeastern University, 2001.

[PR03] Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In Kilov, H. and Baclavski, K., editor, *Practical Foundations of Business and System Specifications*, pages 281–297. Kluwer Academic Publishers, 2003.

[Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, 2014.

[RK96] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In B. Harvey and H. Kilov, editors, *Object-Oriented Behavioral Specifications*, pages 265–286. Kluwer Academic Publishers, 1996.

[RKB95] Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technischer Bericht TUM-I9510, TU München, Deutschland, März 1995.

[RRRW15] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 6(1):33–57, 2015.

[RRW13a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In *Software Engineering Workshopband (SE'13)*, LNI 215, pages 155–170, 2013.

[RRW13b] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013.

[RRW14] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Aachener Informatik-Berichte, Software Engineering, Band 20. Shaker Verlag, December 2014.

[RSW+15] Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, Jan Oliver Ringert, and Peter Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *Software Product Line Conference (SPLC'15)*, pages 141–150. ACM, 2015.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996.

[Rum02] Bernhard Rumpe. Executable Modeling with UML - A Vision or a Nightmare? In T. Clark and J. Warmer, editors, *Issues & Trends of Information Technology Management in Contemporary Associations, Seattle*, pages 697–701. Idea Group Publishing, London, 2002.

[Rum03] Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In *Symposium on Formal Methods for Components and Objects (FMCO'02)*, LNCS 2852, pages

380–402. Springer, November 2003.

[Rum04] Bernhard Rumpe. Agile Modeling with the UML. In *Workshop on Radical Innovations of Software and Systems Engineering in the Future (RISSEF'02)*, LNCS 2941, pages 297–309. Springer, October 2004.

[Rum11] Bernhard Rumpe. *Modellierung mit UML, 2te Auflage*. Springer Berlin, September 2011.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring, 2te Auflage*. Springer Berlin, Juni 2012.

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[SRVK10] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling: State of the Art and Research Challenges. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 57–76. Springer, 2010.

[THR+13] Ulrike Thomas, Gerd Hirzinger, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *Conference on Robotics and Automation (ICRA'13)*, pages 461–466. IEEE, 2013.

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012.

[ZPK+11] Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In *Proceedings of the SESAR Innovation Days*. EUROCONTROL, 2011.